2.1 Introduction

> "I speak Spanish to God, Italian to women, French to men, and German to my horse."
> *Charles V, Holy Roman Emperor, (1500-1558)*

To command a computer's hardware, you must speak its language. The words of a computer's language are called instructions, and its vocabulary is called an *instruction set*. In this chapter, you will see the instruction set of a real computer, both in the form written by people and in the form read by the computer. We introduce instructions in a top-down fashion. Starting from a notation that looks like a restricted programming language, we refine it step-by-step until you see the real language of a real computer. COD Chapter 3 (Arithmetic for Computers) continues our downward descent, unveiling the hardware for arithmetic and the representation of floating-point numbers.

Instruction set: The vocabulary of commands understood by a given architecture.

You might think that the languages of computers would be as diverse as those of people, but in reality, computer languages are quite similar, more like regional dialects than like independent languages. Hence, once you learn one, it is easy to pick up others.

The chosen instruction set comes from MIPS Technologies, and is an elegant example of the instruction sets designed since the 1980s. To demonstrate how easy it is to pick up other instruction sets, we will take a quick look at three other popular instruction sets.

1. ARMv7 is similar to MIPS. More than 100 billion chips with ARM processors will be manufactured in between 2017 and 2020, making it the most popular instruction set in the world.
2. The second example is the Intel x86, which powers both the PC and the cloud of the PostPC Era.
3. The third example is ARMv8, which extends the address size of the ARMv7 from 32 bits to 64 bits. Ironically, as we shall see, this 2013 instruction set is closer to MIPS than it is to ARMv7.

This similarity of instruction sets occurs because all computers are constructed from hardware technologies based on similar underlying principles and because there are a few basic operations that all computers must provide. Moreover, computer designers have a common goal: to find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost and energy. This goal is time-honored; the following quote was written before you could buy a computer, and it is as true today as it was in 1946:

> "It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations …. The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.
> *Burks, Goldstine, and von Neumann, 1946*

The "simplicity of the equipment" is as valuable a consideration for today's computers as it was for those of the 1950s. The goal of this chapter is to teach an instruction set that follows this advice, showing both how it is represented in hardware and the relationship between high-level programming languages and this more primitive one. Our examples are in the C programming language; COD Section 2.15 (Advanced material: Compiling C and interpreting Java) shows how these would change for an object-oriented language like Java.

By learning how to represent instructions, you will also discover the secret of computing: the *stored-program concept*. Moreover, you will exercise your "foreign language" skills by writing programs in the language of the computer and running them on the simulator that comes with this book. You will also see the impact of programming languages and compiler optimization on performance. We conclude with a look at the historical evolution of instruction sets and an overview of other computer dialects.

Stored-program concept: The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored-program computer.

We reveal our first instruction set a piece at a time, giving the rationale along with the computer structures. This top-down, step-by-step tutorial weaves the components with their explanations, making the computer's language more palatable.

PARTICIPATION ACTIVITY 2.1.1: Instruction sets.

1) An instruction set is a particular program provided in the language of a computer.
   - ○ True
   - ○ False

2) The instruction sets of different computers are quite similar to one another.

- ○ True
- ○ False

3) Instructions, as well as data, can be stored in memory as numbers.

- ○ True
- ○ False

2.2 Operations of the computer hardware

> "There must certainly be instructions for performing the fundamental arithmetic operations."
> *Burks, Goldstine, and von Neumann, 1946*

Every computer must be able to perform arithmetic. The MIPS assembly language notation

```
add a, b, c
```

instructs a computer to add the two variables `b` and `c` and to put their sum in `a`.

PARTICIPATION ACTIVITY 2.2.1: A simple add instruction.

Animation content:

Static figure: An assembly code instruction for addition and a table with three rows are displayed.
Begin Assembly code:

add a, b, c
End Assembly code.
Next to the instruction, a table shows the variables and their values as follows:
Row 1, represented by the variable a, has a value that is initially unknown, represented by a question mark.
Row 2, represented by the variable b, has a value of 5.
Row 3, represented by the variable c, has a value of 36.
Step 1: This instruction adds b + c, and puts the sum in a.
The assembly instruction executes, adding the values from b and c, and stores the result in a.
The question mark in row 1 is replaced with 41.
A new assembly code instruction for addition and a second table with two rows are displayed.
Begin Assembly code:
add a, a, b
End Assembly code.
Next to the instruction, a table shows the variables and their values as follows:
Row 1, represented by the variable a, has a value of 3.
Row 2, represented by the variable b, has a value of 4.
Step 2: If a is initially 3 and b is 4, this instruction overwrites a with 7.
The assembly instruction executes, adding the values from a and b, and stores the result in a.
The previous value in row 1, is overwritten with 7.

Animation captions:

1. This instruction adds $b + c$, and puts the sum in a.
2. If a is initially 3 and b is 4, this instruction overwrites a with 7.

PARTICIPATION ACTIVITY 2.2.2: A simple add instruction.

1) Type a MIPS add instruction that computes:
z = x + y

Check **Show answer**

2) Type a MIPS add instruction that computes:
a = a + b

Check **Show answer**

This notation is rigid in that each MIPS arithmetic instruction performs only one operation and must always have exactly three variables. For example, suppose we want to place the sum of four variables `b`, `c`, `d`, and `e` into variable `a`. (In this section we are being deliberately vague about what a "variable" is; in COD Section 2.3 (Operands of the computer hardware) we'll explain in detail.)

The following sequence of instructions adds the four variables:

```
add a, b, c   # The sum of b and c is placed in a
add a, a, d   # The sum of b, c, and d is now in a
add a, a, e   # The sum of b, c, d, and e is now in a
```

Thus, it takes three instructions to sum the four variables.

PARTICIPATION ACTIVITY 2.2.3: Three add instructions executing one at a time.

```
add a, b, c   # The sum of b and c is placed in a
    6  5  1
add a, a, d   # The sum of b, c, and d is now in a
    8  6  2
add a, a, e   # The sum of b, c, d, and e is now in a
   15  8  7
```

? 6 8 15	a
5	b
1	c
2	d
7	e

Animation content:

Static figure: Three assembly code instructions for addition and a table with five rows are displayed.
Begin Assembly code:
add a, b, c # The sum of b and c is placed in a
add a, a, d # The sum of b, c, and d is now in a
add a, a, e # The sum of b, c, d, and e is now in a
End Assembly code.

Next to the instructions, a table shows the variables and their values as follows:
Row 1, represented by the variable a, has a value that is initially unknown, represented by a

question mark.
Row 2, represented by the variable b, has a value of 5.
Row 3, represented by the variable c, has a value of 1.
Row 4, represented by the variable d, has a value of 2.
Row 5, represented by the variable e, has a value of 7.

Step 1: The first instruction executes. a now has b + c.
Execution of assembly instruction, add a, b, c. The question mark in row 1 is replaced with 6.

Step 2: The second instruction executes. a now has b + c + d.
Execution of assembly instruction, add a, a, d. The previous value in row 1 is overwritten with 8.

Step 3: The third instruction executes. a now has b + c + d + e. Because each machine instruction can only add two values, three instructions were needed.
Execution of assembly instruction, add a, a, e. The previous value in row 1 is overwritten with 15.

Animation captions:

1. The first instruction executes. a now has b + c.
2. The second instruction executes. a now has b + c + d.
3. The third instruction executes. a now has b + c + d + e. Because each machine instruction can only add two values, three instructions were needed.

Above, the words to the right of the sharp symbol (#) on each line is a ***comment*** for the human reader, so the computer ignores them. Note that unlike other programming languages, each line of this language can contain at most one instruction. Another difference from C is that comments always terminate at the end of a line.

Similarly to the add instruction, `sub a, b, c` computes b - c and puts the result in a. A table at the end of this section lists more MIPS instructions.

PARTICIPATION ACTIVITY 2.2.4: Basic instruction execution.

Given: b = 2, c = 5, d = 1.

1) `add a, b, c`

Final value of a is ____.

Check **Show answer**

2)
```
add t, d, c
add a, t, c
```

Final value of a is ____.

Check **Show answer**

3)
```
sub t, c, b
add a, t, d
```

Final value of a is ____.

Check **Show answer**

The natural number of operands for an operation like addition is three: the two numbers being added together and a place to put the sum. Requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple: hardware for a variable number of operands is more complicated than hardware for a fixed number. This situation illustrates the first of three underlying principles of hardware design:

Design Principle 1: Simplicity favors regularity.

We can now show, in the two examples that follow, the relationship of programs written in higher-level programming languages to programs in this more primitive notation.

PARTICIPATION ACTIVITY 2.2.5: Example of compiling two C assignment statements into MIPS.


```
a = b + c;
d = a - e;
```

```
add a, b, c      # register a contains b + c
    8  2  6

sub d, a, e      # register d contains a - e
    5  8  3
```

8	a
2	b
6	c
5	d
3	e

Animation content:

Static figure: A table with five rows representing registers is displayed.

Begin table:
Row 1, represented by the register a, has no value.
Row 2, represented by the register b, has a value of 2.
Row 3, represented by the register c, has a value of 6.
Row 4, represented by the register d, has no value.
Row 5, represented by the register e, has a value of 3.
End table.
Step 1: The translation from C to assembly language instructions is performed by the compiler.
Two lines of C code are displayed.
Begin C code:
a = b + c;
d = a - e;
End C code.

Step 2: This assembly instruction operates on two source operands and places the result in one destination operand.
The C code assignment statement, a = b + c, is highlighted and the assembly instruction, add a, b, c, is displayed.

Step 3: The two simple C statements compile directly into two assembly language instructions.
The C code assignment statement, d = a - e, is highlighted and the assembly instruction, sub d, a, e, is displayed.

Step 4: The add instruction puts the result of 2 + 6 in register a. The subtract instruction puts the result of 8 - 3 in register d.
The add instruction, add a, b, c, is highlighted and register a now contains the value 8, which is the sum of b (2) and c (6).
The subtract instruction, sub d, a, e, is highlighted and register d now contains the value 5, which is the result of subtracting e (3) from a (8).

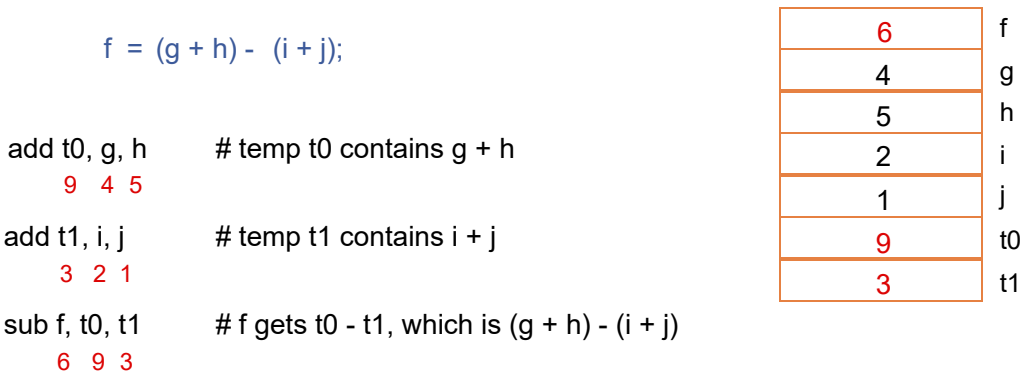
Animation captions:

1. The translation from C to assembly language instructions is performed by the compiler.
2. This assembly instruction operates on two source operands and places the result in one

destination operand.

3. The two simple C statements compile directly into two assembly language instructions.
4. The add instruction puts the result of 2 + 6 in register a. The subtract instruction puts the result of $8 - 3$ in register d.

PARTICIPATION ACTIVITY 2.2.6: Example of compiling a complex C assignment into MIPS.



Animation content:

Static figure: A C code snippet and corresponding assembly language instructions are shown. Adjacent to the code, a table with 7 rows representing registers is displayed.

Begin C code:
f = (g + h) - (i + j);
End C code.

Begin assembly code:
add t0, g, h # temp t0 contains g + h
add t1, i, j # temp t1 contains i + j
sub f, t0, t1 # f gets t0 - t1, which is (g + h) - (i + j)
End assembly code.

Begin table:
Row 1, represented by the register f, has no value.
Row 2, represented by the register g, has a value of 4.
Row 3, represented by the register h, has a value of 5.

Row 4, represented by the register i, has a value of 2.
Row 5, represented by the register j, has a value of 1.
Row 6, represented by the register t0, has no value.
Row 7, represented by the register t1, has no value.
End table.

Step 1: A complex statement contains five variables f, g, h, i, and j. The compiler breaks this statement into several assembly instructions, since only one operation is performed per assembly instruction.
The C code assignment statement, f = (g + h) - (i + j), is displayed.

Step 2: The compiler first generates an instruction to calculate g + h, putting the result in temporary variable t0.
A part of the C code assignment statement, (g + h), is highlighted and the assembly instruction, add t0, g, h, is displayed.

Step 3: Although the next operation is subtract, the compiler must calculate the sum of i and j before the subtract. So, next the compiler generates an instruction to calculate i + j, putting the result in t1.
A part of the C code assignment statement, (i + j), is highlighted and the assembly instruction, add t1, i, j, is displayed.
Step 4: Finally, the compiler generates an instruction to calculate t0 - t1, putting the result in f.
The assembly instruction, sub f, t0, t1, which equals (g + h) - (i + j), is displayed.

Step 5: For the given values of g, h, i, and j, the instructions put the intermediate result of 4 + 5 or 9 into t0, and then 2 + 1 or 3 into t1. Finally, 9 - 3 or 6 is put into register f.
The add instruction, add t0, g, h, is highlighted and register t0 now contains the value 9, which is the sum of g (4) and h (5).
The add instruction, add t1, i, j, is highlighted and register t1 now contains the value 3, which is the sum of i (2) and j (1).
The subtract instruction, sub f, t0, t1, is highlighted and register f now contains the value 6, which is the result of subtracting t1 (3) from t0 (9).

Animation captions:

1. A complex statement contains five variables f, g, h, i, and j. The compiler breaks this statement into several assembly instructions, since only one operation is performed per assembly instruction.
2. The compiler first generates an instruction to calculate g + h, putting the result in temporary variable t0.
3. Although the next operation is subtract, the compiler must calculate the sum of i and j before the subtract. So, next the compiler generates an instruction to calculate i + j, putting the result in t1.

4. Finally, the compiler generates an instruction to calculate t0 - t1, putting the result in f.
5. For the given values of g, h, i, and j, the instructions put the intermediate result of 4 + 5 or 9 into t0, and then 2 + 1 or 3 into t1. Finally, $\mathbf{9 - 3}$ or 6 is put into register f.

PARTICIPATION ACTIVITY 2.2.7: Compiling an expression.

Order the assembly instructions to calculate the expression: a = b + c + d - e

How to use this tool

```
add t0, b, c    sub a, t1, e    add t1, t0, d
```

1

2

3

Reset

PARTICIPATION ACTIVITY 2.2.8: Compiling an expression.

Which instruction is *incorrect* for calculating a = b + c + d - e?

1)
```
add  t0, c, d
```
```
add  t0, t0, b
```
```
sub  a, e, t0
```


PARTICIPATION ACTIVITY 2.2.9: Instruction set simulator: add and sub.

Use this simulator to experiment with various add and sub instructions.

Registers

Instruction 1	add	b	b	c	2	a
Instruction 2	add	e	e	d	5	b
Instruction 3	---	b	b	c	19	c
Instruction 4	---	b	b	c	3	d
Instruction 5	---	b	b	c		
Instruction 6	---	b	b	c	11	e

Enter simulation **Execute instruction**

PARTICIPATION ACTIVITY 2.2.10: Check yourself: High-level language and lines of code.

For a given function, which programming language likely takes the most lines of source code?

How to use this tool

C **Java** **MIPS assembly language**

1 (Requires most lines)

2

3 (Requires fewest lines)

Reset

Elaboration

To increase portability, Java was originally envisioned as relying on a software interpreter. The instruction set of this interpreter is called Java bytecodes (see COD Section 2.15 (Advanced Material: Compiling C and Interpreting Java)), which is quite different from the MIPS instruction set. To get performance close to the equivalent C program, Java systems today typically compile Java bytecodes into the native

instruction sets like MIPS. Because this compilation is normally done much later than for C programs, such Java compilers are often called Just In Time (JIT) compilers. COD Section 2.12 (Translating and Starting a Program) shows how JITs are used later than C compilers in the start-up process, and COD Section 2.13 (A C Sort Example to Put It All Together) shows the performance consequences of compiling versus interpreting Java programs.

Table 2.2.1: MIPS operands revealed in this chapter (COD Figure 2.1).

Name	Example	Comments
32 registers	`$s0..$s7, $t0..$t9, $zero, $a0..$a3, $v0..$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory [4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

PARTICIPATION ACTIVITY 2.2.11: MIPS registers.

Indicate whether each name refers to a MIPS register.

1) $s3

- ○ Yes
- ○ No

2) $s9

- Yes
- No

3) $t9

- Yes
- No

4) a2

- Yes
- No

5) $zero

- Yes
- No

6) $one

- Yes
- No

7) Memory[0]

- Yes
- No

Table 2.2.2: MIPS assembly language revealed in this chapter (COD Figure 2.1).

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three register operands
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three register operands

	add immediate	`addi $s1, $s2, 20`	`$s1 = $s2 + 20`	Used to add constants
Data transfer	load word	`lw $s1, 20($s2)`	`$s1` = Memory[`$s2` + 20]	Word from memory to register
	store word	`sw $s1, 20($s2)`	Memory[`$s2` + 20] = `$s1`	Word from register to memory
	load half	`lh $s1, 20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	load half unsigned	`lhu $s1, 20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	store half	`sh $s1, 20($s2)`	Memory[`$s2` + 20] = `$s1`	Halfword register to memory
	load byte	`lb $s1, 20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1, 20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	store byte	`sb $s1, 20($s2)`	Memory[`$s2` + 20] = `$s1`	Byte from register to memory
	load linked word	`ll $s1, 20($s2)`	`$s1` = Memory[`$s2` + 20]	Load word as 1st half of atomic swap

	store condition. word	`sc $s1, 20($s2)`	Memory[`$s2` + 20] = `$s1`; `$s1` = 0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1, 20`	`$s1` = $20 * 2^{16}$	Loads constant in upper 16 bits
Logical	and	`and $s1, $s2, $s3`	`$s1 = $s2 & $s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1, $s2, $s3`	`$s1 = $s2 | $s3`	Three reg. operands; bit-by-bit OR
	nor	`nor $s1, $s2, $s3`	`$s1 = ~($s2 | $s3)`	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1, $s2, 20`	`$s1 = $s2 & 20`	Bit-by-bit AND reg with constant
	or immediate	`ori $s1, $s2, 20`	`$s1 = $s2 | 20`	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1, $s2, 10`	`$s1 = $s2 << 10`	Shift left by constant
	shift right logical	`srl $s1, $s2, 10`	`$s1 = $s2 >> 10`	Shift right by constant

Conditional branch	branch on equal	`beq $s1, $s2, 25`	if(`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if(`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative branch
	set on less than	`slt $s1, $s2, $s3`	if(`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1, $s2, $s3`	if(`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than unsigned
	set less than immediate	`slti $s1, $s2, 20`	if(`$s2 < 20`) `$s1` = 1; else `$s1` = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1, $s2, 20`	if(`$s2 < 20`) `$s1` = 1; else `$s1` = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

PARTICIPATION ACTIVITY 2.2.12: MIPS instructions.

Indicate whether each is a valid MIPS instruction.

1) `add $s1, $s2, $s3`

- ○ Valid
- ○ Not valid

2) `addi $s1, $s3, 50`

- ○ Valid
- ○ Not valid

3) `lw $s1, 20($s8)`

- ○ Valid
- ○ Not valid

4) `jump 2500`

- ○ Valid
- ○ Not valid

2.3 Operands of the computer hardware

Unlike programs in high-level languages, the operands of arithmetic instructions are restricted; they must be from a limited number of special locations built directly in hardware called registers. Registers are primitives used in hardware design that are also visible to the programmer when the computer is completed, so you can think of registers as the bricks of computer construction. The size of a register in the MIPS architecture is 32 bits; groups of 32 bits occur so frequently that they are given the name *word* in the MIPS architecture.

Word: The natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the MIPS architecture.

One major difference between the variables of a programming language and registers is the limited number of registers, typically 32 on current computers, like MIPS. Thus, continuing in our top-down, stepwise evolution of the symbolic representation of the MIPS language, in this section we have added the restriction that the three operands of MIPS arithmetic instructions must each be chosen from one of the 32-bit registers.

The reason for the limit of 32 registers may be found in the second of our three underlying design

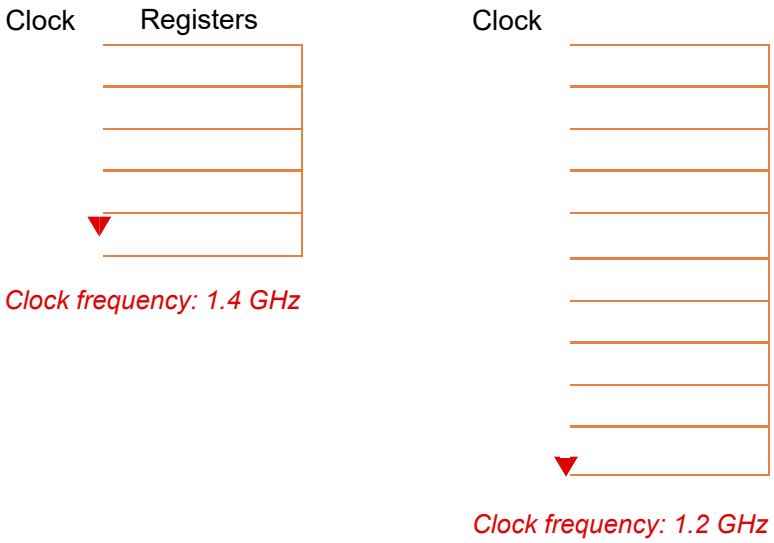
principles of hardware technology:

Design Principle 2: Smaller is faster.

A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.

Guidelines such as "smaller is faster" are not absolutes; 31 registers may not be faster than 32. Yet, the truth behind such observations causes computer designers to take them seriously. In this case, the designer must balance the craving of programs for more registers with the designer's desire to keep the clock cycle fast. Another reason for not using more than 32 is the number of bits it would take in the instruction format.

PARTICIPATION ACTIVITY 2.3.1: Smaller is faster: With fewer registers, the clock frequency can be made faster, because the clock signal requires less time to reach every register.



Animation content:

Static figure: Two groups of registers, each with a clock signal. The first group has 5 registers and a clock frequency of 1.4 GHz. The second group has 10 registers and a clock frequency of 1.2 GHz.
Step 1: The clock signal requires time to reach every register.
A propagation delay occurs in the first group with 5 registers as the clock signal travels from the clock source to reach each of the registers.
Step 2: With more registers, the clock signal requires more time to reach every register, so the clock frequency must be slower.

The reduction in clock frequency in the second group with 10 registers is due to the higher number of registers that the clock signal must reach to maintain signal integrity over the increased distance.

Animation captions:

1. The clock signal requires time to reach every register.
2. With more registers, the clock signal requires more time to reach every register, so the clock frequency must be slower.

PARTICIPATION ACTIVITY 2.3.2: Registers.

1) In the instruction below, a, b, and c are operands. In such an arithmetic instruction, each operand comes from special hardware called a _____.

```
add a, b, c
```

Check **Show answer**

2) In the MIPS architecture, each register is _____ bits wide.

Check **Show answer**

3) More registers may benefit an assembly program, but may directly lead to a _____ clock frequency.

Type: slower, faster, broken.

Check **Show answer**

COD Chapter 4 (The Processor) shows the central role that registers play in hardware construction; as we shall see in this chapter, effective use of registers is critical to program performance.

Although we could simply write instructions using numbers for registers, from 0 to 31, the MIPS convention is to use two-character names following a dollar sign to represent a register. COD Section 2.8 (Supporting Procedures in Computer Hardware) will explain the reasons behind these names. For now, we will use $s0, $s1, … for registers that correspond to variables in C and Java programs and $t0, $t1, … for temporary registers needed to compile the program into MIPS instructions.

Example 2.3.1: Compiling a C Assignment Using Registers.

It is the compiler's job to associate program variables with registers. Take, for instance, the assignment statement from our earlier example:

```
f = (g + h) - (i + j);
```

The variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. What is the compiled MIPS code?

Answer

The compiled program is very similar to the prior example, except we replace the variables with the register names mentioned above plus two temporary registers, $t0 and $t1, which correspond to the temporary variables above:

```
add $t0, $s1, $s2   # register $t0 contains g + h
add $t1, $s3, $s4   # register $t1 contains i + j
sub $s0, $t0, $t1   # f gets $t0 - $t1, which is (g + h) - (i + j)
```

PARTICIPATION ACTIVITY 2.3.3: Associations of variables with registers by the compiler.

Consider the above example.

1) With what register did the compiler associate variable g?
   - ○ $s1
   - ○ $s0

2) Could the compiler have associated g with $s7, assuming $s7 wasn't used for something else?

- Yes
- No

CHALLENGE ACTIVITY 2.3.1: Arithmetic with registers.

622166.4950548.qx3zqy7

Start

Compute: $s4 = $s5 + $s6

add $s4 , $s4 , $s4

Registers	
$s4	0
$s5	44
$s6	16

1 2 3 4 5

Check Next

Memory operands

Programming languages have simple variables that contain single data elements, as in these examples, but they also have more complex data structures—arrays and structures. These complex data structures can contain many more data elements than there are registers in a computer. How can a computer represent and access such large structures?

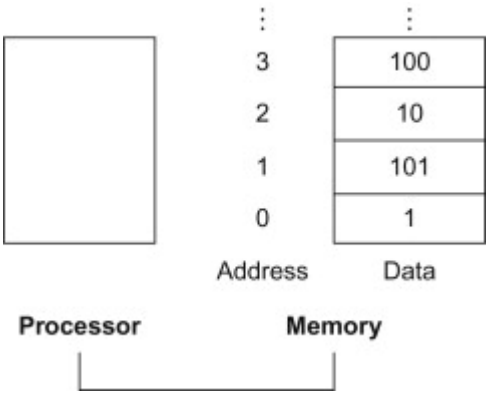
Recall the five components of a computer introduced in COD Chapter 1 (Computer Abstractions and Technology). The processor can keep only a small amount of data in registers, but computer memory contains billions of data elements. Hence, data structures (arrays and structures) are kept in memory.

As explained above, arithmetic operations occur only on registers in MIPS instructions; thus, MIPS must include instructions that transfer data between memory and registers. Such instructions are called *data transfer instructions*. To access a word in memory, the instruction must supply the memory *address*. Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0. For example, in the figure below, the address of the third data element is 2, and the value of Memory [2] is 10.

Data transfer instruction: A command that moves data between memory and registers.

Address : A value used to delineate the location of a specific data element within a memory array.

Figure 2.3.1: Memory addresses and contents of memory at those locations (COD Figure 2.2).



If these elements were words, these addresses would be incorrect, since MIPS actually uses byte addressing, with each word representing four bytes. The animation below (COD Figure 2.3) shows the memory addressing for sequential word addresses.

PARTICIPATION ACTIVITY 2.3.4: Memory data and addresses.

Consider the above example.

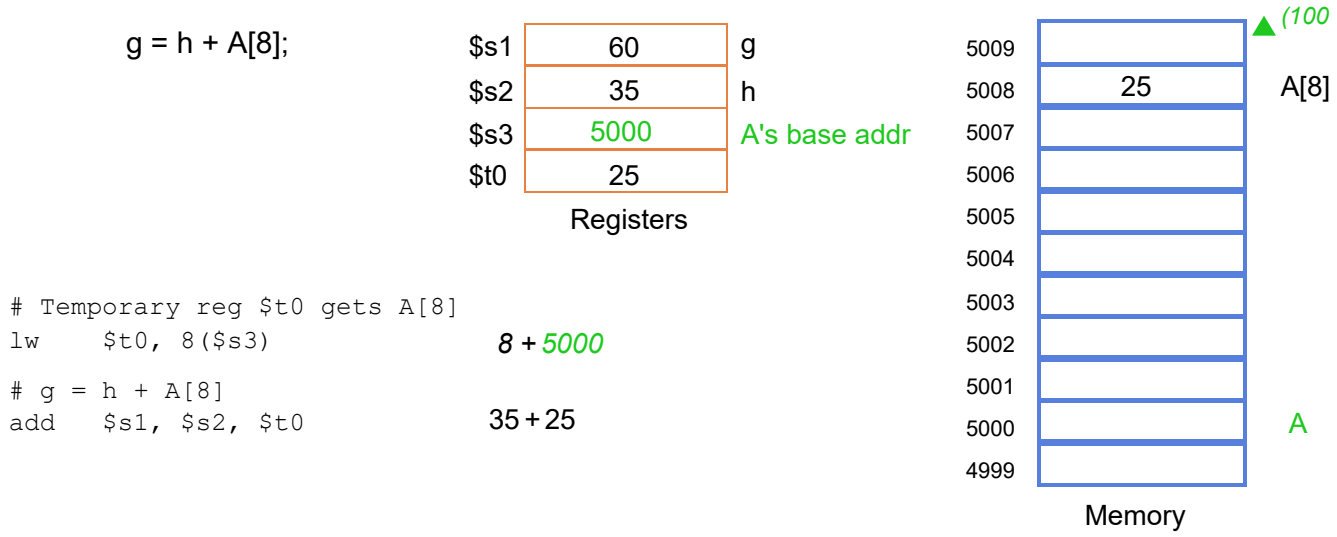
1) What is the data in the memory word with address 3?
   - ○ 100
   - ○ 10

2) What is the data in the memory word with address 1?
   - ○ 0
   - ○ 101

The data transfer instruction that copies data from memory to a register is traditionally called ***load***. The format of the load instruction is the name of the operation followed by the register to be loaded, then a constant and register used to access memory. The sum of the constant portion of the instruction and the contents of the second register forms the memory address. The actual MIPS name for this instruction is `lw`, standing for *load word*.

The animation below illustrates the load instruction. Assume that A is an array of 100 words. We'll be making a slight adjustment to the lw instruction, but we'll use the below simplified version for now. In an lw instruction, a ***base address*** is the starting address of an array in memory (5000 below), a ***base register*** is a register that holds an array's base address ($s3 below), and an ***offset*** is a constant value added to a base address to locate a particular array element (8 below).

PARTICIPATION ACTIVITY 2.3.5: Example of compiling an assignment when an operand is in memory.



Animation content:

Static figure: A C programming statement and its subsequent compilation to MIPS assembly instructions are shown. The C statement 'g = h + A[8];' is shown, indicating an assignment to variable 'g' of the sum of variable 'h' and the 8th element of array 'A'. Next to the C statement are two columns of storage: one labeled "Registers" and the other labeled "Memory." The Registers column consists of four registers: '$s1', '$s2', '$s3', and '$t0', where '$s1' is to be used for variable 'g', '$s2' for 'h', and '$s3' contains the base address of array 'A'. In the Memory column, a sequence of memory locations is shown, starting from address 4999 and increasing to 5009. A segment is highlighted to represent the 100-word array 'A', starting at address 5000. The 8th element of array

'A' is stored at address 5008.

The following MIPS instructions are shown as the compiled C statement:
Begin assembly code:
# Temporary reg $t0 gets A[8]
lw $t0, 8($s3)
# g = h + A[8]
add $s1, $s2, $t0
End assembly code.

Step 1: Goal: Compile the given C statement to assembly instructions. Some registers and memory words are shown.
The given C statement, 'g = h + A[8];', is shown next to the empty registers and memory.

Step 2: Assume the compiler uses '$s1' for g, '$s2' for h.
Assume array A is 100 words with base address 5000, and that the base address is in '$s3'.

Step 3: One of the operands, A[8], is in memory so must first be transferred to a register.
The shown lw instruction copies A[8]'s value into register '$t0'.
The illustration shows the lw assembly instruction as "lw $t0, 8($s3)" where '$t0' is the temporary register to receive the value from memory, and "8($s3)" denotes the memory address calculation with 8 as the constant offset and '$s3' holding the base address of array 'A'.

Step 4: Then, an add instruction can add that value with h ($s2), putting the result in g ($s1).
The add instruction is "add $s1, $s2, $t0", indicating that the value in '$t0' (which is A[8]) is added to the value in '$s2' (register for 'h'), with the result being stored in '$s1' (register for 'g').

Step 5: If A[8]'s value is 25, and h's value is 35, the instructions result in g getting 60.
'$s3' holds the base address 5000 for array 'A'. The memory address calculation next to the lw instruction shows "8 + 5000", which yields the effective memory address 5008. From memory address 5008 the value 25 is loaded into '$t0'. '$s2' has the initial value 35, indicating the value of 'h'; '$t0' has the value 25, representing the value at A[8]. The add instruction performs "35 + 25" and stores the result 60 into '$s1', (register for 'g').

Animation captions:

1. Goal: Compile the given C statement to assembly instructions. Some registers and memory words are shown.
2. Assume the compiler uses $s1 for g, $s2 for h. Assume array A is 100 words with base address 5000, and that the base address is in $s3.
3. One of the operands, A[8], is in memory so must first be transferred to a register. The shown lw instruction copies A[8]'s value into register $t0.
4. Then, an add instruction can add that value with h ($s2), putting the result in g ($s1).

5. If A[8]'s value is 25, and h's value is 35, the instructions result in g getting 60.

PARTICIPATION ACTIVITY 2.3.6: Load word instruction.

Assume $s3 has 5000, and words addressed 5000..5002 have the data shown:
5000: 99
5001: 77
5002: 323

1) What address will be computed by:

```
lw $t0, 2($s3)
```

Check **Show answer**

2) What value will be put in $t0 by:

```
lw $t0, 0($s3)
```

Check **Show answer**

3) What value will be put in $t1 by:

```
lw $t1, 2($s3)
```

Check **Show answer**

4) Assume $s2 has 5001. What value will be put in $t2 by:

```
lw $t2, 1($s2)
```

Check **Show answer**

©zyBooks 05/16/25 23:10 2475274
Jaheim Attri
FIUEEL4709CSpring2025

Hardware/Software Interface

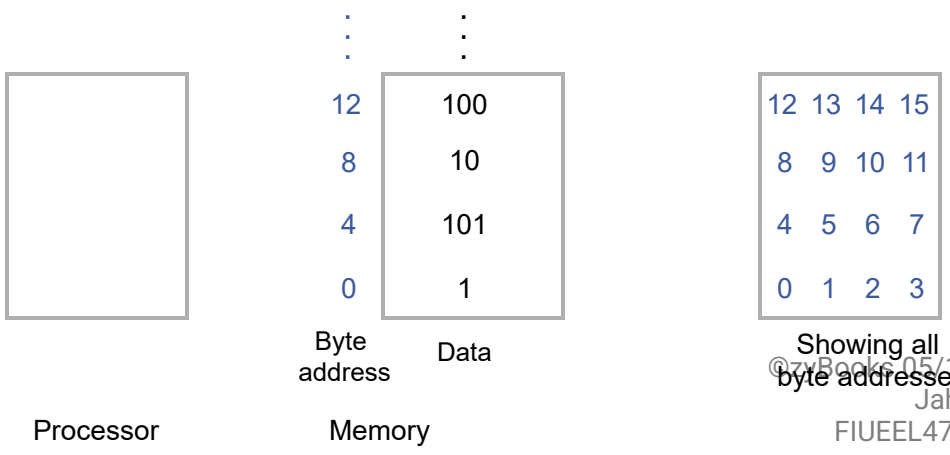
In addition to associating variables with registers, the compiler allocates data structures like arrays and structures to locations in memory. The compiler can then place the proper starting address into the data transfer instructions.

Since 8-bit *bytes* are useful in many programs, virtually all architectures today address individual bytes. Therefore, the address of a word matches the address of one of the 4 bytes within the word, and addresses of sequential words differ by 4. For example, the animation below shows the actual MIPS addresses for the words in the figure above. (Memory addresses and contents of memory at those locations); the byte address of the third word is 8.

In MIPS, words must start at addresses that are multiples of 4. This requirement is called an *alignment restriction*, and many architectures have it. (COD Chapter 4 (The Processor) suggests why alignment leads to faster data transfers.)

Alignment restriction: A requirement that data be aligned in memory on natural boundaries.

PARTICIPATION ACTIVITY 2.3.7: Actual MIPS memory addresses and contents of memory for those words (COD Figure 2.3).



Animation content:

Static figure: A simplified representation of a processor and memory. The memory is depicted as a table labeled Memory with two columns and several rows. The left column is labeled byte address with a sequence of addresses. The right column is labeled Data with data values displayed as binary numbers. Ellipses in each column of the top row indicate continuity beyond what is shown.
The table reads:

12	100
8	10
4	101
0	1
Byte address	Data

Step 1: Since MIPS addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word. The changed addresses are highlighted in blue to contrast with an earlier figure.
The byte address column is filled in with sequential numbers starting at 0 in the bottom row and increasing by 4 each row.
Step 2: For example, the first word actually consists of 4 bytes, with addresses 0, 1, 2, and 3.
A new table that shows all byte addresses appears next to the original memory table, with a row corresponding to each row of the Memory table. Each row has four columns. The new table divides the first word into its individual bytes, marked with addresses 0 through 3 in the four respective columns.
Step 3: Likewise, every other word consists of four bytes. Thus, word addresses are multiples of 4: 0, 4, 8, 12, 16, 20, etc.
The rest of the table that shows all byte addresses is filled out. Each word spans four byte addresses. The bottom row contains the numbers 0 through 3, the next row has 4 through 7, the third row has 8 through 11, and the last row contains 12 through 15.

Animation captions:

1. Since MIPS addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word. The changed addresses are highlighted in blue to contrast with an earlier figure.
2. For example, the first word actually consists of 4 bytes, with addresses 0, 1, 2, and 3.
3. Likewise, each other word consists of four bytes. Thus, word addresses are multiples of 4:

0, 4, 8, 12, 16, 20, etc.

Computers divide into those that use the address of the leftmost or "big end" byte as the word address (_big endian_) versus those that use the rightmost or "little end" byte (_little endian_). MIPS is in the *big-endian camp*. Since the order matters only if you access the identical data both as a word and as four bytes, few need to be aware of the endianess. (COD Appendix A (Assemblers, Linkers, and the SPIM Simulator) shows the two options to number bytes in a word.)

Byte addressing also affects the array index. To get the proper byte address in the code above, *the offset to be added to the base register* `$s3` *must be 4 × 8, or 32,* so that the load address will select `A[8]` and not `A[8/4]`.

PARTICIPATION ACTIVITY 2.3.8: Alignment restriction.

1) Each word consists of _____ bytes.
   - 1
   - 4
   - 8

2) Does every byte in memory have a unique address?
   - Yes
   - No

3) An array of words, A, has a base address of 2000. A[0] is thus at address 2000. What is the address of A[1]?
   - 2000
   - 2001
   - 2004

4) An array of words, A, has a base address of 2000. What is the address of A[9]?
   - 2009
   - 2036
   - 2040

5) Assuming $s3 has 5000, is the following an acceptable instruction?

```
lw $t0, 3($s3)
```

- ○ Yes
- ○ No

6) Consider the 32-bit binary number 11100000 00000000 00000000 00000001, stored in the word with address 5000. For a big-endian architecture, what value is stored in byte 5003?

- ○ 11100000
- ○ 00000000
- ○ 00000001

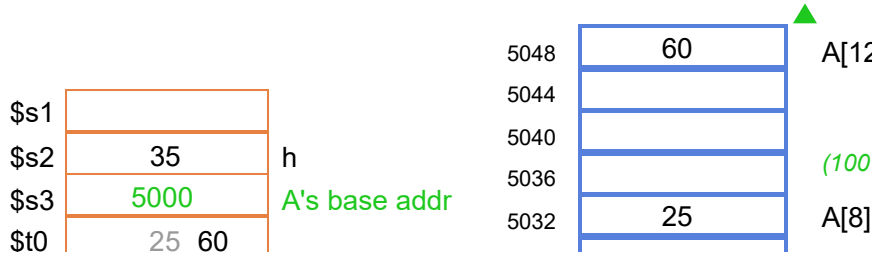
The instruction complementary to load is traditionally called store; it copies data from a register to memory. The format of a store is similar to that of a load: the name of the operation, followed by the register to be stored, then offset to select the array element, and finally the base register. Once again, the MIPS address is specified in part by a constant and in part by the contents of a register. The actual MIPS name is `sw`, standing for *store word*.

Hardware/Software Interface

As the addresses in loads and stores are binary numbers, we can see why the DRAM for main memory comes in binary sizes rather than in decimal sizes. That is, in gibibytes (2^{30}) or tebibytes (2^{40}), not in gigabytes (10^9) or terabytes (10^{12}).

PARTICIPATION ACTIVITY 2.3.9: Example of compiling using load and store.

A[12] = h + A[8];



Animation content:

Static figure: A C programming statement and its subsequent compilation to MIPS assembly instructions are shown. The C statement is A[12] = h + A[8];. Next to the C statement are two columns of storage: one labeled "Registers" and the other labeled "Memory." The Registers column consists of four registers: $s1, $s2, $s3, and $t0, where $s2 is to be used for variable h, and '$s3' contains the base address of array A. In the Memory column, a sequence of memory locations is shown, starting from address 4996 and increasing to 5048. A segment of memory is highlighted to represent the 100-word array A, starting at address 5000. The 8th element of array A is stored at address 5032, and the 12th element of array A is stored at address 5048.

The following MIPS instructions are shown as the compiled C statement:
Begin assembly code:
```
lw   $t0, 32($s3)   # Temporary reg $t0 gets A[8]
add  $t0, $s2, $t0   # Temporary reg $t0 gets h + A[8]
sw $t0, 48($s3)   # Stores h + A[8] back into A[12]
```
End assembly code.

Step 1: Goal: Compile the given C statement to assembly instructions. Similar to an earlier example, except that the result is stored in memory, and proper memory addresses are used (multiples of 4).
The given C statement, A[12] = h + A[8];, is shown next to the empty registers and memory.

Step 2: Assume the compiler uses $s2 for h, and array A is 100 words with base address 5000 in $s3.
$s2 is mapped to h. Memory locations starting at address 5000 are indicated as the storage for array A. $s3 stores the base address of array A, which is 5000.

Step 3: A[8] is in memory so the value must first be transferred to a register. The lw instruction copies A[8]'s value into register $t0. Unlike the earlier example, the proper offset of 32, 8 times 4,

is used.
lw $t0, 32($s3) is executed. Because each word in array A is four byte long, A[8] is 8 times 4 = 32 bytes away from A[0]. Thus, the proper offset is 32. The value in A[8] is stored into $t0.

Step 4: The add instruction adds that value with h ($s2), putting the result into $t0.
add $t0, $s2, $t0 is executed. Values in $t0 and $s2 are added together, and the result is stored into $t0.

Step 5: The sw instruction stores the sum into A[12], using 48, 12 times 4, as the offset and register $s3 as the base register.
sw $t0, 48($s3) is executed. A[12] is 12 times 4 = 48 bytes away from A[0]. Thus, the proper offset is 32. The value of $t0 is stored into A[12].

Step 6: If A[8]'s value is 25, and h's value is 35, the instructions result in A[12] getting 60.
25 is stored at memory address 5032. $s2 stores 35. The lw instruction stores 25 into $t0. The add instruction adds 35 and 25 and stores the result 60 into $t0. The sw instruction stores 60 into the memory at address 5048.

Animation captions:

1. Goal: Compile the given C statement to assembly instructions. Similar to an earlier example, except that the result is stored in memory, and proper memory addresses are used (multiples of 4).
2. Assume the compiler uses $s2 for h, and array A is 100 words with base address 5000 in $s3.
3. A[8] is in memory so the value must first be transferred to a register. The lw instruction copies A[8]'s value into register $t0. Unlike the earlier example, the proper offset of $\mathbf{32(8 \times 4)}$ is used.
4. The add instruction adds that value with h ($s2), putting the result in $t0.
5. The sw instruction stores the sum into A[12], using $\mathbf{48(12 \times 4)}$ as the offset and register $s3 as the base register.
6. If A[8]'s value is 25, and h's value is 35, the instructions result in A[12] getting 60.

Load word and store word are the instructions that copy words between memory and registers in the MIPS architecture. Other brands of computers use other instructions along with load and store to transfer data. An architecture with such alternatives is the Intel x86, described in COD Section 2.19 (Real Stuff: x86 Instructions).

PARTICIPATION ACTIVITY 2.3.10: Store instruction.

1) If $s3 has 900, what address

does this instruction compute?

```
sw $t0, 20($s3)
```

Check **Show answer**

2) If $s3 has 900, $t0 has 77, and memory locations 900, 904, and 908 have 10, 15, 20 respectively, what do those locations have after the following instruction?


```
sw $t0, 4($s3)
```

Type answer as: 10, 15, 20

Check **Show answer**

Hardware/Software Interface

Many programs have more variables than computers have registers. Consequently, the compiler tries to keep the most frequently used variables in registers and places the rest in memory, using loads and stores to move variables between registers and memory. The process of putting less commonly used variables (or those needed later) into memory is called _spilling registers_.

The hardware principle relating size and speed suggests that memory must be slower than registers, since there are fewer registers. This is indeed the case; data accesses are faster if data is in registers instead of memory.

Moreover, data is more useful when in a register. A MIPS arithmetic instruction can read two registers, operate on them, and write the result. A MIPS data transfer instruction only reads one operand or writes one operand, without operating on it.

Thus, registers take less time to access *and* have higher throughput than memory, making data in registers both faster to access and simpler to use. Accessing registers also uses less energy than accessing memory. To achieve the highest performance and conserve energy, an instruction set architecture must have a sufficient number of registers, and compilers must use registers efficiently.

Constant or immediate operands

Many times a program will use a constant in an operation—for example, incrementing an index to point to the next element of an array. In fact, more than half of the MIPS arithmetic instructions have a constant as an operand when running the SPEC CPU2006 benchmarks.

Using only the instructions we have seen so far, we would have to load a constant from memory to use one. (The constants would have been placed in memory when the program was loaded.) For example, to add the constant 4 to register `$s3`, we could use the code

```
lw   $t0, AddrConstant4($s1)    # $t0 = constant 4
add  $s3, $s3, $t0              # $s3 = $s3 + $t0 ($t0 == 4)
```

assuming that `$s1 + AddrConstant4` is the memory address of the constant 4.

An alternative that avoids the load instruction is to offer versions of the arithmetic instructions in which one operand is a constant. This quick add instruction with one constant operand is called *add immediate* or `addi`. To add 4 to register `$s3`, we just write

```
addi $s3, $s3, 4                # $s3 = $s3 + 4
```

Constant operands occur frequently, and by including constants inside arithmetic instructions, operations are much faster and use less energy than if constants were loaded from memory.

The constant zero has another role, which is to simplify the instruction set by offering useful variations. For example, the move operation is just an add instruction where one operand is zero. Hence, MIPS dedicates a register `$zero` to be hard-wired to the value zero. (As you might expect, it is register number 0.) Using frequency to justify the inclusions of constants is another example of the great idea of making the **common case fast**.

PARTICIPATION ACTIVITY 2.3.11: Immediate operands.

Click on the error. Assume $s0 has 5, and $s1 has 20.

1) add $t0, $s0, $s1

   Result: $t0 gets 20

2) add
$t0
,
$s0
,
44

3) add
$t0,
$s0
, 23
Result: $t0 gets
28

4) addi $t0,
$s1
,
99

Result: $t0 gets
79

CHALLENGE ACTIVITY 2.3.2: Loading and storing from memory.

622166.4950548.qx3zqy7

Start

Compute: $t1 = Memory[7064]

lw $t0 , 0 ($t0)

Registers	
$t0	7000
$t1	0

Memory	
7064	293

1 2 3 4 5

Check Next

PARTICIPATION ACTIVITY 2.3.12: Check yourself: Registers trend.

1) Given the importance of registers, what is the rate of increase in the number of registers in a chip over time?

- ○ Very fast, like Moore's Law
- ○ Very slow

Elaboration

Although the MIPS registers in this book are 32 bits wide, there is a 64-bit version of the MIPS instruction set with 32 64-bit registers. To keep them straight, they are officially called MIPS-32 and MIPS-64. In this chapter, we use a subset of MIPS-32. COD Appendix E (A Survey of RISC Architectures for Desktop, Server, and Embedded Computers) shows the differences between MIPS-32 and MIPS-64. COD Sections 2.16 (Real Stuff: ARMv7 (32-bit) Instructions) and 2.18 (Real Stuff: ARMv8 (64-bit) Instructions) show the much more dramatic difference between the 32-bit address ARMv7 and its 64-bit successor, ARMv8.

Elaboration

The MIPS offset plus base register addressing is an excellent match to structures as well as arrays, since the register can point to the beginning of the structure and the offset can select the desired element. We'll see such an example in COD Section 2.13 (A C Sort Example to Put It All Together).

Elaboration

The register in the data transfer instructions was originally invented to hold an index of an array with the offset used for the starting address of an array. Thus, the base register is also called the index register. Today's memories are much larger and the software model of data allocation is more sophisticated, so the base address of the array is normally passed in a register since it won't fit in the offset, as we shall see.

Elaboration

Since MIPS supports negative constants, there is no need for subtract immediate in MIPS.

2.4 Signed and unsigned numbers

First, let's quickly review how a computer represents numbers. Because people have 10 fingers, we are taught to think in base 10, but numbers may be represented in any base. For example, 123 base 10 = 1111011 base 2.

Numbers are kept in computer hardware as a series of high and low electronic signals, and so they are considered base 2 numbers. (Just as base 10 numbers are called decimal numbers, base 2 numbers are called binary numbers.)

A single digit of a binary number is thus the "atom" of computing, since all information is composed of binary digits or bits. This fundamental building block can be one of two values, which can be thought of as several alternatives: high or low, on or off, true or false, or 1 or 0.

Binary digit: Also called a bit. One of the two numbers in base 2 (0 or 1) that are the components of information.

Generalizing the point, in any number base, the value of *i*th digit *d* is

$$d \times \text{Base}^{i}$$

where *i* starts at 0 and increases from right to left. This representation leads to an obvious way to number the bits in the word: simply use the power of the base for that bit. We subscript decimal numbers with *ten* and binary numbers with *two*.

PARTICIPATION ACTIVITY 2.4.1: A base two number's value in base ten.

$1\,0\,1\,1_{\text{two}}$

Animation content:

Static figure: The binary number 1 0 1 1 base 2 is followed by its conversion to base 10.
Step 1: For a base two number, each digit has the weight shown.
The weights are given by the powers of 2, corresponding to the position of each digit in the binary number. The first digit 1 is multiplied by 2 cubed plus the second digit 0 is multiplied by 2 squared plus the third digit 1 is multiplied by 2 to the power of 1 plus and the fourth digit 1 is multiplied by 2 to the power of 0, all calculations in base 10.
Step 2: Converting to base ten.
The products of each binary digit weighted by the corresponding power of 2 are calculated. The calculation simplifies to equal 1 times 8 plus 0 times 4 plus 1 times 2 plus 1 times 1, which equals 8 plus 0 plus 2 plus 1 and equals 11 in base 10. The conversion from the binary number 1 0 1 1 base 2 to its equivalent in base ten equals 11.

Animation captions:

1. For a base two number, each digit has the weight shown.
2. Converting to base ten.

PARTICIPATION ACTIVITY 2.4.2: Binary number tool.

0	0	0	0	0	0	0	0
128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

0 (decimal value)

PARTICIPATION ACTIVITY 2.4.3: Base two and base ten numbers.

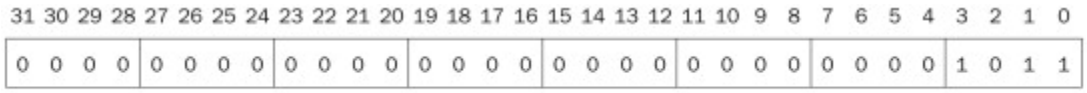
1) What is 1110_{two} in base ten?

