

Chapter 6 :: Architecture

Digital Design and Computer Architecture

David Money Harris and Sarah L. Harris

Copyright © 2007 Elsevier

6-<1>



Chapter 6 :: Topics

- **Introduction**
- **Assembly Language**
- **Machine Language**
- **Programming**
- **Addressing Modes**
- **Lights, Camera, Action: Compiling, Assembling, and Loading**
- **Odds and Ends**

Copyright © 2007 Elsevier

6-<2>



Introduction

- Jumping up a few levels of abstraction.
- **Architecture:** the programmer's view of the computer
 - Defined by instructions (operations) and operand locations
- **Microarchitecture:** how to implement an architecture in hardware (covered in Chapter 7)

Application Software	programs
Operating Systems	device drivers
Architecture	instructions registers
Micro-architecture	datapaths controllers
Logic	adders memories
Digital Circuits	AND gates NOT gates
Analog Circuits	amplifiers filters
Devices	transistors diodes
Physics	electrons

Copyright © 2007 Elsevier

6-<3>



Assembly Language

- To command a computer, you must understand its language.
 - **Instructions:** words in a computer's language
 - **Instruction set:** the vocabulary of a computer's language
- Instructions indicate the operation to perform and the operands to use.
 - **Assembly language:** human-readable format of instructions
 - **Machine language:** computer-readable format (1's and 0's)
- MIPS architecture:
 - Developed by John Hennessy and his colleagues at Stanford and in the 1980's.
 - Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco
- Once you've learned one architecture, it's easy to learn others.

Copyright © 2007 Elsevier

6-<4>



John Hennessy

- President of Stanford University
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Co-invented the Reduced Instruction Set Computer (RISC)
- Developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors have been sold



Copyright © 2007 Elsevier

6-<5>



Architecture Design Principles

Underlying design principles, as articulated by Hennessy and Patterson:

1. **Simplicity favors regularity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

Copyright © 2007 Elsevier

6-<6>



Instructions: Addition

High-level code

`a = b + c;`

MIPS assembly code

`add a, b, c`

- **add**: mnemonic indicates what operation to perform
- **b, c**: source operands on which the operation is performed
- **a**: destination operand to which the result is written

Copyright © 2007 Elsevier

6-<7>



Instructions: Subtraction

- Subtraction is similar to addition. Only the mnemonic changes.

High-level code

`a = b - c;`

MIPS assembly code

`sub a, b, c`

- **sub**: mnemonic indicates what operation to perform
- **b, c**: source operands on which the operation is performed
- **a**: destination operand to which the result is written

Copyright © 2007 Elsevier

6-<8>



Design Principle 1

Simplicity favors regularity

- Consistent instruction format
- Same number of operands (two sources and one destination)
 - easier to encode and handle in hardware

Copyright © 2007 Elsevier

6-<9>



Instructions: More Complex Code

- More complex code is handled by multiple MIPS instructions.

High-level code

```
a = b + c - d;  
// single line comment  
/* multiple line  
comment */
```

MIPS assembly code

```
add t, b, c # t = b + c  
sub a, t, d # a = t - d
```

Copyright © 2007 Elsevier

6-<10>



Design Principle 2

Make the common case fast

- MIPS includes only simple, commonly used instructions.
- Hardware to decode and execute the instruction can be simple, small, and fast.
- More complex instructions (that are less common) can be performed using multiple simple instructions.
- MIPS is a *reduced instruction set computer (RISC)*, with a small number of simple instructions.
- Other architectures, such as Intel's IA-32 found in many PC's, are *complex instruction set computers (CISC)*. They include complex instructions that are rarely used, such as the "string move" instruction that copies a string (a series of characters) from one part of memory to another.

Copyright © 2007 Elsevier

6-<11>



Operands

- A computer needs a physical location from which to retrieve binary operands
- A computer retrieves operands from:
 - Registers
 - Memory
 - Constants (also called *immediates*)

Copyright © 2007 Elsevier

6-<12>



Operands: Registers

- Memory is slow.
- Most architectures have a small set of (fast) registers.
- MIPS has thirty-two 32-bit registers.
- MIPS is called a 32-bit architecture because it operates on 32-bit data.

(A 64-bit version of MIPS also exists, but we will consider only MIPS32.)

Copyright © 2007 Elsevier

6-<13>



Design Principle 3

Smaller is Faster

- MIPS includes only a small number of registers
- Just as retrieving data from a few books on your table is faster than sorting through 1000 books, retrieving data from 32 registers is faster than retrieving it from 1000 registers or a large memory.

Copyright © 2007 Elsevier

6-<14>



The MIPS Register Set

Name	Register Number	Usage
\$0	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	procedure return values
\$a0-\$a3	4-7	procedure arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved variables
\$t8-\$t9	24-25	more temporaries
\$k0-\$k1	26-27	OS temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	procedure return address

Copyright © 2007 Elsevier

6-<15>



Operands: Registers

- Registers:
 - Written with a dollar sign (\$) before their name
 - For example, register 0 is written “\$0”, pronounced “register zero” or “dollar zero”.
- Certain registers used for specific purposes:
 - For example,
 - \$0 always holds the constant value 0.
 - the *saved registers*, \$s0-\$s7, are used to hold variables
 - the *temporary registers*, \$t0-\$t9, are used to hold intermediate values during a larger computation.
- For now, we only use the temporary registers (\$t0-\$t9) and the saved registers (\$s0-\$s7).
- We will use the other registers in later slides.

Copyright © 2007 Elsevier

6-<16>



Instructions with registers

- Revisit add instruction

High-level code

```
a = b + c
```

MIPS assembly code

```
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2
```

Copyright © 2007 Elsevier

6-<17>



Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, so it can hold a lot of data
- But it's also slow
- Commonly used variables kept in registers
- Using a combination of registers and memory, a program can access a large amount of data fairly quickly

Copyright © 2007 Elsevier

6-<18>



Word-Addressable Memory

- Each 32-bit data word has a unique address

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Copyright © 2007 Elsevier

6-<19>



Reading Word-Addressable Memory

- Memory reads are called *loads*
- Mnemonic: *load word* (*lw*)
- **Example:** read a word of data at memory address 1 into $\$s3$
- Memory address calculation:
 - add the *base address* ($\$0$) to the *offset* (1)
 - $\text{address} = (\$0 + 1) = 1$
- Any register may be used to store the base address.
- $\$s3$ holds the value 0xF2F1AC07 after the instruction completes.