Check **Show answer**

2) What is 3_{ten} as a *4-bit* base two number?

Check **Show answer**

We number the bits 0, 1, 2, 3, … from *right to left* in a word. The drawing below shows the numbering of bits within a MIPS word and the placement of the number 1011_{two} :



(32 bits wide)

Since words are drawn vertically as well as horizontally, leftmost and rightmost may be unclear. Hence, the phrase *least significant bit* is used to refer to the rightmost bit (bit 0 above) and *most significant bit* to the leftmost bit (bit 31).

Least significant bit: The rightmost bit in a MIPS word.

Most significant bit: The leftmost bit in a MIPS word.

PARTICIPATION ACTIVITY 2.4.4: 32-bit words.

1) Of a word's 32 bits, what is the leftmost bit numbered?

- 32
- 31

2) Given the following 32-bit number, what is the most significant bit's value?
1000 0000 0000 0000 0000 0000 0000 0000

- 1

- ○ 0

The MIPS word is 32 bits long, so it can represent 2^{32} different 32-bit patterns. It is natural to let these combinations represent the numbers from 0 to $2^{32} - 1$ (4,294,967,295$_{ten}$):

$$\begin{aligned}
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}\\
&\qquad\qquad \ldots \qquad\qquad\qquad\qquad \ldots\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4,294,967,293_{ten}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4,294,967,294_{ten}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4,294,967,295_{ten}
\end{aligned}$$

That is, 32-bit binary numbers can be represented in terms of the bit value times a power of 2 (here x^i means the i th bit of x):

$$(x^{31} \times 2^{31}) + (x^{30} \times 2^{30}) + (x^{29} \times 2^{29}) + \ \ldots\ + (x^{1} \times 2^{1}) + (x^{0} \times 2^{0})$$

For reasons we will shortly see, these positive numbers are called unsigned numbers.

PARTICIPATION ACTIVITY 2.4.5: Unsigned numbers.

1) What is the largest base ten number representable in 4 bits (assuming the "natural" representation)?
   - ○ 8
   - ○ 15
   - ○ 16

2) What is the largest base ten number representable in 8 bits (assuming the "natural" representation)?
   - ○ 255
   - ○ 256

3) What is the largest base ten number approximately representable by 32 bits (assuming the "natural" representation)?

- 4 million
- 4 billion
- 4 trillion

Hardware/Software Interface

Base 2 is not natural to human beings; we have 10 fingers and so find base 10 natural. Why didn't computers use decimal? In fact, the first commercial computer *did* offer decimal arithmetic. The problem was that the computer still used on and off signals, so a decimal digit was simply represented by several binary digits. Decimal proved so inefficient that subsequent computers reverted to all binary, converting to base 10 only for the relatively infrequent input/output events.

Keep in mind that the binary bit patterns above are simply *representatives* of numbers. Numbers really have an infinite number of digits, with almost all being 0 except for a few of the rightmost digits. We just don't normally show leading 0s.

Hardware can be designed to add, subtract, multiply, and divide these binary bit patterns. If the number that is the proper result of such operations cannot be represented by these rightmost hardware bits, ***overflow*** is said to have occurred. It's up to the programming language, the operating system, and the program to determine what to do if overflow occurs.

Overflow: when the results of an operation are larger than can be represented in a register.

Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative. The most obvious solution is to add a separate sign, which conveniently can be represented in a single bit. A ***sign and magnitude representation*** is a signed number representation where a single bit is used to represent the sign, and the remaining bits represent the magnitude.

Alas, sign and magnitude representation has several shortcomings. First, it's not obvious where to put the sign bit. To the right? To the left? Early computers tried both. Second, adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the

proper sign will be. Finally, a separate sign bit means that sign and magnitude has both a positive and a negative zero, which can lead to problems for inattentive programmers. As a result of these shortcomings, sign and magnitude representation was soon abandoned.

In the search for a more attractive alternative, the question arose as to what would be the result for unsigned numbers if we tried to subtract a large number from a small one. The answer is that it would try to borrow from a string of leading 0s, so the result would have a string of leading 1s.

Given that there was no obvious better alternative, the final solution was to pick the representation that made the hardware simple: leading 0s mean positive, and leading 1s mean negative. This convention for representing signed binary numbers is called *two's complement* representation:

$$\begin{aligned}
&\mathbf{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}}\\
&\mathbf{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}}\\
&\mathbf{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}}\\
&\qquad\qquad\mathbf{\ldots}\qquad\qquad\qquad\qquad\qquad\mathbf{\ldots}\\
&\mathbf{0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2,147,483,645_{ten}}\\
&\mathbf{0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2,147,483,646_{ten}}\\
&\mathbf{0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2,147,483,647_{ten}}\\
&\mathbf{1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2,147,483,648_{ten}}\\
&\mathbf{1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2,147,483,647_{ten}}\\
&\mathbf{1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2,147,483,646_{ten}}\\
&\qquad\qquad\mathbf{\ldots}\qquad\qquad\qquad\qquad\qquad\mathbf{\ldots}\\
&\mathbf{1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}}\\
&\mathbf{1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}}\\
&\mathbf{1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}}
\end{aligned}$$

The positive half of the numbers, from 0 to 2,147,483,647$_{ten}$ ($2^{31} - 1$), use the same representation as before. The following bit pattern (1000 … 0000$_{two}$) represents the most negative number −2,147,483,648$_{ten}$ (-2^{31}). It is followed by a declining set of negative numbers: −2,147,483,647$_{ten}$ (1000 … 0001$_{two}$) down to −1$_{ten}$ (1111 … 1111$_{two}$).

Two's complement does have one negative number, −2,147,483,648$_{ten}$, that has no corresponding positive number. Such imbalance was also a worry to the inattentive programmer, but sign and magnitude had problems for both the programmer and the hardware designer. Consequently, every

Two's complement: A signed number representation where a leading 0 indicates a positive number and a leading 1 indicates a negative number. The complement of a value is obtained by complementing each bit (0 → 1 or 1 → 0), and then adding one to the result (explained further below).

computer today uses two's complement binary representations for signed numbers.

Two's complement representation has the advantage that all negative numbers have a 1 in the most significant bit. Consequently, hardware needs to test only this bit to see if a number is positive or negative (with the number 0 considered positive). This bit is often called the sign bit. By recognizing the role of the sign bit, we can represent positive and negative 32-bit numbers in terms of the bit value times a power of 2:

$$(x^{31} \times -2^{31}) + (x^{30} \times 2^{30}) + (x^{29} \times 2^{29}) + \ldots + (x^{1} \times 2^{1}) + (x^{0} \times 2^{0})$$

The sign bit is multiplied by -2^{31}, and the rest of the bits are then multiplied by positive versions of their respective base values.

Example 2.4.1: Binary to decimal conversion (two's complement representation).

What is the decimal value of this 32-bit two's complement number?

$$1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1100_{\text{two}}$$

Answer

Substituting the number's bit values into the formula above:

$$(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^{2}) + (0 \times 2^{1}) + (0 \times 2^{0})$$
$$= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0$$
$$= -2,147,483,648_{\text{ten}} + 2,147,483,644_{\text{ten}}$$
$$= -4_{\text{ten}}$$

We'll see a shortcut to simplify conversion from negative to positive soon.

PARTICIPATION

ACTIVITY 2.4.6: Two's complement representation.

1) Sign and magnitude representation and two's complement representation are used about equally in modern computers.
   - ○ True
   - ○ False

2) In two's complement, is the following number positive or negative?
   1111 0000 0000 0000 0000 0000 0000 0000
   - ○ Positive
   - ○ Negative

3) In two's complement, is the following number positive or negative?
   0000 0000 0000 0000 0000 0000 0000 1111
   - ○ Positive
   - ○ Negative

4) Knowing that 2^{31} is 2,147,483,648, what is the base ten value of the following two's complement number?
   1000 0000 0000 0000 0000 0000 0000 0000
   - ○ -2,147,483,648
   - ○ -1

5) How is 0 represented in two's complement?
   All 0's: 0000 0000 0000 0000 0000 0000 0000 0000
   or
   All 1's: 1111 1111 1111 1111 1111 1111 1111 1111
   - ○ All 0's
   - ○ All 1's

6) In a two's complement representation,

the magnitude of the largest negative value is one greater than the magnitude of the largest positive number.

- ○ True
- ○ False

Just as an operation on unsigned numbers can overflow the capacity of hardware to represent the result, so can an operation on two's complement numbers. Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits to the left (the sign bit is incorrect): a 0 on the left of the bit pattern when the number is negative or a 1 when the number is positive.

PARTICIPATION ACTIVITY 2.4.7: Two's complement: Overflow.

Indicate if the binary operation resulted in overflow.

1)

```
  1000 1111 0000 0000 0000 0000 0000 0000
+ 1000 0000 1111 1111 1111 1111 0000 0000
-----------------------------------------
  0000 1111 1111 1111 1111 1111 0000 0000
```

- ○ Overflow
- ○ No overflow

2)

```
  0000 1111 0000 0000 0000 0000 0000 0000
+ 0111 0000 0000 0000 0000 0000 0000 0000
-----------------------------------------
  0111 1111 0000 0000 0000 0000 0000 0000
```

- ○ Overflow
- ○ No overflow

3)

```
  0111 0000 0000 0000 0000 0000 0000 0000
+ 1111 0000 0000 0000 0000 0000 0000 0000
-----------------------------------------
  0110 0000 0000 0000 0000 0000 0000 0000
```

- ○ Overflow

- No overflow

Hardware/Software Interface

Signed versus unsigned applies to loads as well as to arithmetic. The function of a signed load is to copy the sign repeatedly to fill the rest of the register, known as a ***signextension***The purpose of a signed load is to place a correct representation of the number within that register. Unsigned loads simply fill with 0s to the left of the data, since the number represented by the bit pattern is unsigned.

When loading a 32-bit word into a 32-bit register, the point is moot; signed and unsigned loads are identical. MIPS does offer two flavors of byte loads: *load byte* (`lb`) treats the byte as a signed number and thus sign-extends to fill the 24 left-most bits of the register, while *load byte unsigned* (`lbu`) works with unsigned integers. Since C programs almost always use bytes to represent characters rather than consider bytes as very short signed integers, `lbu` is used practically exclusively for byte loads.

Hardware/Software Interface

Unlike the numbers discussed above, memory addresses naturally start at 0 and continue to the largest address. Put another way, negative addresses make no sense. Thus, programs want to deal sometimes with numbers that can be positive or negative and sometimes with numbers that can be only positive. Some programming languages reflect this distinction. C, for example, names the former integers (declared as `int` in the program) and the latter *unsigned integers* (`unsigned int`). Some C style guides even recommend declaring the former as `signed int` to keep the distinction clear.

Let's examine two useful shortcuts when working with two's complement numbers. The first shortcut is a quick way to negate a two's complement binary number. Simply invert every 0 to 1 and every 1 to 0, then add one to the result. This shortcut is based on the observation that the sum of a number and its inverted representation must be 111 ... 111_{two}, which represents −1. Since $x + \bar{x} =$

−1, therefore $x + \bar{x} + 1 = 0$ or $\bar{x} + 1 = -x$. (We use the notation $\bar{x}$ to mean invert every bit in x from 0 to 1 and vice versa.)

Example 2.4.2: Negation shortcut.

Negate 2_{ten}, and then check the result by negating -2_{ten}.

Answer

$$\mathbf{2_{ten} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}}$$

Negating this number by inverting the bits and adding one,

$$\begin{array}{rr} & \mathbf{1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two}} \\ + & \mathbf{1_{two}} \\ \hline = & \mathbf{1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}} \\ = & \mathbf{-2_{ten}} \end{array}$$

Going the other direction,

$$\mathbf{1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}}$$

is first inverted and then incremented:

$$\begin{array}{rr} & \mathbf{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}} \\ + & \mathbf{1_{two}} \\ \hline = & \mathbf{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}} \\ = & \mathbf{2_{ten}} \end{array}$$

PARTICIPATION ACTIVITY 2.4.8: Negation shortcut for two's complement representation.

1) If $+3_{ten}$ is 00000011_{two}, what is -3_{ten} in an 8-bit two's complement representation?

Check **Show answer**

2) What is -9_{ten} in an 8-bit two's complement representation?

Check **Show answer**

3) Assuming a 32-bit two's-complement representation, what is 1111 1111 1111 1111 1111 1111 1111 1100_{two} in base ten?

Check **Show answer**

Our next shortcut tells us how to convert a binary number represented in *n* bits to a number represented with more than *n* bits. For example, the immediate field in the load, store, branch, add, and set on less than instructions contains a two's complement 16-bit number, representing $-32{,}768_{ten}$ (-2^{15}) to $32{,}767_{ten}$ ($2^{15} - 1$). To add the immediate field to a 32-bit register, the computer must convert that 16- bit number to its 32-bit equivalent. The shortcut is to take the most significant bit from the smaller quantity—the sign bit—and replicate it to fill the new bits of the larger quantity. The old nonsign bits are simply copied into the right portion of the new word. This shortcut is commonly called *sign extension*.

PARTICIPATION ACTIVITY 2.4.9: Example of sign extension shortcut.

0000 0000 0000 0010 $_{two}$ = 2 $_{ten}$

0000 0000 0000 0000 0000 0000 0000 0010 $_{two}$ = 2 $_{ten}$

1111 1111 1111 1110 $_{two}$ = -2 $_{ten}$

1111 1111 1111 1111 1111 1111 1111 1110 $_{two}$ = -2 $_{ten}$

Animation content:

Static Figure:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 base two equals two base ten.
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 base two equals two base ten.
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 base two equals negative two base ten.
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 base two equals negative two base ten.

Step 1: 2 in 16-bit base two is 0000 0000 0000 0010.
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 base two equals two base ten.

Step 2: Converting to 32 bits involves copying the sign bit (0) to the bits on the left.
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 base two equals two base ten. The first 0 is highlighted and 16 zeros are added to the beginning of 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 to become 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0.

Step 3: -2 in 16-bit base two is 1111 1111 1111 1110.
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 base two equals negative two base ten.

Step 4: Converting to 32 bits involves copying the sign bit (1) to the bits on the left.
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 base two equals negative two base ten. The first 1 is highlighted and 16 ones are added to the beginning of 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 to become 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 base two equals negative two base ten.

Animation captions:

1. 2 in 16-bit base two is 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0.
2. Converting to 32 bits involves copying the sign bit (0) to the bits on the left.
3. -4 in 16-bit base two is 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0.
4. Converting to 32 bits involves copying the sign bit (1) to the bits on the left.

This trick works because positive two's complement numbers really have an infinite number of 0s on the left and negative two's complement numbers have an infinite number of 1s. The binary bit pattern representing a number hides leading bits to fit the width of the hardware, sign extension simply restores some of them.

PARTICIPATION ACTIVITY 2.4.10: Negation shortcut for two's complement representation.

1) Given -5 in 8-bit two's complement:

11111011.
Extending to 16 bits yields: ____

- 00000000 11111011
- 11111111 11111011

Summary

The main point of this section is that we need to represent both positive and negative integers within a computer word, and although there are pros and cons to any option, the unanimous choice since 1965 has been two's complement.

Elaboration

For signed decimal numbers, we used "−" to represent negative because there are no limits to the size of a decimal number. Given a fixed word size, binary and hexadecimal bit strings can encode the sign; hence we do not normally use "+" or "−" with binary or hexadecimal notation.

Elaboration

Two's complement gets its name from the rule that the unsigned sum of an n-bit number and its n-bit negative is 2^n; hence, the negation or complement of a number x is $2^n - x$, or its "two's complement."

A third alternative representation to two's complement and sign and magnitude is called one's complement. The negative of a one's complement is found by inverting each bit, from 0 to 1 and from 1 to 0, or $\bar{x}$. This relation helps explain its name since the complement of x is $2^n - x - 1$. It was also an attempt to be a better solution than sign and magnitude, and several early scientific computers did use the notation. This representation is similar to two's complement except that it also has two 0s: 00 … 00_{two} is positive 0 and 11 … 11_{two} is negative 0. The most negative number, 10 … 000_{two}, represents $-2{,}147{,}483{,}647_{ten}$, and so the positives and negatives are balanced. One's complement adders did need an extra step to subtract a number, and hence two's complement dominates today.

A final notation, which we will look at when we discuss floating point in COD Chapter 3 (Arithmetic for Computers), is to represent the most negative value by 00 … 000_{two}

and the most positive value$_{two}$ by 11 … 11$_{two}$, with 0 typically having the value 10 … 00$_{two}$. This is called a biased notation, since it biases the number such that the number plus the bias has a non-negative representation.

One's complement: A notation that represents the most negative value by 10 … 000$_{two}$ and the most positive value by 01 … 11$_{two}$, leaving an equal number of negatives and positives but ending up with two zeros, one positive (00 … 00$_{two}$) and one negative (11 … 11$_{two}$). The term is also used to mean the inversion of every bit in a pattern: 0 to 1 and 1 to 0.

Biased notation: A notation that represents the most negative value by 00 … 000$_{two}$ and the most positive value by 11 … 11$_{two}$, with 0 typically having the value 10 … 00$_{two}$, thereby biasing the number such that the number plus the bias has a non-negative representation.

PARTICIPATION ACTIVITY 2.4.11: Check yourself: Two's complement.

1) Decimal value of the following 16-bit two's complement number:
1111 1111 1111 1000$_{two}$

- −4$_{ten}$
- −8$_{ten}$
- −16$_{ten}$
- 65,528$_{ten}$

2) Decimal value of the following 16-bit unsigned number:
1111 1111 1111 1000$_{two}$

- −4$_{ten}$
- −8$_{ten}$
- −16$_{ten}$
- 65,528$_{ten}$

2.5 Representing instructions in the computer

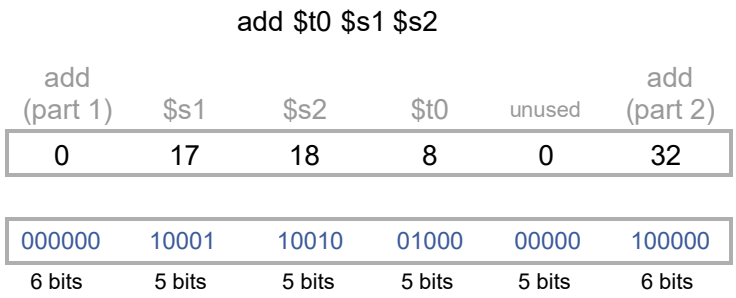
We are now ready to explain the difference between the way humans instruct computers and the way computers see instructions.

Instructions are kept in the computer as a series of high and low electronic signals and may be represented as numbers. In fact, each piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the instruction.

Since registers are referred to in instructions, there must be a convention to map register names into numbers. In MIPS assembly language, registers `$s0` to `$s7` map onto registers 16 to 23, and registers `$t0` to `$t7` map onto registers 8 to 15. Hence, `$s0` means register 16, `$s1` means register 17, `$s2` means register 18, ..., `$t0` means register 8, `$t1` means register 9, and so on. We'll describe the convention for the rest of the 32 registers in the following sections.

The below animation shows the conversion of an assembly instruction into a machine instruction consisting of 0's and 1's. A machine instruction is composed of ***fields***, each field having several bits and representing some part of the instruction.

PARTICIPATION ACTIVITY 2.5.1: Example of translating a MIPS assembly instruction into a machine instruction.



Animation content:

Static figure: An assembly instruction, add $t0 $s1 $s2, and the corresponding machine instruction are displayed. The machine instruction is divided into six fields. The label and value of each field are as follows:
add (part 1), 0 in base ten or 000000 in base two
$s1, 17 in base ten or 10001 in base two
$s2, 18 in base ten or 10010 in base two

$t0, 8 in base ten or 01000 in base two
unused, 0 in base ten or 00000 in base two
add (part 2), 32 in base ten or 100000 in base two

Step 1: The assembly instruction will be converted to a machine instruction. The machine instruction has 6 fields.

Step 2: The first and last fields (containing 0 and 32) in combination tell the MIPS computer that this instruction performs addition.
The operator of the assembly instruction, add, is highlighted. The field labeled add (part 1) now contains 0, and the field labeled add (part 2) now contains 32.

Step 3: The second field gives the number of the register that is the first source operand (17 means $s1), and the third field gives the other source operand (18 means $s2).
Two operands of the assembly instruction, $s1 and $s2, are highlighted. The field labeled $s1 now contains 17, and the field labeled $s2 now contains 18.

Step 4: The fourth field contains the number of the register that is to receive the sum (8 means $t0). The fifth field is unused in this instruction, so is set to 0.
An operand of the assembly instruction, $t0, is highlighted. The field labeled $t0 now contains 8, and the field labeled unused now contains 0.

Step 5: This instruction can be represented as fields of binary numbers as opposed to decimal. The same six fields of the machine instruction are now represented in binary: 000000, 10001, 10010, 01000, 00000, 100000. Each binary value is annotated with the field's bit length: 6 bits, 5 bits, 5 bits, 5 bits, 5 bits, 6 bits.

Animation captions:

1. The assembly instruction will be converted to a machine instruction. The machine instruction has 6 fields.
2. The first and last fields (containing 0 and 32) in combination tell the MIPS computer that this instruction performs addition.
3. The second field gives the number of the register that is the first source operand (17 means $s1), and the third field gives the other source operand (18 means $s2).
4. The fourth field contains the number of the register that is to receive the sum (8 means $t0). The fifth field is unused in this instruction, so is set to 0.
5. This instruction can be represented as fields of binary numbers as opposed to decimal.

This layout of the instruction is called the *instruction format*. As you can see from counting the number of bits, this MIPS instruction takes exactly 32 bits—the same size as a data word. In keeping with our design principle that simplicity favors regularity, all MIPS instructions are 32 bits

long.

Instruction format: A form of representation of an instruction composed of fields of binary numbers.

To distinguish it from assembly language, we call the numeric version of instructions *machine language* and a sequence of such instructions machine code.

Machine language: Binary representation used for communication within a computer system.

PARTICIPATION ACTIVITY 2.5.2: MIPS instruction fields.

Match the assembly instructions item with the instruction field.

Assembly instruction: `add $t1, $s4, $s3`

MIPS instruction fields:

Field 1	Field 2	Field 3	Field 4	Field 5	Field 6

How to use this tool

$s4 **$t1** **$s3** **unused** **add (part 2)** **add (part 1)**

Field 1

Field 2

Field 3

Field 4

Field 5

Field 6

Reset

PARTICIPATION ACTIVITY 2.5.3: MIPS machine instruction fields and values.

1) How many bits are used to indicate that an instruction performs add?

Check **Show answer**

2) An add instruction has fields for three registers. How many bits are used for any one of those registers?

Check **Show answer**

3) How many total bits is a machine instruction?

Check **Show answer**

4) Given: add $t5, $s0, $s1
What decimal value is stored in field 2?

Check **Show answer**

5) Given: add $t5, $s0, $s1
What 5-bit value is stored in field 3?

Check **Show answer**

It would appear that you would now be reading and writing long, tedious strings of binary numbers. We avoid that tedium by using a higher base than binary that converts easily into binary. Since almost all computer data sizes are multiples of 4, *hexadecimal* (base 16) numbers are popular. Since base 16 is a power of 2, we can trivially convert by replacing each group of four binary digits by a single hexadecimal digit, and vice versa. The figure below converts between hexadecimal and binary.

Hexadecimal: Numbers in base 16.

Figure 2.5.1: The hexadecimal-binary conversion table (COD Figure 2.4).

Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of 4, go from right to left.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

PARTICIPATION ACTIVITY 2.5.4: Hexadecimal.

1) What is 0011 as a hexadecimal digit?

Check **Show answer**

2) What is 1011 as a hexadecimal digit?

Check **Show answer**

3) What is 11110000 in 2-digit hexadecimal? Write answer as:

a1

Check **Show answer**

4) What is 2f in 8-bit binary?

Check **Show answer**

PARTICIPATION ACTIVITY 2.5.5: Binary and hex tool.

Start

dec	bin	hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

©zyBooks 05/16/25 23:10 2475274
Jaheim Attri
FIUEEL4709CSpring2025

1 2 3 4

Check Next

Because we frequently deal with different number bases, to avoid confusion we will subscript decimal numbers with *ten*, binary numbers with *two*, and hexadecimal numbers with *hex*. (If there is no subscript, the default is base 10.) By the way, C and Java use the notation 0x*nnnn* for hexadecimal numbers.

Example 2.5.1: Binary to hexadecimal and back.

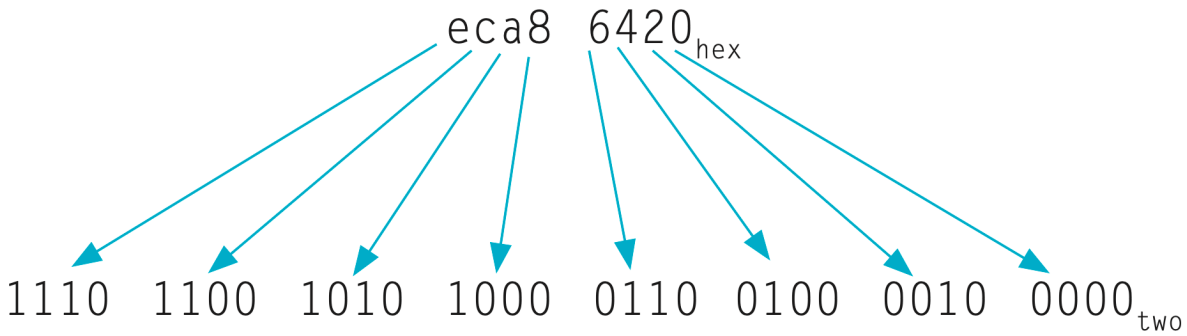
Convert the following hexadecimal and binary numbers into the other base:

$$\mathbf{eca8\ \ 6420_{hex}}$$

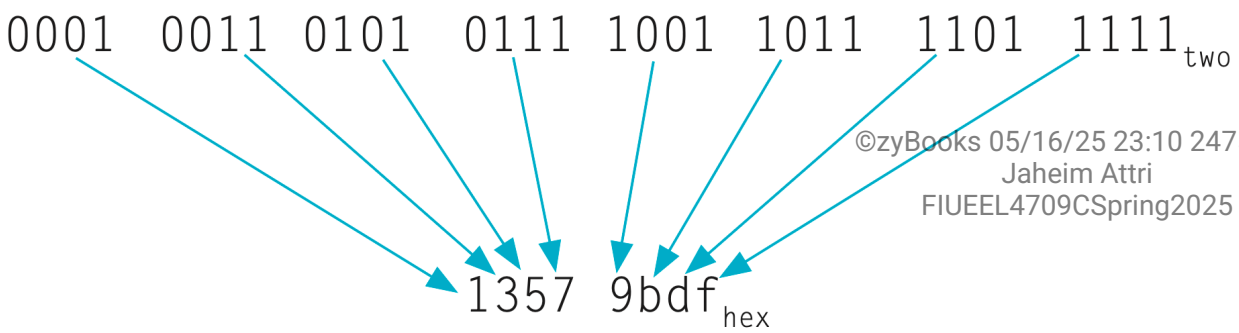
$$\mathbf{0001\ \ 0011\ \ 0101\ \ 0111\ \ 1001\ \ 1011\ \ 1101\ \ 1111_{two}}$$

Answer

Using the previous figure, the answer is just a table lookup one way:



And then the other direction:



MIPS fields

MIPS fields are given names to make them easier to discuss:

Here is the meaning of each name of the fields in MIPS instructions:

- *op* : Basic operation of the instruction, traditionally called the *opcode*.
- *rs*: The first register source operand.
- *rt*: The second register source operand.
- *rd*: The register destination operand. It gets the result of the operation.
- *shamt*: Shift amount. (COD Section 2.6 (Logical operations) explains shift instructions and this term; it will not be used until then, and hence the field contains zero in this section.)
- *funct*: Function. This field, often called the function code, selects the specific variant of the operation in the op field.

Opcode: The field that denotes the operation and format of an instruction.

A problem occurs when an instruction needs longer fields than those shown above. For example, the load word instruction must specify two registers and a constant. If the address were to use one of the 5-bit fields in the format above, the constant within the load word instruction would be limited to only 2^5 or 32. This constant is used to select elements from arrays or data structures, and it often needs to be much larger than 32. This 5-bit field is too small to be useful.

Hence, we have a conflict between the desire to keep all instructions the same length and the desire to have a single instruction format. This leads us to the final hardware design principle:

Design Principle 3: Good design demands good compromises.

The compromise chosen by the MIPS designers is to keep all instructions the same length, thereby requiring different kinds of instruction formats for different kinds of instructions. For example, the format above is called *R-type* (for register) or *R-format*. A second type of instruction format is called *I-type* (for immediate) or *I-format* and is used by the immediate and data transfer instructions. The fields of I-format are

The 16-bit address means a load word instruction can load any word within a region of $\pm 2^{15}$ or 32,768 bytes ($\pm 2^{13}$ or 8192 words) of the address in the base register rs. Similarly, add immediate is limited to constants no larger than $\pm 2^{15}$. We see that more than 32 registers would be difficult in this format, as the rs and rt fields would each need another bit, making it harder to fit everything in one word.

Let's look at the load word instruction:

```
lw   $t0, 32($s3)   # Temporary reg $t0 gets A[8]
```

Here, 19 (for `$s3`) is placed in the rs field, 8 (for `$t0`) is placed in the rt field, and 32 is placed in the address field. Note that the meaning of the rt field has changed for this instruction: in a load word instruction, the rt field specifies the *destination register*, which receives the result of the load. A ***destination register*** is a register that receives the result of an operation.

Although multiple formats complicate the hardware, we can reduce the complexity by keeping the formats similar. For example, the first three fields of the R-type, D-type, and I-type formats are the same size and have the same names; the length of the fourth field in I-type is equal to the sum of the lengths of the last three fields of R-type.

In case you were wondering, the formats are distinguished by the values in the first field: each format is assigned a distinct set of values in the first field (op) so that the hardware knows whether to treat the last half of the instruction as three fields (R-type) or as a single field (I-type). The following figure shows the numbers used in each field for the MIPS instructions covered so far.

PARTICIPATION ACTIVITY 2.5.6: MIPS R-type and I-type instruction encoding (COD Figure 2.5).

Instruction formats

Instruction	Format	op	rs	rt	rd	shamt	funct	*(R-type)*
					Constant or address			*(I-type)*
`add`	R	0	reg	reg	reg	0	32	
`sub` (subtract)	R	0	reg	reg	reg	0	34	
`add immediate`	I	8	reg	reg	constant			
`lw` (load word)	I	35	reg	reg	address			
`sw` (store word)	I	43	reg	reg	address			

Sample instructions

	op	rs	rt	rd	shamt	funct
`add $t0, $s2, $s3`	0	18	19	8	0	32
`sub $s1, $s2, $s3`	0	18	19	17	0	34
`addi $s1, $s2, 20`	8	18	17	20		
`lw $t0, 1200($t1)`	35	9	8	1200		
`sw $t0, 1200($t1)`	43	9	8	1200		

Register locations

$t0	8
$t1	9
$s1	17
$s2	18
$s3	19

Animation content:

Static figure: Two tables are displayed. The first table defines instruction formats for R-type and I-type instructions. The first column identifies the instruction, the second column identifies the format, the third column identifies the field op, the fourth column identifies the field rs, and the fifth column identifies the field rt. For an R-type instruction, the sixth column identifies the field rd, the seventh column identifies the field shamt, and the eighth column identifies the field funct. For an I-type instruction, the sixth column identifies the field that contains either a constant value or an address. The second table provides some sample instructions showing how each field is used.

Begin Instruction formats table:
Row 1: The instruction add has a format of R, op equals 0, rs equals reg (short for register), rt equals reg, rd equals reg, shamt equals 0, and funct equals 32.
Row 2: The instruction sub (subtract) has a format of R, op equals 0, rs equals reg, rt equals reg, rd equals reg, shamt equals 0, and funct equals 34.
Row 3: The instruction add immediate has a format of I, op equals 8, rs equals reg, rt equals reg, and a constant.
Row 4: The instruction lw (load word) has a format of I, op equals 35, rs equals reg, rt equals reg, and an address.
Row 5: The instruction sw (store word) has a format of I, op equals 43, rs equals reg, rt equals reg, and an address.
End table.

Begin Sample instructions table:
Row 1: The instruction, add $t0, $s2, $s3, has an op equal to 0, rs equals 18, rt equals 19, rd equals 8, shamt equals 0, and funct equals 32.
Row 2: The instruction, sub $s1, $s2, $s3, has an op equal to 0, rs equals 18, rt equals 19, rd equals 17, shamt equals 0, and funct equals 34.
Row 3: The instruction, addi $s1, $s2, 20, has an op equal to 8, rs equals 18, rt equals 17, and a constant value equal to 20.
Row 4: The instruction, lw $t0, 1200($t1), has an op equal to 35, rs equals 9, rt equals 8, and an address equal to 1200.
Row 5: The instruction, sw $t0, 1200($t1), has an op equal to 43, rs equals 9, rt equals 8, and an address equal to 1200.
End table.

Following the Sample instructions table, a list labeled Register locations is displayed.
Begin Register locations:
$t0 is 8
$t1 is 9
$s1 is 17

$s2 is 18
$s3 is 19
End Register locations.

Step 1: "add is an 'R-type' instruction (R for register) with 6 fields: op, rs, rt, rd, shamt, and funct. op of 0 and funct of 32 means add."
The first row of the table that defines the instruction formats appears, that defines the instruction format for the add instruction.

Step 2: "sub is similar, except funct is 34 instead of 32 for add."
The second row of the table that defines the instruction formats appears, that defines the instruction format for the subtract instruction.

Step 3: "addi is an 'I-type' (I for immediate) rather than 'R-type' instruction. The last 16 bits form one field, having the immediate instruction's constant value."
The third row of the table that defines the instruction formats appears, that defines the instruction format for the add immediate instruction.

Step 4: "lw and sw are also I-type."
The last two rows of the table that defines the instruction formats appear, that defines the instruction format for the load word and store word instructions.

Step 5: "Some sample R-type and I-type instructions, showing how each field is used."
The table that provides the sample instructions appears. Next to it is the list that defines register locations.

Animation captions:

1. add is an "R-type" instruction (R for register) with 6 fields: op, rs, rt, rd, shamt, and funct. op of 0 and funct of 32 means add.
2. sub is similar, except funct is 34 instead of 32 for add.
3. addi is an "I-type" (I for immediate) rather than "R-type" instruction. The last 16 bits form one field, having the immediate instruction's constant value.
4. lw and sw are also I-type.
5. Some sample R-type and I-type instructions, showing how each field is used.

PARTICIPATION ACTIVITY 2.5.7: R-type and I-type instructions.

1) What type of instruction is add?

- ○ R-type

- I-type

2) What type of instruction is addi (add immediate)?
   - R-type
   - I-type

3) What type of instruction is sw (store word)?
   - R-type
   - I-type

4) For both add and addi instructions, field 3 (rt) represents a register.
   - True
   - False

5) Because I-type instructions involve a constant, an I-type instruction uses more bits.
   - True
   - False

Example 2.5.2: Translating MIPS assembly language into machine language.

We can now take an example all the way from what the programmer writes to what the computer executes. If `$t1` has the base of the array `A` and `$s2` corresponds to `h`, the assignment statement

```
A[300] = h + A[300];
```

is compiled into


```
lw  $t0, 1200($t1) # Temporary reg $t0 gets A[300]
add $t0, $s2, $t0  # Temporary reg $t0 gets h + A[300]
sw  $t0, 1200($t1) # Stores h + A[300] back into A[300]
```

What is the MIPS machine language code for these three instructions?

Answer

For convenience, let's first represent the machine language instructions using decimal numbers. From the previous figure, we can determine the three machine language instructions:

op	rs	rt	rd	shamt	funct	
			Constant or address			(I-type)
35	9	8	1200			
0	18	8	8	0	32	(R-type)
43	9	8	1200			

The `lw` instruction is identified by 35 (see the animation above) in the first field (op). The base register 9 (`$t1`) is specified in the second field (rs), and the destination register 8 (`$t0`) is specified in the third field (rt). The offset to select `A[300]` (1200 = 300 × 4) is found in the final field (address).

The `add` instruction that follows is specified with 0 in the first field (op) and 32 in the last field (funct). The three register operands (18, 8, and 8) are found in the second, third, and fourth fields and correspond to `$s2`, `$t0`, and `$t0`.

The `sw` instruction is identified with 43 in the first field. The rest of this final instruction is identical to the `lw` instruction.

Since 1200_{ten} = 0000 0100 1011 0000_{two}, the binary equivalent to the decimal form is:

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Note the similarity of the binary representations of the first and last instructions. The only difference is in the third bit from the left, which is highlighted here.

PARTICIPATION ACTIVITY 2.5.8: MIPS machine language example.

Given these MIPS machine instructions, indicate what each value represents.

op	rs	rt	rd	shamt	funct	
			Constant or address			(I-type)
35	9	8	1200			
0	18	8	8	0	32	(R-type)
43	9	8	1200			

How to use this tool

18 (in row 2) **1200 (in row 1)** **9 (in row 1)** **32 (in row 2)** **35 (in row 1)**

Opcode for a load word instruction.

$t1, containing a base address.

An offset

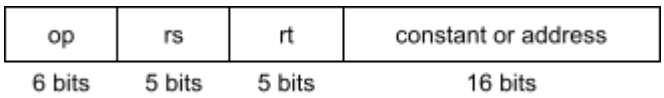
$s2

Combined with opcode, indicates an add instruction.

Reset

PARTICIPATION ACTIVITY 2.5.9: Translating MIPS instructions to machine language.

Translate `addi $t7, $t4, 5` to the corresponding MIPS machine language code. The fields of an I-format instruction are provided below:



1) What 6-bit value is stored in the op field?

Check **Show answer**

2) What 5-bit value is stored in rs field?

Check **Show answer**

3) What 5-bit value is stored in rt

field?

Check **Show answer**

4) What 16-bit value is stored in the constant or address field?

Check **Show answer**

Hardware/Software Interface

The desire to keep all instructions the same size is in conflict with the desire to have as many registers as possible. Any increase in the number of registers uses up at least one more bit in every register field of the instruction format. Given these constraints and the design principle that smaller is faster, most instruction sets today have 16 or 32 general purpose registers.

The following figure summarizes the portions of MIPS machine language described in this section. As we shall see in COD Chapter 4 (The Processor), the similarity of the binary representations of related instructions simplifies hardware design. These similarities are another example of regularity in the MIPS architecture.

Figure 2.5.2: MIPS architecture revealed thus far (COD Figure 2.6).

The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an *op* field, giving the base operation; an *rs* field, giving one of the sources; and the *rt* field, which specifies the other source operand, except for load word and add immediate, where it specifies the destination register. R-format divides the last 16 bits into an *rd* field, specifying the destination register; the *shamt* field, which COD Section 2.6 (Logical operations) explains; and the *funct* field, which specifies the specific operation of R-format instructions. I-format combines the last 16 bits into a single *address* field.

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

PARTICIPATION ACTIVITY 2.5.10: MIPS machine language.

1) Opcode 35 indicates a _____ instruction.
   - load word
   - store word

2) Opcode _____ indicates a store word instruction.
   - 41
   - 43

3) Opcode 0 indicates an _____ instruction.
   - add
   - (insufficient information)

4) Opcode 0 and a funct field of 34 indicates a(n) _____ instruction.
   - add immediate
   - subtract

5) addi's opcode is _____.
   - 8
   - 5

The Big Picture

Today's computers are built on two key principles:

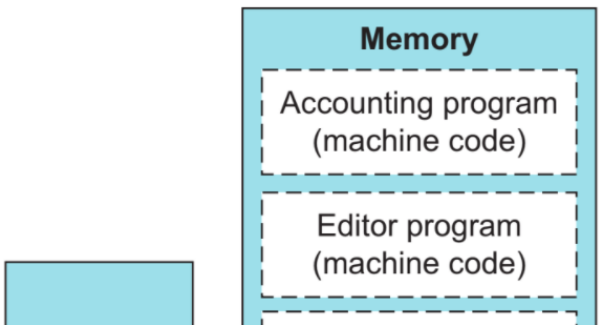
1. Instructions are represented as numbers.
2. Programs are stored in memory to be read or written, just like data.

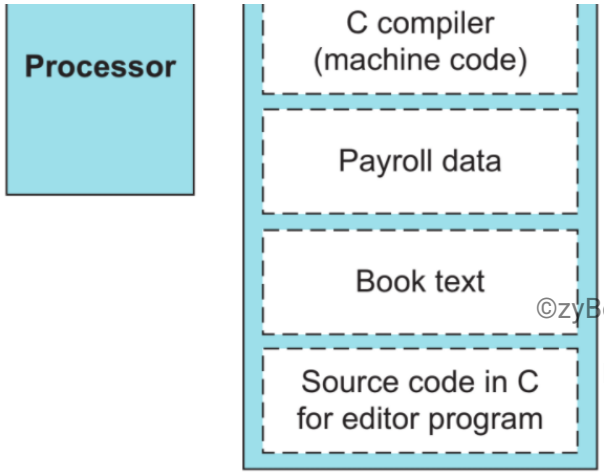
These principles lead to the *stored-program concept*; its invention let the computing genie out of its bottle. The following figure shows the power of the concept; specifically, memory can contain the source code for an editor program, the corresponding compiled machine code, the text that the compiled program is using, and even the compiler that generated the machine code.

One consequence of instructions as numbers is that programs are often shipped as files of binary numbers. The commercial implication is that computers can inherit ready-made software provided they are compatible with an existing instruction set. Such "binary compatibility" often leads industry to align around a small number of instruction set architectures.

Figure 2.5.3: The stored-program concept (COD Figure 2.7).

Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given location in memory. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed for data can also be used for programs, and programs like compilers, for instance, can translate code written in a notation far more convenient for humans into code that the computer can understand.





PARTICIPATION ACTIVITY 2.5.11: Stored-program concept.

1) The stored-program concept means:

- Programs are stored in memory along with data.
- A computer supports a store instruction.
- Programs are stored on external disks.

PARTICIPATION ACTIVITY 2.5.12: Check yourself: MIPS instructions.

1) Which MIPS instruction does the following represent?

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

- `sub $t0, $t1, $t2`
- `add $t2, $t0, $t1`
- `sub $t2, $t1, $t0`
- `sub $t2, $t0, $t1`

2) If a person is age 40_{ten}, what is their age in hexadecimal?

- 101000

○ 28

2.6 Logical operations

> "Contrariwise," continued Tweedledee, "if it was so, it might be, and if it were so, it would be; but as it isn't, it ain't. That's logic."
> *Lewis Carroll, Alice's Adventures in Wonderland, 1865*

Although the first computers operated on full words, it soon became clear that it was useful to operate on fields of bits within a word or even on individual bits. Examining characters within a word, each of which is stored as 8 bits, is one example of such an operation (see COD Section 2.9 (Communicating with people)). It follows that operations were added to programming languages and instruction set architectures to simplify, among other things, the packing and unpacking of bits into words. These instructions are called logical operations. The figure below shows logical operations in C, Java, and MIPS.

Figure 2.6.1: C and Java logical operators and their corresponding MIPS instructions (COD Figure 2.8).

MIPS implements NOT using a NOR with one operand being zero.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

The first class of such operations is called *shifts*. A _shift_ moves all the bits in a word to the left or right, filling the emptied bits with 0s.

PARTICIPATION ACTIVITY 2.6.1: Example of shifting left by 4.

```
sll $t2, $s0, 4    # reg $t2 = reg $s0 << 4 bits
```

shift left by 4

Animation content:

Static figure: Two rows of binary numbers. The first row represents the original value and shows a sequence of binary digits 0000 0000 0000 0000 0000 0000 0000 1001. This binary number corresponds to the decimal value 9. The second row represents the new value, equal to the original value shifted left by 4, and shows a sequence of binary digits 0000 0000 0000 0000 0000 0000 1001 0000. This binary number corresponds to the decimal value 144. Also shown is the instruction: sll $t2, $s0, 4 and a comment (indicated with a #) reg $t2 = reg $s0 << 4 bits

Step 1: A shift moves bits left or right. If a register originally stores the bits shown....
The original value row is shown as a sequence of binary digits 0000 0000 0000 0000 0000 0000 0000 1001, which corresponds to the decimal value 9.

Step 2: ....then shifting left by 4 moves the bits over by 4 places. The leftmost 4 bits are discarded, and 4 0's (green) are shifted into the rightmost 4 bits.
The new value row is shown but does not contain a value. A copy of the original value row is moved to the new value row, but shifted to the left by 4 as compared to the original value row. The leftmost 0000 of the original value copy are shown initially, then greyed out. Additional bits 0000 (shown in green) move into the rightmost positions in the new value row. This binary number of the new value row corresponds to the decimal value 144.

Step 3: This instruction would shift $s0's value and store the result in $t2.
The following instruction is shown: sll $t2, $s0, 4 # reg $t2 = reg $s0 << 4 bits

Animation captions:

1. A shift moves bits left or right. If a register originally stores the bits shown....
2. ....then shifting left by 4 moves the bits over by 4 places. The leftmost 4 bits are discarded, and 4 0's (green) are shifted into the rightmost 4 bits.
3. This instruction would shift $s0's value and store the result in $t2.

The dual of a shift left is a shift right. The actual name of the two MIPS shift instructions are called *shift left logical* (`sll`) and *shift right logical* (`srl`). The following instruction performs the operation above, assuming that the original value was in register `$s0` and the result should go in register `$t2`:

```
sll $t2, $s0, 4   # reg $t2 = reg $s0 << 4 bits
```


We delayed explaining the *shamt* field in the R-format. Used in shift instructions, it stands for *shift amount*. Hence, the machine language version of the instruction above is

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

The encoding of `sll` is 0 in both the op and funct fields, rd contains 10 (register `$t2`), rt contains 16 (register `$s0`), and shamt contains 4. The rs field is unused and thus is set to 0.

Shift left logical provides a bonus benefit. Shifting left by *i* bits gives the same result as multiplying by 2^i, just as shifting a decimal number by *i* digits is equivalent to multiplying by 10^i. For example, the above `sll` shifts by 4, which gives the same result as multiplying by 2^4 or 16. The first bit pattern above represents 9, and 9 × 16 = 144, the value of the second bit pattern.

PARTICIPATION ACTIVITY 2.6.2: Shifting.

Given the following initial value, determine the resulting value for the given operation.
0011 0000 0000 0000 0000 0000 1111 0001

How to use this tool

Multiply by 2 **Shift left by 2** **Shift left by 8** **Shift right by 8**

0000 0000 0000 0000 1111 0001 0000 0000

0000 0000 0011 0000 0000 0000 0000 0000

1100 0000 0000 0000 0000 0011 1100 0100

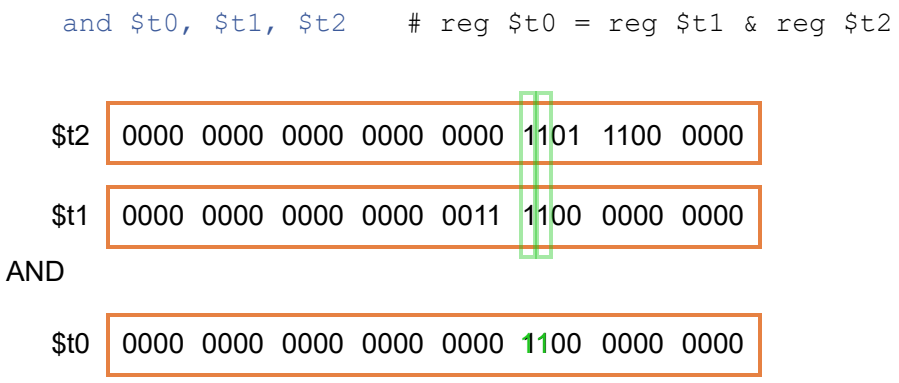
0110 0000 0000 0000 0000 0001 1110 0010

Reset

Another useful operation that isolates fields is *AND*. (We capitalize the word to avoid confusion between the operation and the English conjunction.) AND is a bit-by-bit operation that leaves a 1 in the result only if both bits of the operands are 1.

AND: A logical bit- by-bit operation with two operands that calculates a 1 only if there is a 1 in both operands.

PARTICIPATION ACTIVITY 2.6.3: Example of an AND operation.



Animation content:

Static figure:
Begin assembly code:
and $t0, $t1, $t2 # reg $t0 = reg $t1 & reg $t2
End assembly code.

Register $t2 stores 0000 0000 0000 0000 0000 1101 1100 0000
Register $t1 stores 0000 0000 0000 0000 0011 1100 0000 0000
Register $t0 stores 0000 0000 0000 0000 0000 1100 0000 0000
The binary bits of registers $t2, $t1, and $t0 are vertically aligned with each other in an AND operation.
The 10th and 11th bits from the rightmost bit of all three registers are highlighted.

Step 1: An AND yields a 1 in the result only if both bits of the operands are 1.
Since the 10th bit from the rightmost bit of both $t2 and $t1 are 1, the corresponding bit of $t0

becomes 1.
Since the 11th bit from the rightmost bit of both $t2 and $t1 are 1, the corresponding bit of $t0 becomes 1.

Step 2: If either operand bit is a 0, AND yields a 0.
Since the 8th bit from the rightmost bit of $t2 is 1, while the corresponding bit of $t1 is 0, the corresponding bit of $t0 becomes 0.

Step 3: No other pair of operand bits are both 1's, so AND yields 0's for all other bits.
The rest of the bits of $t0 become 0.

Animation captions:

1. An AND yields a 1 in the result only if both bits of the operands are 1.
2. If either operand bit is a 0, AND yields a 0.
3. No other pair of operand bits are both 1's, so AND yields 0's for all other bits.

As you can see, AND can apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern. Such a bit pattern in conjunction with AND is traditionally called a ***mask***, since the mask "conceals" some bits.

To place a value into one of these seas of 0s, there is the dual to AND, called *OR*. It is a bit-by-bit operation that places a 1 in the result if either operand bit is a 1. To elaborate, if the registers `$t1` and `$t2` are unchanged from the preceding example, the result of the MIPS instruction

```
or $t0, $t1, $t2   # reg $t0 = reg $t1 | reg $t2
```

is this value in register `$t0`:

$$0000\ 0000\ 0000\ 0000\ 0011\ 1101\ 1100\ 0000_{two}$$

OR: A logical bit-by-bit operation with two operands that calculates a 1 if there is a 1 in either operand.

The final logical operation is a contrarian. *NOT* takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa. Using our prior notation, it calculates $\sim$.

NOT: A logical bit-by-bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

In keeping with the three-operand format, the designers of MIPS decided to include the instruction *NOR* (NOT OR) instead of NOT. If one operand is zero, then it is equivalent to NOT: A NOR 0 = NOT

(A OR 0) = NOT (A).

NOR: A logical bit-by-bit operation with two operands that calculates the NOT of the OR of the two operands. That is, it calculates a 1 only if there is a 0 in both operands.

If the register `$t1` is unchanged from the preceding example and register `$t3` has the value 0, the result of the MIPS instruction


```
nor $t0, $t1, $t3   # reg $t0 = ~ (reg $t1 | reg $t3)
```

is this value in register `$t0`:

$1111\ 1111\ 1111\ 1111\ 1100\ 0011\ 1111\ 1111_{two}$

PARTICIPATION ACTIVITY 2.6.4: AND, OR, and NOT.

Given the following initial values, determine the resulting value for the given operation.
x: 0011 0000 0000 0000 0000 0000 1111 0001
y: 0000 0000 0000 0000 0000 0000 1111 1111

How to use this tool

x AND y **NOT x** **x OR y** **x NOR y**

0000 0000 0000 0000 0000 0000 1111 0001

0011 0000 0000 0000 0000 0000 1111 1111

1100 1111 1111 1111 1111 1111 0000 1110

1100 1111 1111 1111 1111 1111 0000 0000

Reset

The figure above shows the relationship between the C and Java operators and the MIPS instructions. Constants are useful in AND and OR logical operations as well as in arithmetic

operations, so MIPS also provides the instructions *and immediate* (`andi`) and *or immediate* (`ori`). Constants are rare for NOR, since its main use is to invert the bits of a single operand; thus, the MIPS instruction set architecture has no immediate version of NOR.

Elaboration

The full MIPS instruction set also includes exclusive or (XOR), which sets the bit to 1 when two corresponding bits differ, and to 0 when they are the same. C allows bit fields or fields to be defined within words, both allowing objects to be packed within a word and to match an externally enforced interface such as an I/O device. All fields must fit within a single word. Fields are unsigned integers that can be as short as 1 bit. C compilers insert and extract fields using logical instructions in MIPS: `and`, `or`, `sll`, *and* `srl`.

Elaboration

Logical AND immediate and logical OR immediate put 0s into the upper 16 bits to form a 32-bit constant, unlike add immediate, which does sign extension.

PARTICIPATION ACTIVITY 2.6.5: Logical operations.

1) AND can be used to mask out particular bits (forcing those bits to 0's).
   - ○ True
   - ○ False

2) OR can be used to set particular bits to 1.
   - ○ True
   - ○ False

3) Goal: Force the rightmost bit of $s3 to

be 1.

```
____ $s3, $s3, 1
```

- ○ andi
- ○ ori

4) Goal: Invert every bit of $s3. Ex: If $s3 is 1010…1010, make $s3 0101…0101.

```
____ $s3, $s3, ____
```

- ○ ori / $zero
- ○ nor / $zero

5) To isolate bits 7..4 in the rightmost 4 bits of a 32-bit word, one can first shift left 24 bits, then right ____ bits.

- ○ 24
- ○ 28

6) Goal: multiply $s2 by 8.

```
sll $s2, $s2, ____
```

- ○ 3
- ○ 8

PARTICIPATION ACTIVITY 2.6.6: Check yourself: Masking operators.

1) An AND operation can isolate a field in a word.

- ○ True
- ○ False

2) A shift left followed by a shift right operation can isolate a field in a word.

- ○ True
- ○ False

CHALLENGE ACTIVITY 2.6.1: Logical operations.

Note: The type of operands may change depending on instruction choices.

622166.4950548.qx3zqy7

Start

Compute: $s3 = ($s4 >> 4) & 7

srl $s3, $s3, 0

andi $s3, $s3, 0

Registers

$s3 0

$s4 0..01111110100000

$s5 0

1 2 3 4 5

Check Next

2.7 Instructions for making decisions

> "The utility of an automatic computer lies in the possibility of using a given sequence of instructions repeatedly, the number of times it is iterated being dependent upon the results of the computation …. This choice can be made to depend upon the sign of a number (zero being reckoned as plus for machine purposes). Consequently, we introduce an [instruction] (the conditional transfer [instruction]) which will, depending on the sign of a given number, cause the proper one of two routines to be executed.
> *Burks, Goldstine, and von Neumann, 1946*

What distinguishes a computer from a simple calculator is its ability to make decisions. Based on the input data and the values created during computation, different instructions execute. Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *go to* statements and labels. MIPS assembly language includes two decision-making instructions, similar to an *if* statement with a *go to*. The first instruction is


```
beq register1, register2, L1
```

This instruction means go to the statement labeled `L1` if the value in `register1` equals the value in `register2`. The mnemonic beq stands for *branch if equal*. The second instruction is

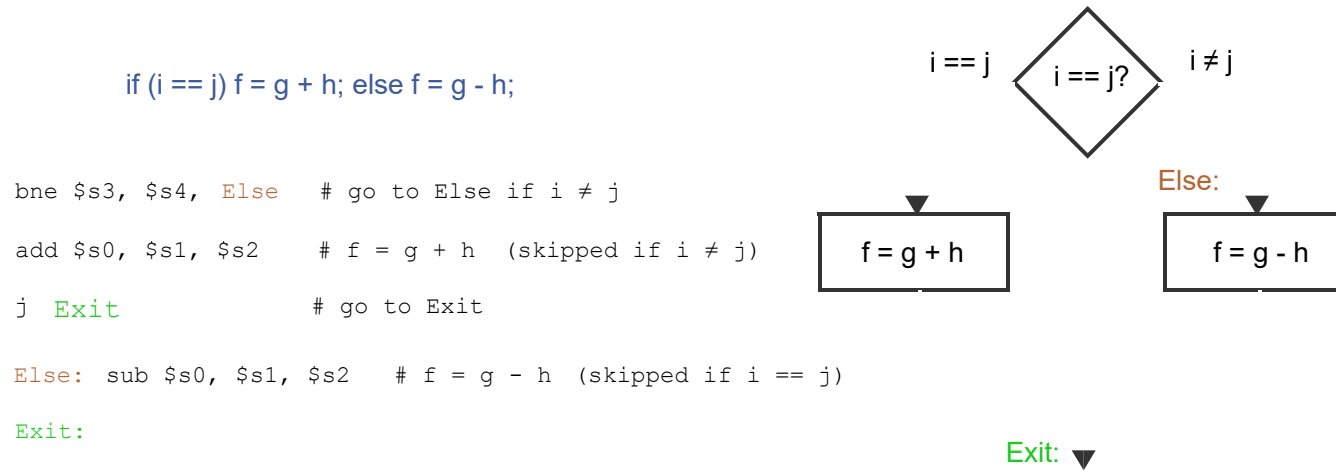
```
bne register1, register2, L1
```

It means go to the statement labeled `L1` if the value in `register1` does *not* equal the value in `register2`. The mnemonic `bne` stands for *branch if not equal*. These two instructions are traditionally called *conditional branches*.

Conditional branch: An instruction that requires the comparison of two values and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison.

In contrast, an ***unconditional branch*** is an instruction that always follows the branch, as in the "j" instruction in the animation below.

PARTICIPATION ACTIVITY 2.7.1: Example of compiling if-then-else into conditional branches (COD Figure 2.9).



Animation content:

Static figure: C code, assembly instructions, and a flowchart are displayed.

Begin C code:
if (i == j) f = g + h; else f = g - h;
End C code.

Begin assembly instructions:
bne $s3, $s4, Else # go to Else if i ≠ j
add $s0, $s1, $s2 # f = g + h (skipped if i ≠ j)
j Exit # go to Exit

Else: sub $s0, $s1, $s2 # f = g - h (skipped if i == j)
Exit:
End assembly instructions.

The flowchart contains a diamond-shaped decision node labeled i == j?, from which two branches emerge: one labeled i == j, leading to a block containing f = g + h, and the other labeled i ≠ j, leading to another block labeled Else:, containing f = g - h. Following each of these two blocks, arrows converge to an arrow labeled Exit.

Step 1: The C if-else statement uses variables f, g, h, i, and j. The flowchart illustrates the statement's behavior.

Step 2: For example, if i is 9 and j is 9, the add executes.

Step 3: f = g + h and f = g - h are compiled to these assembly instructions.
Two lines of assembly code are displayed.
Begin assembly instructions:
add $s0, $s1, $s2 # f = g + h
sub $s0, $s1, $s2 # f = g - h
End assembly instructions.

Step 4: If i and j are NOT equal, a branch occurs to the Else part (which needs a label). One might consider comparing for equal instead, but testing for not equal yields more efficient code.
The assembly instructions are updated and a new instruction is added.
Begin assembly instructions:
bne $s3, $s4, Else # go to Else if i ≠ j
add $s0, $s1, $s2 # f = g + h (skipped if i ≠ j)
Else: sub $s0, $s1, $s2 # f = g - h
End assembly instructions.

Step 5: If i equals j, the first part executes. A jump to Exit skips over the Else part.
The assembly instructions are updated and two new instructions are added.
Begin assembly instructions:
bne $s3, $s4, Else # go to Else if i ≠ j
add $s0, $s1, $s2 # f = g + h (skipped if i ≠ j)
j Exit # go to Exit
Else: sub $s0, $s1, $s2 # f = g - h (skipped if i == j)
Exit:
End assembly instructions.

Step 6: If i == j, the add executes and execution jumps to the end, skipping the subtract (the Else part).

Step 7: If i ≠ j, the add is skipped, and the subtract (the Else part) executes. Either path ends at the Exit label.

Animation captions:

1. The C if-else statement uses variables f, g, h, i, and j. The flowchart illustrates the statement's behavior.
2. For example, if i is 9 and j is 9, the add executes.
3. f = g + h and f = g - h are compiled to these assembly instructions.
4. If i and j are NOT equal, a branch occurs to the Else part (which needs a label). One might consider comparing for equal instead, but testing for not equal yields more efficient code.
5. If i equals j, the first part executes. A jump to Exit skips over the Else part.
6. If i == j, the add executes and execution jumps to the end, skipping the subtract (the Else part).
7. If i ≠ j, the add is skipped, and the subtract (the Else part) executes. Either path ends at the Exit label.

Notice that the assembler relieves the compiler and the assembly language programmer from the tedium of calculating addresses for branches, just as it does for calculating data addresses for loads and stores.

Hardware/Software Interface

Compilers frequently create branches and labels where they do not appear in the programming language. Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is a reason coding is faster at that level.

PARTICIPATION ACTIVITY 2.7.2: Branches.

Assume $s1 has 50 and $s2 has 30. Given this code:

```
bne $s3, $s4, Else
add $s0, $s1, $s2
j Exit
```

```
Else: sub $s0, $s1, $s2
Exit:
```

1) What is "bne" short for?

- branch equal
- branch not equal
- be nice everyone

2) If $s3 is 9 and $s4 is 9, which instruction executes after bne?

- add
- sub

3) `j Exit` is executed when $s3 and $s4 values _____.

- are equal
- are not equal

4) If the first instruction were beq rather than bne, what instruction should then appear immediately after beq?

- `add $s0, $s1, $s2`
- `sub $s0, $s1, $s2`

5) Given the C statement "if (i == j) f = g + h", what instruction is needed (assuming variables are mapped to registers properly)?

```
      _____ $s3, $s4, Exit
      add   $s0, $s1, $s2
Exit:
```

- bne
- beq

CHALLENGE ACTIVITY 2.7.1: Write branch conditions.

Notes:

- Some instructions are deliberately unchangeable.
- Pseudocode "if (a condition is true) then ..." is implemented as a MIPS instruction that branches when the *opposite* condition is true.
- A label on an empty line after all the instructions marks the exit point of the program.

622166.4950548.qx3zqy7

Start

Convert pseudocode to MIPS:

```
if ($t0 != $t1)
   $t2 = $t2 + $t2;
```

```
add $t2 , $t2 , $t2
add $t2 , $t2 , $t2
L1:     # Code ends here
```

1 2 3 4 5

Check Next

Loops

Decisions are important both for choosing between two alternatives—found in *if* statements—and for iterating a computation—found in loops. The same assembly instructions are the building blocks for both cases.

PARTICIPATION ACTIVITY 2.7.3: Compiling a C while loop.


```
while (x == y) {
   // Loop body
}
```

```
Loop:  bne $s0, $s1, Exit
   # Loop body
   j Loop
Exit:
```

Animation content:

Static figure: Two separate code blocks side by side, with the left block showing C code, and the right block showing assembly language instructions.
Begin C code:

```
while (x == y) {
   // Loop body
}
```

End C code.
Begin assembly code:

```
Loop: bne $s0, $s1, Exit
  # Loop body
  j Loop
Exit:
```

End assembly code.

Step 1: The loop condition determines if the loop body should be entered or if instead execution should jump past the loop. Various loop conditions are possible; this condition uses ==.
The left block code while (x == y) { and the right block code Loop: bne $s0, $s1, Exit are both highlighted at the same time.
Step 2: If $s0 equals $s1 (assume x is $s0 and y is $s1), the loop body should execute. Assume equal: bne branches if NOT equal, so when equal execution falls through to the loop body.
The left block code // Loop body and the right block code # Loop body are both highlighted at the same time.

Step 3: After the loop body, execution jumps unconditionally to the start of the loop again.
The left block code } and the right block code j Loop are both highlighted at the same time.
Then, the left block code while (x == y) { and the right block code Loop: bne $s0, $s1, Exit are both highlighted at the same time.

Step 4: If the values are not equal, execution should jump past the loop. bne branches when not equal, so jumps to Exit.
The blank space below the } of the left block code and the right block code Exit: are both highlighted at the same time.

©zyBooks 05/16/25 23:10 2475274
Jaheim Attri
FIUEEL4709CSpring2025

Animation captions:

1. The loop condition determines if the loop body should be entered or if instead execution should jump past the loop. Various loop conditions are possible; this condition uses ==.
2. If $s0 equals $s1 (assume x is $s0 and y is $s1), the loop body should execute. Assume equal: bne branches if NOT equal, so when equal execution falls through to the loop body.
3. After the loop body, execution jumps unconditionally to the start of the loop again.

4. If the values are not equal, execution should jump past the loop. bne branches when not equal, so jumps to Exit.

Example 2.7.1: Compiling a while loop in C.

Here is a traditional loop in C:

```
while (save[i] == k)
      i += 1;
```

Assume that `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array save is in `$s6`. What is the MIPS assembly code corresponding to this C segment?

Answer

The first step is to load `save[i]` into a temporary register. Before we can load `save[i]` into a temporary register, we need to have its address. Before we can add `i` to the base of array `save` to form the address, we must multiply the index `i` by 4 due to byte addressing . Fortunately, we can use shift left logical, since shifting left by 2 bits multiplies by 2^2 or 4. We need to add the label `Loop` to it so that we can branch back to that instruction at the end of the loop:

```
Loop: sll $t1, $s3, 2      # Temp reg $t1 = i * 4
```

To get the address of `save[i]`, we need to add `$t1` and the base of save in `$s6`:

```
      add $t1, $t1, $s6    # $t1 = address of save[i]
```

Now we can use that address to load `save[i]` into a temporary register:

```
      lw $t0, 0($t1)       # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if `save[i] ≠ k`:


```
      bne $t0, $s5, Exit   # go to Exit if save[i] ≠ k
```

The next instruction adds 1 to `i`:

```
      addi $s3, $s3, 1     # i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. We just add the `Exit` label after it, and we're done:

```
      j     Loop            # go to Loop
Exit:
```

(See the exercises for an optimization of this sequence.)

Hardware/Software Interface

Such sequences of instructions that end in a branch are so fundamental to compiling that they are given their own buzzword: a *basic block* is a sequence of instructions without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning. One of the first early phases of compilation is breaking the program into basic blocks.

Basic block: A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

PARTICIPATION ACTIVITY 2.7.4: Loops.

1) In the above example, the first three instructions (sll, add, lw) deal with _____.
   - ○ getting save[i]'s value
   - ○ executing the loop body

2) In the above example, the loop condition uses ==, but the compiled code uses bne (branch if not equal). Use of bne rather than beq is _____.
   - ○ a mistake
   - ○ correct

3) Given the C loop: while (x != y) { .... }. != means not equal, and assume $s0 is

x and $s1 is y. Complete the compiled loop:

```
Loop: _____  $s0 $s1 Exit
      # Loop body
      j Loop
Exit:
```


- ○ beq
- ○ bne

The test for equality or inequality is probably the most popular test, but sometimes it is useful to see if a variable is less than another variable. For example, a *for* loop may want to test to see if the index variable is less than 0. Such comparisons are accomplished in MIPS assembly language with an instruction that compares two registers and sets a third register to 1 if the first is less than the second; otherwise, it is set to 0. The MIPS instruction is called *set on less than*, or `slt`. For example,

```
slt  $t0, $s3, $s4   # $t0 = 1 if $s3 < $s4
```

means that register `$t0` is set to 1 if the value in register `$s3` is less than the value in register `$s4`; otherwise, register `$t0` is set to 0.

Constant operands are popular in comparisons, so there is an immediate version of the set on less than instruction. To test if register `$s2` is less than the constant 10, we can just write

```
slti $t0, $s2, 10    # $t0 = 1 if $s2 < 10
```

Hardware/Software Interface

MIPS compilers use the `slt`, `slti`, `beq`, `bne`, and the fixed value of 0 (always available by reading register `$zero`) to create all relative conditions: equal, not equal, less than, less than or equal, greater than, greater than or equal.

Heeding von Neumann et al's warning about the simplicity of the "equipment", the MIPS architecture doesn't include branch on less than because it is too complicated; either it would stretch the clock cycle time or it would take extra clock cycles per instruction. Two faster instructions are more useful.

PARTICIPATION ACTIVITY 2.7.5: Loops with relative comparisons.

1) Given: for (i = 0; i < 9; ++i) { .... } where i is $s0. Which comparison instruction is most appropriate?

- `slt  $t0, $s0, 9`
- `slti  $t0, $s0, 9`
- `slti  $t0, $s0, 0`

2) Given: for (i = 50; i > 10; --i) { .... } where i is $s0. Which comparison instruction is most appropriate?

- `slti  $t0, 10, 50`
- `slti  $t0, 10, $s0`
- None of the above

3) Given: for (i = 1; i < j; ++i) { loop body } where i is $s0, j is $s1. Complete the indicated instruction:

```
Loop: slt   $t3, $s0, $s1
      _____ $t3, $zero,
Exit
      # Loop body
      j Loop
Exit:
```

- beq
- bne
- slti

4) What does slt stand for?

- shift on less than
- set on lowly triumph
- set on less than

CHALLENGE ACTIVITY 2.7.2: Write loops.

622166.4950548.qx3zqy7

Start

Convert pseudocode to MIPS:

```
while ($s1 != $s2){
   $s1 = $s1 << 1;
}
```



```
Loop:  sll $s1, $s1, 1
       sll $s1, $s1, 1
       sll $s1, $s1, 1
Exit:
```

1	2	3	4	5

Check Next

Hardware/Software Interface

Comparison instructions must deal with the dichotomy between signed and unsigned numbers. Sometimes a bit pattern with a 1 in the most significant bit represents a negative number and, of course, is less than any positive number, which must have a 0 in the most significant bit. With unsigned integers, on the other hand, a 1 in the most significant bit represents a number that is *larger* than any that begins with a 0. (We'll soon take advantage of this dual meaning of the most significant bit to reduce the cost of the array bounds checking.)

MIPS offers two versions of the set on less than comparison to handle these alternatives. *Set on less than* (`slt`) and *set on less than immediate* (`slti`) work with signed integers. Unsigned integers are compared using *set on less than unsigned* (`sltu`) and *set on less than immediate unsigned* (`sltiu`).

Example 2.7.2: Signed versus unsigned comparison.

Suppose register `$s0` has the binary number

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$

and that register `$s1` has the binary number

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}$

What are the values of registers `$t0` and `$t1` after these two instructions?

```
slt   $t0, $s0, $s1   # signed comparison
sltu  $t1, $s0, $s1   # unsigned comparison
```

Answer

The value in register `$s0` represents -1_{ten} if it is an integer and $4{,}294{,}967{,}295_{ten}$ if it is an unsigned integer. The value in register `$s1` represents 1_{ten} in either case. Then register `$t0` has the value 1, since $-1_{ten} < 1_{ten}$, and register `$t1` has the value 0, because $4{,}294{,}967{,}295_{ten} > 1_{ten}$.

Treating signed numbers as if they were unsigned gives us a low cost way of checking if $0 \le x < y$, which matches the index out-of-bounds check for arrays. The key is that negative integers in two's complement notation look like large numbers in unsigned notation; that is, the most significant bit is a sign bit in the former notation but a large part of the number in the latter. Thus, an unsigned comparison of $x < y$ also checks if x is negative as well as if x is less than y.

Example 2.7.3: Bounds check shortcut.

Use this shortcut to reduce an index-out-of-bounds check: jump to `IndexOutOfBounds` if `$s1` $\ge$ `$t2` or if `$s1` is negative.

Answer

The checking code just uses `u` to do both checks:

```
sltu $t0, $s1, $t2                  # $t0 = 0 if $s1 >= length or $s1 <
beq  $t0, $zero, IndexOutOfBounds   # if bad, goto Error
```

PARTICIPATION ACTIVITY 2.7.6: Signed and unsigned comparisons.

Assume $s0 has the binary number 1111 0000 0000 0000 0000 0000 0000 1111_{two} and $s1 has the binary number 0000 0000 0000 0000 0000 0000 0000 1111_{two}.

1) What is the value of $t0 after the following instruction?


```
slt $t0, $s0, $s1
```

- 1_{ten}
- 0_{ten}

2) What is the value of $t0 after the following instruction?

```
sltu $t0, $s0, $s1
```

- 1_{ten}
- 0_{ten}

3) Assume the values in $s0 and $s1 are signed binary numbers, and $s1 is positive. Complete the following code to jump to `L2` if the bounds check 0 ≤ $s0 < $s1 fails.

```
_____ $t0, $s0, $s1
beq   $t0, $zero, L2
```

- sltu
- slt

Case/switch statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement switch is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a *jump address table* or *jump table*, and the program needs only to index into the table and then jump to the appropriate sequence. The jump table is then just an array of words containing addresses that correspond to labels in the code. The program loads the

appropriate entry from the jump table into a register. It then needs to jump using the address in the register. To support such situations, computers like MIPS include a *jump register* instruction (`jr`), meaning an unconditional jump to the address specified in a register. Then it jumps to the proper address using this instruction. We'll see an even more popular use of `jr` in the next section.

Jump address table: Also called ***jump table***. A table of addresses of alternative instruction sequences.

Hardware/Software Interface

Although there are many statements for decisions and loops in programming languages like C and Java, the bedrock statement that implements them at the instruction set level is the conditional branch.

PARTICIPATION ACTIVITY 2.7.7: Check yourself: Branches.

C has many kinds of statements for decisions and loops, while MIPS has few. Which of the following explain this imbalance?

1) More kinds of decision statements make code easier to read and understand.
   - ○ True
   - ○ False

2) Fewer kinds of decision statements simplify the task of the underlying layer that is responsible for execution.
   - ○ True
   - ○ False

3) More kinds of decision statements mean fewer lines of code, which generally reduces coding time.
   - ○ True

- False

4) More kinds of decision statements mean fewer lines of code, which generally results in the execution of fewer operations.

- True
- False

PARTICIPATION ACTIVITY 2.7.8: Check yourself: Logical operators and branches.

1) Why does C provide two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||), while MIPS doesn't?

- Logical operations AND and OR implement & and |, while conditional branches implement && and ||.
- The above choice has it backwards: && and || correspond to logical operations, while & and | map to conditional branches.
- They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C.

2.8 Supporting procedures in computer hardware

A *procedure* or function is one tool programmers use to structure programs, both to make them easier to understand and to allow code to be reused. Procedures allow the programmer to concentrate on just one portion of the task at a time; parameters act as an interface between the procedure and the rest of the program

and data, since they can pass values and return results. We describe the equivalent to procedures in Java in COD Section 2.15 (Advanced Material: Compiling C and Interpreting Java), but Java needs everything from a computer that C needs. Procedures are one way to implement **abstraction** in software.

Procedure: A stored subroutine that performs a specific task based on the parameters with which it is provided.

You can think of a procedure like a spy who leaves with a secret plan, acquires resources, performs the task, covers his or her tracks, and then returns to the point of origin with the desired result. Nothing else should be perturbed once the mission is complete. Moreover, a spy operates on only a "need to know" basis, so the spy can't make assumptions about his employer.

Similarly, in the execution of a procedure, the program must follow these six steps:

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

As mentioned above, registers are the fastest place to hold data in a computer, so we want to use them as much as possible. MIPS software follows the following convention for procedure calling in allocating its 32 registers:

- `$a0 – $a3`: four argument registers in which to pass parameters
- `$v0 – $v1`: two value registers in which to return values
- `$ra`: one return address register to return to the point of origin

In addition to allocating these registers, MIPS assembly language includes an instruction just for the procedures: it jumps to an address and simultaneously saves the address of the following instruction in register `$ra`. The *jump-and-link instruction* (`jal`) is simply written

```
jal ProcedureAddress
```


jump-and-link instruction: An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register (`$ra` in MIPS).

The *link* portion of the name means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address. This "link", stored in register `$ra` (register 31), is called the *return address*. The return address is needed because the same procedure could

be called from several parts of the program.

Return address: A link to the calling site that allows a procedure to return to the proper address; in MIPS it is stored in register `$ra`.

To support such situations, computers like MIPS use *jump register* instruction (`jr`), introduced above to help with case statements, meaning an unconditional jump to the address specified in a register:


```
jr $ra
```

The jump register instruction jumps to the address stored in register `$ra`—which is just what we want. Thus, the calling program, or *caller*, puts the parameter values in `$a0 – $a3` and uses `jal X` to jump to procedure `X` (sometimes named the *callee*). The callee then performs the calculations, places the results in `$v0` and `$v1`, and returns control to the caller using `jr $ra`.

Caller: The program that instigates a procedure and provides the necessary parameter values.

Callee: A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

Implicit in the stored-program idea is the need to have a register to hold the address of the current instruction being executed. For historical reasons, this register is almost always called the *program counter*, abbreviated *PC* in the MIPS architecture, although a more sensible name would have been *instruction address register*. The `jal` instruction actually saves PC + 4 in register `$ra` to link to the following instruction to set up the procedure return.

Program counter _(PC)_: The register containing the address of the instruction in the program being executed.

PARTICIPATION ACTIVITY 2.8.1: Procedure basics.

Idea	*MIPS*
Main program:	Main program:
InstrA	InstrA
InstrB	InstrB
Call ProcX with f, g	Set $a0-$a3
Instr operating on r	jal ProcXAddress *(sets $ra to next Instr's address)*
	Instr operating on $v0-$v1

Animation content:

Static Figure:
Begin program:
Idea
Main program:
  InstrA
  InstrB
  Call ProcX with f, g
  Instr operating on r
  ...
ProcX:
  Instrs operating on f and g
  Return r
End program.
The line of code Call ProcX with f, g is highlighted blue. The line of code Return r is highlighted blue.
An arrow is shown starting at the line Call ProcX with f, g and pointing to the line of code Instrs operating on f and g. Another arrow is shown starting at the line of code Return r and pointing to the line of code Instr operating on r.

Begin MIPS program:
Main program:
  InstrA
  InstrB
  Set $a0-$a3
  jal ProcXAddress (sets $ra to next Instr's address)
  Instr operating on $v0-$v1
  ...
ProcX:
  Instrs operating on $a0-$a3
  Set $v0-$v1
  jr $ra
End MIPS program.

©zyBooks 05/16/25 23:10 2475274
Jaheim Attri
FIUEEL4709CSpring2025

The line of code jal ProcXAddress is highlighted blue. The line of code jr $ra is highlighted blue. The line of code Set $a0-$a3 is highlighted purple. The line of code Set $v0-$v1 is highlighted purple.
An arrow is shown starting at the line jal ProcXAddress and pointing to the line of code Instrs operating on $a0-$a3. Another arrow is shown starting at the line of code jr $ra and pointing to the line of code Instr operating on $v0-$v1.

Step 1: Procedure ProcX has instructions for a particular computation, like power, average, etc. A main program starts by executing normal instructions InstrA and InstrB...
In the main program, the line of code InstrA is highlighted. Then, the line of code InstrB is highlighted.

Step 2: ...and then "calls" ProcX, "passing" values f and g ("parameters"). Execution jumps to the procedure's instructions, while the next main instruction's address is also remembered.
In the main program, the line of code Call ProcX with f, g is highlighted. Then the lines of code ProcX:
  Instrs operating on f and g
are highlighted.
An arrow is shown starting at the line Call ProcX with f, g and pointing to the line of code Instrs operating on f and g.

Step 3: The procedure's instructions execute, operating on the passed values. When done, a resulting value is returned, and execution resumes at the next main instruction.
In the main program, the line of code Return r is highlighted. Then, the line of code Instr operating on r is highlighted. Then, the line of code ... is highlighted.
An arrow is shown starting at the line of code Return r and pointing to the line of code Instr operating on r.

Step 4: In MIPS, "argument" registers $a0, $a1, $a2, and $a3 are for passing parameters.
jal instruction jumps to address while saving next instruction's address in $ra.
In the MIPS program, the lines of code
  InstrA
  InstrB
  Set $a0-$a3
are highlighted.

Then, the line of code
  jal ProcXAddress
is highlighted, and the text (sets $ra to next Instr's address) appears next to the line of code.
Next, the lines of code
ProcX:
  Instrs operating on $a0-$a3

are highlighted.
An arrow is shown starting at the line jal ProcXAddress and pointing to the line of code Instrs operating on $a0-$a3.

Step 5: Procedures operate on parameter values. MIPS provides registers $v0 and $v1 for returning values. jr $ra jumps to address in $ra, which is the next main instruction.
In the MIPS program, the lines of code
  Set $v0-$v1
  jr $ra
are highlighted.
Next, the lines of code
  Instr operating on $v0-$v1
  ...
are highlighted.
An arrow is shown starting at the line of code jr $ra and pointing to the line of code Instr operating on $v0-$v1.

Animation captions:

1. Procedure ProcX has instructions for a particular computation, like power, average, etc. A main program starts by executing normal instructions InstrA and InstrB...
2. ...and then "calls" ProcX, "passing" values f and g ("parameters"). Execution jumps to the procedure's instructions, while the next main instruction's address is also remembered.
3. The procedure's instructions execute, operating on the passed values. When done, a resulting value is returned, and execution resumes at the next main instruction.
4. In MIPS, "argument" registers $a0, $a1, $a2, and $a3 are for passing parameters. jal instruction jumps to address while saving next instruction's address in $ra.
5. Procedures operates on parameter values. MIPS provides registers $v0 and $v1 for returning values. jr $ra jumps to address in $ra, which is the next main instruction.

PARTICIPATION ACTIVITY 2.8.2: Basic procedure call and return.

1) A main program will call a procedure Power for computing x^y. Currently, x is in $s0, y is in $s1. How might the program pass the parameter values to Power?

   ○
   ```
   add $a0, $s0, $zero
   add $a1, $s1, $zero
   ```

- `add $s0, $a0, $zero`
  `add $s1, $a1, $zero`
- `add $v0, $s0, $zero`
  `add $v1, $s1, $zero`

2) A first part of a main program calls procedure Power to compute x^y, where x is in $s0, y is in $s1. Later, the program is to call Power again, but this time x is in $s3 and y is in $s7. How might the program pass the parameter values to Power?

- Copy $s3 to $a0, and $s7 to $a1.
- Not possible; x and y must be in $s0 and $s1.

3) A main program calls a Power procedure using the instruction: `jal Power`. That instruction is at address 1000. What happens to $ra?

- Nothing; jal is unrelated to $ra.
- $ra is set to 1000.
- $ra is set to 1004.

4) A procedure Power computes $a0 to the power of $a1. In which register should Power write the result before returning?

- $a0
- $v0
- $s0

5) A procedure Power computes $a0 to the power of $a1. How should the procedure jump back to the next instruction in the caller?

- `jr Caller`
- `jr $ra`

- `jal $ra`

Using more registers

Suppose a compiler needs more registers for a procedure than the four argument and two return value registers. Since we must cover our tracks after our mission is complete, any registers needed by the caller must be restored to the values that they contained *before* the procedure was invoked. This situation is an example in which we need to spill registers to memory, as mentioned in the *Hardware/Software Interface* in COD Section 2.3 (Operands of the computer hardware).

The ideal data structure for spilling registers is a *stack*—a last-in-first-out queue. A stack needs a pointer to the most recently allocated address in the stack to show where the next procedure should place the registers to be spilled or where old register values are found. The *stack pointer* is adjusted by one word for each register that is saved or restored. MIPS software reserves register 29 for the stack pointer, giving it the obvious name `$sp`. Stacks are so popular that they have their own buzzwords for transferring data to and from the stack: placing data onto the stack is called a *push*, and removing data from the stack is called a *pop*.

Stack: A data structure for spilling registers organized as a last-in- first-out queue.

Stack pointer: A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In MIPS, it is register `$sp`.

Push: Add element to stack.

Pop: Remove element from stack.

By historical precedent, stacks "grow" from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer. Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

Example 2.8.1: Compiling a C procedure that doesn't call another procedure.

Let's turn the following example into a C procedure:

```
int leaf_example (int g, int h, int i, int j)
```

```
{
      int f;

      f = (g + h) - (i + j);
      return f;
}
```

What is the compiled MIPS assembly code?

Answer

The parameter variables `g`, `h`, `i`, and `j` correspond to the argument registers `$a0`, `$a1`, `$a2`, and `$a3`, and `f` corresponds to `$s0`. The compiled program starts with the label of the procedure:

```
leaf_example:
```

The next step is to save the registers used by the procedure. The C assignment statement in the procedure body is identical to the example in Section 2.2 (Operations of the Computer Hardware), which uses two temporary registers. Thus, we need to save three registers: `$s0`, `$t0`, and `$t1`. We "push" the old values onto the stack by creating space for three words (12 bytes) on the stack and then store them:

```
addi $sp, $sp, -12      # adjust stack to make room for 3 items
sw $t1, 8($sp)          # save register $t1 for use afterwards
sw $t0, 4($sp)          # save register $t0 for use afterwards
sw $s0, 0($sp)          # save register $s0 for use afterwards
```

The animation below shows the stack before, during, and after the procedure call. The next three statements correspond to the body of the procedure:

```
add $t0, $a0, $a1      # register $t0 contains g + h
add $t1, $a2, $a3      # register $t1 contains i + j
sub $s0, $t0, $t1      # f = $t0 - $t1, which is (g + h)-(i + j)
```

To return the value of f, we copy it into a return value register:


```
add $v0, $s0, $zero    # returns f ($v0 = $s0 + 0)
```

Before returning, we restore the three old values of the registers we saved by "popping" them from the stack:

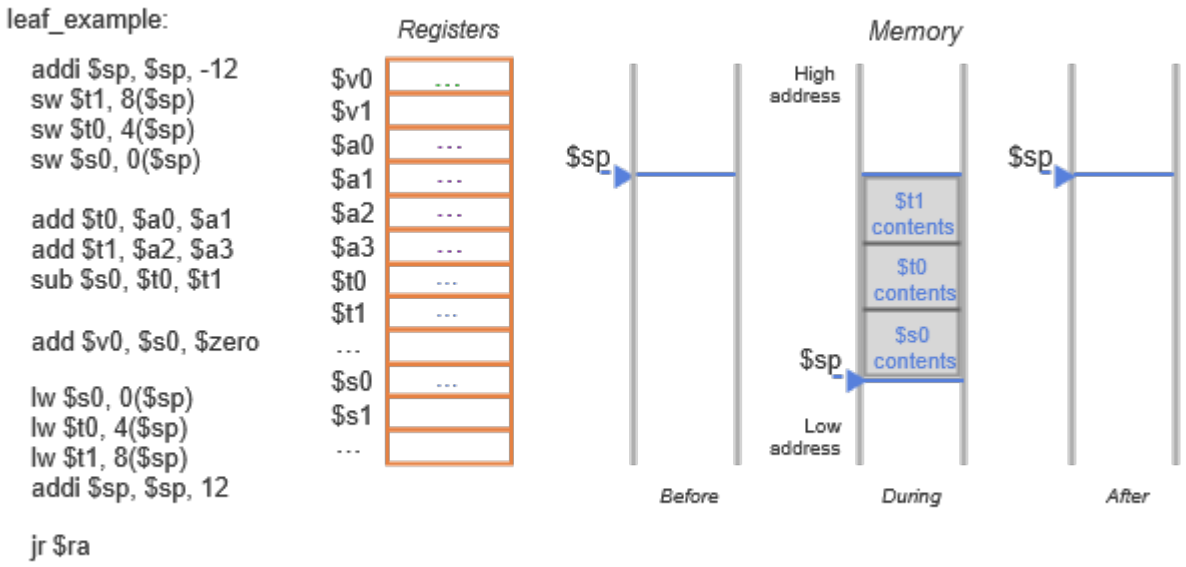
```
lw $s0, 0($sp)         # restore register $s0 for caller
lw $t0, 4($sp)         # restore register $t0 for caller
```

```
lw $t1, 8($sp)          # restore register $t1 for caller
addi $sp, $sp, 12       # adjust stack to delete 3 items
```

The procedure ends with a jump register using the return address:

```
jr $ra                  # jump back to calling routine
```


PARTICIPATION ACTIVITY 2.8.3: The values of the stack pointer and the stack before, during, and after the procedure call (COD Figure 2.10).



The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Animation content:

Static figure: The procedure that is executed is a list of instructions labeled "leaf example":
addi $sp, $sp, -12
sw $t1, 8($sp)
sw $t0, 4($sp)
sw $s0, 0($sp)
add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
add $v0, $s0, $zero
lw $s0, 0($sp)
lw $t0, 4($sp)
lw $t1, 8($sp)

addi $sp, $sp, 12
jr $ra
A list of registers $v0, $v1, $a0, $a1, $a2, $a3, $t0, $t1, $s0, $s1. The memory stack before, during, and after the procedure call. The memory stack is empty before and after the procedure call. The memory stack during the procedure call contains $t1 contents, $t0 contents, and $s0 contents. The stack pointer $sp points to the bottom of the stack, which is $s0 contents. In memory, addresses toward the top of the stack have high addresses, and addresses toward the bottom of the stack have low addresses.

Step 1: A main program (not shown) first puts parameter values in $a0-$a3, then calls procedure leaf_example.

Step 2: Procedure leaf_example uses $t1, $t0, and $s0, so makes room on stack, and copies those registers' values to stack. The instruction addi $sp, $sp, -12 is executed and the position of the stack pointer is updated. The instruction sw $t1, 8($sp) is executed and the contents of $t1 are added to the stack. The instruction sw $t0, 4($sp) is executed and the contents of $t0 are added to the stack. The instruction sw $s0, 0($sp) is executed and the contents of $s0 are added to the stack. The stack pointer points to the bottom of the stack, which is the location of the contents of $s0.

Step 3: The procedure can freely modify $t0, $t1, and $s0 during computations. When done, the procedure copies the computed value to $v0, which the main program will later read. The following instructions are executed:
sw $s0, 0($sp)
add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
add $v0, $s0, $zero
The value of the result of these computations are stored in register $v0.

Step 4: The procedure copies the saved values of $s0, $t0, and $t1 back, so the main program won't notice those registers were changed. The stack pointer is restored too. The instruction lw $s0, 0($sp) is executed. The previously stored value of $s0 in the stack is loaded into the register $s0. The instruction lw $t0, 4($sp) is executed. The previously stored value of $t0 in the stack is loaded into the register $t0. The instruction lw $t1, 8($sp) is executed. The previously stored value of $t1 is loaded into the register $t1. The instruction addi $sp, $sp, 12 is executed. The stack pointer's position is updated.

Step 5: Finally, the procedure returns to the instruction following jal in main. The instruction jr $ra is executed.

Animation captions:

1. A main program (not shown) first puts parameter values in $a0-$a3, then calls procedure leaf_example.
2. Procedure leaf_example uses $t1, $t0, and $s0, so makes room on stack, and copies those registers' values to stack.
3. The procedure can freely modify $t0, $t1, and $s0 during computations. When done, the procedure copies the computed value to $v0, which the main program will later read.
4. The procedure copies the saved values of $s0, $t0, and $t1 back, so the main program won't notice those registers were changed. The stack pointer is restored too.
5. Finally, the procedure returns to the instruction following jal in main.

In the previous example, we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, MIPS software separates 18 of the registers into two groups:

- `$t0 – $t9`: temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- `$s0 – $s7`: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

This simple convention reduces register spilling. In the example above, since the caller does not expect registers `$t0` and `$t1` to be preserved across a procedure call, we can drop two stores and two loads from the code. We still must save and restore `$s0`, since the callee must assume that the caller needs its value.

PARTICIPATION ACTIVITY 2.8.4: Procedure call using the stack.

1) The stack is a region in the set of registers.
   - ○ True
   - ○ False

2) The jal instruction copies registers to the stack.
   - ○ True
   - ○ False

3) A procedure should copy all of registers $t0-$t9 and $s0-$s7 to the stack, before executing the procedure's computations.

- ○ True
- ○ False

4) If a procedure will update registers $s0, $s1, $s2, and $s3, the procedure should make room on the stack by adding 16 to $sp.

- ○ True
- ○ False

5) Upon computing a value to return, the procedure might copy that value into register $v0.

- ○ True
- ○ False

6) MIPS allows a procedure to modify registers $t0-$t9 without saving those registers to the stack and restoring those registers upon returning.

- ○ True
- ○ False

Nested procedures

Procedures that do not call others are called *leaf* procedures. Life would be simple if all procedures were leaf procedures, but they aren't. Just as a spy might employ other spies as part of a mission, who in turn might use even more spies, so do procedures invoke other procedures. Moreover, recursive procedures even invoke "clones" of themselves. Just as we need to be careful when using registers in procedures, more care must also be taken when invoking nonleaf procedures.

For example, suppose that the main program calls procedure A with an argument of 3, by placing the value 3 into register `$a0` and then using jal A. Then suppose that procedure A calls procedure B via jal B with an argument of 7, also placed in `$a0`. Since A hasn't finished its task yet, there is a conflict over the use of register `$a0`. Similarly, there is a conflict over the return address in register `$ra`, since it now has the return address for B. Unless we take steps to prevent the problem, this conflict will eliminate procedure A's ability to return to its caller.

One solution is to push all the other registers that must be preserved onto the stack, just as we did with the saved registers. The caller pushes any argument registers (`$a0 - $a3`) or temporary registers (`$t0 - $t9`) that are needed after the call. The callee pushes the return address register `$ra` and any saved registers (`$s0 - $s7`) used by the callee. The stack pointer `$sp` is adjusted to

account for the number of registers placed on the stack. Upon the return, the registers are restored from memory and the stack pointer is readjusted.

Example 2.8.2: Compiling a recursive C procedure, showing nested procedure linking.

Let's tackle a recursive procedure that calculates factorial:

```
int fact (int n)
{
      if (n < 1) return (1);
            else return (n * fact(n - 1));
}
```

What is the MIPS assembly code?

Answer

The parameter variable n corresponds to the argument register `$a0`. The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and `$a0`:

```
Fact:
addi  $sp, $sp, -8   # adjust stack for 2 items
      sw    $ra, 4($sp)    # save the return address
      sw    $a0, 0($sp)    # save the argument n
```

The first time `fact` is called, `sw` saves an address in the program that called `fact`. The next two instructions test whether `n` is less than 1, going to `L1` if `n ≥ 1`.

```
slti  $t0,$a0,1      # test for n < 1
beq   $t0,$zero,L1   # if n >= 1, go to L1
```

If `n` is less than 1, `fact` returns 1 by putting 1 into a value register: it adds 1 to 0 and places that sum in `$v0`. It then pops the two saved values off the stack and jumps to the return address:


```
addi  $v0,$zero,1   # return 1
addi  $sp,$sp,8     # pop 2 items off stack
jr    $ra           # return to caller
```

Before popping two items off the stack, we could have loaded `$a0` and `$ra`. Since `$a0`

and `$ra` don't change when `n` is less than 1, we skip those instructions.

If `n` is not less than 1, the argument `n` is decremented and then `fact` is called again with the decremented value:

```
L1: addi $a0,$a0,-1   # n >= 1: argument gets (n - 1)
    jal fact          # call fact with (n -1)
```


The next instruction is where `fact` returns. Now the old return address and old argument are restored, along with the stack pointer:

```
lw   $a0, 0($sp)   # return from jal: restore argument n
lw   $ra, 4($sp)   # restore the return address
addi $sp, $sp, 8   # adjust stack pointer to pop 2 items
```

Next, the value register `$v0` gets the product of old argument `$a0` and the current value of the value register. We assume a multiply instruction is available, even though it is not covered until COD Chapter 3 (Arithmetic for Computers):

```
mul $v0, $a0, $v0   # return n * fact (n - 1)
```

Finally, `fact` jumps again to the return address:

```
jr $ra              # return to the caller
```

Hardware/Software Interface

A C variable is generally a location in storage, and its interpretation depends both on its *type* and *storage class*. Examples include integers and characters. C has two storage classes: *automatic* and *static*. Automatic variables are local to a procedure and are discarded when the procedure exits. Static variables exist across exits from and entries to procedures. C variables declared outside all procedures are considered static, as are any variables declared using the keyword *static*. The rest are automatic. To simplify access to static data, MIPS software reserves another register, called the *global pointer*, or `$gp`.

The figure below summarizes what is preserved across a procedure call. Note that several schemes preserve the stack, guaranteeing that the caller will get the same data back on a load

Global pointer: The register that is reserved to point to the static area.

from the stack as it stored onto the stack. The stack above `$sp` is preserved simply by making sure the callee does not write above `$sp`; `$sp` is itself preserved by the callee adding exactly the same amount that was subtracted from it; and the other registers are preserved by saving them on the stack (if they are used) and restoring them from there.

Figure 2.8.1: What is and what is not preserved across a procedure call (COD Figure 2.11).

If the software relies on the frame pointer register or on the global pointer register, discussed in the following subsections, they are also preserved.

Preserved	Not preserved
Saved registers: `$s0–$s7`	Temporary registers: `$t0–$t9`
Stack pointer register: `$sp`	Argument registers: `$a0–$a3`
Return address register: `$ra`	Return value registers: `$v0–$v1`
Stack above the stack pointer	Stack below the stack pointer

PARTICIPATION ACTIVITY 2.8.5: Non-leaf procedures.

Consider a procedure P that calls another procedure Q.

1) P is a leaf procedure.
   - ○ True
   - ○ False

2) P should always save $ra on the stack.
   - ○ True
   - ○ False

3) If P will write to $a0 to pass a parameter to Q, P might first need to save $a0 to the stack.
   - ○ True

- ○ False

4) When P calls Q, P should expect that Q might pop less from the stack than Q pushed to the stack.

- ○ True
- ○ False

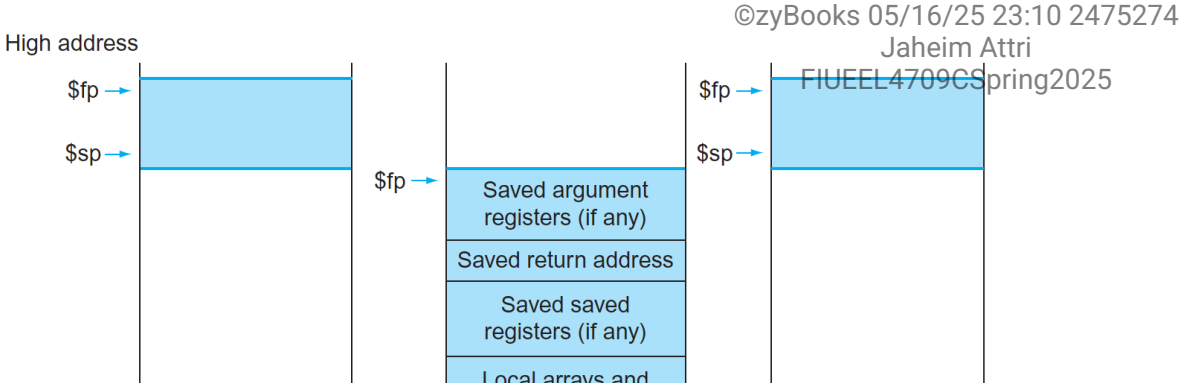
Allocating space for new data on the stack

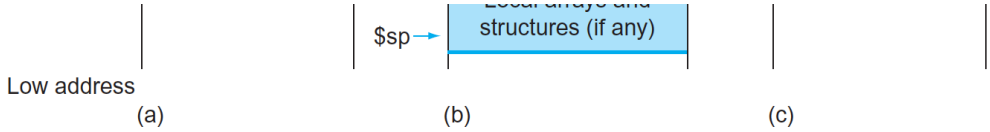
The final complexity is that the stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures. The segment of the stack containing a procedure's saved registers and local variables is called a *procedure frame* or *activation record*. The figure below shows the state of the stack before, during, and after the procedure call.

Procedure frame: Also called ***activation record***. The segment of the stack containing a procedure's saved registers and local variables.

Figure 2.8.2: Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call (COD Figure 2.12).

The frame pointer (`$fp`) points to the first word of the frame, often a saved argument register, and the stack pointer (`$sp`) points to the top of the stack. The stack is adjusted to make room for all the saved registers and any memory-resident local variables. Since the stack pointer may change during program execution, it's easier for programmers to reference variables via the stable frame pointer, although it could be done just with the stack pointer and a little address arithmetic. If there are no local variables on the stack within a procedure, the compiler will save time by not setting and restoring the frame pointer. When a frame pointer is used, it is initialized using the address in `$sp` on a call, and `$sp` is restored using `$fp`.





Some MIPS software uses a *frame pointer* (`$fp`) to point to the first word of the frame of a procedure. A stack pointer might change during the procedure, and so references to a local variable in memory might have different offsets depending on where they are in the procedure, making the procedure harder to understand. Alternatively, a frame pointer offers a stable base register within a procedure for local memory-references. Note that an activation record appears on the stack whether or not an explicit frame pointer is used. We've been avoiding using `$fp` by avoiding changes to `$sp` within a procedure: in our examples, the stack is adjusted only on entry and exit of the procedure.

Frame pointer: A value denoting the location of the saved registers and local variables for a given procedure.

PARTICIPATION ACTIVITY 2.8.6: Activation record / procedure frame.

1) Procedure P needs 2 local variables. P will almost certainly need to put those local variables on the stack.
   - ○ True
   - ○ False

2) Procedure X needs 30 local variables. X will almost certainly need to put some of those local variables on the stack.
   - ○ True
   - ○ False

3) Whether a procedure puts a local variable in a register or on the stack doesn't impact performance.
   - ○ True
   - ○ False

4) Local variables v and w saved to the

stack can be accessed as offsets from the frame pointer.

- ○ True
- ○ False

5) All of the saved registers and local variables for a procedure call are referred to as an activation record.

- ○ True
- ○ False

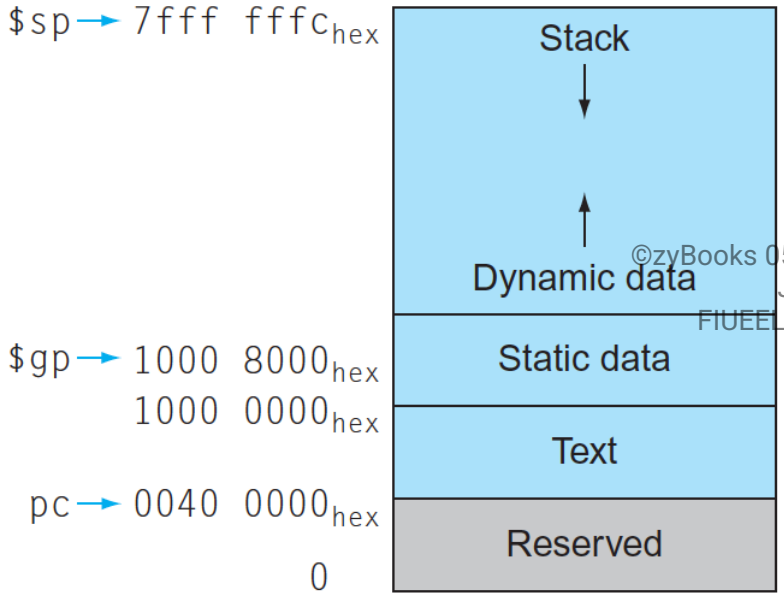
Allocating space for new data on the heap

In addition to automatic variables that are local to procedures, C programmers need space in memory for static variables and for dynamic data structures. The figure below shows the MIPS convention for allocation of memory. The stack starts in the high end of memory and grows down. The first part of the low end of memory is reserved, followed by the home of the MIPS machine code, traditionally called the *text segment*. Above the code is the *static data segment*, which is the place for constants and other static variables. Although arrays tend to be a fixed length and thus are a good match to the static data segment, data structures like linked lists tend to grow and shrink during their lifetimes. The segment for such data structures is traditionally called the *heap*, and it is placed next in memory. Note that this allocation allows the stack and heap to grow toward each other, thereby allowing the efficient use of memory as the two segments wax and wane.

Text segment: The segment of a UNIX object file that contains the machine language code for routines in the source file.

Figure 2.8.3: The MIPS memory allocation for program and data (COD Figure 2.13).