Assembly code

```
lw $s3, 1($0) # read memory word 1 into $s3
```

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Copyright © 2007 Elsevier

6-<20>



Writing Word-Addressable Memory

- Memory writes are called *stores*
- Mnemonic: *store word (sw)*
- **Example:** Write (store) the value held in \$t4 into memory address 7
- Offset can be written in decimal (default) or hexadecimal
- Memory address calculation:
 - add the base address (\$0) to the offset (0x7)
 - address: ($\$0 + 0x7$) = 7
- Any register may be used to store the base address

Assembly code

```
sw $t4, 0x7($0) # write the value in $t4
                 # to memory word 7
```

Word Address	Data	
...
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Copyright © 2007 Elsevier

6-<21>



Byte-Addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (lb) and store byte (sb)
- Each 32-bit words has 4 bytes, so the word address increments by 4

Word Address	Data	
...
0000000C	4 0 F 3 0 7 8 8	Word 3
00000008	0 1 E E 2 8 4 2	Word 2
00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

← width = 4 bytes →

Copyright © 2007 Elsevier

6-<22>



Reading Byte-Addressable Memory

- The address of a memory word must now be multiplied by 4. For example,
 - the address of memory word 2 is $2 \times 4 = 8$
 - the address of memory word 10 is $10 \times 4 = 40$ (0x28)
- Load a word of data at memory address 4 into \$s3.
- \$s3 holds the value 0xF2F1AC07 after the instruction completes.
- **MIPS is byte-addressed, not word-addressed**

MIPS assembly code

```
lw $s3, 4($0) # read word at address 4 into $s3
```

Word Address	Data	
...
0000000C	4 0 F 3 0 7 8 8	Word 3
00000008	0 1 E E 2 8 4 2	Word 2
00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

← width = 4 bytes →

Copyright © 2007 Elsevier

6-<23>



Writing Byte-Addressable Memory

- **Example:** stores the value held in \$t7 into memory address 0x2C (44)

MIPS assembly code

```
sw $t7, 44($0) # write $t7 into address 44
```

Word Address	Data	
...
0000000C	4 0 F 3 0 7 8 8	Word 3
00000008	0 1 E E 2 8 4 2	Word 2
00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

← width = 4 bytes →

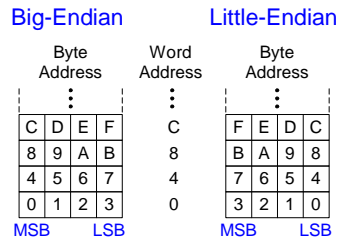
Copyright © 2007 Elsevier

6-<24>



Big-Endian and Little-Endian Memory

- How to number bytes within a word?
- Word address is the same for big- or little-endian
- Little-endian: byte numbers start at the little (least significant) end
- Big-endian: byte numbers start at the big (most significant) end



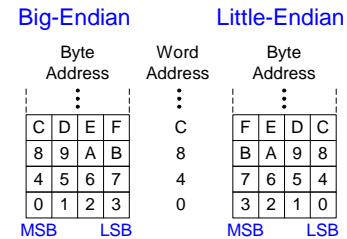
Copyright © 2007 Elsevier

6-<25>



Big-Endian and Little-Endian Memory

- From Jonathan Swift's *Gulliver's Travels* where the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end.
- As indicated by the farcical name, it doesn't really matter which addressing type is used – except when the two systems need to share data!



Copyright © 2007 Elsevier

6-<26>



Big- and Little-Endian Example

- Suppose \$t0 initially contains 0x23456789. After the following program is run on a big-endian system, what value does \$s0 contain? In a little-endian system?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

Copyright © 2007 Elsevier

6-<27>

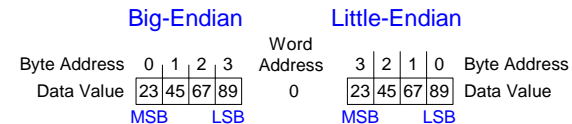


Big- and Little-Endian Example

- Suppose \$t0 initially contains 0x23456789. After the following program is run on a big-endian system, what value does \$s0 contain? In a little-endian system?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

- Big-endian: 0x00000045
- Little-endian: 0x00000067



Copyright © 2007 Elsevier

6-<28>



Design Principle 4

Good design demands good compromises

- Multiple instruction formats allow flexibility
 - add, sub: use 3 register operands
 - lw, sw: use 2 register operands and a constant
- Number of instruction formats kept small
 - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster).

Copyright © 2007 Elsevier

6-<29>



Operands: Constants/Immediates

- lw and sw illustrate the use of constants or *immediates*
- Called immediates because they are *immediately* available from the instruction
- Immediates don't require a register or memory access.
- The add immediate (addi) instruction adds an immediate to a variable (held in a register).
- An immediate is a 16-bit two's complement number.
- Is subtract immediate (subi) necessary?

High-level code

```
a = a + 4;  
b = a - 12;
```

MIPS assembly code

```
# $s0 = a, $s1 = b  
addi $s0, $s0, 4  
addi $s1, $s0, -12
```

Copyright © 2007 Elsevier

6-<30>



Machine Language

- Computers only understand 1's and 0's
- Machine language: binary representation of instructions
- 32-bit instructions
 - Again, simplicity favors regularity: 32-bit data and instructions
- Three instruction formats:
 - R-Type: register operands
 - I-Type: immediate operand
 - J-Type: for jumping (we'll discuss later)

Copyright © 2007 Elsevier

6-<31>



R-Type

- *Register-type*
- 3 register operands:
 - rs, rt: source registers
 - rd: destination register
- Other fields:
 - op: the *operation code* or *opcode* (0 for R-type instructions)
 - funct: the *function*
together, the opcode and function tell the computer what operation to perform
 - shamt: the *shift amount* for shift instructions, otherwise it's 0

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Copyright © 2007 Elsevier

6-<32>



R-Type Examples

Assembly Code

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

Machine Code

op	rs	rt	rd	shamt	funct
000000	10001	10010	10000	00000	100000
000000	01011	01101	01000	00000	100010

(0x02328020) (0x016D4022)