These addresses are only a software convention, and not part of the MIPS architecture. The stack pointer is initialized to `7fff fffc`$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at `0040 0000`$_{hex}$. The static data starts at `1000 0000`$_{hex}$. Dynamic data, allocated by malloc in C and by new in Java, is next. It grows up toward the stack in an area called the heap. The global pointer, `$gp`, is set to an address to make it easy to access data. It is initialized to `1000 8000`$_{hex}$ so that it can access from `1000 0000`$_{hex}$ to `1000 ffff`$_{hex}$ using the positive and negative 16-bit offsets from `$gp`.



C allocates and frees space on the heap with explicit functions. `malloc()` allocates space on the heap and returns a pointer to it, and `free()` releases space on the heap to which the pointer points. Memory allocation is controlled by programs in C, and it is the source of many common and difficult bugs. Forgetting to free space leads to a "memory leak", which eventually uses up so much memory that the operating system may crash. Freeing space too early leads to "dangling pointers", which can cause pointers to point to things that the program never intended. Java uses automatic memory allocation and garbage collection primarily to avoid such bugs.

PARTICIPATION ACTIVITY 2.8.7: Heap and stack.

1) A procedure's local variables are normally stored on the heap.
   - ○ True
   - ○ False

2) If a running program should create a new local array, the array is normally stored on the heap.
   - ○ True

- False

3) Memory is allocated for the stack and the heap so that the stack and heap grow toward each other.
   - True
   - False

4) If a program allocated huge dynamically allocated arrays, the program may cause the heap to run into the stack, causing an error.
   - True
   - False

5) If a program has a procedure P that calls itself recursively, and a bug causes P to just keep calling itself recursively without end, eventually the stack will run into the heap, causing an error.
   - True
   - False

The figure below summarizes the register conventions for the MIPS assembly language. This convention is another example of making the **common case fast**: most procedures can be satisfied with up to 4 arguments, 2 registers for a return value, 8 saved registers, and 10 temporary registers without ever going to memory.

Figure 2.8.4: MIPS register conventions (COD Figure 2.14).

Register 1, called `$at`, is reserved for the assembler, and registers 26—27, called `$k0`—`$k1`, are reserved for the operating system.

Name	Register number	Usage	Preserved on call?
`$zero`	0	The constant value 0	n.a.
`$v0–$v1`	2–3	Values for results and expression evaluation	no
`$a0–$a3`	4–7	Arguments	no
`$t0–$t7`	8–15	Temporaries	no
`$s0–$s7`	16–23	Saved	yes

$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Elaboration

What if there are more than four parameters? The MIPS convention is to place the extra parameters on the stack below the frame pointer. The procedure then expects the first four parameters to be in registers `$a0` through `$a3` and the rest in memory, addressable via the frame pointer.

The frame pointer is convenient because all references to variables in the stack within a procedure will have the same offset. The frame pointer is not necessary, however. The GNU MIPS C compiler uses a frame pointer, but the C compiler from MIPS does not; it treats register 30 as another save register (`$s8`).

Elaboration

Some recursive procedures can be implemented iteratively without using recursion. Iteration can significantly improve performance by removing the overhead associated with recursive procedure calls. For example, consider a procedure used to accumulate a sum:

```
int sum (int n, int acc) {
   if (n > 0)
      return sum(n - 1, acc + n);
   else
      return acc;
}
```


Consider the procedure call `sum(3, 0)`. This will result in recursive calls to `sum(2, 3)`, `sum(1, 5)`, and `sum(0, 6)`, and then the result 6 will be returned four times. This recursive call of sum is referred to as a tail call, and this example use of tail recursion can be implemented very efficiently (assume `$a0 = n` and `$a1 = acc`):

```
sum: slti $t0, $a0, 1           # test if n <= 0
     bne  $t0, $zero, sum_exit  # goto sum_exit if n <= 0
     add  $a1, $a1, $a0         # add n to acc
     addi $a0, $a0, -1          # subtract 1 from n
     j sum                      # go to sum
sum_exit:
     add $v0, $a1, $zero        # return value acc
     jr $ra                     # return to caller
```


PARTICIPATION ACTIVITY 2.8.8: Check yourself: Memory management in C and Java.

1) C programmers manage data explicitly, while data management is automatic in Java.
   - ○ True
   - ○ False

2) C leads to more pointer bugs and memory leak bugs than does Java.
   - ○ True
   - ○ False

2.9 Communicating with people

> "!(@ |= > (wow open tab at bar is great)
> *Fourth line of the keyboard poem "Hatless Atlas", 1991 (some give names to ASCII characters: "!" is "wow", "(" is open, "|" is bar, and so on).*

Computers were invented to crunch numbers, but as soon as they became commercially viable they were used to process text. Most computers today offer 8-bit bytes to represent characters, with the ***American Standard Code for Information Interchange*** (***ASCII***) being the representation that nearly everyone follows. The following figure summarizes ASCII.

Figure 2.9.1: ASCII representation of characters (COD Figure 2.15).

Note that upper- and lowercase letters differ by exactly 32; this observation can lead to shortcuts in checking or changing upper- and lowercase. Values not shown include formatting characters. For example, 8 represents a backspace, 9 represents a tab character, and 13 a carriage return. Another useful value is 0 for null, the value the programming language C uses to mark the end of a string.

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Example 2.9.1: ASCII versus binary numbers.

We could represent numbers as strings of ASCII digits instead of as integers. How much does storage increase if the number 1 billion is represented in ASCII versus a 32-bit integer?

Answer

One billion is 1,000,000,000, so it would take 10 ASCII digits, each 8 bits long. Thus the storage expansion would be (10 × 8)/32 or 2.5. Beyond the expansion in storage, the hardware to add, subtract, multiply, and divide such decimal numbers is difficult and would consume more energy. Such difficulties explain why computing professionals are raised to believe that binary is natural and that the occasional decimal computer is bizarre.

PARTICIPATION ACTIVITY 2.9.1: ASCII bit codes (and decimal number equivalents).

Type a character: A ASCII bit code: **1000001** ASCII number: **65**

PARTICIPATION ACTIVITY 2.9.2: ASCII.

1) What is the ASCII decimal value for a lower-case 'a'?

Check **Show answer**

2) What is the ASCII decimal value for an exclamation point?

Check **Show answer**

3) What two-letter word does this pair of decimal values represent in ASCII? Pay attention to upper/lower case. Use the above ASCII table.
72 105

Check **Show answer**

4) What two digits does this pair of decimal values represent in ASCII? Use the above ASCII table.
49 50

Check **Show answer**

A series of instructions can extract a byte from a word, so load word and store word are sufficient for transferring bytes as well as words. Because of the popularity of text in some programs, however, MIPS provides instructions to move bytes. *Load byte* (`lb`) loads a byte from memory, placing it in the rightmost 8 bits of a register. *Store byte* (`sb`) takes a byte from the rightmost 8 bits of a register and writes it to memory. Thus, we copy a byte with the sequence

```
lb $t0, 0($sp)    # Read byte from source
sb $t0, 0($gp)    # Write byte to destination
```


PARTICIPATION ACTIVITY 2.9.3: lb and sb.

Assume memory at the indicated addresses have the given decimal values:
2000: 65
2001: 66
2002: 67
2003: 68

1) If the memory values represent a string of characters in ASCII, what are those letters?
   - abcd
   - ABCD
   - 65666768

2) If $t1 holds 2000, what is $t2 (in decimal) after:
   `lb $t2, 0($t1)`
   - 2000
   - 65
   - Low byte is 65; rest unknown.

3) If $t1 holds 2000 and is increased by 1, what is $t2 (in decimal) after:
   `lb $t2, 0($t1)`
   - 65
   - 66
   - 67

4) If $t1 holds 2000, what is $t2 (in

decimal) after:

```
lw $t2, 0($t1)
```

- ○ 65
- ○ 65 + 66 + 67 + 68
- ○ A very large number.

5) Suppose $t1 holds 3000, and $t2 holds 97 (in decimal). Suppose writing to address 3000 prints a character to the screen. What is printed after the following instruction?

```
sb $t2, 0($t1)
```

- ○ 97
- ○ a
- ○ 3000

Characters are normally combined into strings, which have a variable number of characters. There are three choices for representing a string: (1) the first position of the string is reserved to give the length of a string, (2) an accompanying variable has the length of the string (as in a structure), or (3) the last position of a string is indicated by a character used to mark the end of a string. C uses the third choice, terminating a string with a byte whose value is 0 (named null in ASCII). Thus, the string "Cal" is represented in C by the following 4 bytes, shown as decimal numbers: 67, 97, 108, 0. (As we shall see, Java uses the first option.)

Example 2.9.2: Compiling a string copy procedure, showing how to use C strings.

The procedure `strcpy` copies string `y` to string `x` using the null byte termination convention of C:

```
void strcpy (char x[], char y[])
{
    int i;
    i = 0;
    while ((x[i] = y[i]) != '\0') /* copy & test byte */
        i += 1;
}
```


What is the MIPS assembly code?

Answer

Below is the basic MIPS assembly code segment. Assume that base addresses for arrays `x` and `y` are found in `$a0` and `$a1`, while `i` is in `$s0`. `strcpy` adjusts the stack pointer and then saves the saved register `$s0` on the stack:


```
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 more item
    sw   $s0, 0($sp)        # save $s0
```

To initialize `i` to 0, the next instruction sets `$s0` to 0 by adding 0 to 0 and placing that sum in `$s0`:

```
    add $s0, $zero, $zero   # i = 0 + 0
```

This is the beginning of the loop. The address of `y[i]` is first formed by adding `i` to `y[]`:

```
L1: add $t1, $s0, $a1       # address of y[i] in $t1
```

Note that we don't have to multiply `i` by 4 since `y` is an array of *bytes* and not of words, as in prior examples.

To load the character in `y[i]`, we use load byte unsigned, which puts the character into `$t2`:

```
    lbu $t2, 0($t1)         # $t2 = y[i]
```

A similar address calculation puts the address of `x[i]` in `$t3`, and then the character in `$t2` is stored at that address.

```
    add $t3, $s0, $a0       # address of x[i] in $t3
    sb  $t2, 0($t3)         # x[i] = y[i]
```

Next, we exit the loop if the character was 0. That is, we exit if it is the last character of the string:


```
    beq $t2, $zero, L2      # if y[i] == 0, go to L2
```

If not, we increment `i` and loop back:

```
    addi $s0, $s0, 1        # i = i + 1
    j    L1                 # go to L1
```

If we don't loop back, it was the last character of the string; we restore `$s0` and the stack pointer, and then return.

```
L2: lw   $s0, 0($sp)         # y[i] == 0: end of string.
                             # Restore old $s0
    addi $sp, $sp, 4         # pop 1 word off stack
    jr   $ra                 # return
```

String copies usually use pointers instead of arrays in C to avoid the operations on `i` in the code above. See COD Section 2.14 (Arrays versus Pointers) for an explanation of arrays versus pointers.

Since the procedure `strcpy` above is a leaf procedure, the compiler could allocate `i` to a temporary register and avoid saving and restoring `$s0`. Hence, instead of thinking of the `$t` registers as being just for temporaries, we can think of them as registers that the callee should use whenever convenient. When a compiler finds a leaf procedure, it exhausts all temporary registers before using registers it must save.

PARTICIPATION ACTIVITY 2.9.4: String copy example.

Consider the above example of a C string copy procedure.

1) In C, `(x[i] = y[i])` not only assigns x[i] with y[i], but also evaluates the value that was assigned.
   - ○ Yes
   - ○ No

2) In the C code, the comparison with '\0' (known as null) checks whether this character is the last character in the string.
   - ○ Yes
   - ○ No

3) In the MIPS assembly code, the first two instructions (addi, sw) are needed because `strcpy` is a procedure.
   - ○ Yes
   - ○ No

4) If $a1 holds the value 5000, what is the value loaded into $t1 the first time this statement is executed:
`L1: add $t1, $s0, $a1`

- ○ 5000
- ○ 5001

5) If $t1 holds 5000, would lw work the same as lbu in the assembly code?

- ○ Yes
- ○ No

6) lbu is used to load a value into array y, and again to load a value into array x.

- ○ Yes
- ○ No

7) The `j L1` instruction executes if y's current character is 0.

- ○ Yes
- ○ No

8) The instructions at L2 restore $s0's value, adjust the stack pointer, and return to the caller program.

- ○ Yes
- ○ No

Characters and strings in Java

Unicode is a universal encoding of the alphabets of most human languages. The figure below gives a list of Unicode alphabets; there are almost as many *alphabets* in Unicode as there are useful *symbols* in ASCII. To be more inclusive, Java uses Unicode for characters. By default, it uses 16 bits to represent a character.

Figure 2.9.2: Example alphabets in Unicode (COD Figure 2.16).

Unicode version 4.0 has more than 160 "blocks", which is their name for a collection of symbols. Each block is a multiple of 16. For example, Greek starts at 0370_{hex}, and Cyrillic

at 0400_{hex}. The first three columns show 48 blocks that correspond to human languages in roughly Unicode numerical order. The last column has 16 blocks that are multilingual and are not in order. A 16-bit encoding, called UTF-16, is the default. A variable-length encoding, called UTF-8, keeps the ASCII subset as eight bits and uses 16 or 32 bits for the other characters. UTF-32 uses 32 bits per character. New Unicode versions are released every June, with version 12.0 in 2019. Versions 9.0 to 12.0 added various Emojis, while earlier versions added new language blocks and hieroglyphs. To learn more, see www.unicode.org.

Latin	Malayalam	Tagbanwa	General Punctuation
Greek	Sinhala	Khmer	Spacing Modifier Letters
Cyrillic	Thai	Mongolian	Currency Symbols
Armenian	Lao	Limbu	Combining Diacritical Marks
Hebrew	Tibetan	Tai Le	Combining Marks for Symbols
Arabic	Myanmar	Kangxi Radicals	Superscripts and Subscripts
Syriac	Georgian	Hiragana	Number Forms
Thaana	Hangul Jamo	Katakana	Mathematical Operators
Devanagari	Ethiopic	Bopomofo	Mathematical Alphanumeric Symbols
Bengali	Cherokee	Kanbun	Braille Patterns
Gurmukhi	Unified Canadian Aboriginal Syllabic	Shavian	Optical Character Recognition
Gujarati	Ogham	Osmanya	Byzantine Musical Symbols
Oriya	Runic	Cypriot Syllabary	Musical Symbols
Tamil	Tagalog	Tai Xuan Jing Symbols	Arrows
Telugu	Hanunoo	Yijing Hexagram Symbols	Box Drawing
Kannada	Buhid	Aegean Numbers	Geometric Shapes

The MIPS instruction set has explicit instructions to load and store such 16- bit quantities, called *halfwords*. *Load half* (`lh`) loads a halfword from memory, placing it in the rightmost 16 bits of a register. Like load byte, *load half* (`lh`) treats the halfword as a signed number and thus sign-extends to fill the 16 leftmost bits of the register, while *load halfword unsigned* (`lhu`) works with unsigned integers. Thus, `lhu` is the more popular of the two. *Store half* (`sh`) takes a halfword from the rightmost 16 bits of a register and writes it to memory. We copy a halfword with the sequence


```
lhu $t0, 0($sp)    # Read halfword (16 bits) from source
sh  $t0, 0($gp)    # Write halfword (16 bits) to destination
```

Strings are a standard Java class with special built-in support and predefined methods for concatenation, comparison, and conversion. Unlike C, Java includes a word that gives the length of the string, similar to Java arrays.

Elaboration

MIPS software tries to keep the stack aligned to word addresses, allowing the program to always use `lw` and `sw` (which must be aligned) to access the stack. This convention means that a `char` variable allocated on the stack occupies 4 bytes, even though it needs less. However, a C string variable or an array of bytes will pack 4 bytes per word, and a Java string variable or array of shorts packs 2 halfwords per word.

Elaboration

Reflecting the international nature of the web, most web pages today use Unicode instead of ASCII.

PARTICIPATION ACTIVITY 2.9.5: Check yourself: Character and string representation.

1) A string is a single-dimension array of characters in C and Java.
   - ○ True
   - ○ False

2) A string in C takes about half the memory as the same string in Java.
   - ○ True
   - ○ False

3) Strings in C and Java use null (0) to mark the end of a string.
   - ○ True
   - ○ False

4) Operations on strings, like length, are faster in C than in Java.
   - ○ True

○ False

PARTICIPATION ACTIVITY 2.9.6: Check yourself: Memory requirements.

1) Which type of variable that can contain $1{,}000{,}000{,}000_{ten}$ takes the most memory space?

- ○ int in C
- ○ string in C
- ○ string in Java

The Big Picture

Newcomers to computing are surprised that the type of the data is not encoded inside the data, but instead in the *program* that operates on that data.

To Illustrate, we give an example from natural languages. What does the word "won" mean? You cannot answer that question without knowing the context, specifically the language that it is supposed to be in. Here are four alternatives:

1. In English, it is the verb that is the past tense of win.
2. In Korean, it is a noun that is the monetary unit of South Korea.
3. In Polish, it is an adjective that means nice smelling.
4. In Russian, it is an adjective that means stinks.

A binary number can also represent several types of data. For example, the 32-bit pattern

```
01100010 01100001 01010000 00000000
```

could represent the following:

1. 1,650,544,640 if the program treats it as an unsigned integer.
2. +1,650,544,640 if the program treats it as a signed integer.
3. "baP" if the program treats it as a null-terminated ASCII string.
4. The color dark blue if the program treats the bit pattern as a mixture of the four base colors cyan, magenta, yellow, and black of the Pantone

color-matching system.

The Big Picture in COD Section 2.5 reminds us that instructions are also represented as numbers, so the bit pattern could represent the MIPS machine language instruction

```
011000 10011 00001 01010 00000 000000
```

Which corresponds to the assembly language instruction multiply (see Chapter 3):

```
mult, $t2, $s3, $at
```


If you accidentally give a word-processing program an image, it will try to interpret it as text, and you will see bizarre images on the screen; you will run into similar issues if you give text data to a graphics display program. This unrestricted behavior of a stored program computer is why file systems have a naming convention of the suffix giving the type of the file (for example, .jpg, .pdf, .txt) to enable the program to check for mismatches by file name to reduce such embarrassing scenarios.

2.10 MIPS addressing for 32-bit immediates and addresses

Although keeping all MIPS instructions 32 bits long simplifies the hardware, there are times where it would be convenient to have a 32-bit constant or 32-bit address. This section starts with the general solution for large constants, and then shows the optimizations for instruction addresses used in branches and jumps.

32-bit immediate operands

Although constants are frequently short and fit into the 16-bit field, sometimes they are bigger. The MIPS instruction set includes the instruction *load upper immediate* (`lui`) specifically to set the upper 16 bits of a constant in a register, allowing a subsequent instruction to specify the lower 16 bits of the constant. The figure below shows the operation of lui.

PARTICIPATION ACTIVITY 2.10.1: The lui instruction, and how to load a 32-bit constant (COD Figure 2.17 (The effect of the lui instruction)).

Animation content:

Static figure: The instruction "lui $t0, 255 # $t0 is register 8", and its binary representation "001111 00000 01000 0000 0000 1111 1111" with an arrow pointing towards the register $t0 which contains "0000 0000 1111 1111 0000 0000 0000 0000" showing that the immediate constant field value "0000 0000 1111 1111" is loaded into the upper 16 bits of $t0, and the lower 16 bits are filled with zeros. An example of how to load a 32-bit constant into the register $s0. The 32-bit constant "0000 0000 0111 1101 0000 1001 0000 0000" is shown. The upper 16 bits represent 61 in decimal and the lower 16 bits represent 2304 in decimal. The instruction "lui $s0, 61" sets the upper half of the $s0 register with the value 61, resulting with the register $s0 containing "0000 0000 0111 1101 0000 0000 0000 0000". The instruction "ori $s0, $s0, 2304" sets the lower half of the $s0 register with the value 2304, resulting with the register $s0 containing "0000 0000 0111 1101 0000 1001 0000 0000"

Step 1: The instruction lui transfers the 16-bit immediate constant field value into the leftmost 16 bits of the register, filling the lower 16 bits with 0s. The instruction "lui $t0, 255 # t0 is register 8" is executed. The binary representation of the instruction is "001111 00000 01000 0000 0000 1111 1111". The register $t0 now contains "0000 0000 1111 1111 0000 0000 0000 0000".

Step 2: How to load a given 32-bit constant into $s0? Start by loading the constant's upper 16 bits into the leftmost 16 bits of the $s0 register (representing 61 in decimal). The instruction "lui $s0, 61" and a 32-bit binary constant "0000 0000 0011 1101 0000 1001 0000 0000" is shown. This instruction sets the upper half of the $s0 register with the constant's upper 16 bits, representing the decimal value 61, and the resulting register $s0 contains "0000 0000 0011 1101 0000 0000

0000 0000".

Step 3: The next step inserts the lower 16 bits (representing 2304 in decimal) using instruction ori.
The instruction "ori $s0, $s0, 2304" sets the lower half of the $s0 register with the constant's lower 16 bits, representing the decimal value 2304, and the resulting register $s0 contains "0000 0000 0011 1101 0000 1001 0000 0000".

Animation captions:

1. The instruction lui transfers the 16-bit immediate constant field value into the leftmost 16 bits of the register, filling the lower 16 bits with 0s.
2. How to load a given 32-bit constant into $s0? Start by loading the constant's upper 16 bits into the leftmost 16 bits of the $s0 register (representing 61 in decimal).
3. The next step inserts the lower 16 bits (representing 2304 in decimal) using instruction ori.

Hardware/Software Interface

Either the compiler or the assembler must break large constants into pieces and then reassemble them into a register. As you might expect, the immediate field's size restriction may be a problem for memory addresses in loads and stores as well as for constants in immediate instructions. If this job falls to the assembler, as it does for MIPS software, then the assembler must have a temporary register available in which to create the long values. This need is a reason for the register `$at` (assembler temporary), which is reserved for the assembler.

Hence, the symbolic representation of the MIPS machine language is no longer limited by the hardware, but by whatever the creator of an assembler chooses to include. We stick close to the hardware to explain the architecture of the computer, noting when we use the enhanced language of the assembler that is not found in the processor.

Elaboration

Creating 32-bit constants needs care. The instruction `addi` *copies the left-most bit of the 16-bit immediate field of the instruction into the upper 16 bits of a word. The*

instruction `ori` (Logical or immediate, from COD Section 2.6 (Logical operations)) loads 0s into the upper 16 bits and hence is used by the assembler in conjunction with `lui` to create 32-bit constants.

PARTICIPATION ACTIVITY 2.10.2: 32-bit immediate operands.

1) Given: `lui $s0, 7`. How is the immediate operand 7 represented in the instruction?
   - 0000 0000 0000 0111
   - 0000 0000 0000 0000 0000 0000 0111

2) What is $s0 after:
```
lui $s0, 7
```
   - 0000 0000 0000 0000 0000 0000 0111
   - 0000 0000 0000 0000 0000 0111 0000
   - 0000 0000 0000 0111 0000 0000 0000 0000

3) What is $s0 after:
```
lui $s0, 7
ori $s0, $s0, 8
```
   - 0000 0000 0000 0000 0000 0000 1111
   - 0000 0000 0000 1111 0000 0000 0000 0000
   - 0000 0000 0000 0111 0000 0000 0000 1000

©zyBooks 05/16/25 23:10 2475274
Jaheim Attri
FIUEEL4709CSpring2025

Addressing in branches and jumps

The MIPS jump instructions have the simplest addressing. They use the final MIPS instruction format, called the *J-type*, which consists of 6 bits for the operation field and the rest of the bits for

the address field. Thus,

```
j 10000   # go to location 10000
```

could be assembled into this format (it's actually a bit more complicated, as we will see):

where the value of the jump opcode is 2 and the jump address is `10000`.

Unlike the jump instruction, the conditional branch instruction must specify two operands in addition to the branch address. Thus,

```
bne $s0, $s1, Exit   # go to Exit if $s0 ≠ $s1
```

is assembled into this instruction, leaving only 16 bits for the branch address:

If addresses of the program had to fit in this 16-bit field, it would mean that no program could be bigger than 2^{16}, which is far too small to be a realistic option today. An alternative would be to specify a register that would always be added to the branch address, so that a branch instruction would calculate the following:

Program counter = Register + Branch address

This sum allows the program to be as large as 2^{32} and still be able to use conditional branches, solving the branch address size problem. Then the question is, which register?

The answer comes from seeing how conditional branches are used. Conditional branches are found in loops and in *if* statements, so they tend to branch to a nearby instruction. For example, about half of all conditional branches in SPEC benchmarks go to locations less than 16 instructions away. Since the *program counter* (PC) contains the address of the current instruction, we can branch within $\pm 2^{15}$ words of the current instruction if we use the PC as the register to be added to the address. Almost all loops and *if* statements are much smaller than 2^{16} words, so the PC is the ideal choice.

This form of branch addressing is called *PC-relative addressing*. As we shall see in COD Chapter 4 (The Processor), it is convenient for the hardware to increment the PC early to point to the next instruction. Hence, the MIPS address is actually relative to the address of the following instruction (PC + 4) as opposed to the current instruction (PC). It is yet another example of making the **common case fast**, which in this case is addressing nearby instructions.

PC-relative addressing: An addressing regime in which the address is the sum of the *program counter* (PC) and a constant in the instruction.

Like most recent computers, MIPS uses PC-relative addressing for all conditional branches, because the destination of these instructions is likely to be close to the branch. On the other hand, jump-and-link instructions invoke procedures that have no reason to be near the call, so they normally use other forms of addressing. Hence, the MIPS architecture offers long addresses for procedure calls by using the J-type format for both jump and jump-and-link instructions.

Since all MIPS instructions are 4 bytes long, MIPS stretches the distance of the branch by having PC-relative addressing refer to the number of *words* to the next instruction instead of the number of bytes. Thus, the 16-bit field can branch four times as far by interpreting the field as a relative word address rather than as a relative byte address. Similarly, the 26-bit field in jump instructions is also a word address, meaning that it represents a 28-bit byte address.

Elaboration

Since the PC is 32 bits, 4 bits must come from somewhere else for jumps. The MIPS jump instruction replaces only the lower 28 bits of the PC, leaving the upper 4 bits of the PC unchanged. The loader and linker must be careful to avoid placing a program across an address boundary of 256 MB (64 million instructions); otherwise, a jump must be replaced by a jump register instruction preceded by other instructions to load the full 32-bit address into a register.

Example 2.10.1: Showing branch offset in machine language.

The *while* loop was compiled into this MIPS assembler code:

```
Loop: sll  $t1, $s3, 2      # Temp reg $t1 = 4 * i
      add  $t1, $t1, $s6    # $t1 = address of save[i]
      lw   $t0, 0($t1)      # Temp reg $t0 = save[i]
      bne  $t0, $s5, Exit   # go to Exit if save[i] ≠ k
      addi $s3, $s3, 1      # i = i + 1
      j    Loop             # go to Loop
Exit:
```


If we assume we place the loop starting at location 80000 in memory, what is the MIPS machine code for this loop?

Answer

The assembled instructions and their addresses are:

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

Remember that MIPS instructions have byte addresses, so addresses of sequential words differ by 4, the number of bytes in a word. The `bne` instruction on the fourth line adds 2 words or 8 bytes to the address of the *following* instruction (80016), specifying the branch destination relative to that following instruction (8 + 80016) instead of relative to the branch instruction (12 + 80012) or using the full destination address (80024). The jump instruction on the last line does use the full address (20000 × 4 = 80000), corresponding to the label `Loop`.

Hardware/Software Interface

Most conditional branches are to a nearby location, but occasionally they branch far away, farther than can be represented in the 16 bits of the conditional branch instruction. The assembler comes to the rescue just as it did with large addresses or constants: it inserts an unconditional jump to the branch target, and inverts the condition so that the branch decides whether to skip the jump.

Example 2.10.2: Branching far away.

Given a branch on register `$s0` being equal to register `$s1`,


```
beq $s0, $s1, L1
```

replace it by a pair of instructions that offers a much greater branching distance.

Answer

These instructions replace the short-address conditional branch:

```
      bne $s0, $s1, L2
      j   L1
L2:
```


MIPS addressing mode summary

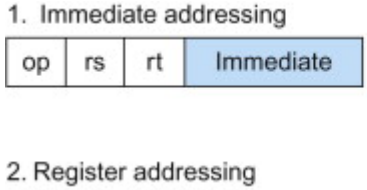
Multiple forms of addressing are generically called *addressing modes*. The figure below shows how operands are identified for each addressing mode. The MIPS addressing modes are the following:

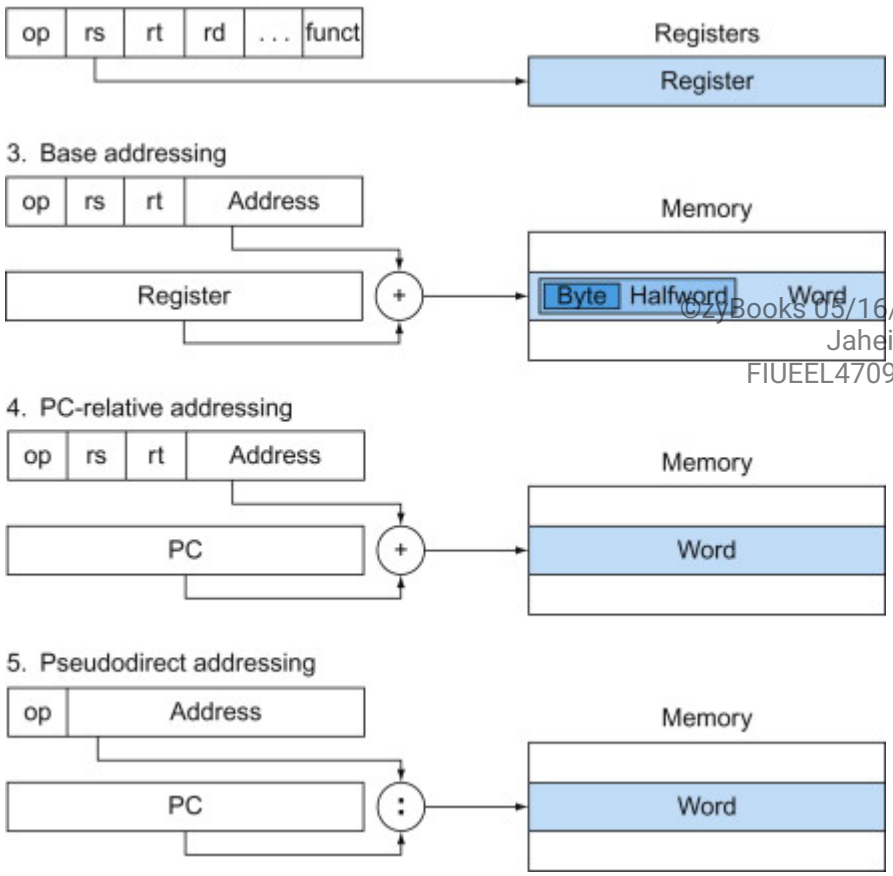
1. ***Immediate addressing***: The operand is a constant within the instruction itself
2. ***Register addressing***: The operand is a register
3. ***Base addressing / displacement addressing***: The operand is at the memory location whose address is the sum of a register and a constant in the instruction
4. ***PC-relative addressing***: The branch address is the sum of the PC and a constant in the instruction
5. ***Pseudodirect addressing***: The jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

Addressing mode: One of several addressing regimes delimited by their varied use of operands and/or addresses.

Figure 2.10.1: Illustration of the five MIPS addressing modes (COD Figure 2.18).

The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC. Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (`addi`) and register (`add`) addressing.





Hardware/Software Interface

Although we show MIPS as having 32-bit addresses, nearly all microprocessors (including MIPS) have 64-bit address extensions (see COD Appendix E (A Survey of RISC Architectures for Desktop, Server, and Embedded Computers)). These extensions were in response to the needs of software for larger programs. The process of instruction set extension allows architectures to expand in such a way that is able to move software compatibly upward to the next generation of architecture.

PARTICIPATION ACTIVITY 2.10.3: Addressing modes.

How to use this tool

Immediate addressing **PC-relative addressing** **Base or displacement addressing**

Register addressing **Pseudodirect addressing**

The operand is a constant within the instruction itself.

The operand is a register.

The operand is at the memory location whose address is the sum of a register and a constant in the instruction.

The branch address is the sum of the PC and a constant in the instruction.

The jump address is the 26 bits of the instruction concatenated with the upper bits of the PC.

Reset

Decoding machine language

Sometimes you are forced to reverse-engineer machine language to create the original assembly language. One example is when looking at "core dump". The figure below shows the MIPS encoding of the fields for the MIPS machine language. This figure helps when translating by hand between assembly language and machine language.

Figure 2.10.2: MIPS instruction encoding (COD Figure 2.19).

This notation gives the value of a field by row and by column. For example, the top portion of the figure shows **load word** in row number 4 (100_{two} for bits 31—29 of the instruction) and column number 3 (011_{two} for bits 28—26 of the instruction), so the corresponding value of the op field (bits 31—26) is 100011_{two}. Underscore means the field is used elsewhere. For example, `R-format` in row 0 and column 0 (op = 000000_{two}) is defined in the bottom part of the figure. Hence, `subtract` in row 4 and column 2 of the bottom

section means that the funct field (bits 5—0) of the instruction is 100010_{two} and the op field (bits 31—26) is 000000_{two}. The `floating point` value in row 2, column 1 is defined in in COD Chapter 3 (Arithmetic for Computers). `Bltz/gez` is the opcode for four instructions found in COD Appendix A (Assemblers, Linkers, and the SPIM Simulator): `bltz, bgez, bltzal,` and `bgezal`. This chapter describes instructions given in full name using color, while COD Chapter 3 (Arithmetic for Computers) describes instructions given in mnemonics using color. Appendix A (Assemblers, Linkers, and the SPIM Simulator) covers all instructions.

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

op(31:26)=010000 (TLB), rs(25:21)								
23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

Example 2.10.3: Decoding machine code.

What is the assembly language statement corresponding to this machine instruction?

```
00af8020hex
```

Answer

The first step in converting hexadecimal to binary is to find the op fields:

```
(Bits: 31 28 26                          5   2 0)
       0000 0000 1010 1111 1000 0000 0010 0000
```

We look at the op field to determine the operation. Referring to figure above (MIPS instruction encoding), when bits 31—29 are 000 and bits 28—26 are 000, it is an R-format instruction. Let's reformat the binary instruction into R-format fields, listed in the figure below (MIPS instruction formats):

```
op        rs       rt       rd       shamt    funct
000000    00101    01111    10000    00000    100000
```

The bottom portion of the figure above (MIPS instruction encoding) determines the operation of an R-format instruction. In this case, bits 5—3 are 100 and bits 2—0 are 000, which means this binary pattern represents an `add` instruction.

We decode the rest of the instruction by looking at the field values. The decimal values are 5 for the rs field, 15 for rt, and 16 for rd (shamt is unused). COD Figure 2.14 (MIPS register conventions) shows that these numbers represent registers `$a1`, `$t7`, and `$s0`. Now we can reveal the assembly instruction:

```
add $s0, $a1, $t7
```

Figure 2.10.3: MIPS instruction formats (COD Figure 2.20).

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

The figure above shows all the MIPS instruction formats. COD Figure 2.1 (MIPS assembly language revealed in this chapter) shows the MIPS assembly language revealed in this chapter. The remaining hidden portion of MIPS instructions deals mainly with arithmetic and real numbers, which are covered in the next chapter.

PARTICIPATION ACTIVITY 2.10.4: Check yourself: 32-bit immediates and addresses.

1) What is the range of addresses for conditional branches in MIPS (K = 1024)?

- Addresses between 0 and 256K - 1
- Addresses up to about 32K before the branch to about 32K after
- Addresses up to about 128K before the branch to about 128K after

2) What is the range of addresses for jump and jump and link in MIPS (M = 1024K)?

- Addresses between 0 and 64M - 1
- Addresses up to about 128M before the branch to about 128M after
- Anywhere within a block of 256M addresses where the PC supplies the upper 4 bits

3) What is the MIPS assembly language instruction corresponding to the machine instruction with the value $0000\ 0000_{hex}$?

- j
- sll

Undefined opcode: there is no legal instruction that corresponds to 000000

2.11 Parallelism and instructions: synchronization

Parallel execution is easier when tasks are independent, but often they need to cooperate. Cooperation usually means some tasks are writing new values that others must read. To know when a task is finished writing so that it is safe for another to read, the tasks need to synchronize. If they don't synchronize, there is a danger of a *data race*, where the results of the program can change depending on how events happen to occur.

Data race: Two memory accesses form a data race if they are from different threads to same location, at least one is a write, and they occur one after another.

For example, recall the analogy of the eight reporters writing in COD Chapter 1 (Computer Abstractions and Technology). Suppose one reporter needs to read all the prior sections before writing a conclusion. Hence, he or she must know when the other reporters have finished their sections, so that there is no danger of sections being changed afterwards. That is, they had better synchronize the writing and reading of each section so that the conclusion will be consistent with what is printed in the prior sections.

In computing, synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions. In this section, we focus on the implementation of *lock* and *unlock* synchronization operations. Lock and unlock can be used straightforwardly to create regions where only a single processor can operate, called a *mutual exclusion*, as well as to implement more complex synchronization mechanisms.

The critical ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to *atomically* read and modify a memory location. That is, nothing else can interpose itself between the read and the write of the memory location. Without such a capability, the cost of building basic synchronization primitives will be high and will increase unreasonably as the processor count increases.

There are a number of alternative formulations of the basic hardware primitives, all of which provide the ability to atomically read and modify a location, together with some way to tell if the read and write were performed atomically. In general, architects do not expect users to employ the basic hardware primitives, but instead expect that the primitives will be used by system programmers to build a synchronization library, a process that is often complex and tricky.

Let's start with one such hardware primitive and show how it can be used to build a basic synchronization primitive. One typical operation for building synchronization operations is the *atomic exchange* or *atomic swap*, which interchanges a value in a register for a value in memory.

To see how to use this to build a basic synchronization primitive, assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and 1 is used to indicate that the lock is unavailable. A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock. The value returned from the exchange instruction is 1 (unavailable) if some other processor had already claimed access, and 0 (free) otherwise. In the latter case, the value is also changed to 1 (unavailable), preventing any competing exchange in another processor from also retrieving a 0 (free).

For example, consider two processors that each try to do the exchange simultaneously: this race is broken, since exactly one of the processors will perform the exchange first, returning 0 (free), and the second processor will return 1 (unavailable) when it does the exchange. The key to using the exchange primitive to implement synchronization is that the operation is atomic: the exchange is indivisible, and two simultaneous exchanges will be ordered by the hardware. It is impossible for two processors trying to set the synchronization variable in this manner to both think they have simultaneously set the variable.

Implementing a single atomic memory operation introduces some challenges in the design of the processor, since it requires both a memory read and a write in a single, uninterruptible instruction.

An alternative is to have a pair of instructions in which the second instruction returns a value showing whether the pair of instructions was executed as if the pair were atomic. The pair of instructions is effectively atomic if it appears as if all other operations executed by any processor occurred before or after the pair. Thus, when an instruction pair is effectively atomic, no other processor can change the value between the instruction pair.

In MIPS this pair of instructions includes a special load called a *load linked* and a special store called a *store conditional*. These instructions are used in sequence: if the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails. The store conditional is defined to both store the value of a (presumably different) register in memory *and* to change the value of that register to a 1 if it succeeds and to a 0 if it fails. Since the load linked returns the initial value, and the store conditional returns 1 only if it succeeds, the following sequence implements an atomic exchange on the memory location specified by the contents of `$s1`:


```
again: add  $t0, $zero, $s4    #copy locked value
       ll   $t1, 0($s1)        #load linked
       sc   $t0, 0($s1)        #store conditional
       beq  $t0, $zero, again  #branch if store fails
       add  $s4, $zero, $t1    #put load value in $s4
```

Any time a processor intervenes and modifies the value in memory between the `ll` and `sc`

instructions, the `sc` returns 0 in `$t0`, causing the code sequence to try again. At the end of this sequence the contents of `$s4` and the memory location specified by `$s1` have been atomically exchanged.

Elaboration

Although it was presented for multiprocessor synchronization, atomic exchange is also useful for the operating system in dealing with multiple processes in a single processor. To make sure nothing interferes in a single processor, the store conditional also fails if the processor does a context switch between the two instructions (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)).

Elaboration

An advantage of the load linked/store conditional mechanism is that it can be used to build other synchronization primitives, such as atomic compare and swap or atomic fetch-and-increment, which are used in some parallel programming models. These involve more instructions between the `ll` and the `sc`, but not too many.

Since the store conditional will fail after either another attempted store to the load linked address or any exception, care must be taken in choosing which instructions are inserted between the two instructions. In particular, only register-register instructions can safely be permitted; otherwise, it is possible to create deadlock situations where the processor can never complete the `sc` because of repeated page faults. In addition, the number of instructions between the load linked and the store conditional should be small to minimize the probability that either an unrelated event or a competing processor causes the store conditional to fail frequently.

PARTICIPATION ACTIVITY 2.11.1: Check yourself: Synchronization.

When are primitives like load linked and store conditional used?

1) When in a parallel program the cooperating threads need synchronization to get proper

behavior for reading and writing shared data.

- Yes
- No

2) When on a uniprocessor the cooperating processes need synchronization to reading and writing shared data.

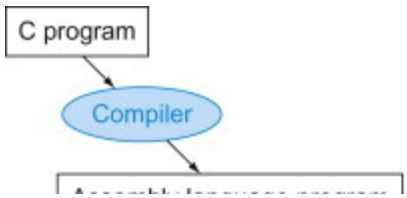
- Yes
- No

2.12 Translating and starting a program

This section describes the four steps in transforming a C program in nonvolatile storage into a program running on a computer. The following figure shows the translation hierarchy. Some systems combine these steps to reduce translation time, but these are the logical four phases that programs go through. This section follows this translation hierarchy.

A high-level language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine code into the proper memory locations for execution by the processor. To speed up the translation process, some steps are skipped or combined. Some compilers produce object modules directly, and some systems use linking loaders that perform the last two steps. To identify the type of file, UNIX follows a suffix convention for files: C source files are named `x.c`, assembly files are `x.s`, object files are named `x.o`, statically linked library routines are `x.a`, dynamically linked library routines are `x.so`, and executable files by default are called `a.out`. MS-DOS uses the suffixes `.C`, `.ASM`, `.OBJ`, `.LIB`, `.DLL`, and `.EXE` to the same effect.

PARTICIPATION ACTIVITY 2.12.1: A translation hierarchy for C (COD Figure 2.21).



Animation content:

Static figure: A flow diagram illustrating the steps of translating a C program into an executable in computer memory. Nodes represent the data and translation steps. An arrow indicates the flow from one node to the next. The nodes are labeled as follows:

1. C program
2. Compiler
3. Assembly language program
4. Assembler
5. Object: Machine language module
6. Object: Library routine (machine language)
7. Linker
8. Executable: Machine language program
9. Loader
10. Memory

Arrows show the progression from a C program through the steps of compilation, assembly, linking, and loading until the Executable: Machine language program is placed into Memory.

Step 1: A compiler converts a high-level language program, like a C program, to an assembly language program.

Step 2: An assembler converts an assembly language program to an object module in machine language.

Step 3: A linker combines multiple object modules, such as library routines, into an executable machine language program.

Step 4: Finally, a loader places the executable file into memory, to be run by the computer.

Animation captions:

1. A compiler converts a high-level language program, like a C program, to an assembly language program.
2. An assembler converts an assembly language program to an object module in machine language.
3. A linker combines multiple object modules, such as library routines, into an executable machine language program.
4. Finally, a loader places the executable file into memory, to be run by the computer.

Compiler

The compiler transforms the C program into an *assembly language program*, a symbolic form of what the machine understands. High-level language programs take many fewer lines of code than assembly language, so programmer productivity is much higher.

In 1975, many operating systems and assemblers were written in *assembly language* because memories were small and compilers were inefficient. The million-fold increase in memory capacity per single DRAM chip has reduced program size concerns, and optimizing compilers today can produce assembly language programs nearly as well as an assembly language expert, and sometimes even better for large programs.

Assembly language: A symbolic language that can be translated into binary machine language.

Assembler

Since assembly language is an interface to higher-level software, the assembler can also treat common variations of machine language instructions as if they were instructions in their own right. The hardware need not implement these instructions; however, their appearance in assembly language simplifies translation and programming. Such instructions are called *pseudoinstructions*.

Pseudoinstruction: A common variation of assembly language instructions often treated as if it were an instruction in its own right.

As mentioned above, the MIPS hardware makes sure that register `$zero` always has the value 0. That is, whenever register `$zero` is used, it supplies a 0, and the programmer cannot change the

value of register `$zero`. Register `$zero` is used to create the assembly language instruction that copies the contents of one register to another. Thus the MIPS assembler accepts this instruction even though it is not found in the MIPS architecture:

```
move $t0, $t1          # register $t0 gets register $t1
```

The assembler converts this assembly language instruction into the machine language equivalent of the following instruction:


```
add $t0, $zero, $t1   # register $t0 gets 0 + register $t1
```

The MIPS assembler also converts `blt` (branch on less than) into the two instructions `slt` and `bne`. Other examples include `bgt`, `bge`, and `ble`. It also converts branches to faraway locations into a branch and jump. As mentioned above, the MIPS assembler allows 32-bit constants to be loaded into a register despite the 16-bit limit of the immediate instructions.

In summary, pseudoinstructions give MIPS a richer set of assembly language instructions than those implemented by the hardware. The only cost is reserving one register, `$at`, for use by the assembler. If you are going to write assembly programs, use pseudoinstructions to simplify your task. To understand the MIPS architecture and be sure to get best performance, however, study the real MIPS instructions found in COD Figures 2.1 (MIPS assembly language revealed in this chapter) and 2.19 (MIPS instruction encoding).

Assemblers will also accept numbers in a variety of bases. In addition to binary and decimal, they usually accept a base that is more succinct than binary yet converts easily to a bit pattern. MIPS assemblers use hexadecimal.

Such features are convenient, but the primary task of an assembler is assembly into machine code. The assembler turns the assembly language program into an *object file*, which is a combination of machine language instructions, data, and information needed to place instructions properly in memory.

To produce the binary version of each instruction in the assembly language program, the assembler must determine the addresses corresponding to all labels. Assemblers keep track of labels used in branches and data transfer instructions in a *symbol table*. As you might expect, the table contains pairs of symbols and addresses.

The object file for UNIX systems typically contains six distinct pieces:

- The *object file header* describes the size and position of the other pieces of the object file.
- The *text segment* contains the machine language code.
- The *static data segment* contains data allocated for the life of the program. (UNIX allows programs to use both *static data*, which is allocated throughout the program, and *dynamic data*, which can grow or shrink as needed by the program.)
- The *relocation information* identifies instructions and data words that depend on absolute addresses when the program is loaded into memory.

- The *symbol table* contains the remaining labels that are not defined, such as external references.
- The *debugging information* contains a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable.

The next subsection shows how to attach such routines that have already been assembled, such as library routines.

Symbol table: A table that matches names of labels to the addresses of the memory words that instructions occupy.

PARTICIPATION ACTIVITY 2.12.2: Compiler and assembler.

1) A compiler converts a high-level program, such as a C program, into an assembly language program.
   - ○ True
   - ○ False

2) An assembly language program is a series of 0s and 1s that a machine understands.
   - ○ True
   - ○ False

3) An assembler is used to convert an assembly language program to a machine language program.
   - ○ True
   - ○ False

4) An instruction at address 985 has a label SUM. The assembler might put the following in a symbol table:
   SUM: 985
   - ○ True
   - ○ False

5) An instruction jumps to label SUM. If

a symbol table has SUM's address as 985, then the jump will be to address 985.

- ○ True
- ○ False

6) An object file is output by a compiler, and is a combination of machine language instructions, data, and information needed to place the instructions properly in memory.

- ○ True
- ○ False

Linker

What we have presented so far suggests that a single change to one line of one procedure requires compiling and assembling the whole program. Complete retranslation is a terrible waste of computing resources. This repetition is particularly wasteful for standard library routines, because programmers would be compiling and assembling routines that by definition almost never change. An alternative is to compile and assemble each procedure independently, so that a change to one line would require compiling and assembling only one procedure. This alternative requires a new systems program, called a *link editor* or *linker*, which takes all the independently assembled machine language programs and "stitches" them together.

Linker: Also called ***link editor***. A systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.

There are three steps for the linker:

1. Place code and data modules symbolically in memory.
2. Determine the addresses of data and instruction labels.
3. Patch both the internal and external references.

The linker uses the relocation information and symbol table in each object module to resolve all undefined labels. Such references occur in branch instructions , jump instructions, and data addresses, so the job of this program is much like that of an editor: it finds the old addresses and replaces them with the new addresses. Editing is the origin of the name "link editor," or linker for short. The reason a linker is useful is that it is much faster to patch code than it is to recompile and reassemble.

If all external references are resolved, the linker next determines the memory locations each

module will occupy. Recall that COD Figure 2.13 (The MIPS memory allocation for program and data) shows the MIPS convention for allocation of program and data to memory. Since the files were assembled in isolation, the assembler could not know where a module's instructions and data would be placed relative to other modules. When the linker places a module in memory, all *absolute* references--that is, memory addresses that are not relative to a register--must be *relocated* to reflect its true location.

The linker produces an *executable file* that can be run on a computer. Typically, this file has the same format as an object file, except that it contains no unresolved references. It is possible to have partially linked files, such as library routines, that still have unresolved addresses and hence result in object files.

Executable file: A functional program in the format of an object file that contains no unresolved references. It can contain symbol tables and debugging information. A "stripped executable" does not contain that information. Relocation information may be included for the loader.

Example 2.12.1: Linking object files.

Link the two object files below. Show updated addresses of the first few instructions of the completed executable file. We show the instructions in assembly language just to make the example understandable; in reality, the instructions would be numbers.

Note that in the object files we have highlighted the addresses and symbols that must be updated in the link process: the instructions that refer to the addresses of procedures `A` and `B` and the instructions that refer to the addresses of data `X` and `Y`.

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	
Object file header			
	Name	Procedure B	

	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Answer

Procedure `A` needs to find the address for the variable labeled `X` to put in the load instruction and to find the address of procedure `B` to place in the `jal` instruction. Procedure `B` needs the address of the variable labeled `Y` for the store instruction and the address of procedure `A` for its `jal` instruction.

The text segment starts at address $40\ 0000_{hex}$ and the data segment at $1000\ 0000_{hex}$. The text of procedure `A` is placed at the first address and its data at the second. The object file header for procedure `A` says that its text is 100_{hex} bytes and its data is 20_{hex} bytes, so the starting address for procedure `B` text is $40\ 0100_{hex}$, and its data starts at $1000\ 0020_{hex}$.

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	`lw $a0,` 8000_{hex}`($gp)`
	$0040\ 0004_{hex}$	`jal` $40\ 0100_{hex}$
	...	...
	$0040\ 0100_{hex}$	`sw $a1,` 8020_{hex}`($gp)`
	$0040\ 0104_{hex}$	`jal` $40\ 0000_{hex}$
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

Now the linker updates the address fields of the instructions. It uses the instruction type field to know the format of the address to be edited. We have two types here:

1. The `jal`s are easy because they use pseudodirect addressing. The `jal` at address `40 0004`$_{hex}$ gets `40 0100`$_{hex}$ (the address of procedure `B`) in its address field, and the `jal` at `40 0104`$_{hex}$ gets `40 0000`$_{hex}$ (the address of procedure `A`) in its address field.
2. The load and store addresses are harder because they are relative to a base register. This example uses the global pointer as the base register. COD Figure 2.13 (The MIPS memory allocation for program and data) shows that `$gp` is initialized to `1000 8000`$_{hex}$. To get the address `1000 0000`$_{hex}$ (the address of word `X`), we place `8000`$_{hex}$ in the address field of `lw` at address `40 0000`$_{hex}$. The address field of `lw` is sign extended, so `8000`$_{hex}$ becomes `FFFF 8000`$_{hex}$, or `-32768`. Similarly, we place `8020`$_{hex}$ in the address field of `sw` at address `40 0100`$_{hex}$ to get the address `1000 0020`$_{hex}$ (the address of word `Y`).

Elaboration

Recall that MIPS instructions are word aligned, so `jal` drops the right two bits to increase the instruction's address range. Thus, it uses 26 bits to create a 28-bit byte address. Hence, the actual address in the lower 26 bits of the `jal` instruction in this example is `10 0040`$_{hex}$, rather than `40 0100`$_{hex}$.

Loader

Now that the executable file is on disk, the operating system reads it to memory and starts it. The *loader* follows these steps in UNIX systems:

1. Reads the executable file header to determine size of the text and data segments.
2. Creates an address space large enough for the text and data.
3. Copies the instructions and data from the executable file into memory.
4. Copies the parameters (if any) to the main program onto the stack.
5. Initializes the machine registers and sets the stack pointer to the first free location.
6. Jumps to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine

terminates the program with an `exit` system call.

Sections COD A.3 (Linkers) and COD A.4 (Loading) in COD Appendix A (Assemblers, Linkers, and the SPIM Simulator) describe linkers and loaders in more detail.

Loader: A systems program that places an object program in main memory so that it is ready to execute.

PARTICIPATION ACTIVITY 2.12.3: Linkers and loaders.

1) A linker combines independently assembled machine language programs and then recompiles the programs.
   - ○ True
   - ○ False

2) An executable file is a program that can be run on the computer and has no unresolved references.
   - ○ True
   - ○ False

3) An executable file is executed after a loader places the file into main memory.
   - ○ True
   - ○ False

> "Virtually every problem in computer science can be solved by another level of indirection."
> *David Wheeler*

Dynamically linked libraries

The first part of this section describes the traditional approach to linking libraries before the program is run. Although this static approach is the fastest way to call library routines, it has a few disadvantages:

- The library routines become part of the executable code. If a new version of the library is released that fixes bugs or supports new hardware devices, the statically linked program keeps using the old version.
- It loads all routines in the library that are called anywhere in the executable, even if those calls are not executed. The library can be large relative to the program; for example, the standard C library is 2.5 MB.

These disadvantages lead to *dynamically linked libraries (DLLs)*, where the library routines are not linked and loaded until the program is run. Both the program and library routines keep extra information on the location of nonlocal procedures and their names. In the initial version of DLLs, the loader ran a dynamic linker, using the extra information in the file to find the appropriate libraries and to update all external references.

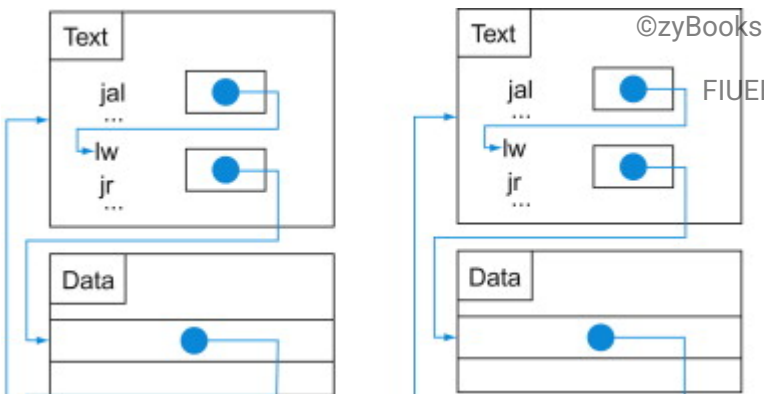
Dynamically linked libraries (***DLLs***): Library routines that are linked to a program during execution.

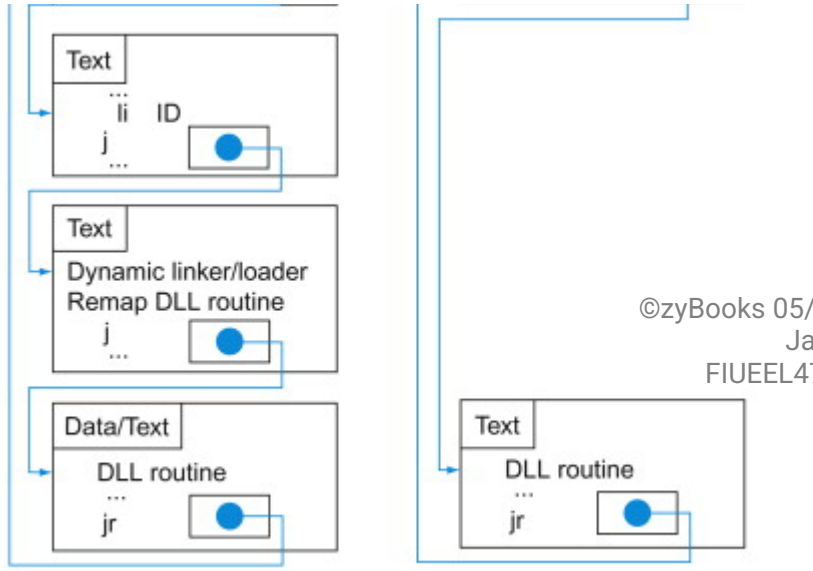
The downside of the initial version of DLLs was that it still linked all routines of the library that might be called, versus only those that are called during the running of the program. This observation led to the lazy procedure linkage version of DLLs, where each routine is linked only *after* it is called.

Like many innovations in our field, this trick relies on a level of indirection. The figure below shows the technique. It starts with the nonlocal routines calling a set of dummy routines at the end of the program, with one entry per nonlocal routine. These dummy entries each contain an indirect jump.

Figure 2.12.1: Dynamically linked library via lazy procedure linkage (COD Figure 2.22).

(a) Steps for the first time a call is made to the DLL routine. (b) The steps to find the routine, remap it, and link it are skipped on subsequent calls. As we will see in COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy), the operating system may avoid copying the desired routine by remapping it using virtual memory management.





(a) First call to DLL routine (b) Subsequent calls to DLL routine

The first time the library routine is called, the program calls the dummy entry and follows the indirect jump. It points to code that puts a number in a register to identify the desired library routine and then jumps to the dynamic linker/loader. The linker/loader finds the desired routine, remaps it, and changes the address in the indirect jump location to point to that routine. It then jumps to it. When the routine completes, it returns to the original calling site. Thereafter, the call to the library routine jumps indirectly to the routine without the extra hops.

In summary, DLLs require extra space for the information needed for dynamic linking, but do not require that whole libraries be copied or linked. They pay a good deal of overhead the first time a routine is called, but only a single indirect jump thereafter. Note that the return from the library pays no extra overhead. Microsoft's Windows relies extensively on dynamically linked libraries, and it is also the default when executing programs on UNIX systems today.

PARTICIPATION ACTIVITY 2.12.4: Dynamically linked libraries.

1) A compiler and assembler using dynamically linked libraries (DLL) create a complete object program that can be loaded and run.
   - ○ True
   - ○ False

2) A DLL approach results in a larger object program.
   - ○ True
   - ○ False

3) A DLL approach better supports library updates.

- True
- False

4) A DLL approach results in faster-running programs.

- True
- False

5) In a lazy procedure linkage approach to DLL's, each routine is only loaded if a call to the routine exists in the object program.

- True
- False

Starting a Java program

The discussion above captures the traditional model of executing a program, where the emphasis is on fast execution time for a program targeted to a specific instruction set architecture, or even a specific implementation of that architecture. Indeed, it is possible to execute Java programs just like C. Java was invented with a different set of goals, however. One was to run safely on any computer, even if it might slow execution time.

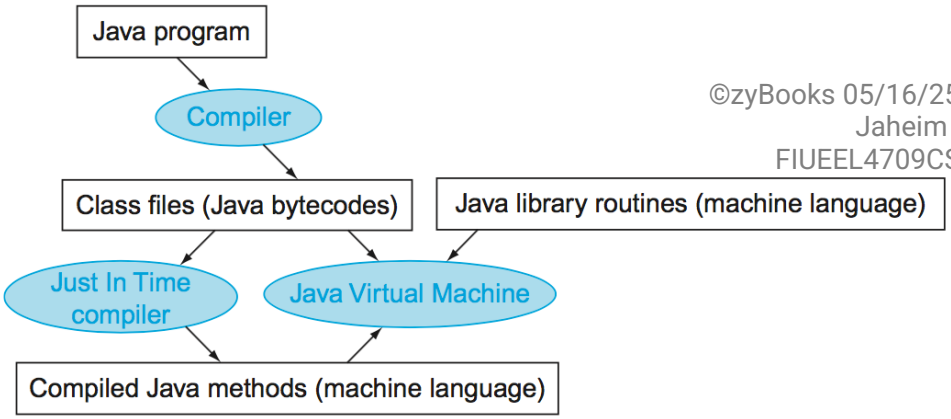
The figure below shows the typical translation and execution steps for Java. Rather than compile to the assembly language of a target computer, Java is compiled first to instructions that are easy to interpret: the *Java bytecode* instruction set. This instruction set is designed to be close to the Java language so that this compilation step is trivial. Virtually no optimizations are performed. Like the C compiler, the Java compiler checks the types of data and produces the proper operation for each type. Java programs are distributed in the binary version of these bytecodes.

Java bytecode: Instruction from an instruction set designed to interpret Java programs.

Figure 2.12.2: A translation hierarchy for Java (COD Figure 2.23).

A Java program is first compiled into a binary version of Java bytecodes, with all addresses defined by the compiler. The Java program is now ready to run on the interpreter, called the *Java Virtual Machine* (JVM). The JVM links to desired methods in the

Java library while the program is running. To achieve greater performance, the JVM can invoke the JIT compiler, which selectively compiles methods into the native machine language of the machine on which it is running.



A software interpreter, called a *Java Virtual Machine* (JVM), can execute Java bytecodes. An interpreter is a program that simulates an instruction set architecture. For example, the MIPS simulator used with this book is an interpreter. There is no need for a separate assembly step since either the translation is so simple that the compiler fills in the addresses or JVM finds them at runtime.

Java Virtual Machine (***JVM***): The program that interprets Java bytecodes.

The upside of interpretation is portability. The availability of software Java virtual machines meant that most people could write and run Java programs shortly after Java was announced. Today, Java virtual machines are found in hundreds of millions of devices, in everything from cell phones to Internet browsers.

The downside of interpretation is lower performance. The incredible advances in performance of the 1980s and 1990s made interpretation viable for many important applications, but the factor of 10 slowdown when compared to traditionally compiled C programs made Java unattractive for some applications.

To preserve portability and improve execution speed, the next phase of Java development was compilers that translated *while* the program was running. Such *Just In Time compilers* (*JIT*) typically profile the running program to find where the "hot" methods are and then compile them into the native instruction set on which the virtual machine is running. The compiled portion is saved for the next time the program is run, so that it can run faster each time it is run. This balance of interpretation and compilation evolves over time, so that frequently run Java programs suffer little of the overhead of interpretation.

Just In Time compiler (***JIT***) : The name commonly given to a compiler that operates at runtime, translating the interpreted code segments into the native code of the computer.

As computers get faster so that compilers can do more, and as researchers invent better ways to compile Java on the fly, the performance gap between Java and C or C++ is closing. COD Section 2.15 (Advanced Material: Compiling C and Interpreting Java) goes into much greater depth on the implementation of Java, Java bytecodes, JVM, and JIT compilers.

PARTICIPATION ACTIVITY 2.12.5: Check yourself: Advantages of an interpreter over a translator.

1) Which of the advantages of an interpreter over a translator was likely the most important for the designers of Java?
   - ○ Ease of writing an interpreter
   - ○ Higher performance
   - ○ Smaller object code
   - ○ Machine independence

2.13 A C sort example to put it all together

One danger of showing assembly language code in snippets is that you will have no idea what a full assembly language program looks like. In this section, we derive the MIPS code from two procedures written in C: one to swap array elements and one to sort them.

Figure 2.13.1: A C procedure that swaps two locations in memory (COD Figure 2.24).

This subsection uses this procedure in a sorting example.

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
```

```
        v[k+1] = temp;
    }
```

The procedure `swap`

Let's start with the code for the procedure `swap` in the above figure. This procedure simply swaps two locations in memory. When translating from C to assembly language by hand, we follow these general steps:

1. Allocate registers to program variables.
2. Produce code for the body of the procedure.
3. Preserve registers across the procedure invocation.

This section describes the `swap` procedure in these three pieces, concluding by putting all the pieces together.

Register allocation for `swap`

The MIPS convention on parameter passing is to use registers `$a0`, `$a1`, `$a2`, and `$a3`. Since swap has just two parameters, `v` and `k`, they will be found in registers `$a0` and `$a1`. The only other variable is `temp`, which we associate with register `$t0` since swap is a leaf procedure. This register allocation corresponds to the variable declarations in the first part of the swap procedure in the above figure.

Code for the body of the procedure `swap`

The remaining lines of C code in swap are

```
temp = v[k];
v[k] = v[k + 1];
v[k + 1] = temp;
```


Recall that the memory address for MIPS refers to the *byte* address, and so words are really 4 bytes apart. Hence we need to multiply the index `k` by 4 before adding it to the address. *Forgetting that sequential word addresses differ by 4 instead of by 1 is a common mistake in assembly language programming.* Hence the first step is to get the address of `v[k]` by multiplying `k` by 4 via a shift left by 2:

```
sll   $t1, $a1, 2     # reg $t1 = k * 4
add   $t1, $a0, $t1   # reg $t1 = v + (k * 4)
                      # reg $t1 has the address of v[k]
```

Now we load `v[k]` using `$t1`, and then `v[k+1]` by adding 4 to `$t1`:

```
lw    $t0, 0($t1)     # reg $t0 (temp) = v[k]
lw    $t2, 4($t1)     # reg $t2 = v[k + 1]
                      # refers to next element of v
```

Next we store `$t0` and `$t2` to the swapped addresses:


```
sw    $t2, 0($t1)     # v[k] = reg $t2
sw    $t0, 4($t1)     # v[k + 1] = reg $t0 (temp)
```

Now we have allocated registers and written the code to perform the operations of the procedure. What is missing is the code for preserving the saved registers used within `swap`. Since we are not using saved registers in this leaf procedure, there is nothing to preserve.

The full `swap` procedure

We are now ready for the whole routine, which includes the procedure label and the return jump. To make it easier to follow, we identify in the figure below each block of code with its purpose in the procedure.

Figure 2.13.2: MIPS assembly code of the procedure swap (COD Figure 2.25).

Procedure body

```
swap: sll    $t1, $a1, 2          # reg $t1 = k * 4
      add    $t1, $a0, $t1        # reg $t1 = v + (k * 4)
                                  # reg $t1 has the address of v[k]
      lw     $t0, 0($t1)          # reg $t0 (temp) = v[k]
      lw     $t2, 4($t1)          # reg $t2 = v[k + 1]
                                  # refers to next element of v
      sw     $t2, 0($t1)          # v[k] = reg $t2
      sw     $t0, 4($t1)          # v[k+1] = reg $t0 (temp)
```

Procedure return

```
      jr     $ra                  # return to calling routine
```


PARTICIPATION ACTIVITY 2.13.1: Swap procedure in C and assembly.

Consider the above code for swap(). Assume:

- function header is void swap(int v[], int k)
- array v is {9, 47, 6, 25}
- k is 1

- variable temp is associated with register $t0

1) What is the value of v[k] after the swap function executes?

Check **Show answer**

2) Which register holds v[]?

Check **Show answer**

3) What value is in $t1 after: `sll $t1, $a1, 2`

Check **Show answer**

4) Suppose v is located at address 4008. After executing `add $t1, $a0, $t1` , what is in $t1?

Check **Show answer**

5) What is the value of $t0 after: `lw $t0, 0($t1)`?

Check **Show answer**

6) What is the value of $t2 after: `lw $t2, 4($t1)` ?

Check **Show answer**

7) What is the value of v[1] after this instruction: `sw $t2, 0($t1)`?

Check **Show answer**

8) What is the value of v[k+1] after this instruction: `sw $t0, 4($t1)`?

Check **Show answer**

The procedure `sort`

To ensure that you appreciate the rigor of programming in assembly language, we'll try a second, longer example. In this case, we'll build a routine that calls the swap procedure. This program sorts an array of integers, using bubble or exchange sort, which is one of the simplest if not the fastest sorts. The figure below shows the C version of the program. Once again, we present this procedure in several steps, concluding with the full procedure.

Figure 2.13.3: A C procedure that performs a sort on the array v (COD Figure 2.26).

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v, j);
        }
    }
}
```

Register allocation for `sort`

The two parameters of the procedure `sort`, `v` and `n`, are in the parameter registers `$a0` and `$a1`, and we assign register `$s0` to `i` and register `$s1` to `j`.

Code for the body of the procedure `sort`

The procedure body consists of two nested *for* loops and a call to `swap` that includes parameters. Let's unwrap the code from the outside to the middle.

The first translation step is the first *for* loop:

```
for (i = 0; i < n; i += 1) {
```

Recall that the C *for* statement has three parts: initialization, loop test, and iteration increment. It takes just one instruction to initialize `i` to 0, the first part of the *for* statement:

```
move $s0, $zero          # i = 0
```

(Remember that `move` is a pseudoinstruction provided by the assembler for the convenience of the assembly language programmer.) It also takes just one instruction to increment `i`, the last part of the *for* statement:

```
addi $s0, $s0, 1         # i += 1
```

The loop should be exited if `i < n` is *not* true or, said another way, should be exited if `i ≥ n`. The set on less than instruction sets register `$t0` to 1 if `$s0 < $a1` and to 0 otherwise. Since we want to test if `$s0 ≥ $a1`, we branch if register `$t0` is 0. This test takes two instructions:

```
for1tst: slt $t0, $s0, $a1        # reg $t0 = 0 if $s0 ≥ $a1 (i ≥ n)
         beq $t0, $zero, exit1    # go to exit1 if $s0 ≥ $a1 (i ≥ n)
```

The bottom of the loop just jumps back to the loop test:

```
         jal x0, for1tst               # jump to test of outer loop
exit1:
```


The skeleton code of the first for loop is then

```
         move $s0, $zero          # i = 0
for1tst: slt  $t0, $s0, $a1       # reg $t0 = 0 if $s0 ≥ $a1 (i ≥ n)
         beq  $t0, $zero, exit1   # go to exit1 if $s0 ≥ $a1 (i ≥ n)
            ...
```

```
            (body of first for loop)
            ...
         addi $s0, $s0, 1          # i+=1
         jal x0, for1tst                  # jump to test of outer loop
exit1:
```

Voila! (The exercises explore writing faster code for similar loops.)

The second *for* loop looks like this in C:

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

The initialization portion of this loop is again one instruction:

```
addi $s1, $s0, -1         # j = i - 1
```

The decrement of `j` at the end of the loop is also one instruction:

```
addi $s1, $s1, -1         # j -= 1
```

The loop test has two parts. We exit the loop if either condition fails, so the first test must exit the loop if it fails (`j` < 0):

```
for2tst: slti $t0, $s1, 0          # reg $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2    # goto exit2 if $s1 < 0 (j < 0)
```

This branch will skip over the second condition test. If it doesn't skip, `j` ≥ 0.

The second test exits if `v[j] > v[j + 1]` is *not* true, or exits if `v[j] ≤ v[j + 1]`. First we create the address by multiplying `j` by 4 (since we need a byte address) and add it to the base address of `v`:

```
sll $t1, $s1, 2           # reg $t1 = j * 4
add $t2, $a0, $t1         # reg $t2 = v + (j * 4)
```

Now we load `v[j]`:

```
lw $t3, 0($t2)            # reg $t3 = v[j]
```


Since we know that the second element is just the following word, we add 4 to the address in register `$t2` to get `v[j + 1]`:

```
lw $t4, 4($t2)            # reg $t4 = v[j + 1]
```

The test of `v[j] ≤ v[j + 1]` is the same as `v[j + 1] ≥ v[j]`, so the two instructions of the exit test are

```
slt $t0, $t4, $t3        # reg $t0 = 0 if $t4 ≥ $t3
beq $t0, $zero, exit2    # go to exit2 if $t4 ≥ $t3
```

The bottom of the loop jumps back to the inner loop test:

```
jal x0, for2tst               # jump to test of inner loop
```


Combining the pieces, the skeleton of the second *for* loop looks like this:

```
         addi $s1, $s0, -1        # j = i - 1
for2tst: slti $t0, $s1, 0         # reg $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2   # goto exit2 if $s1 < 0 (j < 0)
         sll  $t1, $s1, 2         # reg $t1 = j * 4
         add  $t2, $a0, $t1       # reg $t2 = v + (j * 4)
         lw   $t3, 0($t2)         # reg $t3 = v[j]
         lw   $t4, 4($t2)         # reg $t4 = v[j + 1]
         slt  $t0, $t4, $t3       # reg $t0 = 0 if $t4 ≥ $t3
         beq $t0, $zero, exit2    # go to exit2 if $t4 ≥ $t3
            ...
            (body of second for loop)
            ...
         addi $s1, $s1, -1        # j -= 1
         jal x0, for2tst               # jump to test of inner loop
exit2:
```

The procedure call in `sort`

The next step is the body of the second *for* loop:

```
swap(v, j);
```

Calling swap is easy enough:

```
jal swap
```


Passing parameters in `sort`

The problem comes when we want to pass parameters because the `sort` procedure needs the values in registers `$a0` and `$a1`, yet the swap procedure needs to have its parameters placed in those same registers. One solution is to copy the parameters for `sort` into other registers earlier in the procedure, making registers `$a0` and `$a1` available for the call of `swap`. (This copy is faster

than saving and restoring on the stack.) We first copy `$a0` and `$a1` into `$s2` and `$s3` during the procedure:

```
move $s2, $a0   # copy parameter $a0 into $s2
move $s3, $a1   # copy parameter $a1 into $s3
```

Then we pass the parameters to `swap` with these two instructions:


```
move $a0, $s2   # first swap parameter is v
move $a1, $s1   # second swap parameter is j
```

Preserving registers in `sort`

The only remaining code is the saving and restoring of registers. Clearly, we must save the return address in register `$ra`, since `sort` is a procedure and is called itself. The `sort` procedure also uses the saved registers `$s0, $s1, $s2,` and `$s3`, so they must be saved. The prologue of the `sort` procedure is then

```
addi $sp, $sp, -20   # make room on stack for 5 registers
sw   $ra, 16($sp)    # save $ra on stack
sw   $s3, 12($sp)    # save $s3 on stack
sw   $s2, 8($sp)     # save $s2 on stack
sw   $s1, 4($sp)     # save $s1 on stack
sw   $s0, 0($sp)     # save $s0 on stack
```

The tail of the procedure simply reverses all these instructions, then adds a `jr` to return.

The full procedure `sort`

Now we put all the pieces together in the figure below, being careful to replace references to registers `$a0` and `$a1` in the *for* loops with references to registers `$s2` and `$s3`. Once again, to make the code easier to follow, we identify each block of code with its purpose in the procedure. In this example, nine lines of the `sort` procedure in C became 35 lines in the MIPS assembly language.

Figure 2.13.4: MIPS assembly version of procedure sort (COD Figure 2.27).

Procedure body			
Move parameters		move $s2, $a0	# copy parameter $a0 into $s2 (save $a0)
		move $s3, $a1	# copy parameter $a1 into $s3 (save $a1)
Outer loop		move $s0, $zero	# i = 0
	for1tst:	slt $t0, $s0, $s3	# reg $t0 = 0 if $s0 Š $s3 (i Š n)
		beq $t0, $zero, exit1	# go to exit1 if $s0 Š $s3 (i Š n)
Inner loop		addi $s1, $s0, -1	# j = i - 1
	for2tst:	slti $t0, $s1, 0	# reg $t0 = 1 if $s1 < 0 (j < 0)
		bne $t0, $zero, exit2	# go to exit2 if $s1 < 0 (j < 0)
		sll $t1, $s1, 2	# reg $t1 = j * 4
		add $t2, $s2, $t1	# reg $t2 = v + (j * 4)
		lw $t3, 0($t2)	# reg $t3 = v[j]
		lw $t4, 4($t2)	# reg $t4 = v[j + 1]
		slt $t0, $t4, $t3	# reg $t0 = 0 if $t4 Š $t3
		beq $t0, $zero, exit2	# go to exit2 if $t4 Š $t3
Pass parameters and call		move $a0, $s2	# 1st parameter of swap is v (old $a0)
		move $a1, $s1	# 2nd parameter of swap is j
		jal swap	# swap code shown in Figure 2.25
Inner loop		addi $s1, $s1, -1	# j -- 1
		j for2tst	# jump to test of inner loop
Outer loop	exit2:	addi $s0, $s0, 1	# i += 1
		j for1tst	# jump to test of outer loop

Restoring registers			
	exit1:	lw $s0, 0($sp)	# restore $s0 from stack
		lw $s1, 4($sp)	# restore $s1 from stack
		lw $s2, 8($sp)	# restore $s2 from stack
		lw $s3,12($sp)	# restore $s3 from stack
		lw $ra,16($sp)	# restore $ra from stack
		addi $sp,$sp, 20	# restore stack pointer

Procedure return			
		jr $ra	# return to calling routine

Elaboration

One optimization that works with this example is procedure inlining. Instead of passing arguments in parameters and invoking the code with a `jal` *instruction, the compiler would copy the code from the body of the* `swap` *procedure where the call to* `swap` *appears in the code. Inlining would avoid four instructions in this example. The downside of the inlining optimization is that the compiled code would be bigger if the inlined procedure is called from several locations. Such a code expansion might turn into lower performance if it increased the cache miss rate; see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy).*

PARTICIPATION ACTIVITY 2.13.2: For loop in assembly.

1) What is the value of $s0 after: `move $s0, $zero`?

Check **Show answer**

2) How much is $s0 increased by: `addi $s0, $s0, 1`?

Check **Show answer**

3) If the value of $s0 is 3 and the value of $a1 is 3, what is the value of $t0 after: `slt $t0, $s0, $a1`?

Check **Show answer**

PARTICIPATION ACTIVITY 2.13.3: Building an assembly program from a C program.

1) By convention, MIPS uses 4 registers for parameter passing allocation.
   - ○ True
   - ○ False

2) Sequential word addresses differ by 1 byte.
   - ○ True
   - ○ False

3) `move` is a pseudoinstruction as a convenience for the assembly language programmer.
   - ○ True
   - ○ False

Understanding program performance

The figure below (Comparing performance, instruction count, and CPI …) shows the impact of compiler optimization on sort program performance, compile time, clock cycles, instruction count, and CPI. Note that unoptimized code has the best CPI, and O1 optimization has the lowest instruction count, but O3 is the fastest, reminding us that time is the only accurate measure of program performance.

The figure below (Performance of two sort algorithms in C and Java …) compares the impact of programming languages, compilation versus interpretation, and algorithms on performance of sorts. The fourth column shows that the unoptimized C program is 8.3 times faster than the interpreted Java code for Bubble Sort. Using the JIT compiler makes Java 2.1 times faster than the unoptimized C and within a factor of 1.13 of the highest optimized C code. (COD Section 2.15 (Advanced Material: Compiling C and Interpreting Java) gives more details on interpretation versus compilation of Java and the Java and MIPS code for Bubble Sort.) The ratios aren't as close for Quicksort in Column 5, presumably because it is harder to amortize the cost of runtime compilation over the shorter execution time. The last column demonstrates the impact of a better algorithm, offering three orders of magnitude a performance increases by when sorting 100,000 items. Even comparing interpreted Java in Column 5 to the C compiler at highest optimization in Column 4, Quicksort beats Bubble Sort by a factor of 50 (0.05 × 2468, or 123 times faster than the unoptimized C code versus 2.41 times faster).

Figure 2.13.5: Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort (COD Figure 2.28).

The programs sorted 100,000 words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM. It used Linux version 2.4.20.

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66

03 (procedure integration)	2.41	65,747	44,993	1.46

Figure 2.13.6: Performance of two sort algorithms in C and Java using interpretation and optimizing compilers relative to unoptimized C version (COD Figure 2.29).

The last column shows the advantage in performance of Quicksort over Bubble Sort for each language and execution option. These programs were run on the same system as in the figure above. The JVM is Sun version 1.3.1, and the JIT is Sun Hotspot version 1.3.1.

Language	Execution method	Optimization	Bubble Sort relative performance	Quicksort relative performance	Speedup Quicksort vs. Bubble Sort
C	Compiler	None	1.00	1.00	2468
	Compiler	O1	2.37	1.50	1562
	Compiler	O2	2.38	1.50	1555
	Compiler	O3	2.41	1.91	1955
Java	Interpreter	–	0.12	0.05	1050
	JIT compiler	–	2.13	0.29	338

Elaboration

The MIPS compilers always save room on the stack for the arguments in case they need to be stored, so in reality they always decrement `$sp` *by 16 to make room for all four argument registers (16 bytes). One reason is that C provides a* `vararg` *option that allows a pointer to pick, say, the third argument to a procedure. When the compiler encounters the rare* `vararg`*, it copies the four argument registers onto the stack into the four reserved locations.*

2.14 Arrays versus pointers

A challenge for any new C programmer is understanding pointers. Comparing assembly code that uses arrays and array indices to the assembly code that uses pointers offers insights about pointers. This section shows C and MIPS assembly versions of two procedures to clear a sequence of words in memory: one using array indices and one using pointers. The figure below shows the

two C procedures.

Figure 2.14.1: Two C procedures for setting an array to all zeros (COD Figure 2.30).

`clear1` uses indices, while `clear2` uses pointers. The second procedure needs some explanation for those unfamiliar with C. The address of a variable is indicated by `&`, and the object pointed to by a pointer is indicated by `*`. The declarations declare that `array` and `p` are pointers to integers. The first part of the *for* loop in `clear2` assigns the address of the first element of `array` to the pointer `p`. The second part of the *for* loop tests to see if the pointer is pointing beyond the last element of `array`. Incrementing a pointer by one, in the last part of the *for* loop, means moving the pointer to the next sequential object of its declared size. Since `p` is a pointer to integers, the compiler will generate MIPS instructions to increment p by four, the number of bytes in a MIPS integer. The assignment in the loop places 0 in the object pointed to by `p`.


```
clear1(int array[], int size)
{
    int i;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}

clear2(int *array, int size)
{
    int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}
```


The purpose of this section is to show how pointers map into MIPS instructions and not to endorse a dated programming style. We'll see the impact of modern compiler optimization on these two procedures at the end of the section.

Array version of clear

Let's start with the array version, `clear1`, focusing on the body of the loop and ignoring the

procedure linkage code. We assume that the two parameters `array` and `size` are found in the registers `$a0` and `$a1`, and that `i` is allocated to register `$t0`.

The initialization of `i`, the first part of the *for* loop, is straightforward:

```
        move $t0, $zero           # i = 0 (register $t0 = 0)
```

To set `array[i]` to 0 we must first get its address. Start by multiplying `i` by 4 to get the byte address:


```
loop1: sll $t1, $t0, 2            # $t1 = i * 4
```

Since the starting address of the array is in a register, we must add it to the index to get the address of `array[i]` using an add instruction:

```
        add $t2, $a0, $t1         # $t2 = address of array[i]
```

Finally, we can store 0 in that address:

```
        sw $zero, 0($t2)          # array[i] = 0
```

This instruction is the end of the body of the loop, so the next step is to increment `i`:

```
        addi $t0, $t0, 1          # i = i + 1
```

The loop test checks if i is less than size:

```
        slt  $t3, $t0, $a1        # $t3 = (i < size)
        bne  $t3, $zero, loop1    # if (i < size) go to loop1
```

We have now seen all the pieces of the procedure. Here is the MIPS code for clearing an array using indices:

```
        move  $t0,$zero           # i = 0
loop1:  sll   $t1, $t0, 2         # $t1 = i * 4
        add   $t2, $a0, $t1       # $t2 = address of array[i]
        sw    $zero, 0($t2)       # array[i] = 0
        addi  $t0, $t0, 1         # i = i + 1
        slt   $t3, $t0, $a1       # $t3 = (i < size)
        bne   $t3, $zero, loop1   # if (i < size) gotoloop1
```


(This code works as long as `size` is greater than 0; ANSI C requires a test of size before the loop, but we'll skip that legality here.)

Pointer version of clear

The second procedure that uses pointers allocates the two parameters `array` and `size` to the registers `$a0` and `$a1` and allocates `p` to register `$t0`. The code for the second procedure starts with assigning the pointer `p` to the address of the first element of the array:

```
        move $t0, $a0           # p = address of array[0]
```

The next code is the body of the *for* loop, which simply stores 0 into `p`:


```
loop2: sw $zero, 0($t0)         # Memory[p] = 0
```

This instruction implements the body of the loop, so the next code is the iteration increment, which changes `p` to point to the next word:

```
        addi $t0, $t0, 4        # p = p + 4
```

Incrementing a pointer by 1 means moving the pointer to the next sequential object in C. Since `p` is a pointer to integers, each of which uses 4 bytes, the compiler increments p by 4.

The loop test is next. The first step is calculating the address of the last element of `array`. Start with multiplying `size` by 4 to get its byte address:

```
        sll $t1, $a1, 2         # $t1 = size * 4
```

and then we add the product to the starting address of the array to get the address of the first word *after* the array:

```
        add $t2, $a0, $t1       # $t2 = address of array[size]
```

The loop test is simply to see if `p` is less than the last element of `array`:

```
        slt  $t3, $t0, $t2      # $t3 = (p < &array[size])
        bne  $t3, $zero, loop2  # if (p < &array[size]) go to loop2
```

With all the pieces completed, we can show a pointer version of the code to zero an array:

```
        move $t0, $a0           # p = address of array[0]
loop2: sw $zero, 0($t0)         # Memory[p] = 0
        addi $t0, $t0, 4        # p = p + 4
        sll $t1, $a1, 2         # $t1 = size * 4
        add $t2, $a0, $t1       # $t2 = address of array[size]
        slt $t3, $t0, $t2       # t3 = (p < &array[size])
        bne $t3, $zero, loop2   # if (p < &array[size]) go to loop2
```


As in the first example, this code assumes `size` is greater than 0.

Note that this program calculates the address of the end of the array in every iteration of the loop, even though it does not change. A faster version of the code moves this calculation outside the loop:

```
        move $t0, $a0           # p = address of array[0]
        sll $t1, $a1, 2         # $t1 = size * 4
        add $t2, $a0, $t1       # $t2 = address of array[size]
loop2:  sw $zero, 0($t0)        # Memory[p] = 0
        addi $t0, $t0, 4        # p = p + 4
        slt $t3, $t0, $t2       # $t3 = (p < &array[size])
        bne $t3, $zero, loop2   # if (p < &array[size]) go to loop2
```


Comparing the two versions of clear

Comparing the two code sequences side by side illustrates the difference between array indices and pointers (the changes introduced by the pointer version are highlighted):

```
       move  $t0,$zero      # i = 0                    move  $t0,$a0       # p = & array[0]
loop1: sll   $t1,$t0,2      # $t1 = i * 4              sll   $t1,$a1,2     # $t1 = size * 4
       add   $t2,$a0,$t1    # $t2 = &array[i]          add   $t2,$a0,$t1   # $t2 = &array[size]
       sw    $zero, 0($t2)  # array[i] = 0      loop2: sw    $zero,0($t0)  # Memory[p] = 0
       addi  $t0,$t0,1      # i = i + 1                addi  $t0,$t0,4     # p = p + 4
       slt   $t3,$t0,$a1    # $t3 = (i < size)         slt   $t3,$t0,$t2   # $t3=(p<&array[size])
       bne   $t3,$zero,loop1# if () go to loop1        bne   $t3,$zero,loop2# if () go to loop2
```

The version on the left must have the "multiply" and add inside the loop because `i` is incremented and each address must be recalculated from the new index. The memory pointer version on the right increments the pointer `p` directly. The pointer version moves the scaling shift and the array bound addition outside the loop, thereby reducing the instructions executed per iteration from 6 to 4. This manual optimization corresponds to the compiler optimization of strength reduction (shift instead of multiply) and induction variable elimination (eliminating array address calculations within loops). COD Section 2.15 (Advanced Material: Compiling C and Interpreting Java) describes these two and many other optimizations.

Elaboration

As mentioned earlier, a C compiler would add a test to be sure that `size` is greater than 0. One way would be to add a jump just before the first instruction of the loop to the `slt` instruction.

PARTICIPATION ACTIVITY 2.14.1: Using pointers in assembly language.

1) In C, one way to declare an array as a function parameter is: int *array.
   - True
   - False

2) If int* array is allocated to register $a0 and int* p is allocated to register $t0, the proper way to get p to point to array[0] is: `move $a0, $t0`.
   - True
   - False

3) The instruction `addi $t0, $t0, 1` can be used either to increment an integer or to increment a pointer to an integer.
   - True
   - False

4) If an array's size is 5 elements, and int size has a value of 5, &array[size] returns the address of the first word after the array.
   - True
   - False

5) If int *p is allocated to register $t0 and &array[size-1] is allocated to register $t2, then the proper loop test involves `slt $t3, $t0, $t2`.
   - True
   - False

Understanding program performance

People used to be taught to use pointers in C to get greater efficiency than that

available with arrays: "Use pointers, even if you can't understand the code". Modern optimizing compilers can produce code for the array version that is just as good. Most programmers today prefer that the compiler do the heavy lifting.

2.15 Advanced material: Compiling C and interpreting Java

This section gives a brief overview of how the C compiler works and how Java is executed. Because the compiler will significantly affect the performance of a computer, understanding compiler technology today is critical to understanding performance. Keep in mind that the subject of compiler construction is usually taught in a one- or two-semester course, so our introduction will necessarily only touch on the basics.

The second part of this section is for readers interested in seeing how an *object oriented language* like Java executes on a MIPS architecture. It shows the Java byte-codes used for interpretation and the MIPS code for the Java version of some of the C segments in prior sections, including Bubble Sort. It covers both the Java Virtual Machine and just-in-time (JIT) compilers.

Object oriented language: A programming language that is oriented around objects rather than actions, or data versus logic.

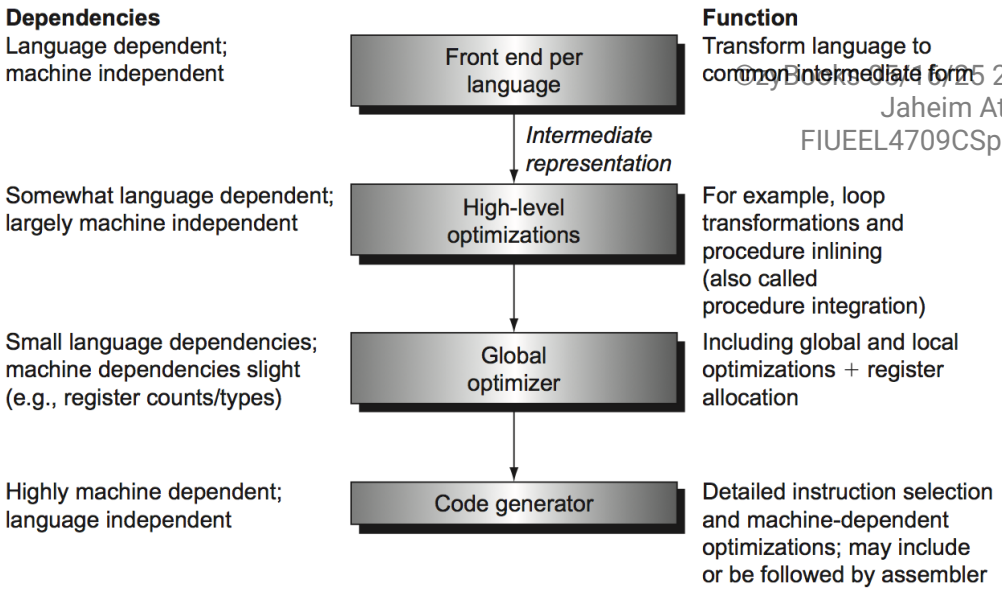
Compiling C

This first part of the section introduces the internal **anatomy** of a compiler. To start, the figure below shows the structure of recent compilers, and we describe the optimizations in the order of the passes of that structure. To illustrate the concepts in this part of this section, we will use the following C version of a while loop:


```
while (save[i] == k)
      i += 1;
```

Figure 2.15.1: The structure of a modern optimizing compiler consists of a number of passes or phases (COD Figure e2.15.1).

Logically, each pass can be thought of as running to completion before the next occurs. In practice, some passes may handle one procedure at a time, essentially interleaving with another pass.



PARTICIPATION ACTIVITY 2.15.1: Modern optimizing compiler phases.

How to use this tool

Code generator **High-level optimizations** **Global optimizer** **Front end**

Reads in a source program and translates the program to an intermediate representation.

Relatively-big changes to a program that yield the same function, but may optimize a program's performance or other metric.

Global and local optimizations.

Converts the optimized intermediate representation into a

processor's machine instructions.

Reset

The front end

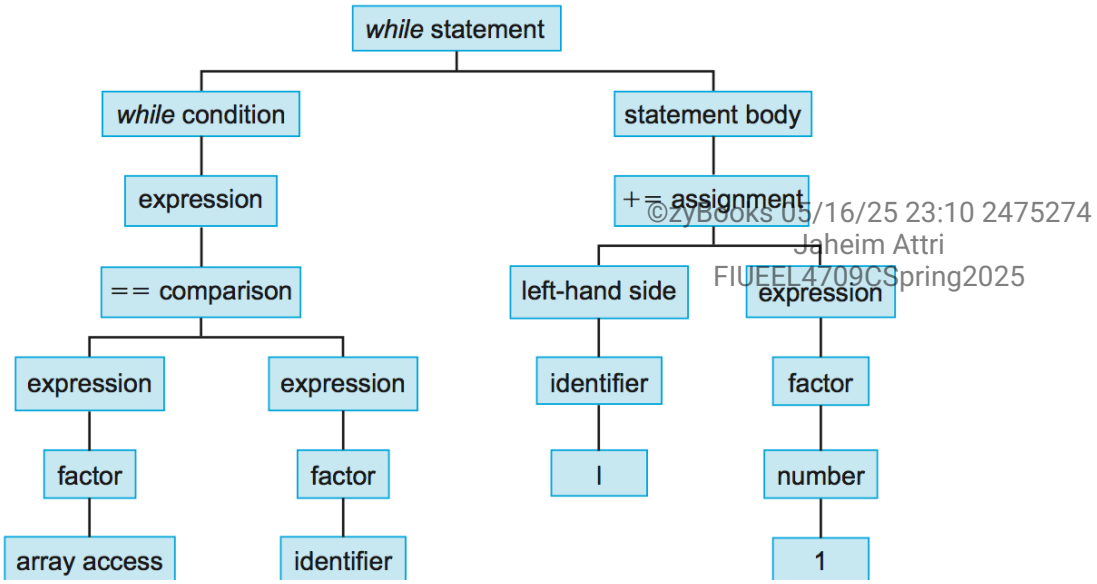
The function of the front end is to read in a source program; check the syntax and semantics, and translate the source program to an intermediate form that interprets most of the language-specific operation of the program. As we will see, intermediate forms are usually simple, and some are in fact similar to the Java bytecodes.

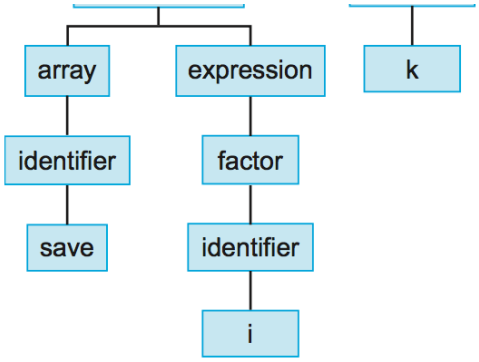
The front end is usually broken into four separate functions:

1. *Scanning* reads in individual characters and creates a string of tokens. Examples of *tokens* are reserved words, names, operators, and punctuation symbols. In the above example, the token sequence is `while`, `(`, `save`, `[`, `i`, `]`, `==`, `k`, `)`, `i`, `+=`, `1`. A word like `while` is recognized as a reserved word in C, but `save`, `i`, and `j` are recognized as names, and `1` is recognized as a number.
2. *Parsing* takes the token stream, ensures the syntax is correct, and produces an *abstract syntax tree*, which is a representation of the syntactic structure of the program. The figure below shows what the abstract syntax tree might look like for this program fragment.

Figure 2.15.2: An abstract syntax tree for the while example (COD Figure e2.15.2).

The roots of the tree consist of the informational tokens such as numbers and names. Long chains of straight-line descendents are often omitted in constructing the tree.





3. *Semantic analysis* takes the abstract syntax tree and checks the program for semantic correctness. Semantic checks normally ensure that variables and types are properly declared and that the types of operators and objects match, a step called *type checking*. During this process, a symbol table representing all the named objects—classes, variables, and functions—is usually created and used to type-check the program.
4. *Generation of the intermediate representation* (IR) takes the symbol table and the abstract syntax tree and generates the intermediate representation that is the output of the front end. Intermediate representations usually use simple operations on a small set of primitive types, such as integers, characters, and reals. Java bytecodes represent one type of intermediate form. In modern compilers, the most common intermediate form looks much like the MIPS instruction set but with an infinite number of virtual registers; later, we describe how to map these virtual registers to a finite set of real registers. The figure below shows how our example might be represented in such an intermediate form. We capitalize the MIPS instructions in this section when they represent IR forms.

Figure 2.15.3: The while loop example is shown using a typical intermediate representation (COD Figure e2.15.3).

In practice, the names `save`, `i`, and `k` would be replaced by some sort of address, such as a reference to either the local stack pointer or a global pointer, and an offset, similar to the way `save[i]` is accessed. Note that the format of the MIPS instructions is different, because they are intermediate representations here: the operations are capitalized and the registers use `RXX` notation.


```
        # comments are written like this--source code often included
        # while (save[i] == k)
 loop:  li R1,save   # loads the starting address of save into R1
        lw R2,i
        mult R3,R2,4 # Multiply R2 by 4
        add R4,R3,R1
        lw R5,0(R4) # load save[i]
```

```
        lw R6,k
        BNE R5,R6,endwhileloop
        # i += 1
        lw R6, i
        add R7,R6,1  # increment
        sw R7,i
        branch loop # next iteration
endwhileloop:
```


The intermediate form specifies the functionality of the program in a manner independent of the original source. After this front end has created the intermediate form, the remaining passes are largely language independent.

PARTICIPATION ACTIVITY 2.15.2: The front end.

1) `y = x + 1;` is converted to y, =, x, +, 1, and ;.
   - ○ Scanning
   - ○ Parsing
   - ○ Semantic analysis

2) y, =, x, +, 1, and ; are checked for correct syntax, and then converted to an abstract syntax tree.
   - ○ Scanning
   - ○ Parsing
   - ○ Semantic analysis

3) For the abstract syntax tree of `y = x + 1;`, x is determined to be a non-numeric type for which adding 1 doesn't make sense, so an error is reported.
   - ○ Parsing
   - ○ Semantic analysis
   - ○ IR generation

4) After previous steps, a representation

is generated consisting of basic operations, like add t0, t1, t2.

- ○ High-level optimization
- ○ IR generation

High-level optimizations

High-level optimizations are transformations that are done at something close to the source level.

The most common high-level transformation is probably *procedure inlining*, which replaces a call to a function by the body of the function, substituting the caller's arguments for the procedure's parameters. Other high-level optimizations involve loop transformations that can reduce loop overhead, improve memory access, and exploit the hardware more effectively. For example, in loops that execute many iterations, such as those traditionally controlled by a *for* statement, the optimization of *loop-unrolling* is often useful. Loop-unrolling involves taking a loop, replicating the body multiple times, and executing the transformed loop fewer times. Loop-unrolling reduces the loop overhead and provides opportunities for many other optimizations. Other types of high-level transformations include sophisticated loop transformations such as interchanging nested loops and blocking loops to obtain better memory behavior; see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy) for examples.

Loop-unrolling: A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.

PARTICIPATION ACTIVITY 2.15.3: High-level optimizations.

1)
```
for (int i = 0; i < 3; ++i) {
   x[i] = i * i;
}
```

being replaced by:

```
x[0] = 0 * 0;
x[1] = 1 * 1;
x[2] = 2 * 2;
```


is an example of _____.

- ○ Loop unrolling
- ○ Procedure inlining

2)
```
a = max(b, c);
d = max(e, f);
```

being replaced by:

```
a = (b > c) ? b : c;
d = (e > f) ? e : f;
```

is an example of _____.

- ◯ Loop unrolling
- ◯ Procedure inlining

3) High-level optimizations are done on instructions that are close to the assembly level.
   - ◯ True
   - ◯ False

Local and global optimizations

Within the pass dedicated to local and global optimization, three classes of optimizations are performed:

1. *Local optimization* works within a single basic block. A local optimization pass is often run as a precursor and successor to global optimization to "clean up" the code before and after global optimization.
2. *Global optimization* works across multiple basic blocks; we will see an example of this shortly.
3. *Global register allocation* allocates variables to registers for regions of the code. Register allocation is crucial to getting good performance in modern processors.

Several optimizations are performed both locally and globally, including common subexpression elimination, constant propagation, copy propagation, dead store elimination, and strength reduction. Let's look at some simple examples of these optimizations.

Common subexpression elimination finds multiple instances of the same expression and replaces the second one by a reference to the first. Consider, for example, a code segment to add 4 to an array element:

```
x[i] = x[i] + 4
```


The address calculation for `x[i]` occurs twice and is identical since neither the starting address of `x` nor the value of `i` changes. Thus, the calculation can be reused. Let's look at the intermediate code for this fragment, since it allows several other optimizations to be performed. The unoptimized intermediate code is on the left. On the right is the optimized code, using common subexpression elimination to replace the second address calculation with the first. Note that the register allocation has not yet occurred, so the compiler is using virtual register numbers like `R100` here.

```
# x[i] + 4
li R100, x
lw R101, i
mult R102, R101, 4
add R103, R100, R102
lw R104, 0(R103)
#
add R105, R104, 4
li R106, x
lw R107, i
mult R108, R107, 4
add R109, R106, R108
sw R105, 0(R109)
```

```
# x[i] + 4
li R100, x
lw R101, i
mult R102, R101, 4
add R103, R100, R102
lw R104, 0(R103)
# value of x[i] is in R104
add R105, R104, 4
sw R105,0(R103)
```


If the same optimization were possible across two basic blocks, it would then be an instance of *global common subexpression elimination*.

Let's consider some of the other optimizations:

- *Strength reduction* replaces complex operations by simpler ones and can be applied to this code segment, replacing the MULT by a shift left.
- *Constant propagation* and its *sibling constant folding* find constants in code and propagate them, collapsing constant values whenever possible.
- *Copy propagation* propagates values that are simple copies, eliminating the need to reload values and possibly enabling other optimizations, such as common subexpression elimination.
- *Dead store elimination* finds stores to values that are not used again and eliminates the store; its "cousin" is *dead code elimination*, which finds unused code—code that cannot affect the result of the program—and eliminates it. With the heavy use of macros, templates, and the similar techniques designed to reuse code in high-level languages, dead code occurs surprisingly often.

Compilers must be *conservative*. The first task of a compiler is to produce correct code; its second task is usually to produce fast code, although other factors, such as code size, may sometimes be important as well. Code that is fast but incorrect—for any possible combination of inputs—is simply wrong. Thus, when we say a compiler is "conservative," we mean that it performs an optimization only if it knows with 100% certainty that, no matter what the inputs, the code will perform as the user wrote it. Since most compilers translate and optimize one function or procedure at a time, most compilers, especially at lower optimization levels, assume the worst about function calls and about their own parameters.

Understanding program performance

Programmers concerned about performance of critical loops, especially in real-time or embedded applications, often find themselves staring at the assembly language produced by a compiler and wondering why the compiler failed to perform some global optimization or to allocate a variable to a register throughout a loop. The answer often lies in the dictate that the compiler be conservative. The opportunity for improving the code may seem obvious to the programmer, but then the programmer often has knowledge that the compiler does not have, such as the absence of aliasing between two pointers or the absence of side effects by a function call. The compiler may indeed be able to perform the transformation with a little help, which could eliminate the worst-case behavior that it must assume. This insight also illustrates an important observation: programmers who use pointers to try to improve performance in accessing variables, especially pointers to values on the stack that also have names as variables or as elements of arrays, are likely to disable many compiler optimizations. The end result is that the lower-level pointer code may run no better, or perhaps even worse, than the higher-level code optimized by the compiler.