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

Note the order of registers in the assembly code:

```
add rd, rs, rt
```

Copyright © 2007 Elsevier

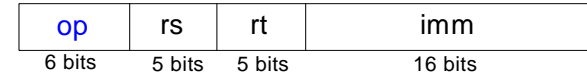
6-<33>



I-Type

- *Immediate-type*
- 3 operands:
 - rs, rt: register operands
 - imm: 16-bit two's complement immediate
- Other fields:
 - op: the opcode
 - Simplicity favors regularity: all instructions have opcode
 - Operation is completely determined by the opcode

I-Type



Copyright © 2007 Elsevier

6-<34>



I-Type Examples

Assembly Code

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw $t2, 32($0)
sw $s1, 4($t1)
```

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4

6 bits 5 bits 5 bits 16 bits

Note the differing order of registers in the assembly and machine codes:

```
addi rt, rs, imm
lw rt, imm(rs)
sw rt, imm(rs)
```

op	rs	rt	imm
001000	10001	10000	0000 0000 0000 0101
001000	10011	01000	1111 1111 1111 0100
100011	00000	01010	0000 0000 0010 0000
101011	01001	10001	0000 0000 0000 0100

(0x22300005) (0x2268FFF4) (0x8C0A0020) (0xAD310004)

6 bits 5 bits 5 bits 16 bits

Copyright © 2007 Elsevier

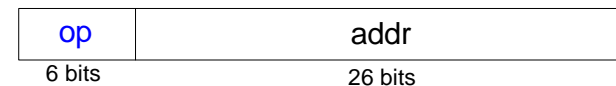
6-<35>



Machine Language: J-Type

- *Jump-type*
- 26-bit address operand (addr)
- Used for jump instructions (j)

J-Type



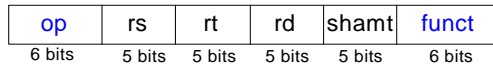
Copyright © 2007 Elsevier

6-<36>

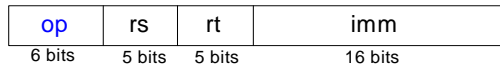


Review: Instruction Formats

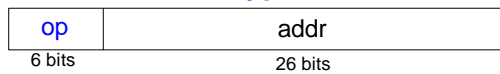
R-Type



I-Type



J-Type



Copyright © 2007 Elsevier

6-<37>



The Power of the Stored Program

- 32-bit instructions and data stored in memory
- Sequence of instructions: only difference between two applications (for example, a text editor and a video game)
- To run a new program:
 - No rewiring required
 - Simply store new program in memory
- The processor hardware executes the program:
 - *fetches* (reads) the instructions from memory in sequence
 - performs the specified operation
- The program counter (PC) keeps track of the current instruction
- In MIPS, programs typically start at memory address 0x00400000

Copyright © 2007 Elsevier

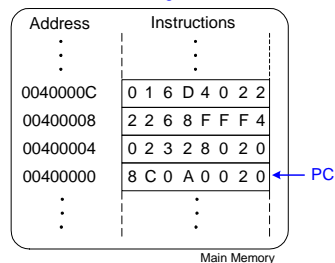
6-<38>



The Stored Program

Assembly Code	Machine Code
lw \$t2, 32(\$0)	0x8C0A0020
add \$s0, \$s1, \$s2	0x02328020
addi \$t0, \$s3, -12	0x2268FFF4
sub \$t0, \$t3, \$t5	0x016D4022

Stored Program



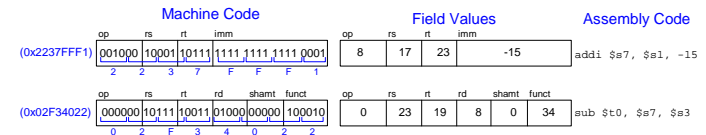
Copyright © 2007 Elsevier

6-<39>



Interpreting Machine Language Code

- Start with opcode
- Opcode tells how to parse the remaining bits
- If opcode is all 0's
 - R-type instruction
 - Function bits tell what instruction it is
- Otherwise
 - opcode tells what instruction it is



Copyright © 2007 Elsevier

6-<40>



Programming

- High-level languages:
 - e.g., C, Java, Python
 - Written at more abstract level
- Common high-level software constructs:
 - if/else statements
 - for loops
 - while loops
 - array accesses
 - procedure calls
- Other useful instructions:
 - Arithmetic/logical instructions
 - Branching

Copyright © 2007 Elsevier

6-<41>



Ada Lovelace, 1815 - 1852

- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage's Analytical Engine
- She was the only legitimate child of the poet Lord Byron



Copyright © 2007 Elsevier

6-<42>



Logical Instructions

- and, or, xor, nor
 - and: useful for *masking* bits
 - Masking all but the least significant byte of a value:
 $0xF234012F \text{ AND } 0x000000FF = 0x0000002F$
 - or: useful for combining bit fields
 - Combine $0xF2340000$ with $0x000012BC$:
 $0xF2340000 \text{ OR } 0x000012BC = 0xF23412BC$
 - nor: useful for inverting bits:
 - $\text{ANOR } \$0 = \text{NOT } A$
- andi, ori, xori
 - 16-bit immediate is zero-extended (*not* sign-extended)
 - nori not needed

Copyright © 2007 Elsevier

6-<43>



Logical Instruction Examples

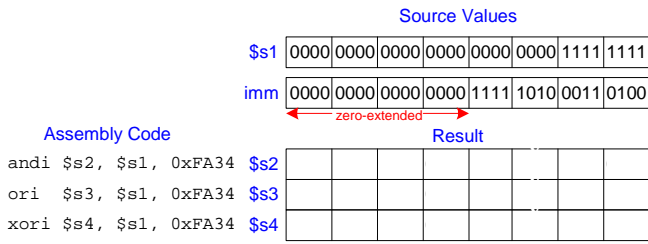
		Source Registers							
\$s1		1111	1111	1111	1111	0000	0000	0000	0000
\$s2		0100	0110	1010	0001	1111	0000	1011	0111
Assembly Code		Result							
and \$s3, \$s1, \$s2	\$s3	0100	0110	1010	0001	0000	0000	0000	0000
or \$s4, \$s1, \$s2	\$s4	1111	1111	1111	1111	1111	0000	1011	0111
xor \$s5, \$s1, \$s2	\$s5	1011	1001	0101	1110	1111	0000	1011	0111
nor \$s6, \$s1, \$s2	\$s6	0000	0000	0000	0000	0000	1111	0100	1000

Copyright © 2007 Elsevier

6-<44>



Logical Instruction Examples



Copyright © 2007 Elsevier

6-<45>



Shift Instructions

- sll: shift left logical
 - **Example:** sll \$t0, \$t1, 5 # \$t0 <= \$t1 << 5
- srl: shift right logical
 - **Example:** srl \$t0, \$t1, 5 # \$t0 <= \$t1 >> 5
- sra: shift right arithmetic
 - **Example:** sra \$t0, \$t1, 5 # \$t0 <= \$t1 >>> 5

Variable shift instructions:

- sllv: shift left logical variable
 - **Example:** sllv \$t0, \$t1, \$t2 # \$t0 <= \$t1 << \$t2
- srlv: shift right logical variable
 - **Example:** srlv \$t0, \$t1, \$t2 # \$t0 <= \$t1 >> \$t2
- srav: shift right arithmetic variable
 - **Example:** srav \$t0, \$t1, \$t2 # \$t0 <= \$t1 >>> \$t2

Copyright © 2007 Elsevier

6-<46>



Shift Instructions

Assembly Code