Global code optimizations

Many global code optimizations have the same aims as those used in the local case, including common subexpression elimination, constant propagation, copy propagation, and dead store and dead code elimination.

There are two other important global optimizations: code motion and induction variable elimination. Both are loop optimizations; that is, they are aimed at code in loops. *Code motion* finds code that is loop invariant: a particular piece of code computes the same value on every iteration of the loop and, hence, may be computed once outside the loop. *Induction variable elimination* is a combination of transformations that reduce overhead on indexing arrays, essentially replacing array indexing with pointer accesses. Rather than examine induction variable elimination in depth, we point the reader to COD Section 2.14 (Arrays versus pointers), which compares the use of array indexing and pointers; for most loops, the transformation from the more obvious array code to the pointer code can be performed by a modern optimizing compiler.

Implementing local optimizations

Local optimizations are implemented on basic blocks by scanning the basic block in instruction execution order, looking for optimization opportunities. In the assignment statement example, the

duplication of the entire address calculation is recognized by a series of sequential passes over the code. Here is how the process might proceed, including a description of the checks that are needed:

1. Determine that the two `li` operations return the same result by observing that the operand `x` is the same and that the value of its address has not been changed between the two `li` operations.
2. Replace all uses of `R106` in the basic block by `R101`.
3. Observe that `i` cannot change between the two `LW`s that reference it. So replace all uses of `R107` with `R101`.
4. Observe that the `mult` instructions now have the same input operands, so that `R108` may be replaced by `R102`.
5. Observe that now the two `add` instructions have identical input operands (`R100` and `R102`), so replace the `R109` with `R103`.
6. Use dead store code elimination to delete the second set of `li, lw, mult`, and `add` instructions since their results are unused.

Throughout this process, we need to know when two instances of an operand have the same value. This is easy to determine when they refer to virtual registers, since our intermediate representation uses such registers only once, but the problem can be trickier when the operands are variables in memory, even though we are only considering references within a basic block.

It is reasonably easy for the compiler to make the common subexpression elimination determination in a conservative fashion in this case; as we will see in the next subsection, this is more difficult when branches intervene.

Implementing global optimizations

To understand the challenge of implementing global optimizations, let's consider a few examples:

- Consider the case of an opportunity for common subexpression elimination, say, of an IR statement like `ADD Rx, R20, R50`. To determine whether two such statements compute the same value, we must determine whether the values of `R20` and `R50` are identical in the two statements. In practice, this means that the values of `R20` and `R50` have not changed between the first statement and the second. For a single basic block, this is easy to decide; it is more difficult for a more complex program structure involving multiple basic blocks and branches.
- Consider the second `LWlw` of `i` into `R107` within the earlier example: how do we know whether its value is used again? If we consider only a single basic block, and we know that all uses of `R107` are within that block, it is easy to see. As optimization proceeds, however, common subexpression elimination and copy propagation may create other uses of a value. Determining that a value is unused and the code is dead is more difficult in the case of multiple basic blocks.

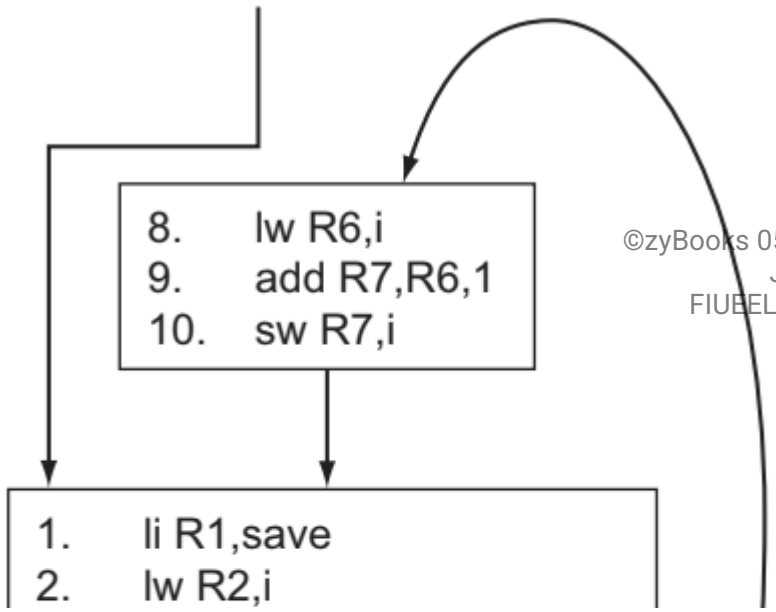
- Finally, consider the load of `k` in our loop, which is a candidate for code motion. In this simple example, we might argue that it is easy to see that `k` is not changed in the loop and is, hence, loop invariant. Imagine, however, a more complex loop with multiple nestings and *if* statements within the body. Determining that the load of `k` is loop invariant is harder in such a case.

The information we need to perform these global optimizations is similar: we need to know where each operand in an IR statement could have been changed or *defined* (use-definition information). The dual of this information is also needed: that is, finding all the uses of that changed operand (definition-use information). *Data flow analysis* obtains both types of information.

Global optimizations and data flow analysis operate on a *control flow graph*, where the nodes represent basic blocks and the arcs represent control flow between basic blocks. The figure below shows the control flow graph for our simple loop example, with one important transformation introduced. We describe the transformation in the caption, but see if you can discover it, and why it was done, on your own!

Figure 2.15.4: A control flow graph for the while loop example (COD Figure e2.15.4).

Each node represents a basic block, which terminates with a branch or by sequential fall-through into another basic block that is also the target of a branch. The IR statements have been numbered for ease in referring to them. The important transformation performed was to move the *while* test and conditional branch to the end. This eliminates the unconditional branch that was formerly inside the loop and places it before the loop. This transformation is so important that many compilers do it during the generation of the IR. The `mult` was also replaced with ("strength-reduced to") an `sll`.

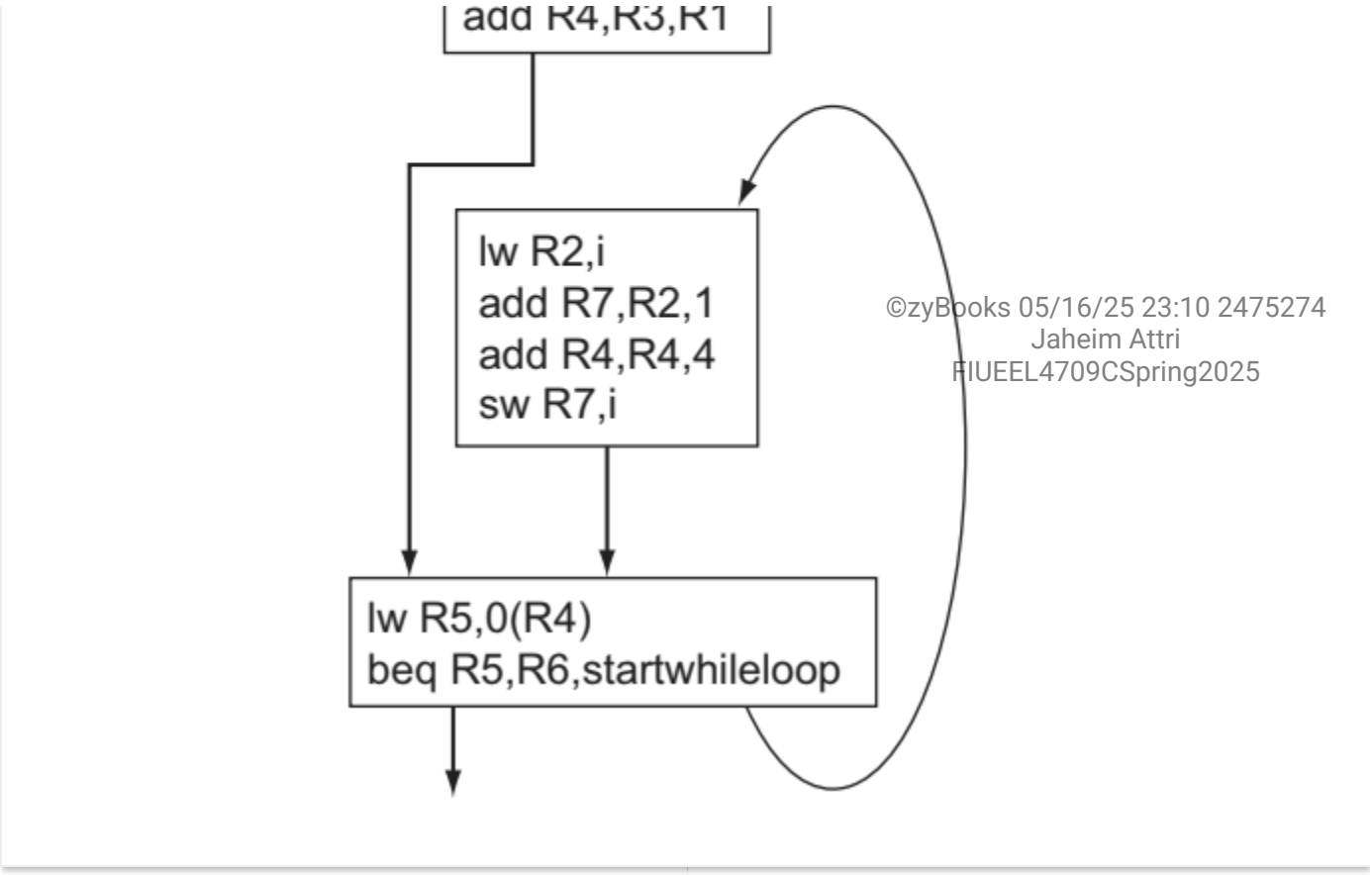


Suppose we have computed the use-definition information for the control flow graph in the above figure. How does this information allow us to perform code motion? Consider IR statements number 1 and 6: in both cases, the use-definition information tells us that there are no definitions (changes) of the operands of these statements within the loop. Thus, these IR statements can be moved outside the loop. Notice that if the `li` of `save` and the `lw` of `k` are executed once, just prior to the loop entrance, the computational effect is the same, but the program now runs faster since these two statements are outside the loop. In contrast, consider IR statement 2, which loads the value of `i`. The definitions of `i` that affect this statement are both outside the loop, where `i` is initially defined, and inside the loop in statement 10 where it is stored. Hence, this statement is not loop invariant.

The figure below shows the code after performing both code motion and induction variable elimination, which simplifies the address calculation. The variable `i` can still be register allocated, eliminating the need to load and store it every time, and we will see how this is done in the next subsection.

Figure 2.15.5: The control flow graph showing the representation of the while loop example after code motion and induction variable elimination (COD Figure e2.15.5).

The number of instructions in the inner loop has been reduced from 10 to 6.



Before we turn to register allocation, we need to mention a caveat that also illustrates the complexity and difficulty of optimizers. Remember that the compiler must be conservative. To be conservative, a compiler must consider the following question: Is there *any way* that the variable `k` could possibly ever change in this loop? Unfortunately, there is one way. Suppose that the variable `k` and the variable `i` actually refer to the same memory location, which could happen if they were accessed by pointers or reference parameters.

I am sure that many readers are saying, "Well, that would certainly be a stupid piece of code!" Alas, this response is not open to the compiler, which must translate the code as it is written. Recall too that the aliasing information must also be conservative; thus, compilers often find themselves negating optimization opportunities because of a possible alias that exists in one place in the code or because of incomplete information about aliasing.

Register allocation

Register allocation is perhaps the most important optimization for modern load-store architectures. Eliminating a load or a store eliminates an instruction. Furthermore, register allocation enhances the value of other optimizations, such as common subexpression elimination. Fortunately, the trend toward larger register counts in modern architectures has made register allocation simpler and more effective. Register allocation is done on both a local basis and a global basis, that is, across multiple basic blocks but within a single function. Local register allocation is usually done late in compilation, as the final code is generated. Our focus here is on the more challenging and more opportunistic global register allocation.

Modern global register allocation uses a region-based approach, where a region (sometimes called a *live range*) represents a section of code during which a particular variable could be allocated to a particular register. How is a region selected? The process is iterative:

1. Choose a definition (change) of a variable in a given basic block; add that block to the region.
2. Find any uses of that definition, which is a data flow analysis problem; add any basic blocks that contain such uses, as well as any basic block that the value passes through to reach a use, to the region.
3. Find any other definitions that also can affect a use found in the previous step and add the basic blocks containing those definitions, as well as the blocks the definitions pass through to reach a use, to the region.
4. Repeat steps 2 and 3 using the definitions discovered in step 3 until convergence.

The set of basic blocks found by this technique has a special property: if the designated variable is allocated to a register in all these basic blocks, then there is no need for loading and storing the variable.

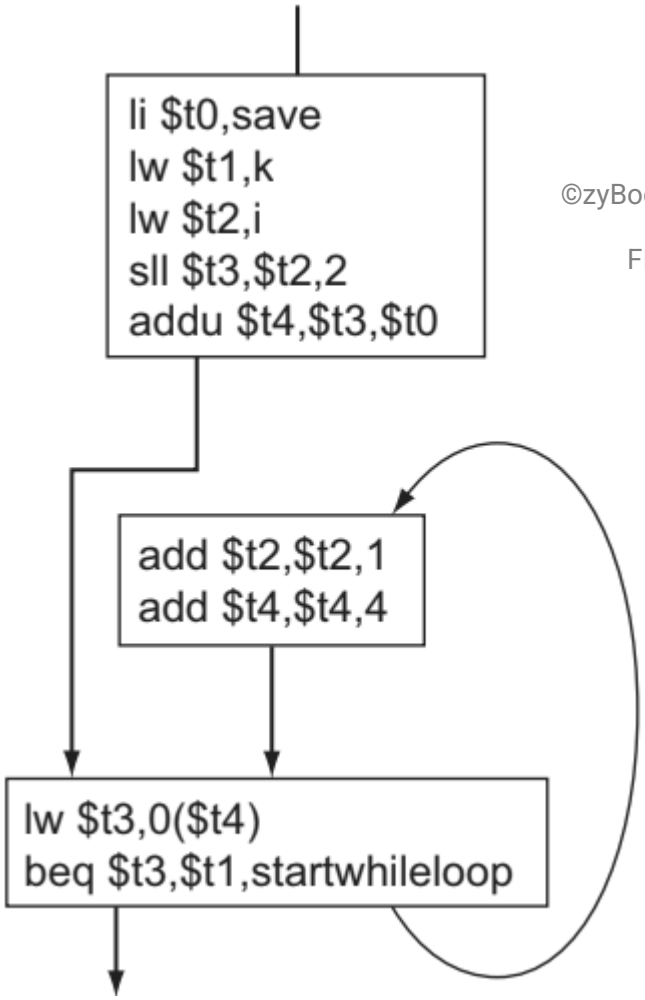
Modern global register allocators start by constructing the regions for every virtual register in a function. Once the regions are constructed, the key question is how to allocate a register to each region: the challenge is that certain regions overlap and may not use the same register. Regions that do not overlap (i.e., share no common basic blocks) can share the same register. One way to represent the interference among regions is with an *interference graph*, where each node represents a region, and the arcs between nodes represent that the regions have some basic blocks in common.

Once an interference graph has been constructed, the problem of allocating registers is equivalent to a famous problem called *graph coloring*: find a color for each node in a graph such that no two adjacent nodes have the same color. If the number of colors equals the number of registers, then coloring an interference graph is equivalent to allocating a register for each region! This insight was the initial motivation for the allocation method now known as region-based allocation, but originally called the graph-coloring approach. The figure below shows the flow graph representation of the *while* loop example after register allocation.

Figure 2.15.6: The control flow graph showing the representation of the while loop example after code motion and induction variable elimination and register allocation, using the MIPS register names (COD Figure e2.15.6).

The number of IR statements in the inner loop has now dropped to only four from six before register allocation and ten before any global optimizations. The value of `i` resides in `$t2` at the end of the loop and may need to be stored eventually to maintain the program semantics. If `i` were unused after the loop, not only could the store be avoided,

but also the increment inside the loop could be eliminated completely!



What happens if the graph cannot be colored using the number of registers available? The allocator must spill registers until it can complete the coloring. By doing the coloring based on a priority function that takes into account the number of memory references saved and the cost of tying up the register, the allocator attempts to avoid spilling for the most important candidates.

Spilling is equivalent to splitting up a region (or live range); if the region is split, fewer other regions will interfere with the two separate nodes representing the original region. A process of splitting regions and successive coloring is used to allow the allocation process to complete, at which point all candidates will have been allocated a register. Of course, whenever a region is split, loads and stores must be introduced to get the value from memory or to store it there. The location chosen to split a region must balance the cost of the loads and stores that must be introduced against the advantage of freeing up a register and reducing the number of interferences.

Modern register allocators are incredibly effective in using the large register counts available in modern processors. In many programs, the effectiveness of register allocation is limited not by the availability of registers but by the possibilities of aliasing that cause the compiler to be conservative in its choice of candidates.

PARTICIPATION ACTIVITY 2.15.4: Local and global optimizations.

How to use this tool

Register allocation **Local optimization** **Global optimization**

A compiler replaces `x = 2 * 3` with `x = 6`.

A loop that uses `i` to iterate from 0 to 9 has a first statement of `x = 6`. A compiler moves the `x = 6` to just before the loop.

The statement `x = y + z` originally uses three registers in the intermediate representation: `add t0, t1, t2`. A compiler decides that `t0` is not needed after that operation, and thus modifies the operation to: `add t1, t1, t2`.

Reset

Code generation

The final steps of the compiler are code generation and assembly. Most compilers do not use a stand-alone assembler that accepts assembly language source code; to save time, they instead perform most of the same functions: filling in symbolic values and generating the binary code as the final stage of code generation.

In modern processors, code generation is reasonably straightforward, since the simple architectures make the choice of instruction relatively obvious. For more complex architectures, such as the x86, code generation is more complex since multiple IR instructions may collapse into a single machine instruction. In modern compilers, this compilation process uses pattern matching with either a tree-based pattern matcher or a pattern matcher driven by a parser.

During code generation, the final stages of machine-dependent optimization are also performed.

These include some constant folding optimizations, as well as localized instruction scheduling (see COD Chapter 4 (The Processor)).

Optimization summary

The figure below gives examples of typical optimizations, and the last column indicates where the optimization is performed in the gcc compiler. It is sometimes difficult to separate some of the simpler optimizations—local and processor-dependent optimizations—from transformations done in the code generator, and some optimizations are done multiple times, especially local optimizations, which may be performed before and after global optimization as well as during code generation.

Figure 2.15.7: Major types of optimizations and explanation of each class (COD Figure e2.15.7).

The third column shows when these occur at different levels of optimization in gcc. The GNU organization calls the three optimization levels medium (O1), full (O2), and full with integration of small procedures (O3).

Optimization name	Explanation	gcc level
High level	*At or near the source level; processor independent*	
Procedure integration	Replace procedure call by procedure body	O3
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	O1
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	O1
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	O1
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	O2
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., *A* = *X*) with *X*	O2
Code motion	Remove code from a loop that computes the same value each iteration of the loop	O2
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	O2
Processor dependent	*Depends on processor knowledge*	
Strength reduction	Many examples; replace multiply by a constant with shifts	O1
Pipeline scheduling	Reorder instructions to improve pipeline performance	O1
Branch offset optimization	Choose the shortest branch displacement that reaches target	O1

One of the most important uses of interprocedural analysis is to obtain so-called alias information. An *alias* occurs when two names may designate the same variable. For example, it is quite helpful to know that two pointers passed to a function may never designate the same variable. Alias information is usually flow-insensitive and must be used conservatively.

Hardware/Software Interface

Today, essentially all programming for desktop and server applications is done in high-level languages, as is most programming for embedded applications. This development means that since most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target. It is tempting to add sophisticated operations in an instruction set. The challenge is that they may not exactly match what the compiler needs to produce or may be so general that they aren't fast. For example, consider special loop instructions found in some computers. Suppose that instead of decrementing by one, the compiler wanted to increment by four, or instead of branching on not equal zero, the compiler wanted to branch if the index was less than or equal to the limit. The loop instruction may be a mismatch. When faced with such objections, the instruction set designer might then generalize the operation, adding another operand to specify the increment and perhaps an option on which branch condition to use. Then the danger is that a common case, say, incrementing by one, will be slower than a sequence of simple operations.

Elaboration

Some more sophisticated compilers, and many research compilers, use an analysis technique called interprocedural analysis to obtain more information about functions and how they are called. Interprocedural analysis attempts to discover what properties remain true across a function call. For example, we might discover that a function call can never change any global variables, which might be useful in optimizing a loop that calls such a function. Such information is called may-information or flow-insensitive information and can be obtained reasonably efficiently, although analyzing a call to a function F requires analyzing all the functions that F calls, which makes the process somewhat time consuming for large programs. A more costly property to discover is that a function must always change some variable; such information is called must-information or flow-sensitive information. Recall the dictate to be conservative: may-information can never be used as must-information—just because a function may change a variable does not mean that it must change it. It is conservative, however, to use the negation of may-information, so the compiler can rely on the fact that a function will never change a variable in optimizations around the call site of that function.

Interpreting Java

This second part of the section is for readers interested in seeing how an *object-oriented language* like Java executes on a MIPS architecture. It shows the Java bytecodes used for interpretation and the MIPS code for the Java version of some of the C segments in prior sections, including Bubble Sort.

Let's quickly review the Java lingo to make sure we are all on the same page. The big idea of object-oriented programming is for programmers to think in terms of abstract objects, and operations are associated with each *type* of object. New types can often be thought of as refinements to existing types, and so some operations for the existing types are used by the new type without change. The hope is that the programmer thinks at a higher level, and that code can be reused more readily if the programmer implements the common operations on many different types.

This different perspective led to a different set of terms. The type of an object is a *class*, which is the definition of a new data type together with the operations that are defined to work on that data type. A particular object is then an *instance* of a class, and creating an object from a class is called *instantiation*. The operations in a class are called *methods*, which are similar to C procedures. Rather than call a procedure as in C, you *invoke* a method in Java. The other members of a class are *fields*, which correspond to variables in C. Variables inside objects are called *instance fields*. Rather than access a structure with a pointer, Java uses an *object reference* to access an object. The syntax for method invocation is `x.y`, where `x` is an object reference and `y` is the method name.

The parent—child relationship between older and newer classes is captured by the verb "extends": a child class *extends* (or sub classes) a parent class. The child class typically will redefine some of the methods found in the parent to match the new data type. Some methods work fine, and the child class *inherits* those methods.

To reduce the number of errors associated with pointers and explicit memory deallocation, Java automatically frees unused storage, using a separate garbage collector that frees memory when it is full. Hence, `new` creates a new instance of a dynamic object on the heap, but there is no `free` in Java. Java also requires array bounds to be checked at runtime to catch another class of errors that can occur in C programs.

Interpretation

As mentioned before, Java programs are distributed as Java bytecodes, and the Java Virtual Machine (JVM) executes Java byte codes. The JVM understands a binary format called the *class file* format. A class file is a stream of bytes for a single class, containing a table of valid methods with their bytecodes, a pool of constants that acts in part as a symbol table, and other information such as the parent class of this class.

When the JVM is first started, it looks for the class method `main`. To start any Java class, the JVM

dynamically loads, links, and initializes a class. The JVM loads a class by first finding the binary representation of the proper class (class file) and then creating a class from that binary representation. Linking combines the class into the runtime state of the JVM so that it can be executed. Finally, it executes the class initialization method that is included in every class.

Figure 2.15.8: Java bytecode architecture versus MIPS (COD Figure e2.15.8).

Although many bytecodes are simple, those in the last half-dozen rows above are complex and specific to Java. Bytecodes are one to five bytes in length, hence their name. The Java mnemonics use the prefix `i` for 32-bit integer, `a` for reference (address), `s` for 16-bit integers (short), and `b` for 8-bit bytes. We use `I8` for an 8-bit constant and `I16` for a 16-bit constant. MIPS uses registers for operands, but the JVM uses a stack. The compiler knows the maximum size of the operand stack for each method and simply allocates space for it in the current frame. Here is the notation in the Meaning column: `TOS`: top of stack; `NOS`: next position below `TOS`; `NNOS`: next position below `NOS`; pop: remove `TOS`; pop2: remove `TOS` and `NOS`; and push: add a position to the stack. `*NOS` and `*NNOS` mean access the memory location pointed to by the address in the stack at those positions. `Const[]` refers to the runtime constant pool of a class created by the JVM, and `Frame[]` refers to the variables of the local method frame. The only missing MIPS instructions are `nor`, `andi`, `ori`, `slti`, and `lui`. The missing bytecodes are a few arithmetic and logical operators, some tricky stack management, compares to 0 and branch, support for branch tables, type conversions, more variations of the complex, Java-specific instructions plus operations on floating-point data, 64-bit integers (longs), and 16-bit characters.

Category	Operation	Java bytecode	Size (bits)	MIPS instr.	Meaning
Arithmetic	add	iadd	8	add	NOS=TOS+NOS; pop
	subtract	isub	8	sub	NOS=TOS–NOS; pop
	increment	iinc I8a I8b	8	addi	Frame[I8a]= Frame[I8a] + I8b
Data transfer	load local integer/address	iload I8/aload I8	16	lw	TOS=Frame[I8]
	load local integer/address	iload_/aload_{0,1,2,3}	8	lw	TOS=Frame[{0,1,2,3}]
	store local integer/address	istore I8/astore I8	16	sw	Frame[I8]=TOS; pop
	load integer/address from array	iaload/aaload	8	lw	NOS=*NOS[TOS]; pop
	store integer/address into array	iastore/aastore	8	sw	*NNOS[NOS]=TOS; pop2
	load half from array	saload	8	lh	NOS=*NOS[TOS]; pop
	store half into array	sastore	8	sh	*NNOS[NOS]=TOS; pop2
	load byte from array	baload	8	lb	NOS=*NOS[TOS]; pop
	store byte into array	bastore	8	sb	*NNOS[NOS]=TOS; pop2
	load immediate	bipush I8, sipush I16	16, 24	addi	push; TOS=I8 or I16
	load immediate	iconst_{–1,0,1,2,3,4,5}	8	addi	push; TOS={–1,0,1,2,3,4,5}
Logical	and	iand	8	and	NOS=TOS&NOS; pop
	or	ior	8	or	NOS=TOS|NOS; pop
	shift left	ishl	8	sll	NOS=NOS << TOS; pop
	shift right	iushr	8	srl	NOS=NOS >> TOS; pop
Conditional branch	branch on equal	if_icompeq I16	24	beq	if TOS == NOS, go to I16; pop2
	branch on not equal	if_icompne I16	24	bne	if TOS != NOS, go to I16; pop2
	compare	if_icomp{lt,le,gt,ge} I16	24	slt	if TOS {<,<=,>,>=} NOS, go to I16; pop2

Unconditional jump	jump	goto I16	24	j	go to I16
	return	ret, ireturn	8	jr	
	jump to subroutine	jsr I16	24	jal	go to I16; push; TOS=PC+3
Stack management	remove from stack	pop, pop2	8		pop, pop2
	duplicate on stack	dup	8		push; TOS=NOS
	swap top 2 positions on stack	swap	8		T=NOS; NOS=TOS; TOS=T
Safety check	check for null reference	ifnull I16, ifnotnull I16	24		if TOS {==,!=} null, go to I16
	get length of array	arraylength	8		push; TOS = length of array
	check if object a type	instanceof I16	24		TOS = 1 if TOS matches type of Const[I16]; TOS = 0 otherwise
Invocation	invoke method	invokevirtual I16	24		Invoke method in Const[I16], dispatching on type
Allocation	create new class instance	new I16	24		Allocate object type Const[I16] on heap
	create new array	newarray I16	24		Allocate array type Const[I16] on heap

The figure above shows Java bytecodes and their corresponding MIPS instructions, illustrating five major differences between the two:

1. To simplify compilation, Java uses a stack instead of registers for operands. Operands are pushed on the stack, operated on, and then popped off the stack.
2. The designers of the JVM were concerned about code size, so bytecodes vary in length between one and five bytes, versus the 4-byte, fixed-size MIPS instructions. To save space, the JVM even has redundant instructions of different lengths whose only difference is size of the immediate. This decision illustrates a code size variation of our third design principle: make the common case *small*.
3. The JVM has safety features embedded in the architecture. For example, array data transfer instructions check to be sure that the first operand is a reference and that the second index operand is within bounds.
4. To allow garbage collectors to find all live pointers, the JVM uses different instructions to operate on addresses versus integers so that the JVM can know what operands contain addresses. MIPS generally lumps integers and addresses together.
5. Finally, unlike MIPS, there are Java-specific instructions that perform complex operations, like allocating an array on the heap or invoking a method.

Example 2.15.1: Compiling a while loop in Java using bytecodes.

Compile the *while* loop, this time using Java bytecodes:


```
while (save[i] == k)
   i += 1;
```

Assume that `i`, `k`, and `save` are the first three local variables. Show the addresses of the bytecodes. The MIPS version of the C loop took six instructions and twenty-four bytes. How big is the bytecode version?

Answer

The first step is to put the array reference in save on the stack:

```
0 aload_3             # Push local variable 3 (save[]) onto stack
```

This 1-byte instruction informs the JVM that an address in local variable 3 is being put on the stack. The 0 on the left of this instruction is the byte address of this first instruction; bytecodes for each method start at 0. The next step is to put the index on the stack:


```
1 iload_1             # Push local variable 1 (i) onto stack
```

Like the prior instruction, this 1-byte instruction is a short version of a more general instruction that takes 2 bytes to load a local variable onto the stack. The next instruction is to get the value from the array element:

```
2 iaload              # Put array element (save[i]) onto stack
```

This 1-byte instruction checks the prior two operands, pops them off the stack, and then puts the value of the desired array element onto the new top of the stack. Next, we place `k` on the stack:

```
3 iload_2             # Push local variable 2 (k) onto stack
```

We are now ready for the *while* test:

```
4 if_icompne, Exit    # Compare and exit if not equal
```

This 3-byte instruction compares the top two elements of the stack, pops them off the stack, and branches if they are not equal. We are finally ready for the body of the loop:

```
7 iinc, 1, 1          # Increment local variable 1 by 1 (i+=1)
```

This unusual 3-byte instruction increments a local variable by 1 without using the operand stack, an optimization that again saves space. Finally, we return to the top of the loop with a 3-byte jump:

```
10 go to 0            # Go to top of Loop (byte address 0)
```


Thus, the bytecode version takes seven instructions and thirteen bytes, almost half the size of the MIPS C code. (As before, we can optimize this code to jump less.)

Compiling for Java

Since Java is derived from C and Java has the same built-in types as C, the assignment statement examples in COD Chapter 2 (Instructions: Language of the Computer) are the same in Java as they are in C. The same is true for the *if* statement.

The Java version of the *while* loop is different, however. The designers of C leave it up to the programmers to be sure that their code does not exceed the array bounds. The designers of Java wanted to catch array bound bugs, and thus require the compiler to check for such violations. To check bounds, the compiler needs to know what they are. Java includes an extra word in every array that holds the upper bound. The lower bound is defined as 0.

Example 2.15.2: Compiling a while loop in Java.

Modify the MIPS code for the *while* loop to include the array bounds checks that are required by Java. Assume that the length of the array is located just before the first element of the array.

Answer

Let's assume that Java arrays reserved the first two words of arrays before the data starts. We'll see the use of the first word soon, but the second word has the array length. Before we enter the loop, let's load the length of the array into a temporary register:

```
lw $t2, 4($s6)                  # Temp reg $t2 = length of array sav
```

Before we multiply `i` by 4, we must test to see if it's less than 0 or greater than the last element of the array. The first step is to check if i is less than 0:

```
Loop: slt $t0, $s3, $zero       # Temp reg $t0 = 1 if i < 0
```

Register `$t0` is set to 1 if i is less than 0. Hence, a branch to see if register `$t0` is *not equal to* zero will give us the effect of branching if `i` is less than 0. This pair of instructions, `slt` and `bne`, implements branch on less than. Register `$zero` always contains 0, so this final test is accomplished using the `bne` instruction and comparing register `$t0` to register `$zero`:

```
bne $t0, $zero, IndexOutOfBounds    # if i < 0, goto Error
```


Since the array starts at 0, the index of the last array element is one less than the length of the array. Thus, the test of the upper array bound is to be sure that `i` is less than the length of the array. The second step is to set a temporary register to 1 if `i` is less than the array length and then branch to an error if it's not less. That is, we branch to an error if the temporary register is *equal* to zero:

```
slt $t0, $s3, $t2               # Temp reg $t0 = 0 if i >= length
```

```
beq $t0, $zero, IndexOutOfBounds   # if i >= length, goto Error
```

Note that these two instructions implement branch on greater than or equal to. The next two lines of the MIPS *while* loop are unchanged from the C version:

```
sll $t1, $s3, 2                    # Temp reg $t1 = 4 * i
add $t1, $t1, $s6                  # $t1 = address of save[i]
```


We need to account for the first 8 bytes that are reserved in Java. We do that by changing the address field of the load from 0 to 8:

```
lw $t0, 8($t1)                     # Temp reg $t0 = save[i]
```

The rest of the MIPS code from the C *while* loop is fine as is:

```
bne $t0, $s5, Exit                 # go to Exit if save[i] ? k
add $s3, $s3, 1                    # i = i + 1
j Loop                             # go to Loop
Exit:
```

(See the exercises for an optimization of this sequence.)

Invoking methods in Java

The compiler picks the appropriate method depending on the type of the object. In a few cases, it is unambiguous, and the method can be invoked with no more overhead than a C procedure. In general, however, the compiler knows only that a given variable contains a pointer to an object that belongs to some subtype of a general class. Since it doesn't know at compile time which subclass the object is, and thus which method should be invoked, the compiler will generate code that first tests to be sure the pointer isn't null and then uses the code to load a pointer to a table with all the legal methods for that type. The first word of the object has the method table address, which is why Java arrays reserve two words. Let's say it's using the fifth method that was declared for that class. (The method order is the same for all subclasses.) The compiler then takes the fifth address from that table and invokes the method at that address.

The cost of object orientation in general is that method invocation includes 1) a conditional branch to be sure that the pointer to the object is valid; 2) a load to get the address of the table of available methods; 3) another load to get the address of the proper method; 4) placing a return address into the return register, and finally 5) a jump register to invoke the method. The next subsection gives a concrete example of method invocation.

A sort example in Java

The figure below shows the Java version of exchange sort. A simple difference is that there is no need to pass the length of the array as a separate parameter, since Java arrays include their length: `v.length` denotes the length of `v`.

Figure 2.15.9: An initial Java procedure that performs a sort on the array v (COD Figure e2.15.9).

Changes from COD Figures e2.24 (A C procedure that swaps two locations in memory) and e2.26 (A C procedure that performs a sort on the array v) are highlighted.

```
public class sort {
   public static void sort (int[] v) {
      for (int i = 0; i < v.length; i += 1) {
         for (int j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
                swap(v, j);
         }
   }

   protected static void swap(int[] v, int k) {
      int temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
   }}
```

A more significant difference is that Java methods are prepended with keywords not found in the C procedures. The `sort` method is declared `public static` while `swap` is declared `protected static`. *Public* means that `sort` can be invoked from any other method, while *protected* means `swap` can only be called by other methods within the same *package* and from methods within derived classes. A *static method* is another name for a class method—methods that perform classwide operations and do not apply to an individual object. Static methods are essentially the same as C procedures.

Public: A Java keyword that allows a method to be invoked by any other method.

Protected: A Java keyword that restricts invocation of a method to other methods in that package.

Package: Basically a directory that contains a group of related classes.

Static method: A method that applies to the whole class rather to an individual object. It is unrelated to static in C.

This straightforward translation from C into static methods means there is no ambiguity on method invocation, and so it can be just as efficient as C. It also is limited to sorting integers, which means a different sort has to be written for each data type.

To demonstrate the object orientation of Java, the figure below shows the new version with the changes highlighted. First, we declare `v` to be of the type `Comparable` and replace `v[j] > v[j + 1]` with an invocation of `compareTo`. By changing `v` to this new class, we can use this code to sort many data types.

Figure 2.15.10: A revised Java procedure that sorts on the array v that can take on more types (COD Figure e2.15.10).

Changes from the above figure are highlighted.

```
public class sort {
   public static void sort(Comparable[] v) {
      for (int i = 0; i < v.length; i += 1) {
         for (int j = i - 1; j >= 0 && v[j].compareTo(v[j + 1]) > 0; j -=1) {
            swap(v, j);
         }
      }
   }

   Protected static void swap(Comparable[] v, int k) {
      Comparable temp = v[k];
      v[k] = v[k + 1];
      v[k + 1] = temp;
   }

   public class Comparable {
      public int compareTo(int x) {
         return value - x;
      }
      public int value;
   }
}
```


The method `compareTo` compares two elements and returns a value greater than 0 if the parameter is larger than the object, 0 if it is equal, and a negative number if it is smaller than the object. These two changes generalize the code so it can sort integers, characters, strings, and so

on, if there are subclasses of `Comparable` with each of these types and if there is a version of `compareTo` for each type. For pedagogic purposes, we redefine the class `Comparable` and the method `compareTo` here to compare integers. The actual definition of `Comparable` in the Java library is considerably different.

Starting from the MIPS code that we generated for C, we show what changes we made to create the MIPS code for Java.

For `swap`, the only significant differences are that we must check to be sure the object reference is not null and that each array reference is within bounds. The first test checks that the address in the first parameter is not zero:

```
swap: beq $a0, $zero, NullPointer    # if $a0 == 0,goto Error
```

Next, we load the length of v into a register and check that index `k` is OK.

```
lw $t2, 4($a0)                       # Temp reg $t2 = length of array v
slt $t0, $a1, $zero                  # Temp reg $t0 = 1 if k < 0
bne $t0, $zero, IndexOutOfBounds     # if k < 0, goto Error
slt $t0, $a1, $t2                    # Temp reg $t0 = 0 if k >= length
beq $t0, $zero, IndexOutOfBounds     # if k >= length, goto Error
```

This check is followed by a check that `k + 1` is within bounds.

```
addi $t1, $a1, 1                     # Temp reg $t1 = k + 1
slt $t0, $t1, $zero                  # Temp reg $t0 = 1 if k + 1 < 0
bne $t0, $zero, IndexOutOfBounds     # if k + 1 < 0, goto Error
slt $t0, $t1, $t2                    # Temp reg $t0 = 0 if k + 1 >= lengt
beq $t0, $zero, IndexOutOfBounds     # if k + 1 >= length, goto Error
```

The figure below highlights the extra MIPS instructions in swap that a Java compiler might produce. We again must adjust the offset in the load and store to account for two words reserved for the method table and length.

Figure 2.15.11: MIPS assembly code of the procedure swap (COD Figure e2.15.11).

Bounds check

```
swap: beq   $a0,$zero,NullPointer      #if $a0==0,goto Error
      lw    $t2,-4($a0)                # Temp reg $t2 = length of array v
      slt   $t0,$a1,$zero              # Temp reg $t0 = 1 if k < 0
      bne   $t0,$zero,IndexOutOfBounds # if k < 0, goto Error
      slt   $t0,$a1,$t2                # Temp reg $t0 = 0 if k >= length
      beq   $t0,$zero,IndexOutOfBounds # if k >= length, goto Error
      addi  $t1,$a1,1                  # Temp reg $t1 = k+1
      slt   $t0,$t1,$zero              # Temp reg $t0 = 1 if k+1 < 0
```

```
bne     $t0,$zero,IndexOutOfBounds  # if k+1 < 0, goto Error
slt     $t0,$t1,$t2                 # Temp reg $t0 = 0 if k+1 >= length
beq     $t0,$zero,IndexOutOfBounds  # if k+1 >= length, goto Error
```

Method body

```
sll     $t1, $a1, 2             # reg $t1 = k * 4
add     $t1, $a0, $t1           # reg $t1 = v + (k * 4)
                                # reg $t1 has the address of v[k]
lw      $t0, 8($t1)             # reg $t0 (temp) = v[k]
lw      $t2, 12($t1)            # reg $t2 = v[k + 1]
                                # refers to next element of v
sw      $t2, 8($t1)             # v[k] = reg $t2
sw      $t0, 12($t1)            # v[k+1] = reg $t0 (temp)
```


Procedure return

```
jr      $ra                     # return to calling routine
```

The figure below shows the method body for those new instructions for `sort`. (We can take the saving, restoring, and return from COD Figure 2.27 (MIPS assembly version of procedure sort).)

Figure 2.15.12: MIPS assembly version of the method body of the Java version of sort (COD Figure e2.15.12).

The new code is highlighted in this figure. We must still add the code to save and restore registers and the return from the MIPS code found in COD Figure 2.27 (MIPS assembly version of procedure sort). To keep the code similar to that figure, we load `v.length` into `$s3` instead of into a temporary register. To reduce the number of lines of code, we make the simplifying assumption that `compareTo` is a leaf procedure and we do not need to push registers to be saved on the stack.

Method body

Move parameters		move	$s2, $a0	# copy parameter $a0 into $s2 (save $a0)
Test ptr null		beq	$a0,$zero,NullPointer	# if $a0==0, goto Error
Get array length		lw	$s3,4($a0)	# $s3 = length of array v
Outer loop	for1tst:	move slt beq	$s0, $zero $t0, $s0, $s3 $t0, $zero, exit1	# i = 0 # reg $t0 = 0 if $s0 ≥ $s3 (i ≥ n) # go to exit1 if $s0 ≥ $s3 (i ≥ n)
Inner loop start	for2tst:	addi slti bne	$s1, $s0, -1 $t0, $s1, 0 $t0, $zero, exit2	# j = i - 1 # reg $t0 = 1 if $s1 < 0 (j < 0) # go to exit2 if $s1 < 0 (j < 0)
Test if j too big		slt beq	$t0,$s1,$s3 $t0,$zero,IndexOutOfBounds	# Temp reg $t0 = 0 if j >= length # if j >= length, goto Error
Get v[j]		sll add lw	$t1, $s1, 2 $t2, $s2, $t1 $t3, 0($t2)	# reg $t1 = j * 4 # reg $t2 = v + (j * 4) # reg $t3 = v[j]
Test if j+1 < 0 or if j+1 too big		addi slt bne slt beq	$t1,$s1,1 $t0,$t1,$zero $t0,$zero,IndexOutOfBounds $t0,$t1,$s3 $t0,$zero,IndexOutOfBounds	# Temp reg $t1 = j+1 # Temp reg $t0 = 1 if j+1 < 0 # if j+1 < 0, goto Error # Temp reg $t0 = 0 if j+1 >= length # if j+1 >= length, goto Error
Get v[j+1]		lw	$t4, 4($t2)	# reg $t4 = v[j + 1]
Load method table		lw	$t5,0($a0)	# $t5 = address of method table
Get method addr		lw	$t5,8($t5)	# $t5 = address of first method
Pass parameters		move move	$a0, $t3 $a1, $t4	# 1st parameter of compareTo is v[j] # 2nd param. of compareTo is v[j+1]

Set return addr		la	$ra,L1	# load return address
Call indirectly		jr	$t5	# call code for compareTo
Test if should skip swap	L1:	slt beq	$t0, $zero, $v0 $t0, $zero, exit2	# reg $t0 = 0 if 0 $\geq$ $v0 # go to exit2 if $t4 $\geq$ $t3
Pass parameters and call swap		move move jal	$a0, $s2 $a1, $s1 swap	# 1st parameter of swap is v # 2nd parameter of swap is j # swap code shown in Figure 2.34
Inner loop end		addi j	$s1, $s1, -1 for2tst	# j -= 1 # jump to test of inner loop
Outer loop	exit2:	addi j	$s0, $s0, 1 for1tst	# i += 1 # jump to test of outer loop

The first test is again to make sure the pointer to `v` is not null:

```
beq $a0, $zero, NullPointer          # if $a0 == 0, goto Error
```

Next, we load the length of the array (we use register `$s3` to keep it similar to the code for the C version of swap):

```
lw $s3, 4($a0)                       # $s3 = length of array v
```

Now we must ensure that the index is within bounds. Since the first test of the inner loop is to test if `j` is negative, we can skip that initial bound test. That leaves the test for too big:

```
slt $t0, $s1, $s3                    # Temp reg $t0 = 0 if j >= length
beq $t0, $zero, IndexOutOfBounds     # if j >= length, goto Error
```

The code for testing `j + 1` is quite similar to the code for checking `k + 1` in `swap`, so we skip it here.

The key difference is the invocation of `compareTo`. We first load the address of the table of legal methods, which we assume is two words before the beginning of the array:

```
lw $t5, 0($a0)                       # $t5 = address of method table
```

Given the address of the method table for this object, we then get the desired method. Let's assume `compareTo` is the third method in the `Comparable` class. To pick the address of the third method, we load that address into a temporary register:

```
lw $t5, 8($t5)                       # $t5 = address of third method
```


We are now ready to call `compareTo`. The next step is to save the necessary registers on the stack. Fortunately, we don't need the temporary registers or argument registers after the method invocation, so there is nothing to save. Thus, we simply pass the parameters for `compareTo`:

```
move $a0, $t3                        # 1st parameter of compareTo is v
move $a1, $t4                        # 2nd parameter of compareTo is v
```

Since we are using a jump register to invoke `compareTo`, we need to pass the return address explicitly. We use the pseudoinstruction load address (`la`) and label where we want to return, and then do the indirect jump:

```
    la $ra, L1                          # load return address
    jr $t5                              # to code for compareTo
```


The method returns, with `$v0` determining which of the two elements is larger. If `$v0 > 0`, then `v[j] > v[j + 1]`, and we need to `swap`. Thus, to skip the `swap`, we need to test if `$v0 ≤ 0`, which is the same as `0 ≥ $v0`. We also need to include the label for the return address:

```
L1: slt $t0, $zero, $v0                 # reg $t0 = 0 if 0 ≥ $v0
    beq $t0, $zero, exit2               # go to exit2 if v[j + 1] ≥ v[j]
```

The MIPS code for `compareTo` is left as an exercise.

Hardware/Software Interface

The main changes for the Java versions of `sort` and `swap` are testing for null object references and index out-of-bounds errors, and the extra method invocation to give a more general compare. This method invocation is more expensive than a C procedure call, since it requires a load, a conditional branch, a pair of chained loads, and an indirect jump. As we see in COD Chapter 4 (The Processor), dependent loads and indirect jumps can be relatively slow on modern processors. The increasing popularity of Java suggests that many programmers today are willing to leverage the high performance of modern processors to pay for error checking and code reuse.

Elaboration

Although we test each reference to j and j + 1 to be sure that these indices are within bounds, an assembly language programmer might look at the code and reason as follows:

1. *The inner for loop is only executed if j ≤ 0 and since j + 1 > j,there is no need to test j + 1 to see if it is less than 0.*
2. *Since i takes on the values, 0, 1, 2, ..., (data.length - 1) and since j takes on the*

values i-1, i-2, …, 2, 1, 0, there is no need to test if j ≤ data.length since the largest value j can be is data.length - 2.

3. *Following the same reasoning, there is no need to test whether j + 1 ≤ data.length since the largest value of j + 1 is data.length - 1.*

There are coding tricks in the rest of COD Chapter 2 (Instructions: Language of the Computer) and superscalar execution in COD Chapter 4 (The Processor) that lower the effective cost of such bounds checking, but only high optimizing compilers can reason this way. Note that if the compiler inlined the swap method into sort, many checks would be unnecessary.

Elaboration

Look carefully at the code for swap in the figure above (MIPS assembly code of the procedure swap). See anything wrong in the code, or at least in the explanation of how the code works? It implicitly assumes that each `Comparable` *element in* `v` *is 4 bytes long. Surely, you need much more than 4 bytes for a complex subclass of* `Comparable`*, which could contain any number of fields. Surprisingly, this code does work, because an important property of Java's semantics forces the use of the same, small representation for all variables, fields, and array elements that belong to* `Comparable` *or its subclasses.*

Java types are divided into primitive types—the predefined types for numbers, characters, and Booleans—and reference types—the built-in classes like String, user-defined classes, and arrays. Values of reference types are pointers (also called references) to anonymous objects that are themselves allocated in the heap. For the programmer, this means that assigning one variable to another does not create a new object, but instead makes both variables refer to the same object. Because these objects are anonymous, and programs therefore have no way to refer to them directly, a program must use indirection through a variable to read or write any objects' fields (variables). Thus, because the data structure allocated for the array v consists entirely of pointers, it is safe to assume they are all the same size, and the same swapping code works for all of `Comparable`*'s subtypes.*

To write sorting and swapping functions for arrays of primitive types requires that we write new versions of the functions, one for each type. This replication is for two reasons. First, primitive type values do not include the references to dispatching tables that we used on `Comparables` *to determine at runtime how to compare values. Second, primitive values come in different sizes: 1, 2, 4, or 8 bytes.*

The pervasive use of pointers in Java is elegant in its consistency, with the penalty being a level of indirection and a requirement that objects be allocated on the heap. Furthermore, in any language where the lifetimes of the heap-allocated anonymous objects are independent of the lifetimes of the named variables, fields, and array elements that reference them, programmers must deal with the problem of deciding when it is safe to deallocate heap-allocated storage. Java's designers chose to use garbage collection. Of course, use of garbage collection rather than explicit user memory management also improves program safety.

C++ provides an interesting contrast. Although programmers can write essentially the same pointer-manipulating solution in C++, there is another option. In C++, programmers can elect to forgo the level of indirection and directly manipulate an array of objects, rather than an array of pointers to those objects. To do so, C++ programmers would typically use the template capability, which allows a class or function to be parameterized by the type of data on which it acts. Templates, however, are compiled using the equivalent of macro expansion. That is, if we declared an instance of sort capable of sorting types X and Y, C++ would create two copies of the code for the class: one for sort<X> and one for sort<Y>, each specialized accordingly. This solution increases code size in exchange for making comparison faster (since the function calls would not be indirect, and might even be subject to inline expansion). Of course, the speed advantage would be canceled if swapping the objects required moving large amounts of data instead of just single pointers. As always, the best design depends on the details of the problem.

PARTICIPATION ACTIVITY 2.15.5: Local and global optimizations.

1) In contrast to C, Java calls the appropriate method (procedure) based on the ____ of the calling object.
   - ○ size
   - ○ type

2) In contrast to C, Java explicitly stores an extra item in every array indicating the array's ____.
   - ○ lower bound
   - ○ upper bound

2.16 Real stuff: ARMv7 (32-bit) instructions

ARM is the most popular instruction set architecture for embedded devices, with more than 100 billion devices through 2016. Standing originally for the Acorn RISC Machine, later changed to Advanced RISC Machine, ARM came out the same year as MIPS and followed similar philosophies. The figure below lists the similarities. The principal difference is that MIPS has more registers and ARM has more addressing modes.

Figure 2.16.1: Similarities in ARM and MIPS instruction sets (COD Figure 2.31).

	ARM	MIPS
Date announced	1985	1985
Instruction size (bits)	32	32
Address space (size, model)	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Integer registers (number, model, size)	15 GPR × 32 bits	31 GPR × 32 bits
I/O	Memory mapped	Memory mapped

There is a similar core of instruction sets for arithmetic-logical and data transfer instructions for MIPS and ARM, as the figure below shows.

Figure 2.16.2: ARM register-register and data transfer instructions equivalent to MIPS core (COD Figure 2.32).

Dashes mean the operation is not available in that architecture or not synthesized in a few instructions. If there are several choices of instructions equivalent to the MIPS core, they are separated by commas. ARM includes shifts as part of every data operation instruction, so the shifts with superscript 1 are just a variation of a move instruction, such as `lsl`. Note that ARM has no divide instruction.