```

sll $t0, $s1, 2
srl $s2, $s1, 2
sra $s3, $s1, 2
    
```

Field Values

op	rs	rt	rd	shamt	funct
0	0	17	8	2	0
0	0	17	18	2	2
0	0	17	19	2	3

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

Machine Code

op	rs	rt	rd	shamt	funct
000000	00000	10001	01000	00010	000000 (0x00114080)
000000	00000	10001	10010	00010	000010 (0x00119082)
000000	00000	10001	10011	00010	000011 (0x00119883)

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

Copyright © 2007 Elsevier

6-<47>



Generating Constants

- 16-bit constants using addi:

High-level code

```

// int is a 32-bit signed word
int a = 0x4f3c;
    
```

MIPS assembly code

```

# $s0 = a
addi $s0, $0, 0x4f3c
    
```

- 32-bit constants using load upper immediate (lui) and ori:

(lui loads the 16-bit immediate into the upper half of the register and sets the lower half to 0.)

High-level code

```

int a = 0xFEDC8765;
    
```

MIPS assembly code

```

# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
    
```

Copyright © 2007 Elsevier

6-<48>



Multiplication, Division

- Special registers: lo, hi
- 32×32 multiplication, 64 bit result
 - mult \$s0, \$s1
 - Result in {hi, lo}
- 32-bit division, 32-bit quotient, 32-bit remainder
 - div \$s0, \$s1
 - Quotient in lo
 - Remainder in hi
- Moves from lo/hi special registers
 - mflo \$s2
 - mfhi \$s3

Copyright © 2007 Elsevier

6-<49>



Branching

- Allows a program to execute instructions out of sequence.
- Types of branches:
 - **Conditional branches**
 - branch if equal (beq)
 - branch if not equal (bne)
 - **Unconditional branches**
 - jump (j)
 - jump register (jr)
 - jump and link (jal)

Copyright © 2007 Elsevier

6-<50>



Review: The Stored Program

Assembly Code	Machine Code
lw \$t2, 32(\$0)	0x8C0A0020
add \$s0, \$s1, \$s2	0x02328020
addi \$t0, \$s3, -12	0x2268FFF4
sub \$t0, \$t3, \$t5	0x016D4022

Stored Program

Address	Instructions
...	...
0040000C	0 1 6 D 4 0 2 2
00400008	2 2 6 8 F F F 4
00400004	0 2 3 2 8 0 2 0
00400000	8 C 0 A 0 0 2 0 ← PC
...	...

Main Memory

Copyright © 2007 Elsevier

6-<51>



Conditional Branching (beq)

MIPS assembly

```

addi $s0, $0, 4      # $s0 = 0 + 4 = 4
addi $s1, $0, 1      # $s1 = 0 + 1 = 1
sll $s1, $s1, 2      # $s1 = 1 << 2 = 4
beq $s0, $s1, target # branch is taken
addi $s1, $s1, 1     # not executed
sub $s1, $s1, $s0    # not executed
    
```

```

target:                # label
add $s1, $s1, $s0     # $s1 = 4 + 4 = 8
    
```

Labels indicate instruction locations in a program. They cannot use reserved words and must be followed by a colon (:).

Copyright © 2007 Elsevier

6-<52>



The Branch Not Taken (bne)

MIPS assembly

```
addi    $s0, $0, 4           # $s0 = 0 + 4 = 4
addi    $s1, $0, 1           # $s1 = 0 + 1 = 1
sll     $s1, $s1, 2           # $s1 = 1 << 2 = 4
bne     $s0, $s1, target     # branch not taken
addi    $s1, $s1, 1           # $s1 = 4 + 1 = 5
sub     $s1, $s1, $s0         # $s1 = 5 - 4 = 1

target:
add     $s1, $s1, $s0         # $s1 = 1 + 4 = 5
```

Copyright © 2007 Elsevier

6-<53>



Unconditional Branching / Jumping (j)

MIPS assembly

```
addi    $s0, $0, 4           # $s0 = 4
addi    $s1, $0, 1           # $s1 = 1
j       target               # jump to target
sra     $s1, $s1, 2           # not executed
addi    $s1, $s1, 1           # not executed
sub     $s1, $s1, $s0         # not executed

target:
add     $s1, $s1, $s0         # $s1 = 1 + 4 = 5
```

Copyright © 2007 Elsevier

6-<54>



Unconditional Branching (jr)

MIPS assembly

```
0x00002000    addi $s0, $0, 0x2010
0x00002004    jr   $s0
0x00002008    addi $s1, $0, 1
0x0000200C    sra  $s1, $s1, 2
0x00002010    lw   $s3, 44($s1)
```

Note: jr is an R-type instruction.

Copyright © 2007 Elsevier

6-<55>



High-Level Code Constructs

- if statements
- if/else statements
- while loops
- for loops

Copyright © 2007 Elsevier

6-<56>



If Statement

High-level code

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

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```

```
L1:
```

Notice that the assembly tests for the opposite case ($i \neq j$) than the test in the high-level code ($i == j$).

Copyright © 2007 Elsevier

6-<57>



If / Else Statement

High-level code

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

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```

```
L1:
done:
```

Copyright © 2007 Elsevier

6-<58>



While Loops

High-level code

```
// determines the power
// of x such that 2x = 128
int pow = 1;
int x = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

MIPS assembly code

```
# $s0 = pow, $s1 = x
```

```
while:  
```

```
done:
```

Notice that the assembly tests for the opposite case ($pow == 128$) than the test in the high-level code ($pow \neq 128$).