	Instruction name	ARM	MIPS
	Add	add	addu, addiu
	Add (trap if overflow)	adds; swivs	add
	Subtract	sub	subu
	Subtract (trap if overflow)	subs; swivs	sub

Register-register	Multiply	mul	mult, multu
	Divide	—	div, divu
	And	and	and
	Or	orr	or
	Xor	eor	xor
	Load high part register	—	lui
	Shift left logical	lsl[1]	sllv, sll
	Shift right logical	lsr[1]	srlv, srl
	Shift right arithmetic	asr[1]	srav, sra
	Compare	cmp, cmn, tst, teq	slt/i,slt/iu
Data transfer	Load byte signed	ldrsb	lb
	Load byte unsigned	ldrb	lbu
	Load halfword signed	ldrsh	lh
	Load halfword unsigned	ldrh	lhu
	Load word	ldr	lw
	Store byte	strb	sb
	Store halfword	strh	sh
	Store word	str	sw
	Read, write special registers	mrs, msr	move
	Atomic Exchange	swp, swpb	ll;sc

Addressing modes

The figure below shows the data addressing modes supported by ARM. Unlike MIPS, ARM does not reserve a register to contain 0. Although MIPS has just three simple data addressing modes, ARM has nine, including fairly complex calculations. For example, ARM has an addressing mode that can shift one register by any amount, add it to the other registers to form the address, and then update one register with this new address.

Figure 2.16.3: Summary of data addressing modes (COD Figure 2.33).

ARM has separate register indirect and register + offset addressing modes, rather than just putting 0 in the offset of the latter mode. To get greater addressing range, ARM shifts the offset left 1 or 2 bits if the data size is halfword or word.

Addressing mode	ARM	MIPS
Register operand	X	X
Immediate operand	X	X
Register + offset (displacement or based)	X	X

Register + register (indexed)	X	—
Register + scaled register (scaled)	X	—
Register + offset and update register	X	—
Register + register and update register	X	—
Autoincrement, autodecrement	X	—
PC-relative data	X	—

Compare and conditional branch

MIPS uses the contents of registers to evaluate conditional branches. ARM uses the traditional four condition code bits stored in the program status word: *negative*, *zero*, *carry*, and *overflow*. They can be set on any arithmetic or logical instruction; unlike earlier architectures, this setting is optional on each instruction. An explicit option leads to fewer problems in a pipelined implementation. ARM uses conditional branches to test condition codes to determine all possible unsigned and signed relations.

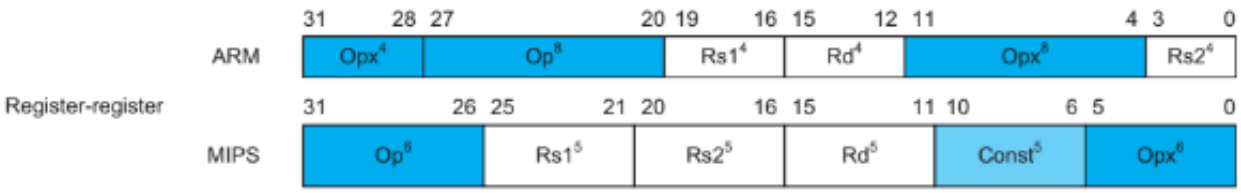
CMP subtracts one operand from the other and the difference sets the condition codes. *Compare negative* (CMN) *adds* one operand to the other, and the sum sets the condition codes. TST performs logical AND on the two operands to set all condition codes but overflow, while TEQ uses exclusive OR to set the first three condition codes.

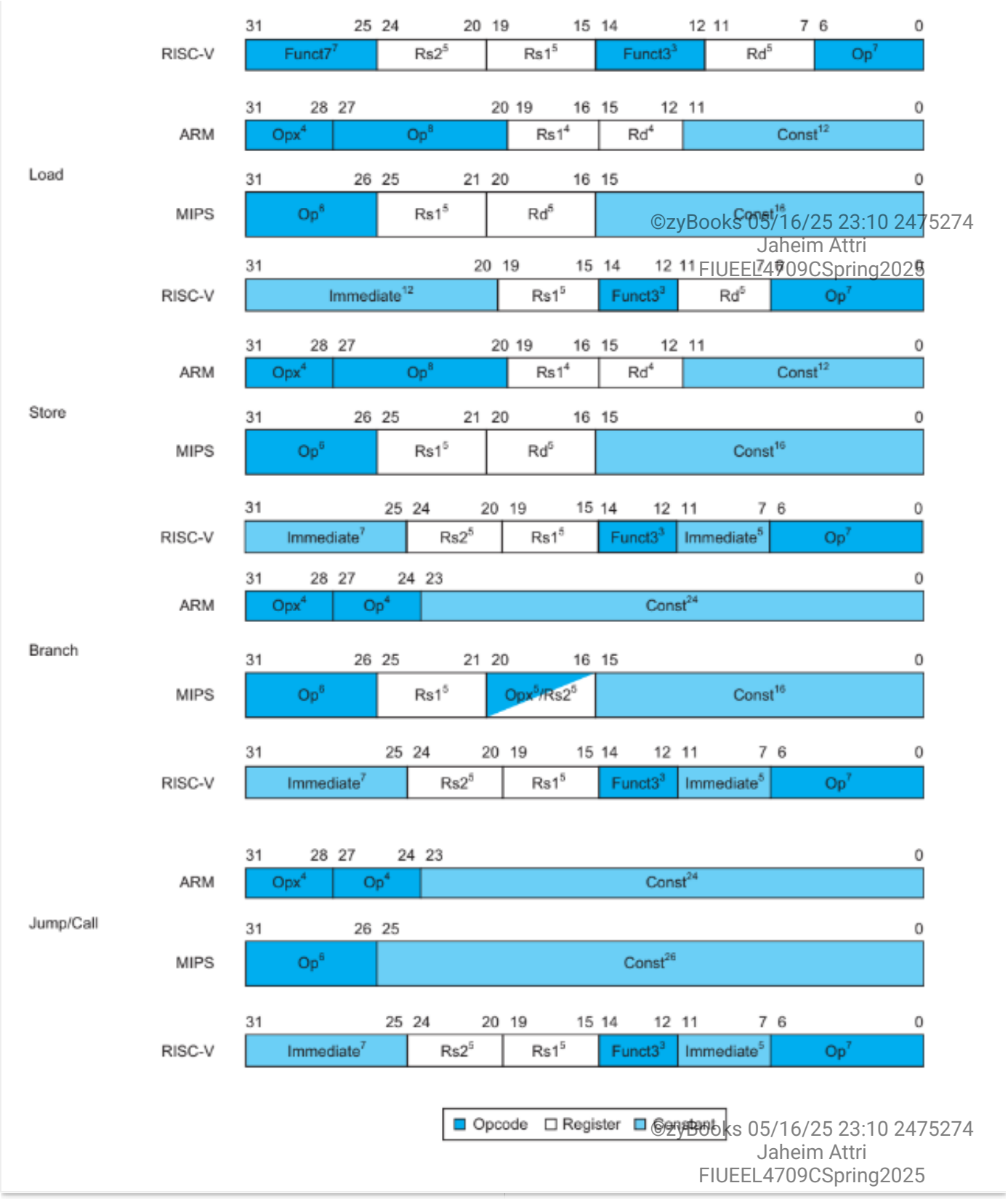
One unusual feature of ARM is that every instruction has the option of executing conditionally, depending on the condition codes. Every instruction starts with a 4-bit field that determines whether it will act as a no operation instruction (nop) or as a real instruction, depending on the condition codes. Hence, conditional branches are properly considered as conditionally executing the unconditional branch instruction. Conditional execution allows avoiding a branch to jump over a single instruction. It takes less code space and time to simply conditionally execute one instruction.

The figure below shows the instruction formats for ARM and MIPS. The principal differences are the 4-bit conditional execution field in every instruction and the smaller register field, because ARM has half the number of registers.

Figure 2.16.4: Instruction formats, ARM and MIPS (COD Figure 2.34).

The differences result from whether the architecture has 16 or 32 registers like MIPS and RISC-V.





Unique features of ARM

The figure below shows a few arithmetic-logical instructions not found in MIPS. Since ARM does not have a dedicated register for 0, it has separate opcodes to perform some operations that MIPS can do with `$zero`. In addition, ARM has support for multiword arithmetic.

Figure 2.16.5: ARM arithmetic/logical instructions not found in MIPS (COD Figure 2.35).

Name	Definition	ARM	MIPS
Load immediate	Rd = Imm	mov	addi $0,
Not	Rd = ~(Rs1)	mvn	nor $0,
Move	Rd = Rs1	mov	or $0,
Rotate right	Rd = Rs i >> i $Rd_{0...i-1} = Rs_{31-i...31}$	ror	
And not	Rd = Rs1 & ~(Rs2)	bic	
Reverse subtract	Rd = Rs2 – Rs1	rsb, rsc	
Support for multiword integer add	CarryOut, Rd = Rd + Rs1 + OldCarryOut	adcs	—
Support for multiword integer sub	CarryOut, Rd = Rd – Rs1 + OldCarryOut	sbcs	—

ARM's 12-bit immediate field has a novel interpretation. The eight least-significant bits are zero-extended to a 32-bit value, then rotated right the number of bits specified in the first four bits of the field multiplied by two. One advantage is that this scheme can represent all powers of two in a 32-bit word. Whether this split actually catches more immediates than a simple 12-bit field would be an interesting study.

Operand shifting is not limited to immediates. The second register of all arithmetic and logical processing operations has the option of being shifted before being operated on. The shift options are shift left logical, shift right logical, shift right arithmetic, and rotate right.

ARM also has instructions to save groups of registers, called *block loads and stores*. Under control of a 16-bit mask within the instructions, any of the 16 registers can be loaded or stored into memory in a single instruction. These instructions can save and restore registers on procedure entry and return. These instructions can also be used for block memory copy, and they also reduce code size on procedure entry and exit.

PARTICIPATION ACTIVITY 2.16.1: MIPS vs. ARM.

How to use this tool

ARMv7 **$zero** **MIPS** **slt** **lsl** **div** **cmp**

Divide instruction (MIPS only).

ARM comparison instruction.

Dedicated register for 0 (MIPS only).

ARM shift left logical instruction.

MIPS comparison instruction.

Instruction set architecture with 32 registers and 3 addressing modes.

Instruction set architecture with 16 registers and 9 addressing modes.

Reset

2.17 Real stuff: ARMv8 (64-bit) instruction set

Of the many potential problems in an instruction set, the one that is almost impossible to overcome is having too small a memory address. While the x86 was successfully extended first to 32-bit addresses and then later to 64-bit addresses, many of its brethren were left behind. For example, the 16-bit address MOStek 6502 powered the Apple II, but even given this headstart with the first commercially successful personal computer, its lack of address bits condemned it to the dustbin of history.

ARM architects could see the writing on the wall of their 32-bit address computer, and began design of the 64-bit address version of ARM in 2007. It was finally revealed in 2013. Rather than some minor cosmetic changes to make all the registers 64 bits wide, which is basically what happened to the x86, ARM did a complete overhaul. The good news is that if you know MIPS it will be very easy to pick up ARMv8, as the 64-bit version is called.

First, as compared to MIPS, ARM dropped virtually all of the unusual features of v7:

- There is no conditional execution field, as there was in nearly every instruction in v7.
- The immediate field is simply a 12 bit constant, rather than essentially an input to a function that produces a constant as in v7.
- ARM dropped Load Multiple and Store Multiple instructions.

- The PC is no longer one of the registers, which resulted in unexpected branches if you wrote to it.

Second, ARM added missing features that are useful in MIPS:

- V8 has 32 general-purpose registers, which compiler writers surely love. Like MIPS, one register is hardwired to 0, although in load and store instructions it instead refers to the stack pointer.
- Its addressing modes work for all word sizes in ARMv8, which was not the case in ARMv7.
- It includes a divide instruction, which was omitted from ARMv7.
- It adds the equivalent of MIPS branch if equal and branch if not equal.

As the philosophy of the v8 instruction set is much closer to MIPS than it is to v7, our conclusion is that the main similarity between ARMv7 and ARMv8 is the name.

PARTICIPATION ACTIVITY 2.17.1: ARMv8 instruction set.

1) ARMv8 uses a __-bit address instruction set.
   - ○ 32
   - ○ 64

2) ARMv8 treats the PC as a general-purpose register.
   - ○ True
   - ○ False

3) ARMv8 has 32 registers, but many of those registers are reserved for special-purposes and thus unavailable to the programmer.
   - ○ True
   - ○ False

4) ARMv8 has a divide instruction.
   - ○ True
   - ○ False

5) ARMv8 is more similar to MIPS than to ARMv7.
   - ○ True

○ False

2.18 Real stuff: RISC-V instructions

The instruction set most similar to MIPS also originated in academia. The good news is that if you know MIPS, it will be very easy to pick up RISC-V. However, RISC-V is an open architecture that is controlled by RISC-V International and not a proprietary architecture that is owned by a company like ARM, MIPS, or x86. MIPS and RISC-V share the same design philosophy, despite MIPS being 25 years more senior than RISC-V. Both also have 32-bit address versions and 64-bit address versions. To show their similarity, Figure 2.34 compares instruction formats for ARM, MIPS, and RISC-V. Here are the common features between RISC-V and MIPS:

- All instructions are 32 bit wide for both architectures.
- Both have 32 general-purpose registers, with one register being hardwired to 0.
- The only way to access memory is via load and store instructions on both architectures.
- Unlike some architectures, there are no instructions that can load or store many registers in MIPS or RISC-V.
- Both have instructions that branch if a register is equal to zero and branch if a register is not equal to zero.
- Both sets of addressing modes work for all data sizes.

One of the main differences between MIPS and RISC-V is for conditional branches other than equal or not equal. Whereas RISC-V simply provides branch instructions to compare two registers, MIPS relies on a comparison instruction that sets a register to 0 or 1 depending on whether the comparison is true. Programmers then follow that comparison instruction with a branch on equal to or not equal to zero depending on the desired outcome of the comparison. Keeping with its minimalist philosophy, MIPS only performs less than comparisons, leaving it up to the programmer to switch order of operands or to switch the condition being tested by the branch to get all the desired outcomes.

PARTICIPATION ACTIVITY 2.18.1: RISC-V instructions.

1) How many general purpose registers does RISC-V have?

   ○ 32

   ○ 64

2) RISC-V provides only less than conditional branch instruction.

- ○ True
- ○ False

3) RISC-V does not have any instructions that load multiple registers from memory at the same time.

- ○ True
- ○ False

2.19 Real stuff: x86 instructions

> "Beauty is altogether in the eye of the beholder."
> *Margaret Wolfe Hungerford, Molly Bawn, 1877*

Designers of instruction sets sometimes provide more powerful operations than those found in ARM and MIPS. The goal is generally to reduce the number of instructions executed by a program. The danger is that this reduction can occur at the cost of simplicity, increasing the time a program takes to execute because the instructions are slower. This slowness may be the result of a slower clock cycle time or of requiring more clock cycles than a simpler sequence.

The path toward operation complexity is thus fraught with peril. COD Section 2.21 (Fallacies and pitfalls) demonstrates the pitfalls of complexity.

Evolution of the Intel x86

ARM and MIPS were the vision of single small groups in 1985; the pieces of these architectures fit nicely together, and the whole architecture can be described succinctly. Such is not the case for the x86; it is the product of several independent groups who evolved the architecture over 40 years, adding new features to the original instruction set as someone might add clothing to a packed bag. Here are important x86 milestones.

- **1978**: The Intel 8086 architecture was announced as an assembly language–compatible extension of the then successful Intel 8080, an 8-bit microprocessor. The 8086 is a 16-bit architecture, with all internal registers 16 bits wide. Unlike MIPS, the registers have dedicated uses, and hence the 8086 is not considered a *general-purpose register* architecture.
- **1980**: The Intel 8087 floating-point coprocessor is announced. This architecture extends the 8086 with about 60 floating-point instructions. Instead of using registers, it relies on a stack

General-purpose register (***GPR***): A register that can be used for addresses or for data with virtually any instruction.

(see COD Section 2.21 (Historical perspective and further reading) and COD Section 3.7 (Real Stuff: Streaming SIMD extensions and advanced vector extensions in x86)).

- **1982**: The 80286 extended the 8086 architecture by increasing the address space to 24 bits, by creating an elaborate memory-mapping and protection model (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)), and by adding a few instructions to round out the instruction set and to manipulate the protection model.
- **1985**: The 80386 extended the 80286 architecture to 32 bits. In addition to a 32-bit architecture with 32-bit registers and a 32-bit address space, the 80386 added new addressing modes and additional operations. The added instructions make the 80386 nearly a general-purpose register machine. The 80386 also added paging support in addition to segmented addressing (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)). Like the 80286, the 80386 has a mode to execute 8086 programs without change.
- **1989-95**: The subsequent 80486 in 1989, Pentium in 1992, and Pentium Pro in 1995 were aimed at higher performance, with only four instructions added to the user-visible instruction set: three to help with multiprocessing (COD Chapter 6 (Parallel Processor from Client to Cloud)) and a conditional move instruction.
- **1997**: After the Pentium and Pentium Pro were shipping, Intel announced that it would expand the Pentium and the Pentium Pro architectures with MMX (Multi Media Extensions). This new set of 57 instructions uses the floating-point stack to accelerate multimedia and communication applications. MMX instructions typically operate on multiple short data elements at a time, in the tradition of *single instruction*, *multiple data* (SIMD) architectures (see COD Chapter 6 (Parallel Processor from Client to Cloud)). Pentium II did not introduce any new instructions.
- **1999**: Intel added another 70 instructions, labeled SSE (*Streaming SIMD Extensions*) as part of Pentium III. The primary changes were to add eight separate registers, double their width to 128 bits, and add a single precision floating-point data type. Hence, four 32-bit floating-point operations can be performed in parallel. To improve memory performance, SSE includes cache prefetch instructions plus streaming store instructions that bypass the caches and write directly to memory.
- **2001**: Intel added yet another 144 instructions, this time labeled SSE2. The new data type is double precision arithmetic, which allows pairs of 64-bit floating-point operations in parallel. Almost all of these 144 instructions are versions of existing MMX and SSE instructions that operate on 64 bits of data in parallel. Not only does this change enable more multimedia operations; it gives the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE registers as floating-point registers like those found in other computers. This change boosted the floating-point performance of the Pentium 4, the first microprocessor to include SSE2 instructions.
- **2003**: A company other than Intel enhanced the x86 architecture this time. AMD announced a

set of architectural extensions to increase the address space from 32 to 64 bits. Similar to the transition from a 16- to 32-bit address space in 1985 with the 80386, AMD64 widens all registers to 64 bits. It also increases the number of registers to 16 and increases the number of 128-bit SSE registers to 16. The primary ISA change comes from adding a new mode called *long mode* that redefines the execution of all x86 instructions with 64-bit addresses and data. To address the larger number of registers, it adds a new prefix to instructions. Depending how you count, long mode also adds four to ten new instructions and drops 27 old ones. PC-relative data addressing is another extension. AMD64 still has a mode that is identical to x86 (*legacy mode*) plus a mode that restricts user programs to x86 but allows operating systems to use AMD64 (*compatibility mode*). These modes allowed a more graceful transition to 64-bit addressing than the HP/Intel IA-64 architecture.

- **2004**: Intel capitulates and embraces AMD64, relabeling it *Extended Memory 64 Technology* (EM64T). The major difference is that Intel added a 128-bit atomic compare and swap instruction, which probably should have been included in AMD64. At the same time, Intel announced another generation of media extensions. SSE3 adds 13 instructions to support complex arithmetic, graphics operations on arrays of structures, video encoding, floating-point conversion, and thread synchronization (see COD Section 2.11 (Parallelism and instructions: Synchronization)). AMD added SSE3 in subsequent chips and the missing atomic swap instruction to AMD64 to maintain binary compatibility with Intel.
- **2006**: Intel announces 54 new instructions as part of the SSE4 instruction set extensions. These extensions perform tweaks like sum of absolute differences, dot products for arrays of structures, sign or zero extension of narrow data to wider sizes, population count, and so on. They also added support for virtual machines (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)).
- **2007**: AMD announces 170 instructions as part of SSE5, including 46 instructions of the base instruction set that adds three operand instructions like MIPS.
- **2011**: Intel ships the Advanced Vector Extension that expands the SSE register width from 128 to 256 bits, thereby redefining about 250 instructions and adding 128 new instructions.
- **2015**: Intel ships AVX-512, which widens the registers and operations from 256 to 512 bits, and once again redefining hundreds of instructions as well as adding many more.

This history illustrates the impact of the "golden handcuffs" of compatibility on the x86, as the existing software base at each step was too important to jeopardize with significant architectural changes.

Whatever the artistic failures of the x86, keep in mind that this instruction set largely drove the PC generation of computers and still dominates the Cloud portion of the PostPC era. Manufacturing 250M x86 chips per year may seem small compared to billions of ARMv7 chips, but many companies would love to control such a market. Nevertheless, this checkered ancestry has led to an architecture that is difficult to explain and impossible to love.

Brace yourself for what you are about to see! Do *not* try to read this section with the care you would need to write x86 programs; the goal instead is to give you familiarity with the strengths and weaknesses of the world's most popular desktop architecture.

Rather than show the entire 16-bit, 32-bit, and 64-bit instruction set, in this section we concentrate on the 32-bit subset that originated with the 80386. We start our explanation with the registers and addressing modes, move on to the integer operations, and conclude with an examination of instruction encoding.

PARTICIPATION ACTIVITY 2.19.1: Evolution of the x86 instruction set.

1) _____ is the abbreviation for a register that can be used for addresses or data with virtually any instruction.

Check **Show answer**

2) The 1985 Intel 80386 extended the previous architecture to ____ bits.

Check **Show answer**

3) _____ is the abbreviation for a set of 70 instructions that provided eight 128-bit registers to the Pentium III processor.

Check **Show answer**

4) In 2003, a company called _____ released a 64-bit version of the x86 instruction set.

Check **Show answer**

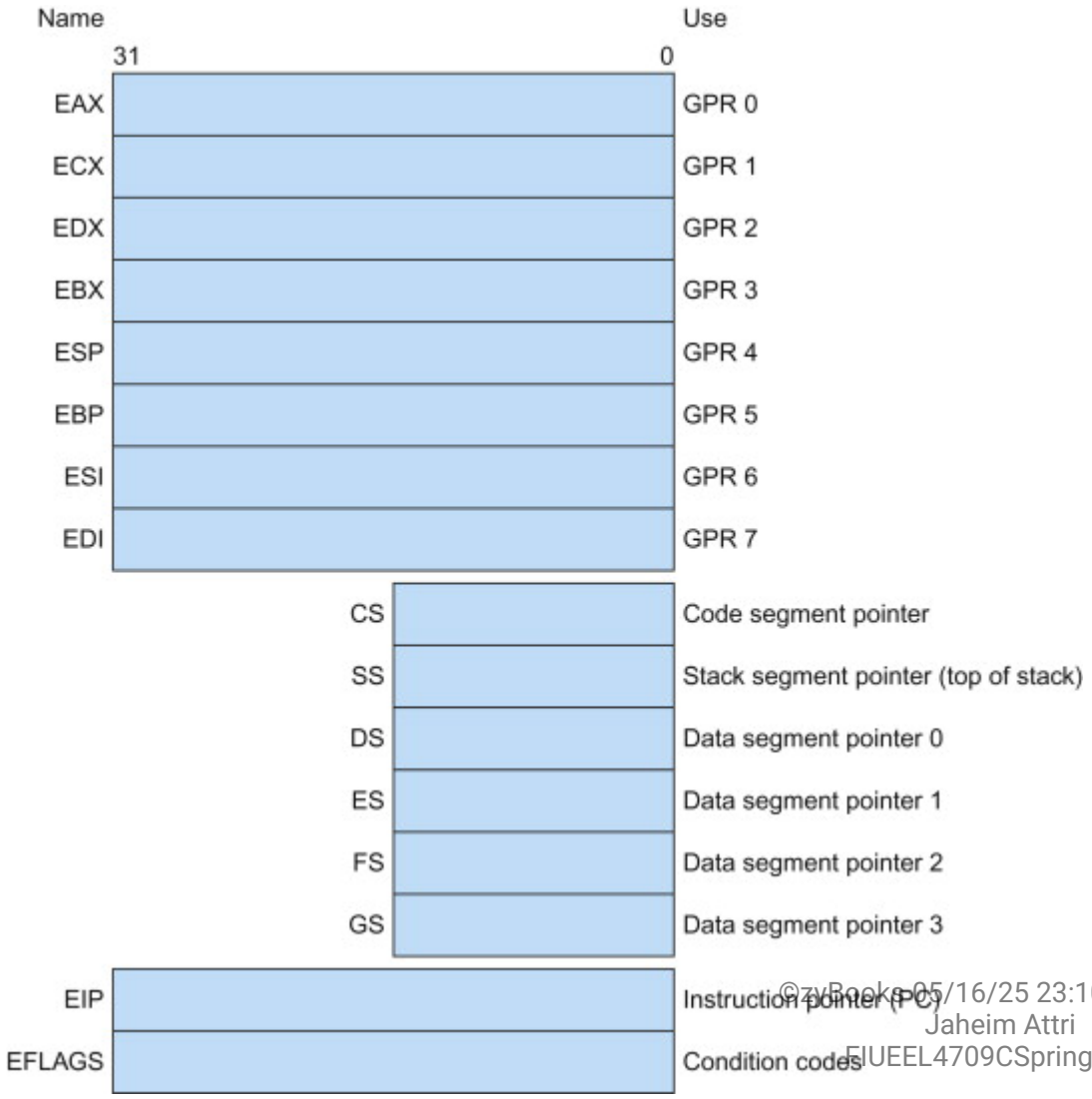
©zyBooks 05/16/25 23:10 2475274
Jaheim Attri
FIUEEL4709CSpring2025

x86 registers and data addressing modes

The registers of the 80386 show the evolution of the instruction set (see figure below). The 80386 extended all 16-bit registers (except the segment registers) to 32 bits, prefixing an E to their name to indicate the 32-bit version. We'll refer to them generically as GPRs (*general-purpose registers*). The 80386 contains only eight GPRs. This means MIPS programs can use four times as many and ARMv7 twice as many.

Figure 2.19.1: The 80386 register set (COD Figure 2.36).

Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers.



The figure below shows the arithmetic, logical, and data transfer instructions are two-operand instructions. There are two important differences here. The x86 arithmetic and logical instructions must have one operand act as both a source and a destination; ARMv7 and MIPS allow separate

registers for source and destination. This restriction puts more pressure on the limited registers, since one source register must be modified. The second important difference is that one of the operands can be in memory. Thus, virtually any instruction may have one operand in memory, unlike ARMv7 and MIPS.

Figure 2.19.2: Instruction types for the arithmetic, logical, and data transfer instructions (COD Figure 2.37).

The x86 allows the combinations shown. The only restriction is the absence of a memory-memory mode. Immediates may be 8, 16, or 32 bits in length; a register is any one of the 14 major registers in the figure above (not EIP or EFLAGS).

Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

Data memory-addressing modes, described in detail below, offer two sizes of addresses within the instruction. These so-called displacements can be 8 bits or 32 bits.

Although a memory operand can use any addressing mode, there are restrictions on which *registers* can be used in a mode. The figure below shows the x86 addressing modes and which GPRs cannot be used with each mode, as well as how to get the same effect using MIPS instructions.

Figure 2.19.3: x86 32-bit addressing modes with register restrictions and the equivalent MIPS code (COD Figure 2.38).

The Base plus Scaled Index addressing mode, not found in ARM or MIPS, is included to avoid the multiples of 4 (scale factor of 2) to turn an index in a register into a byte address (see COD Figures 2.25 (MIPS assembly code of the procedure swap) and 2.27 (MIPS assembly version of procedure sort)). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. A scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel

gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	Not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	Not ESP	`lw $s0,100($s1) # <= 16-bit # displacement`
Base plus scaled index	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,0($t0)`
Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,100($t0) # <-16-bit # displacement`

PARTICIPATION ACTIVITY 2.19.2: x86 registers and addressing modes.

1) An x86 arithmetic and logical instruction can have one operand act as both a source and destination operand, but more than one operand is allowed.
   - ○ True
   - ○ False

2) An x86 instruction can have an operand in memory.
   - ○ True
   - ○ False

3) The base plus scaled index is an addressing mode found in MIPS, ARM, and x86 that can turn an array index in a register, or index register, into a byte address by multiplying the index by 4.
   - ○ True
   - ○ False

x86 integer operations

The 8086 provides support for both 8-bit (*byte*) and 16-bit (*word*) data types. The 80386 adds 32-bit addresses and data (*double words*) in the x86. (AMD64 adds 64-bit addresses and data, called *quad words*; we'll stick to the 80386 in this section.) The data type distinctions apply to register operations as well as memory accesses.

Almost every operation works on both 8-bit data and on one longer data size. That size is determined by the mode and is either 16 bits or 32 bits.

Clearly, some programs want to operate on data of all three sizes, so the 80386 architects provided a convenient way to specify each version without expanding code size significantly. They decided that either 16-bit or 32-bit data dominates most programs, and so it made sense to be able to set a default large size. This default data size is set by a bit in the code segment register. To override the default data size, an 8-bit *prefix* is attached to the instruction to tell the machine to use the other large size for this instruction.

The prefix solution was borrowed from the 8086, which allows multiple prefixes to modify instruction behavior. The three original prefixes override the default segment register, lock the bus to support synchronization, or repeat the following instruction until the register ECX counts down to 0. This last prefix was intended to be paired with a byte move instruction to move a variable number of bytes. The 80386 also added a prefix to override the default address size.

The x86 integer operations can be divided into four major classes:

1. Data movement instructions, including move, push, and pop
2. Arithmetic and logic instructions, including test, integer, and decimal arithmetic operations
3. Control flow, including conditional branches, unconditional jumps, calls, and returns
4. String instructions, including string move and string compare

The first two categories are unremarkable, except that the arithmetic and logic instruction operations allow the destination to be either a register or a memory location. The figure below shows some typical x86 instructions and their functions.

Figure 2.19.4: Some typical x86 instructions and their functions (COD Figure 2.39).

A list of frequent operations appears in the figure below. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

Instruction	Function
je name	if equal(condition code) {EIP=name}; EIP-128 <= name < EIP+128
jmp name	EIP=name
call name	SP=SP-4; M[SP]=EIP+5; EIP=name;
movw EBX,[EDI+45]	EBX=M[EDI+45]
push ESI	SP=SP-4; M[SP]=ESI

pop EDI	EDI=M[SP]; SP=SP+4
add EAX,#6765	EAX= EAX+6765
test EDX,#42	Set condition code (flags) with EDX and 42
movsl	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

Conditional branches on the x86 are based on *condition codes* or *flags*, like ARMv7. Condition codes are set as a side effect of an operation; most are used to compare the value of a result to 0. Branches then test the condition codes. PC-relative branch addresses must be specified in the number of bytes, since unlike ARMv7 and MIPS, 80386 instructions are not all 4 bytes in length.

String instructions are part of the 8080 ancestry of the x86 and are not commonly executed in most programs. They are often slower than equivalent software routines.

The figure below lists some of the integer x86 instructions. Many of the instructions are available in both byte and word formats.

Figure 2.19.5: Some typical operations on the x86 (COD Figure 2.40).

Many operations use register-memory format, where either the source or the destination may be memory and the other may be a register or immediate operand.

Instruction	Meaning
Control	**Conditional and unconditional branches**
jnz, jz	Jump if condition to EIP + 8-bit offset; JNE (forJNZ), JE (for JZ) are alternative names
jmp	Unconditional jump—8-bit or 16-bit offset
call	Subroutine call—16-bit offset; return address pushed onto stack
ret	Pops return address from stack and jumps to it
loop	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX ≠ 0
Data transfer	**Move data between registers or between register and memory**
move	Move between two registers or between register and memory
push, pop	Push source operand on stack; pop operand from stack top to a register
les	Load ES and one of the GPRs from memory
Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
add, sub	Add source to destination; subtract source from destination; register-memory format
cmp	Compare source and destination; register-memory format
shl, shr, rcr	Shift left; shift logical right; rotate right with carry condition code as fill
cbw	Convert byte in eight rightmost bits of EAX to 16-bit word in right of EAX
test	Logical AND of source and destination sets condition codes
inc, dec	Increment destination, decrement destination
or, xor	Logical OR; exclusive OR; register-memory format
String	**Move between string operands; length given by a repeat prefix**
movs	Copies from string source to destination by incrementing ESI and EDI; may be

	repeated
lods	Loads a byte, word, or doubleword of a string into the EAX register

PARTICIPATION ACTIVITY 2.19.3: x86 instructions.

How to use this tool

jmp **jnz** **les** **movs** **lods** **move**

Load ES register and a GPR from memory

Unconditional jump

Copy string from source to destination

Jump if not zero

Move data between two registers

Load a byte, word, or doubleword of a string into the EAX register

Reset

x86 instruction encoding

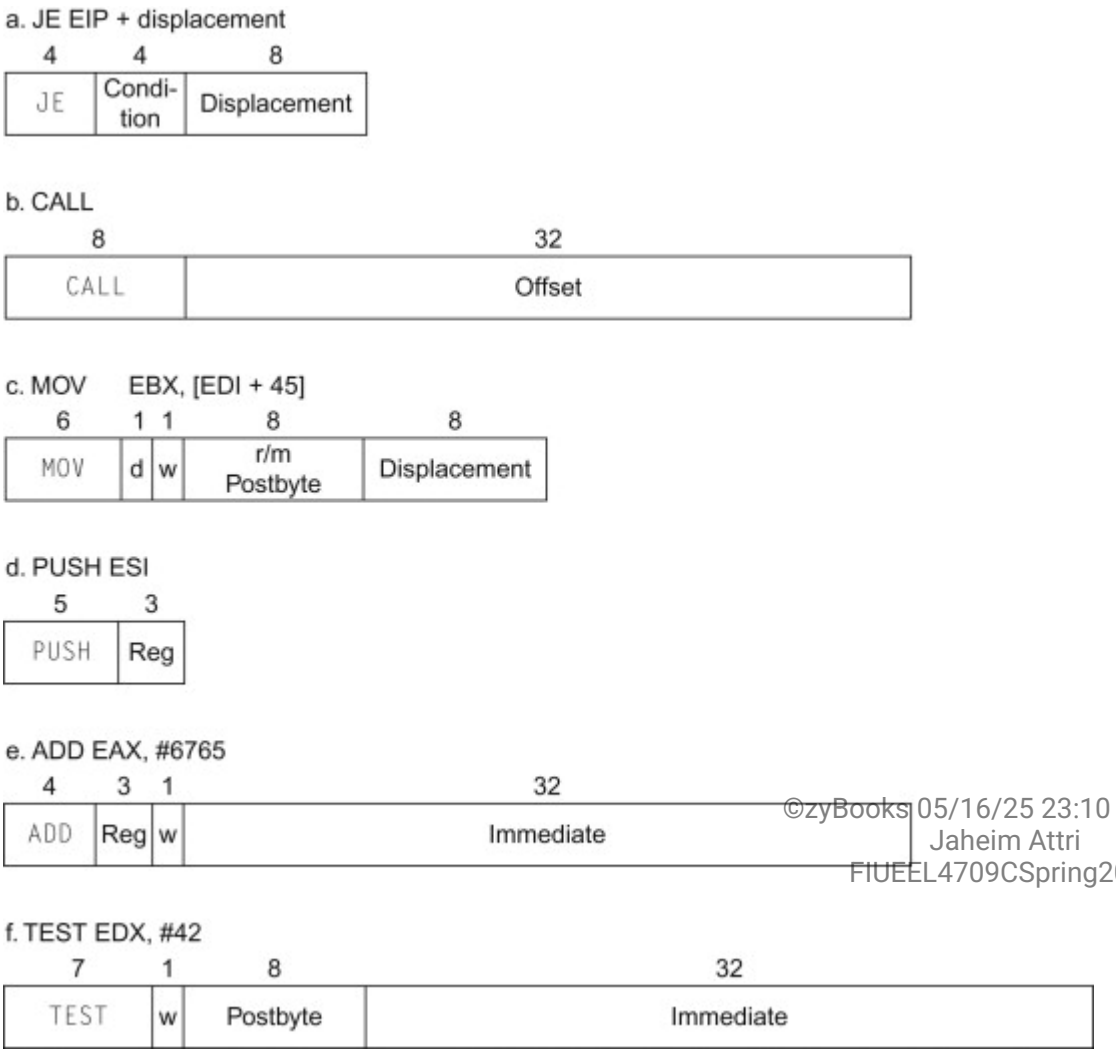
Saving the worst for last, the encoding of instructions in the 80386 is complex, with many different instruction formats. Instructions for the 80386 may vary from 1 byte, when there are no operands, up to 15 bytes.

The figure below shows the instruction format for several of the example instructions in the (Some typical x86 instructions and their functions). The opcode byte usually contains a bit saying whether the operand is 8 bits or 32 bits. For some instructions, the opcode may include the addressing mode and the register; this is true in many instructions that have the form "register = register op immediate". Other instructions use a "postbyte" or extra opcode byte, labeled "mod, reg, r/m", which contains the addressing mode information. This postbyte is used for many of the instructions that

address memory. The base plus scaled index mode uses a second postbyte, labeled "sc, index, base".

Figure 2.19.6: Typical x86 instruction formats (COD Figure 2.41).

The figure below shows the encoding of the postbyte. Many instructions contain the 1-bit field w, which says whether the operation is a byte or a double word. The d field in `MOV` is used in instructions that may move to or from memory and shows the direction of the move. The `ADD` instruction requires 32 bits for the immediate field, because in 32-bit mode, the immediates are either 8 bits or 32 bits. The immediate field in the `TEST` is 32 bits long because there is no 8-bit immediate for test in 32-bit mode. Overall, instructions may vary from 1 to 15 bytes in length. The long length comes from extra 1-byte prefixes, having both a 4-byte immediate and a 4-byte displacement address, using an opcode of 2 bytes, and using the scaled index mode specifier, which adds another byte.



The figure below shows the encoding of the two postbyte address specifiers for both 16-bit and 32-bit mode. Unfortunately, to understand fully which registers and which addressing modes are available, you need to see the encoding of all addressing modes and sometimes even the encoding of the instructions.

Figure 2.19.7: The encoding of the first address specifier of the x86: mod, reg, r/m (COD Figure 2.42).

The first four columns show the encoding of the 3-bit reg field, which depends on the w bit from the opcode and whether the machine is in 16-bit mode (8086) or 32-bit mode (80386). The remaining columns explain the mod and r/m fields. The meaning of the 3-bit r/m field depends on the value in the 2-bit mod field and the address size. Basically, the registers used in the address calculation are listed in the sixth and seventh columns, under mod = 0, with mod = 1 adding an 8-bit displacement and mod = 2 adding a 16-bit or 32-bit displacement, depending on the address mode. The exceptions are 1) r/m = 6 when mod = 1 or mod = 2 in 16-bit mode selects BP plus the displacement; 2) r/m = 5 when mod = 1 or mod = 2 in 32-bit mode selects EBP plus displacement; and 3) r/m = 4 in 32-bit mode when mod does not equal 3, where (sib) means use the scaled index mode shown in the figure above (x86 32-bit addressing modes with register restrictions and the equivalent MIPS code). When mod = 3, the r/m field indicates a register, using the same encoding as the reg field combined with the w bit.

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	AH	SP	ESP	4	addr=SI	*=(sib)*	SI+disp8	*(sib)*+disp8	SI+disp8	*(sib)*+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

x86 Conclusion

Intel had a 16-bit microprocessor two years before its competitors' more elegant architectures, such as the Motorola 68000, and this head start led to the selection of the 8086 as the CPU for the IBM PC. Intel engineers generally acknowledge that the x86 is more difficult to build than computers like ARMv7 and MIPS, but the large market meant in the PC era that AMD and Intel could afford more resources to help overcome the added complexity. What the x86 lacks in style, it made up for in market size, making it beautiful from the right perspective.

Its saving grace is that the most frequently used x86 architectural components are not too difficult to implement, as AMD and Intel have demonstrated by rapidly improving performance of integer programs since 1978. To get that performance, compilers must avoid the portions of the architecture that are hard to implement fast.

In the PostPC era, however, despite considerable architectural and manufacturing expertise, x86 has not yet been competitive in the personal mobile device.

2.20 Going faster: Matrix multiply in C

We start by rewriting the Python program from the Going faster: Matrix multiply in Python section in a previous chapter. The figure below shows a version of a matrix-matrix multiply written in C. This program is commonly called *DGEMM*, which stands for Double precision GEneral Matrix Multiply. Because we are passing the matrix dimension as the parameter n, this version of DGEMM uses single dimensional versions of matrices `C`, `A`, and `B` and address arithmetic to get better performance instead of using the more intuitive two-dimensional arrays that we saw in Python. The comments in the figure remind us of this more intuitive notation. The next figure after that shows the x 86 assembly language output for the inner loop of the figure below. The five floating-point instructions start with a `v` and include `sd` in the name, which stands for scalar double precision.

The final figure shows the performance of the C program as were vary the optimization parameter as compared to the Python program. Even the unoptimized C program is dramatically faster. As we increase optimization levels, it gets even faster; the cost is longer compile time. The reasons for the speed-up are fundamentally using a compiler instead of an interpreter and because the type declarations of C allow the compiler to produce much more efficient code.

Figure 2.20.1: C version of a double precision matrix multiply, widely known as DGEMM for Double-precision General Matrix Multiply (GEMM) (COD Figure 2.43).

```
1  void dgemm (int n, double* A, double* B, double* C)
2  {
3        for (int i = 0; i < n; ++i)
4                for (int j = 0; j < n; ++j)
5                  {
6                     double cij = C[i+j*n]; /* cij = C[i][j] */
7                        for (int k = 0; k < n; ++k)
8                           cij += A[i+k*n] * B[k+j*n]; /* cij += A[i]|
9                   C[i+j*n] = cij; /* C[i][j] = cij */
```

```
10                  }
11   }
```

Figure 2.20.2: The x86 assembly for the body of the nested loops are generated by compiling the unoptimized C code in Figure 2.43 using gcc with -O3 optimization flags (COD Figure 2.44).


```
 1 vmovsd        (%r10),%xmm0                # Load 1 element1 of C into %
 2 mov           %rsi,%rcx                   # register %rcx = %rsi
 3 xor           %eax,%eax                   # register %eax = 0
 4 vmovsd        (%rcx),%xmml                # Load 1 element of B into %x
 5 add           %r9,%rcx                    # register %rcx = %rcx + %r9
 6 vmulsd        (%r8,%rax,8),%xmm1,%xmm1    # Multiply %xmm1, element of
 7 add           $0x1,%rax                   # register %rax = %rax + 1
 8 cmp           %eax,%edi                   # compare %eax to % edi
 9 vaddsd        %xmm1,%xmm0,%xmm0           # Add %xmm1, %xmm0
10 jg            30 <dgemm+0x30>             # jump if %eax > %edi
11 add           $0x1, %r11                  # register %r11 = %r11 + 1
12 vmovsd        %xmm0, (%r10)               # Store %xmm0 into C element
```

Figure 2.20.3: COD Figure 2.45.

Performance over the Python program for the C program in COD Figure 2.43 as we increase the lever of optimization of the GCC C compiler, with -O0 doing no code size or performance optimization, thereby improving compile time and -O3 being the most aggressive for run time and code size. In this case, -O2 and -O3 produce the same x86 code. Most programmers use -O2 as the default compiler flag. GCC also offers a -Os optimization option, which aims at smallest code size.

-O0 (fastest compile time)	-O1	-O2	-O3 (fastest run time)
77	208	212	212

PARTICIPATION ACTIVITY 2.20.1: Matrix multiply in C.

1) An x86 assembly instruction with sd in the name stands for _____.

- ○ scalar double-precision
- ○ static datapath

2) As we increase optimization levels, the program gets even faster; the compile time _____.

- ○ decreases
- ○ increases

2.21 Fallacies and pitfalls

Fallacy: More powerful instructions mean higher performance.

Part of the power of the Intel x86 is the prefixes that can modify the execution of the following instruction. One prefix can repeat the following instruction until a counter counts down to 0. Thus, to move data in memory, it would seem that the natural instruction sequence is to use move with the repeat prefix to perform 32-bit memory-to-memory moves.

An alternative method, which uses the standard instructions found in all computers, is to load the data into the registers and then store the registers back to memory. This second version of this program, with the code replicated to reduce loop overhead, copies at about 1.5 times as fast. A third version, which uses the larger floating-point registers instead of the integer registers of the x86, copies at about 2.0 times as fast than the complex move instruction.

Fallacy: Write in assembly language to obtain the highest performance.

At one time compilers for programming languages produced naïve instruction sequences; the increasing sophistication of compilers means the gap between compiled code and code produced by hand is closing fast. In fact, to compete with current compilers, the assembly language programmer needs to understand the concepts in COD Chapters 4 (The Processor) and 5 (Large and Fast: Exploiting Memory Hierarchy) thoroughly (processor pipelining and memory hierarchy).

This battle between compilers and assembly language coders is another situation in which humans are losing ground. For example, C offers the programmer a chance to give a hint to the compiler about which variables to keep in registers versus spilled to memory. When compilers were poor at register allocation, such hints were vital to performance. In fact, some old C textbooks spent a fair amount of time giving examples that effectively use register hints. Today's C compilers generally ignore such hints, because the compiler does a better job at allocation than the

programmer does.

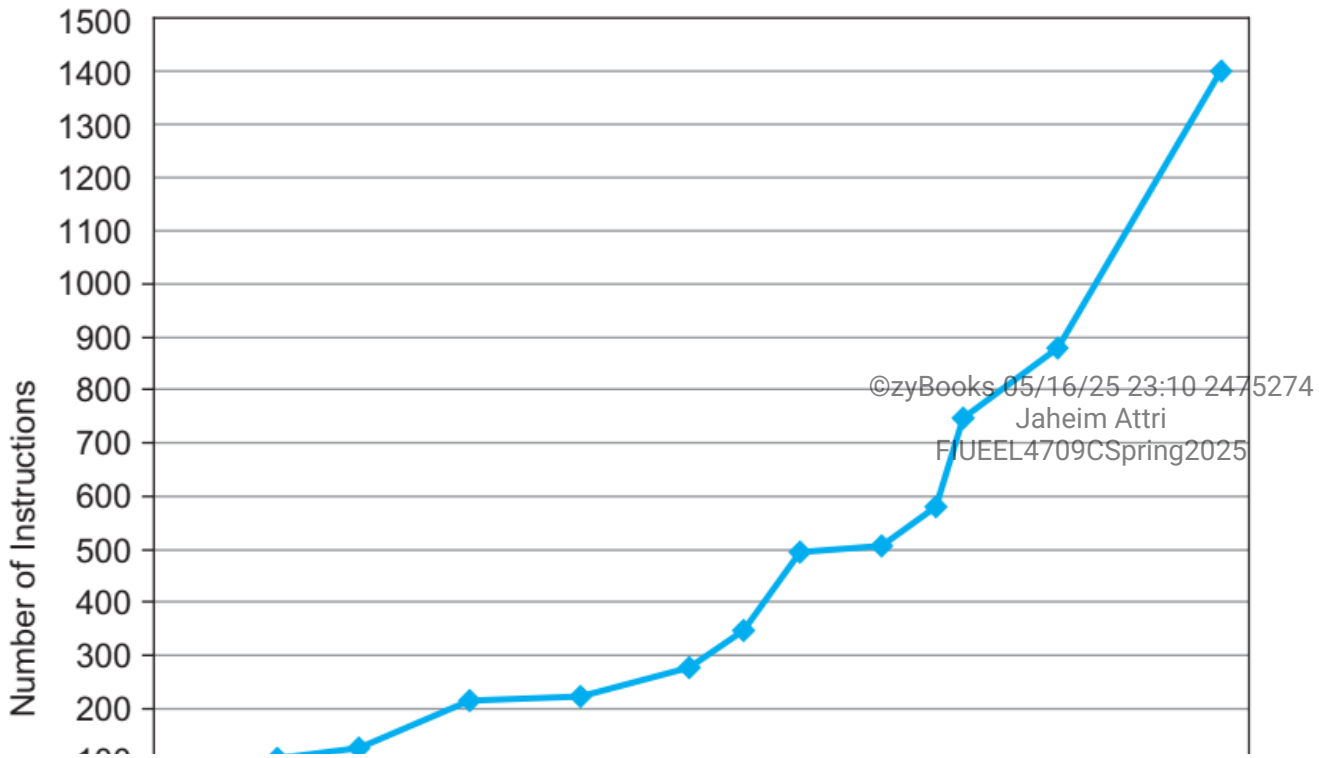
Even *if* writing by hand resulted in faster code, the dangers of writing in assembly language are the longer time spent coding and debugging, the loss in portability, and the difficulty of maintaining such code. One of the few widely accepted axioms of software engineering is that coding takes longer if you write more lines, and it clearly takes many more lines to write a program in assembly language than in C or Java. Moreover, once it is coded, the next danger is that it will become a popular program. Such programs always live longer than expected, meaning that someone will have to update the code over several years and make it work with new releases of operating systems and new models of machines. Writing in higher-level language instead of assembly language not only allows future compilers to tailor the code to future machines; it also makes the software easier to maintain and allows the program to run on more brands of computers.

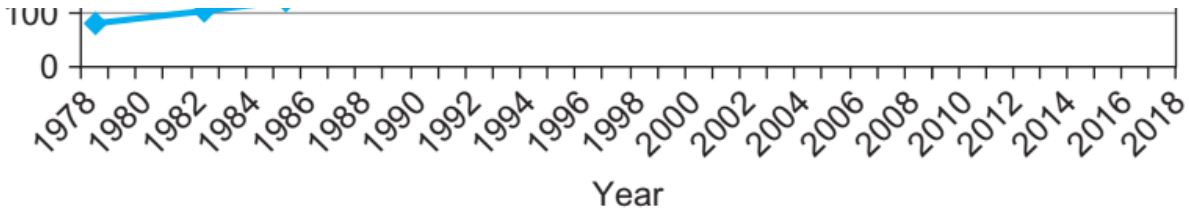
Fallacy: The importance of commercial binary compatibility means successful instruction sets don't change.

While backwards binary compatibility is sacrosanct, the figure below shows that the x86 architecture has grown dramatically. The average is more than one instruction per month over its 40-year lifetime!

Figure 2.21.1: Growth of x86 instruction set over time (COD Figure 2.46).

While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.





Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by one.

Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by one instead of by the word size in bytes. Forewarned is forearmed!

Pitfall: Using a pointer to an automatic variable outside its defining procedure.

A common mistake in dealing with pointers is to pass a result from a procedure that includes a pointer to an array that is local to that procedure. Following the stack discipline in COD Figure 2.12 (Illustration of the stack allocation …), the memory that contains the local array will be reused as soon as the procedure returns. Pointers to automatic variables can lead to chaos.

PARTICIPATION ACTIVITY 2.21.1: Pitfall: Pointing to a local variable.

```
int* GetElement()
{
   // Local array
   int array[3] = {15, 37, 42};

   // Point to array's first element
   int* p = &array[0];

   // Return pointer
   return p;
}


int main()
{
   // Call procedure GetElement()
   int* arrayElement = GetElement();

   // Print value of array's first element
   printf("%d", *arrayElement);
}
```

Memory

p ▶

p no longer points to array

Animation content:

Static figure:
Begin C code:

```
int* GetElement()
{
  // Local array
  int array[3] = {15, 37, 42};

  // Point to array's first element
  int* p = &array[0];

  // Return pointer
  return p;
}

int main()
{
  // Call procedure GetElement()
  int* arrayElement = GetElement();

  // Print value of array's first element
  printf("%d", *arrayElement);
}
```

End C code.

An empty memory block is displayed, with a pointer p pointing to the memory block.
The line of code
printf("%d", *arrayElement);
is highlighted, and the phrase p no longer points to array is displayed next to this line of code.

Step 1: The main procedure calls procedure GetElement(), which declares a local array.
The elements of the array are placed into memory.
An empty memory block is shown.
The lines of code

```
int main()
{
  // Call procedure GetElement()
  int* arrayElement = GetElement();
```

are highlighted.
Next, the lines of code

```
int* GetElement()
{
   // Local array
   int array[3] = {15, 37, 42};
```

are highlighted.
The memory block now contains 3 consecutive memory locations with the values 15, 37, and 42 respectively.