Copyright © 2007 Elsevier

6-<59>



For Loops

The general form of a for loop is:

```
for (initialization; condition; loop operation)
    loop body
```

- **initialization**: executes before the loop begins
- **condition**: is tested at the beginning of each iteration
- **loop operation**: executes at the end of each iteration
- **loop body**: executes each time the condition is met

Copyright © 2007 Elsevier

6-<60>



For Loops

High-level code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
for:
    sum = sum + i
    i = i + 1
    bne i, 10, done
done:
```

Notice that the assembly tests for the opposite case ($i == 10$) than the test in the high-level code ($i != 10$).

Copyright © 2007 Elsevier

6-<61>



Less Than Comparisons

High-level code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
    sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
loop:
    sum = sum + i
    i = i * 2
    bge $s0, 101, done
done:
```

$t1 = 1$ if $i < 101$.

Copyright © 2007 Elsevier

6-<62>



Arrays

- Useful for accessing large amounts of similar data
- Array element: accessed by *index*
- Array *size*: number of elements in the array

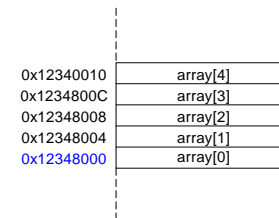
Copyright © 2007 Elsevier

6-<63>



Arrays

- 5-element array
- **Base address** = 0x12348000 (address of the first array element, `array[0]`)
- First step in accessing an array: load base address into a register



Copyright © 2007 Elsevier

6-<64>



Arrays

```
// high-level code
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;

# MIPS assembly code
# array base address = $s0
```

Copyright © 2007 Elsevier

6-<65>



Arrays

```
// high-level code
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;

# MIPS assembly code
# array base address = $s0
lui $s0, 0x1234 # put 0x1234 in upper half of $S0
ori $s0, $s0, 0x8000 # put 0x8000 in lower half of $s0

lw $t1, 0($s0) # $t1 = array[0]
sll $t1, $t1, 1 # $t1 = $t1 * 2
sw $t1, 0($s0) # array[0] = $t1

lw $t1, 4($s0) # $t1 = array[1]
sll $t1, $t1, 1 # $t1 = $t1 * 2
sw $t1, 4($s0) # array[1] = $t1
```

Copyright © 2007 Elsevier

6-<66>



Arrays Using For Loops

```
// high-level code
int array[1000];
int i;

for (i=0; i < 1000; i = i + 1)
    array[i] = array[i] * 8;

# MIPS assembly code
# $s0 = array base address, $s1 = i
```

Copyright © 2007 Elsevier

6-<67>



Arrays Using For Loops

```
# MIPS assembly code
# $s0 = array base address, $s1 = i
# initialization code
lui $s0, 0x23B8 # $s0 = 0x23B80000
ori $s0, $s0, 0xF000 # $s0 = 0x23B8F000
addi $s1, $0, 0 # i = 0
addi $t2, $0, 1000 # $t2 = 1000

loop:
slt $t0, $s1, $t2 # i < 1000?
beq $t0, $0, done # if not then done
sll $t0, $s1, 2 # $t0 = i * 4 (byte offset)
add $t0, $t0, $s0 # address of array[i]
lw $t1, 0($t0) # $t1 = array[i]
sll $t1, $t1, 3 # $t1 = array[i] * 8
sw $t1, 0($t0) # array[i] = array[i] * 8
addi $s1, $s1, 1 # i = i + 1
j loop # repeat
done:
```

Copyright © 2007 Elsevier

6-<68>



ASCII Codes

- *American Standard Code for Information Interchange*
 - assigns each text character a unique byte value
- For example, S = 0x53, a = 0x61, A = 0x41
- Lower-case and upper-case letters differ by 0x20 (32).

Copyright © 2007 Elsevier

6-<69>



Cast of Characters

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	'	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	\$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(38	8	48	H	58	X	68	h	78	x
29)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	
2D	-	3D	=	4D	M	5D]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Copyright © 2007 Elsevier

6-<70>



Procedure Calls

Definitions

- **Caller:** calling procedure (in this case, `main`)
- **Callee:** called procedure (in this case, `sum`)

High-level code

```
void main()
{
    int y;
    y = sum(42, 7);
    ...
}

int sum(int a, int b)
{
    return (a + b);
}
```

Copyright © 2007 Elsevier

6-<71>



Procedure Calls

Procedure calling conventions:

- Caller:
 - passes **arguments** to callee.
 - jumps to the callee
- Callee:
 - **performs the procedure**
 - **returns the result** to caller
 - **returns to the point of call**
 - **must not overwrite** registers or memory needed by the caller

MIPS conventions:

- Call procedure: jump and link (`jal`)
- Return from procedure: jump register (`jr`)
- Argument values: `$a0 - $a3`
- Return value: `$v0`

Copyright © 2007 Elsevier

6-<72>



Procedure Calls

High-level code

```
int main() {
    simple();
    a = b + c;
}

void simple() {
    return;
}
```

MIPS assembly code

```
0x00400200 main: jal simple
0x00400204      add $s0, $s1, $s2
...

0x00401020 simple: jr $ra
```

void means that `simple` doesn't return a value.

Copyright © 2007 Elsevier

6-<73>



Procedure Calls

High-level code

```
int main() {
    simple();
    a = b + c;
}

void simple() {
    return;
}
```

MIPS assembly code

```
0x00400200 main: jal simple
0x00400204      add $s0, $s1, $s2
...

0x00401020 simple: jr $ra
```

`jal`: jumps to `simple` and saves PC+4 in the return address register (`$ra`).
In this case, `$ra = 0x00400204` after `jal` executes.

`jr $ra`: jumps to address in `$ra`, in this case `0x00400204`.