Step 2: Pointer p points to the first element of the array.
The pointer is returned at the end of procedure.
The line of code
int* p = &array[0];
is highlighted.
In the memory block, the letter p with an arrow pointing to the memory location with the value 15, is displayed.
The line of code
return p;
is highlighted.

Step 3: After the procedure returns, the memory that contains the local array may be reused to make room for new data. Pointer p may no longer point to the first element of the local array.
The line of code
int* arrayElement = GetElement();
is highlighted.
In the memory block, the values 15, 37, and 42 are removed from the block.
Next, the line of code
printf("%d", *arrayElement);
is highlighted, and the phrase
p no longer points to array
is displayed next to this line of code.

Animation captions:

1. The main procedure calls procedure GetElement(), which declares a local array. The elements of the array are placed into memory.
2. Pointer p points to the first element of the array. The pointer is returned at the end of procedure.
3. After the procedure returns, the memory that contains the local array may be reused to make room for new data. Pointer p may no longer point to the first element of the local array.

PARTICIPATION ACTIVITY 2.21.2: Fallicies and pitfalls.

1) Using standard instructions can result in higher performance than using powerful instructions crafted specifically for an architecture.
   - ○ True
   - ○ False

2) Current C compilers tend to ignore coded hints regarding register allocation.
   - ○ True
   - ○ False

3) Writing code in a high-level language, such as C, results in more lines of code than writing in assembly language. Therefore, C programs take longer to write and debug.
   - ○ True
   - ○ False

4) Assembly language programs are not easily portable because of the many different architectures of current and future computers.
   - ○ True
   - ○ False

5) Assuming the addresses are in registers, if the size of a word is 4 bytes, the address of the next word can be found by adding 1 to the current word's address.
   - ○ True
   - ○ False

©zyBooks 05/16/25 23:10 2475274
Jaheim Attri
FIUEEL4709CSpring2025

6) Memory containing variables local to a procedure may be reused as soon as the procedure returns.

- ○ True
- ○ False

2.22 Concluding remarks

> "Less is more."
> *Robert Browning, Andrea del Sarto, 1855*

The two principles of the *stored-program* computer are the use of instructions that are indistinguishable from numbers and the use of alterable memory for programs. These principles allow a single machine to aid environmental scientists, financial advisers, and novelists in their specialties. The selection of a set of instructions that the machine can understand demands a delicate balance among the number of instructions needed to execute a program, the number of clock cycles needed by an instruction, and the speed of the clock. As illustrated in this chapter, three design principles guide the authors of instruction sets in making that delicate balance:

1. *Simplicity favors regularity.* Regularity motivates many features of the MIPS instruction set: keeping all instructions a single size, always requiring three register operands in arithmetic instructions, and keeping the register fields in the same place in each instruction format.
2. *Smaller is faster.* The desire for speed is the reason that MIPS has 32 registers rather than many more.
3. *Good design demands good compromises.* One MIPS example was the compromise between providing for larger addresses and constants in instructions and keeping all instructions the same length.

Another big idea in this chapter is that numbers have no inherent type. A given bit pattern can represent an integer number or a string or a color or even an instruction. It is the program that determines the type of the data.

We also saw the great idea of making the **common case fast** applied to instruction sets as well as computer architecture. Examples of making the common MIPS case fast include PC-relative addressing for conditional branches and immediate addressing for larger constant operands.

Above this machine level is assembly language, a language that humans can read. The assembler translates it into the binary numbers that machines can understand, and it even "extends" the instruction set by creating symbolic instructions that aren't in the hardware. For instance, constants or addresses that are too big are broken into properly sized pieces, common variations of

instructions are given their own name, and so on. The figure below lists the MIPS instructions we have covered so far, both real and pseudoinstructions. Hiding details from the higher level is another example of the great idea of **abstraction**.

Figure 2.22.1: The MIPS instruction set covered so far, with the real MIPS instructions on the left and the pseudoinstructions on the right (COD Figure 2.47).

COD Appendix A (Assemblers, Linkers, and the SPIM Simulator) (Section A.10 (MIPS R2000 Assembly Language)) describes the full MIPS architecture. COD Figure 2.1 (MIPS assembly language revealed in this chapter) shows more details of the MIPS architecture revealed in this chapter.

MIPS instructions	Name	Format	Pseudo MIPS	Name	Format
add	add	R	move	move	R
subtract	sub	R	multiply	mult	R
add immediate	addi	I	multiply immediate	multi	I
load word	lw	I	load immediate	li	I
store word	sw	I	branch less than	blt	I
load half	lh	I	branch less than or equal	ble	I
load half unsigned	lhu	I			
store half	sh	I	branch greater than	bgt	I
load byte	lb	I	branch greater than or equal	bge	I
load byte unsigned	lbu	I			
store byte	sb	I			
load linked	ll	I			
store conditional	sc	I			
load upper immediate	lui	I			
and	and	R			
or	or	R			
nor	nor	R			
and immediate	andi	I			
or immediate	ori	I			
shift left logical	sll	R			
shift right logical	srl	R			
branch on equal	beq	I			
branch on not equal	bne	I			
set less than	slt	R			
set less than immediate	slti	I			
set less than immediate unsigned	sltiu	I			
jump	j	J			

jump register	jr	R
jump and link	jal	J

Each category of MIPS instructions is associated with constructs that appear in programming languages:

- Arithmetic instructions correspond to the operations found in assignment statements.
- Transfer instructions are most likely to occur when dealing with data structures like arrays or structures.
- Conditional branches are used in *if* statements and in loops.
- Unconditional jumps are used in procedure calls and returns and for *case/switch* statements.

These instructions are not born equal; the popularity of the few dominates the many. For example, the figure below shows the popularity of each class of instructions for SPEC CPU2006. The varying popularity of instructions plays an important role in the chapters about datapath, control, and pipelining.

Figure 2.22.2: MIPS instruction classes, examples, correspondence to high-level program language constructs, and percentage of MIPS instructions executed by category for the average integer and floating point SPEC CPU2006 benchmarks (COD Figure 2.48).

Instruction class	MIPS examples	HLL correspondence	Frequency	
			Integer	Ft. pt.
Arithmetic	add, sub, addi	Operations in assignment statement s	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	References to data structures, such as arrays	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	Operations in assignment statement s	12%	4%
Conditional branch	beq, bne, slt, slti, sltiu	*If* statements and loops	34%	8%
Jump	j, jr, jal	Procedure calls, returns, and *case/switch* statements	2%	0%

After we explain computer arithmetic in COD Chapter 3 (Arithmetic for Computers), we reveal the rest of the MIPS instruction set architecture.

PARTICIPATION ACTIVITY 2.22.1: MIPS Instructions.

1) Keeping all instructions the same size is a compromise the MIPS instruction set makes to maintain simplicity and

regularity.

- ○ True
- ○ False

2) Adding more registers to MIPS would likely improve performance.

- ○ True
- ○ False

3) PC-relative addressing likely has no influence on the performance of the MIPS architecture.

- ○ True
- ○ False

4) Assembly language adds a level of abstraction to a computer by allowing programmers to write code that is translated into binary numbers that the computer can read.

- ○ True
- ○ False

5) MIPS instructions and pseudoinstructions correspond to operations and constructs used in higher-level programming languages.

- ○ True
- ○ False

2.23 Historical perspective and further reading

This section surveys the history of instruction set architectures over time, and we give a short history of programming languages and compilers. ISAs include accumulator architectures, general-purpose register architectures, stack architectures, and a brief history of ARM and the x86. We also review the controversial subjects of high-level-language computer architectures and reduced instruction set computer architectures. The history of programming languages includes Fortran, Lisp, Algol, C, Cobol, Pascal, Simula, Smalltalk, C++, and Java, and the history of compilers includes

the key milestones and the pioneers who achieved them.

Accumulator architectures

Hardware was precious in the earliest stored-program computers. Consequently, computer pioneers could not afford the number of registers found in today's architectures. In fact, these architectures had a single register for arithmetic instructions. Since all operations would accumulate in a single register, it was called the *accumulator*, and this style of instruction set is given the same name. For example, EDSAC in 1949 had a single accumulator.

> ***Accumulator***: Archaic term for register. On-line use of it as a synonym for "register" is a fairly reliable indication that the user has been around quite a while.
>
> Eric Raymond, The New Hacker's Dictionary, 1991

The three-operand format of MIPS suggests that a single register is at least two registers shy of our needs. Having the accumulator as both a source operand *and* the destination of the operation fills part of the shortfall, but it still leaves us one operand short. That final operand is found in memory. Accumulator architectures have the memory-based operand-addressing mode suggested earlier. It follows that the add instruction of an accumulator instruction set would look like this:

```
add 200
```

This instruction means add the accumulator to the word in memory at address 200 and place the sum back into the accumulator. No registers are specified because the accumulator is known to be both a source and a destination of the operation.

The next step in the evolution of instruction sets was the addition of registers dedicated to specific operations. Hence, registers might be included to act as indices for array references in data transfer instructions, to act as separate accumulators for multiply or divide instructions, and to serve as the top-of-stack pointer. Perhaps the best-known example of this style of instruction set is found in the Intel 8086. This style of instruction set is labeled *extended accumulator*, *dedicated register*, or *special-purpose register*. Like the single-register accumulator architectures, one operand may be in memory for arithmetic instructions. Like the MIPS architecture, however, there are also instructions where all the operands are registers.

General-purpose register architectures

The generalization of the dedicated-register architecture allows all the registers to be used for any purpose, hence the name general-purpose register. MIPS is an example of a general-purpose register architecture. This style of instruction set may be further divided into those that allow one operand to be in memory (as found in accumulator architectures), called a register-memory architecture, and those that demand that operands always be in registers, called either a *load-store*

or a *register-register* architecture. The figure below shows a history of the number of registers in some popular computers.

Load-store architecture: Also called ***register-register*** architecture. An instruction set architecture in which all operations are between registers and data memory may only be accessed via loads or stores.

Figure 2.23.1: The number of general-purpose registers in popular architectures over the years (COD Figure e2.23.1).

Machine	Number of general-purpose registers	Architectural style	Year
EDSAC	1	Accumulator	1949
IBM 701	1	Accumulator	1953
CDC 6600	8	Load-store	1963
IBM 360	16	Register-memory	1964
DEC PDP-8	1	Accumulator	1965
DEC PDP-11	8	Register-memory	1970
Intel 8008	1	Accumulator	1972
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
Intel 80386	8	Register-memory	1985
ARM	16	Load-store	1985
MIPS	32	Load-store	1985
HP PA-RISC	32	Load-store	1986
SPARC	32	Load-store	1987
PowerPC	32	Load-store	1992
DEC Alpha	32	Load-store	1992
HP/Intel IA-64	128	Load-store	2001
AMD64 (EMT64)	16	Register-memory	2003

The first load-store architecture was the CDC 6600 in 1963, considered by many to be the first supercomputer. ARM and MIPS are more recent examples of a load-store architecture.

The 80386 Intel's attempt to transform the 8086 into a general-purpose register-memory instruction set. Perhaps the best-known register-memory instruction set is the IBM 360 architecture, first announced in 1964. This instruction set is still at the core of IBM's mainframe computers—still responsible for $10B per year in annual sales. Register-memory architectures were

the most popular in the 1960s and the first half of the 1970s.

Digital Equipment Corporation's VAX architecture took memory operands one step further in 1977. It allowed an instruction to use any combination of registers and memory operands. A style of architecture in which all operands can be in memory is called *memory-memory*. (In truth the VAX instruction set, like almost all other instruction sets since the IBM 360, is a hybrid, since it also has general-purpose registers.)

Although MIPS has a single add instruction with 32-bit operands, the Intel x86 has many versions of a 32-bit add to specify whether an operand is in memory or is in a register. In addition, the memory operand can be accessed with more than seven addressing modes. This combination of address modes and register-memory operands means that there are dozens of variants of an x86 add instruction. Clearly, this variability makes x86 implementations more challenging.

Compact code and stack architectures

When memory is scarce, it is also important to keep programs small, so architectures like the Intel x86, IBM 360, and VAX had variable-length instructions, both to match the varying operand specifications and to minimize code size. Intel x86 instructions are from 1 to 17 bytes long; IBM 360 instructions are 2, 4, or 6 bytes long; and VAX instruction lengths are anywhere from 1 to 54 bytes.

In the 1960s, a few companies followed a radical approach to instruction sets. In the belief that it was too hard for compilers to utilize registers effectively, these companies abandoned registers altogether! Instruction sets were based on a *stack model* of execution, like that found in the older Hewlett-Packard handheld calculators. Operands are pushed on the stack from memory or popped off the stack into memory. Operations take their operands from the stack and then place the result back onto the stack. In addition to simplifying compilers by eliminating register allocation, stack architectures lent themselves to compact instruction encoding, thereby removing memory size as an excuse not to program in high-level languages.

Memory space was perceived to be precious again for Java, both because memory space is limited to keep costs low in embedded applications and because programs may be downloaded over the Internet or phone lines as Java applets, and smaller programs take less time to transmit. Hence, compact instruction encoding was desirable for Java bytecodes.

High-level-language computer architectures

In the 1960s, systems software was rarely written in high-level languages. For example, virtually every commercial operating system before UNIX was programmed in assembly language, and more recently even OS/2 was originally programmed at that same low level. Some people blamed the code density of the instruction sets, rather than the programming languages and the compiler technology.

Hence, an architecture design philosophy called *high-level-language computer architecture* was

advocated, with the goal of making the hardware more like the programming languages. More efficient programming languages and compilers, plus expanding memory, doomed this movement to a historical footnote. The Burroughs B5000 was the commercial fountainhead of this philosophy, but today there is no significant commercial descendant of this 1960s radical.

Reduced instruction set computer architectures

This language-oriented design philosophy was replaced in the 1980s by *RISC (reduced instruction set computer)*. Improvements in programming languages, compiler technology, and memory cost meant that less programming was being done at the assembly level, so instruction sets could be measured by how well compilers used them, as opposed to how well assembly language programmers used them.

Virtually all new instruction sets since 1982 have followed this RISC philosophy of fixed instruction lengths, load-store instruction sets, limited addressing modes, and limited operations. ARM, ARC, Hitachi SH, IBM PowerPC, MIPS, and RISC-V are all examples of RISC architectures.

PARTICIPATION ACTIVITY 2.23.1: Evolution of computer architectures.

How to use this tool

Load-store architecture **Stack architecture** **Accumulator architecture**

High-level-language computer architecture **RISC architecture**

Reduced instruction set computer architecture

Register-register architecture

Programming language-based architecture

Single register architecture

No registers architecture

Reset

A brief history of the ARM

ARM started as the processor for the Acorn computer, hence its original name of Acorn RISC Machine. Its architecture was influenced by the Berkeley RISC papers.

One of the most important early applications was emulation of the AM 6502, a 16-bit microprocessor. This emulation was to provide most of the software for the Acorn computer. As the 6502 had a variable length instruction set that was a multiple of bytes, 6502 emulation helps explain the emphasis on shifting and masking in the ARM instruction set.

Its popularity as a low-power embedded computer began with its selection as the processor for the ill-fated Apple Newton personal digital assistant. Although the Newton was not as popular as Apple hoped, Apple's blessing gave visibility to ARM, and it subsequently caught on in several markets, including cell phones. Unlike the Newton experience, the extraordinary success of cell phones explains why 100 billion ARM processors were shipped in between 1999 and 2016.

One of the major events in ARM's history is the 64-bit address extension called version 8. ARM took the opportunity to redesign and simplify the instruction set to make it look much more like MIPS than like earlier ARM versions.

A brief history of the x86

The ancestors of the x86 were the first microprocessors, produced starting in 1972. The Intel 4004 and 8008 were extremely simple 4-bit and 8-bit accumulator-style architectures. Morse et al. [1980] described the evolution of the 8086 from the 8080 in the late 1970s as an attempt to provide a 16-bit architecture with better throughput. At that time, almost all programming for microprocessors was done in assembly language—both memory and compilers were in short supply. Intel wanted to keep its base of 8080 users, so the 8086 was designed to be "compatible" with the 8080. The 8086 was never object-code compatible with the 8080, but the architectures were close enough that translation of assembly language programs could be done automatically.

In early 1980, IBM selected a version of the 8086 with an 8-bit external bus, called the 8088, for use in the IBM PC. They chose the 8-bit version to reduce the cost of the architecture. This choice, together with the tremendous success of the IBM PC, has made the 8086 architecture ubiquitous in the PC era. The success of the IBM PC was due in part because IBM opened the architecture of the PC and enabled the PC-clone industry to flourish. As discussed in COD Section 2.19 (Real Stuff: x86 Instructions), the 80286, 80386, 80486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, and newer processors have extended the architecture and provided a series of performance enhancements.

Although the 68000 was chosen for the Macintosh, the Mac was never as pervasive as the PC, partly because Apple did not allow Mac clones based on the 68000, and the 68000 did not acquire the same software following that which the 8086 enjoys. The Motorola 68000 may have been more significant *technically* than the 8086, but the impact of IBM's selection and open architecture

strategy dominated the technical advantages of the 68000 in the market.

Some argue that the inelegance of the x86 instruction set is unavoidable, the price that must be paid for rampant success by any architecture. We reject that notion. Obviously, no successful architecture can jettison features that were added in previous implementations, and over time, some features may be seen as undesirable. The awkwardness of the x86 begins at its core with the 8086 instruction set and was exacerbated by the architecturally inconsistent expansions found in the 8087, 80286, 80386, MMX, SSE, SSE2, SSE3, SSE4, AMD64 (EM64T), and AVX.

A counterexample is the IBM 360/370 architecture, which is much older than the x86. It dominates the mainframe market just as the x86 dominates the PC market. Due undoubtedly to a better base and more compatible enhancements, this instruction set makes much more sense than the x86 55 years after its first implementation.

Extending the x86 to 64-bit addressing means the architecture could last for several more decades. Instruction set anthropologists of the future will peel off layer after layer from such architectures until they uncover artifacts from the first microprocessor. Given such a find, how will they judge today's computer architecture?

PARTICIPATION ACTIVITY 2.23.2: History of ARM and x86.

1) An _____ processor powered the Apple Newton personal digital assistant.
   - ○ x86
   - ○ ARM

2) The first microprocessors were precursors to the _____ architecture.
   - ○ x86
   - ○ ARM

3) The 8088 processors were used to power _____ computers in the 1980s.
   - ○ Apple Macintosh
   - ○ IBM PC
   - ○ Pentium 4

A brief history of programming languages

In 1954, John Backus led a team at IBM to create a more natural notation for scientific programming. The goal of Fortran, for "FORmula TRANslator", was to reduce the time to develop programs. Fortran included many ideas found in programming languages today, including assignment statements, expressions, typed variables, loops, and arrays. The development of the language and the compiler went hand in hand. This language became a standard that has evolved over time to improve programmer productivity and program portability. The evolutionary steps are Fortran I, II, IV, 77, 90-95, 2003, 2008, 2018.

Fortran was developed for IBM's second commercial computer, the 704, which was also the cradle of another important programming language: Lisp. John McCarthy invented the "LISt Processing" language in 1958. Its mantra is that programming can be considered as manipulating lists, so the language contains operations to follow links and to compose new lists from old ones. This list notation is used for the code as well as the data, so modifying or composing Lisp programs is common. The big contribution was dynamic data structures and, hence, pointers. Given that its inventor was a pioneer in artificial intelligence, Lisp became popular in the AI community. Lisp has no type declarations, and Lisp traditionally reclaims storage automatically via built-in garbage collection. Lisp was originally interpreted, although compilers were later developed for it.

Fortran inspired the international community to invent a programming language that was more natural to express algorithms than Fortran, with less emphasis on coding. This language became Algol, for "ALGOrithmic Language". Like Fortran, it included type declarations, but it added recursive procedure calls, nested *if-then-else* statements, *while* loops, *begin-end* statements to structure code, and call-by-name. Algol-60 became the classic language for academics to teach programming in the 1960s.

Although engineers, AI researchers, and computer scientists had their own programming languages, the same could not be said for business data processing. Cobol, for "COmmon Business-Oriented Language", was developed as a standard for this purpose about the same time as Algol-60. Cobol was created to be easy to read, so it follows English vocabulary and punctuation. It added records to programming languages, and separated description of data from description of code.

PARTICIPATION ACTIVITY 2.23.3: History of programming languages.

1) _____, developed for use in business settings, employs English vocabulary and punctuation.

Check **Show answer**

2) _____, developed for IBM's

704 computer in 1954, was one the first widely used programming languages.

Check **Show answer**

3) In the 1970s, Bell Labs built the UNIX operating system using the _____ language.

Check **Show answer**

4) _____ was invented as a way to express algorithms more naturally than its predecessors, with less focus on coding.

Check **Show answer**

5) Dynamic data structures and pointers were major contributions of _____.

Check **Show answer**

6) _____ is currently the most popular language to teach in schools and is the standard language for business data processing applications.

Check **Show answer**

©zyBooks 05/16/25 23:10 2475274
Jaheim Attri
FIUEEL4709CSpring2025

Niklaus Wirth was a member of the Algol-68 committee, which was supposed to update Algol-60. He was bothered by the complexity of the result, and so he wrote a minority report to show that a programming language could combine the algorithmic power of Algol-60 with the record structure from Cobol and be simple to understand, simple to implement, yet still powerful. This minority report became Pascal. It was first implemented with an interpreter and a set of Pascal bytecodes. The ease of implementation led to its being widely deployed, much more than Algol-68, and it soon replaced Algol-60 as the most popular language for academics to teach programming.

In the same period, Dennis Ritchie invented the C programming language to use in building UNIX. Its inventors say it is not a "very high level" programming language or a big one, and it is not aimed at a particular application. Given its birthplace, it was very good at systems programming, and the UNIX operating system and C compiler were written in C. UNIX's popularity helped spur C's popularity.

The concept of object orientation is first captured in Simula-67, a simulation language successor to Algol-60. Invented by Ole-Johan Dahl and Kristen Nygaard at the University of Oslo in 1967, it introduced objects, classes, and inheritance.

Object orientation proved to be a powerful idea. It led Alan Kay and others at Xerox Palo Alto Research Center to invent Smalltalk in the 1970s. Smalltalk-80 married the typeless variables and garbage collection from Lisp and the object orientation of Simula-67. It relied on interpretation that was defined by a Smalltalk virtual machine with a Smalltalk bytecode instruction set. Kay and his colleagues argued that processors were getting faster, and that we must eventually be willing to sacrifice some performance to improve program development. Another example was CLU, which demonstrated that an object-oriented language could be defined that allowed compile-time type checking. Simula-67 also inspired Bjarne Stroustrup of Bell Labs to develop an object-oriented version of C called C++ in the 1980s. C++ became widely used in industry.

Dissatisfied with C++, a group at Sun led by James Gosling invented Oak in the early 1990s. It was invented as an object-oriented C dialect for embedded devices as part of a major Sun project. To make it portable, it was interpreted and had its own virtual machine and bytecode instruction set. Since it was a new language, it had a more elegant object-oriented design than C++ and was much easier to learn and compile than Smalltalk-80. Since Sun's embedded project failed, we might never have heard of it had someone not made the connection between Oak and programmable browsers for the World Wide Web. It was rechristened Java, and in 1995, Netscape announced that it would be shipping with its browser. It soon became extraordinarily popular. Java had the rare distinction of becoming the standard language for new business data processing applications *and* the most popular language for academics to teach programming. Java and languages like it encourage reuse of code, and hence programmers make heavy use of libraries, whereas in the past they were more likely to write everything from scratch.

Several people in this history section won ACM A.M. Turing Awards, at least in part for their contributions to programming languages: John Backus (1977), John McCarthy (1971), Niklaus Wirth (1984), Dennis Ritchie (1983), Ole-Johan Dahl and Kristen Nygaard (2001), and Alan Kay (2003).

A brief history of compilers

Backus and his group were very concerned that Fortran would be unsuccessful if skeptics found examples where the Fortran version ran at half the speed of the equivalent assembly language program. Their success with one of the first compilers created a beachhead that many others followed.

Early compilers were ad hoc programs that performed the steps described in COD Section 2.15 (Advanced Material: Compiling C and interpreting Java). These ad hoc approaches were replaced with a solid theoretical foundation for each of these steps. Each time the theory was established, a tool was created based on that theory that automated the creation of that step.

The theoretical roots underlying scanning and parsing derive from automata theory, and the relationship between languages and automata was known early. The scanning task corresponds to recognition of a language accepted by a finite-state automata, and parsing corresponds to recognition of a language by a push-down automata (basically an automata with a stack). Languages are described by grammars, which are a set of rules that tell how any legal program can be generated.

The scanning pass of a compiler was well understood early, but parsing is harder. The earliest parsers use precedence techniques, which derived from the structure of arithmetic statements, and were then generalized. The great breakthrough in modern parsing was made by Donald Knuth in the invention of LR-parsing, which codified the two key steps in the parsing technique, pushing a token on the stack or reducing a set of tokens on the stack using a grammar rule. The strong theory formulation for scanning and parsing led to the development of automated tools for compiler constructions, such as `lex` and `yacc`, the tools developed as part of UNIX.

Optimizations occurred in many compilers, and it is harder to determine the first examples in most cases. However, Victor Vyssotsky did the first papers on data flow analysis in 1963, and William McKeeman is generally credited with the first peephole optimizer in 1965. The group at IBM, including John Cocke and Fran Allan, developed many of the early optimization concepts, as well as defining and extending the concepts of flow analysis. Important contributions were also made by Al Aho and Jeff Ullman.

One of the biggest challenges for optimization was register allocation. It was so difficult that some architects used stack architectures just to avoid the problem. The breakthrough came when researchers working on compilers for the 801, an early RISC architecture, recognized that coloring a graph with a minimum number of colors was equivalent to allocating a fixed number of registers to the unlimited number of virtual registers used in intermediate forms.

Compilers also played an important role in the open source movement. Richard Stallman's self-appointed mission was to make a public domain version of UNIX. He built the GNU C Compiler (gcc) as an open source compiler in 1987. It soon was ported to many architectures, and is used in many systems today.

PARTICIPATION ACTIVITY 2.23.4: Compilers.

1) lex and yacc, automated tools for compiler construction, were developed as a part of the Windows operating system.
   - ○ True
   - ○ False

2) In early compilers, processes such as parsing and scanning consisted of ad hoc approaches. These processes were replaced by tools derived from automata theory.
   - ○ True
   - ○ False

3) Compilers do not need to perform register allocation in RISC architectures.
   - ○ True
   - ○ False

Further reading

Bayko, J. [1996]. "Great microprocessors of the past and present," search for it on the *www.jbayko.sasktel website.net/cpu.html*
A personal view of the history of both representative and unusual microprocessors, from the Intel 4004 to the Patriot Scientific ShBoom!

Kane, G. and J. Heinrich [1992]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ.
This book describes the MIPS architecture in greater detail than COD Appendix A (Assemblers, Linkers, and the SPIM Simulator).

Levy, H. and R. Eckhouse [1989]. *Computer Programming and Architecture: The VAX*, Digital Press, Boston.
This book concentrates on the VAX, but also includes descriptions of the Intel 8086, IBM 360, and CDC 6600.

Morse, S., B. Ravenal, S. Mazor, and W. Pohlman [1980]. "Intel microprocessors—8080 to 8086," *Computer* 13:10 (October).
The architecture history of the Intel from the 4004 to the 8086, according to the people who participated in the designs.

Wakerly, J. [1989]. *Microcomputer Architecture and Programming*, Wiley, New York.
The Motorola 6800 is the main focus of the book, but it covers the Intel 8086, Motorola 6809, TI 9900, and Zilog Z8000.

2.24 Self study

Instructions as Numbers. Given this binary number:

$0000000101001011010010000010000_{two}$

What is it in hexadecimal format?
Assuming it is an unsigned number, what is it in decimal?
Does the value change if it is considered a signed number?
What assembly language program does it represent?

Instructions as Numbers and Insecurity. Although programs are just numbers in memory, Chapter 5 shows how to tell the computer how to protect the program from being modified by labeling a portion of the address space as read-only. Clever attackers exploit bugs in C programs to nevertheless insert their own code during program execution despite that program being protected.

Here is a simple string copy program that copies what the user types into a local variable on the stack.

```
#include <string.h>
void copyinput (char *input)
{
   char copy[10];
   strcpy(copy, input); // no bounds checking in strcpy
}
int main (int argc, char **argv)
{
   copyinput(argv[1]);
   return 0;
}
```

What happens if the user writes much more than 10 characters as input? What could be the consequence to program execution? How could this let an attacker take over program execution?

While being faster. Here is the MIPS code for a C while loop:

```
Loop: sll $t1,$s3,2             # Temp reg $t1 = i * 4
      add $t1,$t1,$s6           # $t1 = address of save[i]
      lw $t0,0($t1)             # Temp reg $t0 = save[i]
      bne $t0,$s5, Exit         # go to Exit if save[i] ≠k
```

```
      addi $s3,$s3,1            # i = i + 1
      j Loop                    # go to Loop
Exit:
```

Assume the loop typically executes 10 times. Make the loop faster by executing on average one branch instruction per loop rather than both one jump instruction and one branch instruction.

The Anti-Compiler. Here is a portion of MIPS assembly language with comments for the first five instructions.


```
sll $t0, $s0, 2           # $t0 = f * 4
add $t0, $s6, $t0         # $t0 = &A[f]
sll $t1, $s1, 2           # $t1 = g * 4
add $t1, $s7, $t1         # $t1 = &B[g]
lw $s0, 0($t0)            # f = A[f]
addi $t2, $t0, 4          #
lw $t0, 0($t2)            #
add $t0, $t0, $s0         #
sw $t0, 0($t1)            #
```

Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. Assume that the base address of the arrays `A` and `B` are in registers `$s6` and `$s7`, respectively. Complete the comments for the last four instructions, and then show the C code that would have been compiled into these MIPS instructions.

Self Study Answers

Instructions as Numbers.

Binary: **$0000000101001011010010000010000_{two}$**

Hexadecimal: **$014B4820_{hex}$**

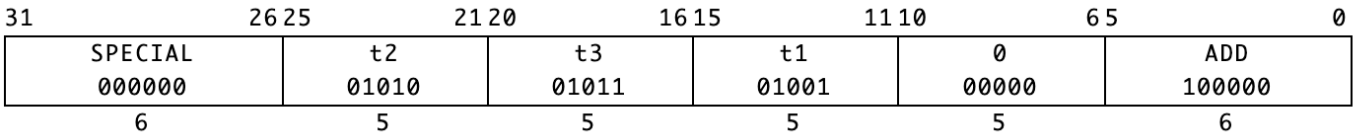
Decimal: **21710880_{ten}**

Since the leading bit is a 0, it is the same decimal value whether signed or unsigned integer.

Assembly language:


```
add t1, t2, t3
```

Machine language:



Instructions as Numbers and Insecurity.

The local variable copy can safely copy user input up to nine characters followed by the null character that terminates a string. Anything longer will overwrite other values on the stack. As the stack grows down, the values below the stack include stack frames from earlier procedure calls, which include return addresses. A careful attacker can not only insert code onto the stack but can overwrite return addresses in the stack so that the program could eventually use the attacker's return address to start executing code placed on the stack after some procedures returned.

While being faster. The trick is to invert the conditional branch and have it jump to the top of the loop rather than having it skip the jump at the bottom of the loop. To match the semantics of the while loop, the code must first check to see if `save[i] == k` before incrementing `i`.

```
      sll $t1,$s3,2                  # Temp reg $t1 = i * 4
       add $t1,$t1,$s6          # $t1 = address of save[i]
       lw $t0,0($t1)                  # Temp reg $t0 = save[i]
       bne $t0,$s5, Exit          # go to Exit if save[i] ≠k
Loop: addi $s3,$s3,1          # i = i + 1
       sll $t1,$s3,2                  # Temp reg $t1 = i * 4
       add $t1,$t1,$s6          # $t1 = address of save[i]
       lw $t0,0($t1)                  # Temp reg $t0 = save[i]
       beq $t0,$s5, Loop          # go to Loop if save[i] = k
Exit:
```

The Anti-Compiler

```
sll $t0, $s0, 2          # $t0 = f * 4
add $t0, $s6, $t0          # $t0 = &A[f]
sll $t1, $s1, 2          # $t1 = g * 4
add $t1, $s7, $t1          # $t1 = &B[g]
lw $s0, 0($t0)          # f = A[f]
addi $t2, $t0, 4          # $t2=$t0+4 => $t2 points to A[f+1] now
lw $t0, 0($t2)          # $t0 = A[f+1]
add $t0, $t0, $s0          # $t0 = $t0 + $s0 => $t0 is now A[f]+A[f+1]
sw $t0, 0($t1)          # store the result into B[g]
```


The C statement equivalent is `B[g] = A[f] + A[f+1];`

2.25 Exercises

COD Appendix A (Assemblers, Linkers, and the SPIM Simulator) describes the MIPS simulator, which is helpful for these exercises. Although the simulator accepts pseudoinstructions, try not to use pseudoinstructions for any exercises that ask you to produce MIPS code. Your goal should be to learn the real MIPS instruction set, and if you are asked to count instructions, your count should reflect the actual instructions that will be executed and not the pseudoinstructions.

There are some cases where pseudoinstructions must be used (for example, the `la` instruction when an actual value is not known at assembly time). In many cases, they are quite convenient and result in more readable code (for example, the `li` and `move` instructions). If you choose to use pseudoinstructions for these reasons, please add a sentence or two to your solution stating which pseudoinstructions you have used and why. Because this interactive zyBook version may have been re-ordered and hence sections renumbered, section numbers below labeled with COD refer to the original hardcopy book's section numbers.

EXERCISE 2.25.1: [5] <COD §2.2>.

(a) For the following C statement, what is the corresponding MIPS assembly code? Assume that the C variables `f`, `g`, and `h`, have already been placed in registers `$s0`, `$s1`, `$s2`, respectively. Use a minimal number of MIPS assembly instructions.

```
f = g + (h - 5);
```

Solution

EXERCISE 2.25.2: [5] <COD §2.2>.

(a) Write a single C statement that corresponds to the two MIPS assembly instructions below.


```
add  f, g, h
add  f, i, f
```

Solution

EXERCISE 2.25.3: [5] <COD §§2.2, 2.3>.

(a) For the following C statement, write the corresponding MIPS assembly code. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. Assume that the base address of the arrays `A` and `B` are in registers `$s6` and `$s7`, respectively.


```
B[8] = A[i - j];
```

Solution

EXERCISE 2.25.4: [5] <COD §§2.2, 2.3>.

(a) For the MIPS assembly instructions below, what is the corresponding C statement? Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. Assume that the base address of the arrays `A` and `B` are in registers `$s6` and `$s7`, respectively.

```
sll  $t0, $s0, 2     # $t0 = f * 4
add  $t0, $s6, $t0   # $t0 = &A[f]
sll  $t1, $s1, 2     # $t1 = g * 4
add  $t1, $s7, $t1   # $t1 = &B[g]
lw   $s0, 0($t0)     # f = A[f]
addi $t2, $t0, 4
lw   $t0, 0($t2)
add  $t0, $t0, $s0
sw   $t0, 0($t1)
```

Solution

EXERCISE 2.25.5: [5] <COD §§.

(a) Show how the value 0xabcdef12 would be arranged in memory of a little-endian and a big-endian machine. Assume the data are stored starting at address 0 and the word size is 4 bytes.

Solution

EXERCISE 2.25.6: [5] <COD §§2.4>.

(a) Translate `0xabcdef12` into decimal.

Solution

EXERCISE 2.25.7: [5] <COD §2.2, 2.3>.

(a) Translate the following C code to MIPS. Assume that the variables, `f`,`g`. `h`, `i`, and `j` are assigned to registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. Assume that the base address of `A` and `B` are in registers `$s6` and `$s7`, respectively. Assume that the elements of the arrays `A` and `B` are 4-byte words:

```
B[8] = A[i] + A[j];
```

Solution

EXERCISE 2.25.8: [10] <COD §2.2, 2.3>.

(a) Translate the following MIPS code to C. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. Assume that the base address of the arrays `A` and `B` are in registers `$s6` and `$s7`, respectively.

```
addi $t0, $s6, 4
add  $t1, $s6, $0
sw   $t1, 0($t0)
lw   $t0, 0($t0)
add  $s0, $t1, $t0
```


Solution

EXERCISE 2.25.9: [20] <COD §§2.3, 2.5>.

(a) For each MIPS instruction in Exercise 2.8, show the value of the opcode (op), source register (rs) and funct field, and destination register (rd) fields. For the I-type instructions, show the value of the immediate field, and for the R-type instructions, show the value of the second source register (rt).

Solution

EXERCISE 2.25.10: [5] <COD §2.4>.

Assume that registers `$s0` and `$s1` hold the values `0x80000000` and `0xD0000000`, respectively.

(a) What is the value of `$t0` for the following assembly code?

```
add $t0, $s0, $s1
```

Solution

(b) Is the result in `$t0` the desired result, or has there been overflow?

Solution

(c) For the contents of registers `$s0` and `$s1` as specified above, what is the value of `$t0` for the following assembly code?

```
sub $t0, $s0, $s1
```

Solution

(d) Is the result in `$t0` the desired result, or has there been overflow?

Solution

(e) For the contents of registers `$s0` and `$s1` as specified above, what is the value of `$t0` for the following assembly code?


```
add $t0, $s0, $s1
add $t0, $t0, $s0
```

Solution

(f) Is the result in `$t0` the desired result, or has there been overflow?

Solution

EXERCISE 2.25.11: [5] <COD §2.4>.

Assume that `$s0` holds the value 128_{ten}.

(a) For the instruction `add $t0, $s0, $s1`, what is the range(s) of values for `$s1` that would result in overflow?

Solution

(b) For the instruction `sub $t0, $s0, $s1`, what is the range(s) of values for `$s1` that would result in overflow?

Solution

(c) For the instruction `sub $t0, $s1, $s0`, what is the range(s) of values for `$s1` that would result in overflow?

Solution

EXERCISE 2.25.12: <COD §§2.4, 2.5>.

(a) Provide the type and assembly language instruction for the following binary value: `0000 0010 0001 0000 1000 0000 0010 0000`$_{two}$. Hint: COD Figure 2.19 (MIPS instruction encoding) may be helpful.

Solution

EXERCISE 2.25.13: <COD §§2.4, 2.5>.

(a) Provide the type and hexadecimal representation of following instructions: `sw $t1, 32($t2)`

Solution

EXERCISE 2.25.14: [5] <COD §2.5>.

(a) Provide the instruction type, assembly language instruction, and binary representation of the instruction described by the following MIPS fields:

```
op = 0, rs = 3, rt = 2, rd = 3, shamt = 0, funct = 34
```


Solution

EXERCISE 2.25.15: [5] <COD §2.5>.

(a) Provide the type, assembly language instruction, and binary representation of the instruction described by the following MIPS fields:

```
op = 0x23, rs = 1, rt = 2, const = 0x4
```

Solution

EXERCISE 2.25.16: [5] <COD §2.5>.

Assume that we would like to expand the MIPS register file to 128 registers and expand the instruction set to contain four times as many instructions.

(a) How would this affect the size of each of the bit fields in the R-type instructions?

(b) How would this affect the size of each of the bit fields in the I-type instructions?

(c) How could each of the two proposed changes decrease the size of an MIPS assembly program? On the other hand, how could the proposed change increase the size of an MIPS assembly program?

EXERCISE 2.25.17: [5] <COD §2.6>.

Assume the following register contents:

```
$t0 = 0xAAAAAAAA, $t1 = 0x12345678
```

(a) For the register values shown above, what is the value of `$t2` for the following

sequence of instructions?

```
sll $t2, $t0, 4
or  $t2, $t2, $t1
```

(b) For the register values shown above, what is the value of `$t2` for the following sequence of instructions?


```
sll  $t2, $t0, 4
andi $t2, $t2, -1
```

(c) For the register values shown above, what is the value of `$t2` for the following sequence of instructions?

```
srl  $t2, $t0, 3
andi $t2, $t2, 0xFFEF
```

EXERCISE 2.25.18: <COD §2.6>.

(a) Find the shortest sequence of MIPS instructions that extracts bits 16 down to 11 from register `$t0` and uses the value of this field to replace bits 31 down to 26 in register `$t1` without changing the other bits of registers `$t0` and `$t1`. (Be sure to test your code using `$t0 = 0` and `$t1 = 0xffffffff`. Doing so may reveal a common oversight.)

EXERCISE 2.25.19: [5] <COD §2.6>.

(a) Provide a minimal set of MIPS instructions that may be used to implement the following pseudoinstruction:

```
not $t1, $t2      // bit-wise invert
```


EXERCISE 2.25.20: [5] <COD §2.6>.

(a) For the following C statement, write a minimal sequence of MIPS assembly instructions that does the identical operation. Assume `$t0 = A`, and `$s0` is the base

address of C.

```
A = C[0] << 4;
```

EXERCISE 2.25.21: [5] <COD §2.7>.

(a) Assume `$t0` holds the value `0x01010000`. What is the value of `$t2` after the following instructions?

```
      slt  $t2, $0, $t0
      bne  $t2, $0, ELSE
      j    DONE
ELSE: addi $t2, $t2, 2
DONE:
```

EXERCISE 2.25.22: [5] <COD §2.10>.

Suppose the program counter (PC) is set to `0x20000000`.

(a) What range of addresses can be reached using the MIPS *jump-and-link* (`jal`) instruction? (In other words, what is the set of possible values for the PC after the jump instruction executes?)

(b) What range of addresses can be reached using the MIPS *branch if equal* (`beq`) instruction? (In other words, what is the set of possible values for the PC after the jump instruction executes?)

EXERCISE 2.25.23: [5] <COD §2.7, 2.10>.

Consider a proposed new instruction named `rpt`. This instruction combines a loop's condition check and counter decrement into a single instruction. For example `rpt $s0, loop` would do the following:

```
if (x29 > 0) {
    x29 = x29 - 1;
    goto loop
}
```

(a) If this instruction were to be added to the MIPS instruction set, what is the most appropriate instruction format?

(b) What is the shortest sequence of MIPS instructions that performs the same operation?

EXERCISE 2.25.24: [5] <COD §2.7>.

Consider the following MIPS loop:

```
LOOP: slt  $t2, $0,  $t1
      beq  $t2, $0,  DONE
      subi $t1, $t1, 1
      addi $s2, $s2, 2
      j    LOOP
DONE:
```

(a) Assume that the register `$t1` is initialized to the value 10. What is the value in register `$s2` assuming `$s2` is initially zero?

(b) For each of the loops above, write the equivalent C code routine. Assume that the registers `$s1`, `$s2`, `$t1`, and `$t2` are integers `A`, `B`, `i`, and `temp`, respectively.

(c) For the loops written in MIPS assembly above, assume that the register `$t1` is initialized to the value N. How many MIPS instructions are executed?

EXERCISE 2.25.25: [10] <COD §2.7>.

(a) Translate the following C code to MIPS assembly code. Use a minimum number of instructions. Assume that the values of `a`, `b`, `i`, and `j` are in registers `$s0`, `$s1`, `$t0`, and `$t1`, respectively. Also, assume that register `$s2` holds the base address of the array `D`.


```
for(i = 0; i < a; i++)
   for(j = 0; j < b; j++)
      D[4 * j] = i + j;
```

Solution

EXERCISE 2.25.26: [5] <COD §2.7>.

(a) How many MIPS instructions does it take to implement the C code from Exercise 2.25? If the variables `a` and `b` are initialized to 10 and 1 and all elements of D are initially 0, what is the total number of MIPS instructions that is executed to complete the loop?

Solution

EXERCISE 2.25.27: [5] <COD §2.7>.

(a) Translate the following loop into C. Assume that the C-level integer `i` is held in register `$t1`, `$s2` holds the C-level integer called `result`, and `$s0` holds the base address of the integer `MemArray`.

```
      addi $t1, $0, 0
LOOP: lw   $s1, 0($s0)
      add  $s2, $s2, $s1
      addi $s0, $s0, 4
      addi $t1, $t1, 1
      slti $t2, $t1, 100
      bne  $t2, $0, LOOP
```

EXERCISE 2.25.28: <COD §2.7>.

(a) Rewrite the loop from Exercise 2.27 to reduce the number of MIPS instructions executed. Hint: Notice that variable `i` is used only for loop control.

EXERCISE 2.25.29: <COD §2.8>.

(a) Implement the following C code in MIPS assembly. Hint: Remember that the stack pointer must remain aligned on a multiple of 16.

```
int fib(int n){
   if (n == 0)
```

```
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```


EXERCISE 2.25.30: <COD §2.8>.

(a) For each function call in Exercise 29, show the contents of the stack after the function call is made. Assume the stack pointer is originally at address `0x7ffffffc`, and follow the register conventions as specified in COD Figure 2.11 (What is and what is not preserved across a procedure call).

EXERCISE 2.25.31: [20] <COD §2.8>.

(a) Translate function `f` into MIPS assembly language. If you need to use registers `$t0` through `$t7`, use the lower-numbered registers first. Assume the function declaration for `func` is `"int func(int a, int b);"`. The code for function `f` is as follows:

```
int f(int a, int b, int c, int d){
   return func(func(a, b), c + d);
}
```

EXERCISE 2.25.32: [5] <COD §2.8>.

(a) Can we use the tail-call optimization in this function? If no, explain why not. If yes, what is the difference in the number of executed instructions in `f` with and without the optimization?

EXERCISE 2.25.33: [5] <COD §2.8>.

(a) Right before your function `f` from Exercise 2.31 returns, what do we know about

contents of registers `$t5`, `$s3`, `$ra`, and `$sp`? Keep in mind that we know what the entire function `f` looks like, but for function `func` we only know its declaration.

EXERCISE 2.25.34: <COD §2.9>.

(a) Write a program in MIPS assembly language to convert an ASCII number string containing positive and negative integer decimal strings, to an integer. Your program should expect register `$a0` to hold the address of a null-terminated string containing some combination of the digits 0 through 9. Your program should compute the integer value equivalent to this string of digits, then place the number in register `$v0`. If a non-digit character appears anywhere in the string, your program should stop with the value −1 in register `$v0`. For example, if register `$t0` points to a sequence of three bytes 50_{ten}, 52_{ten}, 0_{ten} (the null-terminated string "24"), then when the program stops, register `$v0` should contain the value 24_{ten}. The RISC-V mul instruction takes two registers as input. There is no "muli" instruction. Thus, just store the constant 10 in a register.

EXERCISE 2.25.35: [5] <COD §2.9>.

Consider the following code:

```
lbu $t0, 0($t1)
sw  $t0, 4($t1)
```

Assume that the register `$t1` contains the address `0x1000 0000` and the data at address is `0x11223344`.

(a) What value is stored in `0x10000004` on a big-endian machine?

(b) What value is stored in `0x10000004` on a little-endian machine?

EXERCISE 2.25.36: [5] <COD §2.10>.

(a) Write the MIPS assembly code that creates the 32-bit constant `ll`/sc and stores that value to register `$t1`.

EXERCISE 2.25.37: [10] <COD §2.11>.

(a) Write the MIPS assembly code to implement the following C code as an atomic "set max" operation using the `lI/sc` instructions. Here, the argument `shvar` contains the address of a shared variable, which should be replaced by `x` if `x` is greater than the value it points to:


```
void setmax(int* shvar, int x){
   // Begin critical section
   If (x > *shvar)
      *shvar = x;
   // End critical section
}
```

EXERCISE 2.25.38: [5] <COD §2.11>.

(a) Using your code from Exercise 2.37 as an example, explain what happens when two processors begin to execute this critical section at the same time, assuming that each processor executes exactly one instruction per cycle.

EXERCISE 2.25.39: [5] <COD §§1.6, 2.13>.

Assume for a given processor the CPI of arithmetic instructions is 1, the CPI of load/store instructions is 10, and the CPI of branch instructions is 3. Assume a program has the following instruction breakdowns: 500 million arithmetic instructions, 300 million load/store instructions, 100 million branch instructions.

(a) Suppose that new, more powerful arithmetic instructions are added to the instruction set. On average, through the use of these more powerful arithmetic instructions, we can reduce the number of arithmetic instructions needed to execute a program by 25%, and the cost of increasing the clock cycle time by only 10%. Is this a good design choice? Why?

(b) Suppose that we find a way to double the performance of arithmetic instructions. What is the overall speedup of our machine? What if we find a way to improve the performance of arithmetic instructions by 10 times?

EXERCISE 2.25.40: [5] <COD §§1.6, 2.13>.

Assume that for a given program 70% of the executed instructions are arithmetic, 10% are load/store, and 20% are branch.

(a) Given this instruction mix and the assumption that an arithmetic instruction requires 2 cycles, a load/store instruction takes 6 cycles, and a branch instruction takes 3 cycles, find the average CPI.

Solution

(b) For a 25% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions are not improved at all?

Solution

(c) For a 50% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions are not improved at all?

Solution

EXERCISE 2.25.41: [10] <COD §2.21>.

(a) Suppose the MIPS ISA included a scaled offset addressing mode similar to the x86 one described in Section 2.19 (Figure 2.35). Describe how you would use scaled offset loads to further reduce the number of assembly instructions needed to carry out the function given in Excercise 2.4.

Solution

EXERCISE 2.25.42: [10] <COD §2.21>.

(a) Suppose the MIPS ISA included a scaled offset addressing mode similar to the x86 one described in Section 2.19 (Figure 2.35). Describe how you would use scaled offset loads to further reduce the number of assembly instructions needed to carry out the function given in Excercise 2.7.

Solution

2.26 MIPS Simulator

This MIPS simulator (alpha) supports the following language features:

- Instructions
  - add, addi, and, andi, beq, bge, bgt, ble, blt, bne, div, j, jal, jr, lb, lbu, lh, lhu, lui, lw, mfhi, mflo, move, mul, mult, nor, or, ori, sb, sh, sll, slt, slti, sltiu, sltu, srl, sub, sw
- Labels
  - L1: add $s1, $s2, $s3
- Comments
  - add $s1, $s2, $s3 # Adds j + k

PARTICIPATION ACTIVITY 2.26.1: Addition, subtraction, and registers.

1. Run the simulation step-by-step, observing register values.
2. Change $s0's value to 10 by clicking the register value on the right, then run again.
3. Modify line 3 to be `sub $s4, $t1, $t0`, then run again.

Assembly

```
Line 1 add $t0, $s0, $s1
Line 2 add $t1, $s2, $s3
Line 3 sub $s4, $t0, $t1
```

Registers

$t0	0
$t1	0
$s0	5
$s1	4
$s2	3
$s3	2
$s4	1

Mem

4000

ENTER SIMULATION STEP RUN

More options

PARTICIPATION ACTIVITY 2.26.2: Load, store, and memory.

1. Run the simulation step-by-step, observing memory values.
2. Change M[4000]'s value to 45 by clicking the memory value on the right, then run again.
3. Store the addition result in M[4008] by appending `sw $t2, 8($s0)`

Assembly

```
Line 1 lw $t0, 0($s0)     # Load M[4000]
Line 2 lw $t1, 4($s0)     # Load M[4004]
Line 3 add $t2, $t0, $t1 # Add M[4000] and M[4004]
```


Registers

$t0	0
$t1	0
$t2	0
$s0	4000

Mem

4000	
4004	
4008	

ENTER SIMULATION STEP RUN

More options

PARTICIPATION ACTIVITY 2.26.3: Branching and labels.

1. Run the simulation step-by-step, observing register values.
2. Change $s0's value to 5, then run again.

Assembly

```
Line 1 add $t0, $zero, $zero
Line 2
Line 3 loop:
Line 4    slt $t1, $t0, $s0
Line 5    beq $t1, $zero, exit
Line 6    addi $t0, $t0, 1
Line 7    j loop
Line 8 exit:
```

Registers

$zero	0
$t0	0
$t1	0
$s0	2

Mer

4000

ENTER SIMULATION STEP RUN

More options