Copyright © 2007 Elsevier

6-<74>



Input Arguments and Return Values

MIPS conventions:

- Argument values: `$a0 - $a3`
- Return value: `$v0`

Copyright © 2007 Elsevier

6-<75>



Input Arguments and Return Values

High-level code

```
int main()
{
    int y;
    ...
    y = diffofsums(2, 3, 4, 5); // 4 arguments
    ...
}

int diffofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result;           // return value
}
```

Copyright © 2007 Elsevier

6-<76>



Input Arguments and Return Values

MIPS assembly code

```
# $s0 = y

main:
...
addi $a0, $0, 2 # argument 0 = 2
addi $a1, $0, 3 # argument 1 = 3
addi $a2, $0, 4 # argument 2 = 4
addi $a3, $0, 5 # argument 3 = 5
jal diffofsums # call procedure
add $s0, $v0, $0 # y = returned value
...

# $s0 = result
diffofsums:
add $t0, $a0, $a1 # $t0 = f + g
add $t1, $a2, $a3 # $t1 = h + i
sub $s0, $t0, $t1 # result = (f + g) - (h + i)
add $v0, $s0, $0 # put return value in $v0
jr $ra # return to caller
```

Copyright © 2007 Elsevier

6-<77>



Input Arguments and Return Values

MIPS assembly code

```
# $s0 = result
diffofsums:
add $t0, $a0, $a1 # $t0 = f + g
add $t1, $a2, $a3 # $t1 = h + i
sub $s0, $t0, $t1 # result = (f + g) - (h + i)
add $v0, $s0, $0 # put return value in $v0
jr $ra # return to caller
```

- diffofsums overwrote 3 registers: \$t0, \$t1, and \$s0
- diffofsums can use the *stack* to temporarily store registers

Copyright © 2007 Elsevier

6-<78>



The Stack

- Memory used to temporarily save variables
- Like a stack of dishes, last-in-first-out (LIFO) queue
- *Expands*: uses more memory when more space is needed
- *Contracts*: uses less memory when the space is no longer needed



Copyright © 2007 Elsevier

6-<79>



The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: \$sp, points to top of the stack

Address	Data		Address	Data
7FFFFFFC	12345678	←\$sp	7FFFFFFC	12345678
7FFFFFF8			7FFFFFF8	AABBCCDD
7FFFFFF4			7FFFFFF4	11223344
7FFFFFF0			7FFFFFF0	
⋮	⋮		⋮	⋮
⋮	⋮		⋮	⋮

Copyright © 2007 Elsevier

6-<80>



How Procedures use the Stack

- Called procedures must have no other unintended side effects.
- But `diffofsums` overwrites 3 registers: `$t0`, `$t1`, `$s0`

```
# MIPS assembly
# $s0 = result
diffofsums:
    add $t0, $a0, $a1 # $t0 = f + g
    add $t1, $a2, $a3 # $t1 = h + i
    sub $s0, $t0, $t1 # result = (f + g) - (h + i)
    add $v0, $s0, $0  # put return value in $v0
    jr  $ra           # return to caller
```

Copyright © 2007 Elsevier

6-81>



Storing Register Values on the Stack

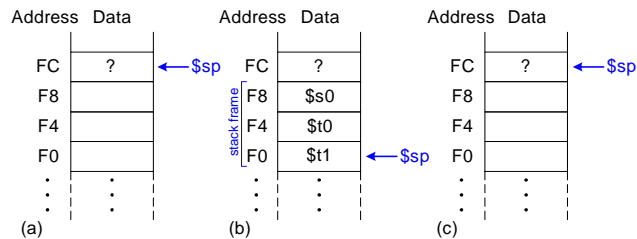
```
# $s0 = result
diffofsums:
    addi $sp, $sp, -12 # make space on stack
                        # to store 3 registers
    sw  $s0, 8($sp)    # save $s0 on stack
    sw  $t0, 4($sp)    # save $t0 on stack
    sw  $t1, 0($sp)    # save $t1 on stack
    add $t0, $a0, $a1 # $t0 = f + g
    add $t1, $a2, $a3 # $t1 = h + i
    sub $s0, $t0, $t1 # result = (f + g) - (h + i)
    add $v0, $s0, $0  # put return value in $v0
    lw  $t1, 0($sp)   # restore $t1 from stack
    lw  $t0, 4($sp)   # restore $t0 from stack
    lw  $s0, 8($sp)   # restore $s0 from stack
    addi $sp, $sp, 12 # deallocate stack space
    jr  $ra           # return to caller
```

Copyright © 2007 Elsevier

6-82>



The Stack during `diffofsums` Call



Copyright © 2007 Elsevier

6-83>



Registers

Preserved <i>Callee-Saved</i>	Nonpreserved <i>Caller-Saved</i>
<code>\$s0 - \$s7</code>	<code>\$t0 - \$t9</code>
<code>\$ra</code>	<code>\$a0 - \$a3</code>
<code>\$sp</code>	<code>\$v0 - \$v1</code>
stack above <code>\$sp</code>	stack below <code>\$sp</code>

Copyright © 2007 Elsevier

6-84>



Multiple Procedure Calls

```
proc1:
  addi $sp, $sp, -4 # make space on stack
  sw   $ra, 0($sp) # save $ra on stack
  jal  proc2
  ...
  lw   $ra, 0($sp) # restore $s0 from stack
  addi $sp, $sp, 4 # deallocate stack space
  jr   $ra         # return to caller
```

Copyright © 2007 Elsevier

6-<85>



Storing Saved Registers on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4 # make space on stack to
                   # store one register
  sw   $s0, 0($sp) # save $s0 on stack
                   # no need to save $t0 or $t1

  add $t0, $a0, $a1 # $t0 = f + g
  add $t1, $a2, $a3 # $t1 = h + i
  sub $s0, $t0, $t1 # result = (f + g) - (h + i)
  add $v0, $s0, $0 # put return value in $v0
  lw   $s0, 0($sp) # restore $s0 from stack
  addi $sp, $sp, 4 # deallocate stack space
  jr   $ra         # return to caller
```

Copyright © 2007 Elsevier

6-<86>



Recursive Procedure Call

High-level code

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

Copyright © 2007 Elsevier

6-<87>



Recursive Procedure Call

MIPS assembly code

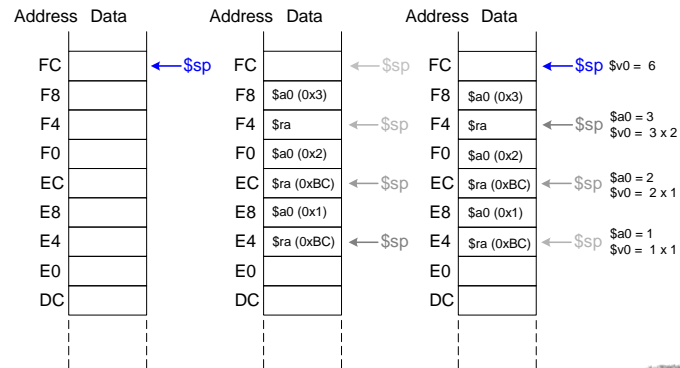
```
0x90 factorial: addi $sp, $sp, -8 # make room
0x94           sw   $a0, 4($sp) # store $a0
0x98           sw   $ra, 0($sp) # store $ra
0x9C           addi $t0, $0, 2
0xA0           slt  $t0, $a0, $t0 # a <= 1 ?
0xA4           beq  $t0, $0, else # no: go to else
0xA8           addi $v0, $0, 1 # yes: return 1
0xAC           addi $sp, $sp, 8 # restore $sp
0xB0           jr   $ra         # return
0xB4           else: addi $a0, $a0, -1 # n = n - 1
0xB8           jal  factorial # recursive call
0xBC           lw   $ra, 0($sp) # restore $ra
0xC0           lw   $a0, 4($sp) # restore $a0
0xC4           addi $sp, $sp, 8 # restore $sp
0xC8           mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC           jr   $ra         # return
```

Copyright © 2007 Elsevier

6-<88>



Stack during Recursive Call



Copyright © 2007 Elsevier

6-89>



Procedure Call Summary

- Caller
 - Put arguments in \$a0-\$a3
 - Save any registers that are needed (\$ra, maybe \$t0-t9)
 - jal callee
 - Restore registers
 - Look for result in \$v0
- Callee
 - Save registers that might be disturbed (\$s0-\$s7)
 - Perform procedure
 - Put result in \$v0
 - Restore registers
 - jr \$ra

Copyright © 2007 Elsevier

6-90>



Addressing Modes

How do we address the operands?

- Register Only
- Immediate
- Base Addressing
- PC-Relative
- Pseudo Direct

Copyright © 2007 Elsevier

6-91>



Addressing Modes

Register Only Addressing

- Operands found in registers
 - Example: add \$s0, \$t2, \$t3
 - Example: sub \$t8, \$s1, \$0

Immediate Addressing

- 16-bit immediate used as an operand
 - Example: addi \$s4, \$t5, -73
 - Example: ori \$t3, \$t7, 0xFF

Copyright © 2007 Elsevier

6-92>



Addressing Modes

Base Addressing

- Address of operand is:
base address + sign-extended immediate

– Example: `lw $s4, 72($0)`

- Address = \$0 + 72

– Example: `sw $t2, -25($t1)`

- Address = \$t1 - 25

Copyright © 2007 Elsevier

6-<93>



Addressing Modes

PC-Relative Addressing

```

0x10      beq  $t0, $0, else
0x14      addi $v0, $0, 1
0x18      addi $sp, $sp, i
0x1C      jr   $ra
0x20      else: addi $a0, $a0, -1
0x24      jal  factorial
    
```

Assembly Code

Field Values

op	rs	rt	imm
4	8	0	3
6 bits	5 bits	5 bits	5 bits 5 bits 6 bits

```

beq $t0, $0, else
(beq $t0, $0, 3)
    
```

Copyright © 2007 Elsevier

6-<94>



Addressing Modes

Pseudo-direct Addressing

```

0x0040005C      jal  sum
...
0x004000A0      sum: add  $v0, $a0, $a1
    
```

JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)
 26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)
 0 1 0 0 0 0 2 8

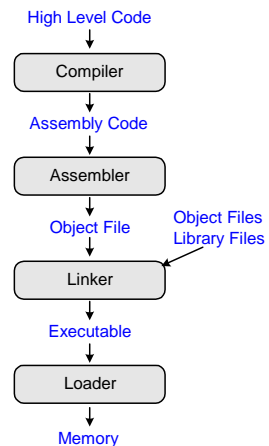
Field Values		Machine Code	
op	imm	op	addr
3	0x0100028	000011	00 0001 0000 0000 0000 0010 1000
6 bits	26 bits	6 bits	26 bits

Copyright © 2007 Elsevier

6-<95>



How do we compile & run an application?



Copyright © 2007 Elsevier

6-<96>



Grace Hopper, 1906 - 1992

- Graduated from Yale University with PhD in mathematics
- Developed first compiler
- Helped develop the COBOL programming language
- Highly awarded naval officer
- Received World War II Victory Medal and National Defense Service Medal, among others



Copyright © 2007 Elsevier

6-<97>



What needs to be stored in memory?

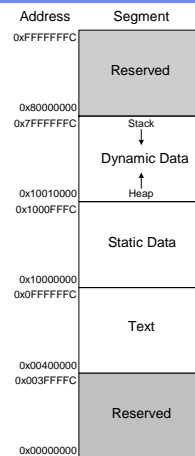
- Instructions (also called *text*)
- Data
 - Global/static: allocated before program begins
 - Dynamic: allocated within program
- How big is memory?
 - At most $2^{32} = 4$ gigabytes (4 GB)
 - From address 0x00000000 to 0xFFFFFFFF

Copyright © 2007 Elsevier

6-<98>



The MIPS Memory Map



Copyright © 2007 Elsevier

6-<99>



Example Program: C Code

```
int f, g, y; // global variables

int main(void)
{
    f = 2;
    g = 3;
    y = sum(f, g);

    return y;
}

int sum(int a, int b) {
    return (a + b);
}
```

Copyright © 2007 Elsevier

6-<100>



Example Program: Assembly Code

```

int f, g, y; // global
int main(void)
{
    f = 2;
    g = 3;
    y = sum(f, g);
    return y;
}

int sum(int a, int b) {
    return (a + b);
}

```

```

.data
f:
g:
y:
.text
main:
    addi $sp, $sp, -4 # stack frame
    sw $ra, 0($sp) # store $ra
    addi $a0, $0, 2 # $a0 = 2
    sw $a0, f # f = 2
    addi $a1, $0, 3 # $a1 = 3
    sw $a1, g # g = 3
    jal sum # call sum
    sw $v0, y # y = sum()
    lw $ra, 0($sp) # restore $ra
    addi $sp, $sp, 4 # restore $sp
sum:
    add $v0, $a0, $a1 # $v0 = a + b
    jr $ra # return to OS

```

Copyright © 2007 Elsevier

Example Program: Symbol Table

Symbol	Address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

Copyright © 2007 Elsevier

6-104>

Example Program: Executable

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)

Text segment	Address	Instruction
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008

Data segment	Address	Data
	0x10000000	f
	0x10000004	g
	0x10000008	y

```

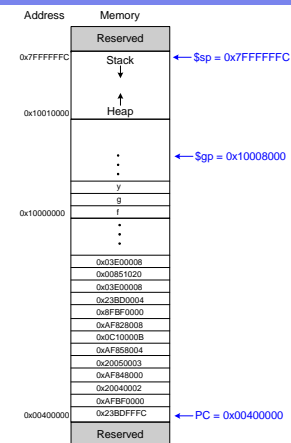
addi $sp, $sp, -4
sw $ra, 0($sp)
addi $a0, $0, 2
sw $a0, 0x8000($gp)
addi $a1, $0, 3
sw $a1, 0x8004($gp)
jal 0x0040002C($gp)
sw $v0, 0x8008($gp)
lw $ra, 0($sp)
addi $sp, $sp, 4
jr $ra
add $v0, $a0, $a1
jr $ra

```

Copyright © 2007 Elsevier

6-103>

Example Program: In Memory



Copyright © 2007 Elsevier

6-104>

Odds and Ends

- Pseudoinstructions
- Exceptions
- Signed and unsigned instructions
- Floating-point instructions

Copyright © 2007 Elsevier

6-<105>



Pseudoinstruction Examples

Pseudoinstruction	MIPS Instructions
li \$s0, 0x1234AA77	lui \$s0, 0x1234 ori \$s0, 0xAA77
mul \$s0, \$s1, \$s2	mult \$s1, \$s2 mflo \$s0
clear \$t0	add \$t0, \$0, \$0
move \$s1, \$s2	add \$s2, \$s1, \$0
nop	sll \$0, \$0, 0

Copyright © 2007 Elsevier

6-<106>



Exceptions

- Unscheduled procedure call to the *exception handler*
- Caused by:
 - Hardware, also called an *interrupt*, e.g. keyboard
 - Software, also called *traps*, e.g. undefined instruction
- When exception occurs, the processor:
 - Records the cause of the exception
 - Jumps to the exception handler at instruction address 0x80000180
 - Returns to program

Copyright © 2007 Elsevier

6-<107>



Exception Registers

- Not part of the register file.
 - Cause
 - Records the cause of the exception
 - EPC (Exception PC)
 - Records the PC where the exception occurred
- EPC and Cause: part of Coprocessor 0
- Move from Coprocessor 0
 - mfc0 \$t0, EPC
 - Moves the contents of EPC into \$t0

Copyright © 2007 Elsevier

6-<108>



Exception Causes

Exception	Cause
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Divide by 0	0x00000024
Undefined Instruction	0x00000028
Arithmetic Overflow	0x00000030

Copyright © 2007 Elsevier

6-<109>



Exceptions

- Processor saves cause and exception PC in Cause and EPC
- Processor jumps to exception handler (0x80000180)
- Exception handler:
 - Saves registers on stack
 - Reads the Cause register
`mfc0 $t0, Cause`
 - Handles the exception
 - Restores registers
 - Returns to program
`mfc0 $k0, EPC`
`jr $k0`

Copyright © 2007 Elsevier

6-<110>



Signed and Unsigned Instructions

- Addition and subtraction
- Multiplication and division
- Set less than

Copyright © 2007 Elsevier

6-<111>



Addition and Subtraction

- **Signed:** `add`, `addi`, `sub`
 - Same operation as unsigned versions
 - But processor takes exception on overflow
- **Unsigned:** `addu`, `addiu`, `subu`
 - Doesn't take exception on overflow
 - **Note:** `addiu` sign-extends the immediate

Copyright © 2007 Elsevier

6-<112>



Multiplication and Division

- **Signed:** mult, div
- **Unsigned:** multu, divu

Copyright © 2007 Elsevier

6-<113>



Set Less Than

- **Signed:** slt, slti
- **Unsigned:** sltu, sltiu
 - **Note:** sltiu sign-extends the immediate before comparing it to the register

Copyright © 2007 Elsevier

6-<114>



Loads

- **Signed:**
 - Sign-extends to create 32-bit value to load into register
 - Load halfword: lh
 - Load byte: lb
- **Unsigned:** addu, addiu, subu
 - Zero-extends to create 32-bit value
 - Load halfword unsigned: lhu
 - Load byte: lbu

Copyright © 2007 Elsevier

6-<115>



Floating-Point Instructions

- Floating-point coprocessor (Coprocessor 1)
- 32 32-bit floating-point registers (\$f0 - \$f31)
- Double-precision values held in two floating point registers
 - e.g., \$f0 and \$f1, \$f2 and \$f3, etc.
 - So, double-precision floating point registers: \$f0, \$f2, \$f4, etc.

Copyright © 2007 Elsevier

6-<116>



Floating-Point Instructions

Name	Register Number	Usage
\$fv0 - \$fv1	0, 2	return values
\$ft0 - \$ft3	4, 6, 8, 10	temporary variables
\$fa0 - \$fa1	12, 14	procedure arguments
\$ft4 - \$ft8	16, 18	temporary variables
\$fs0 - \$fs5	20, 22, 24, 26, 28, 30	saved variables

Copyright © 2007 Elsevier

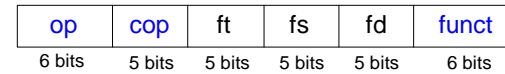
6-<117>



F-Type Instruction Format

- Opcode = 17 (010001₂)
- Single-precision:
 - cop = 16 (010000₂)
 - add.s, sub.s, div.s, neg.s, abs.s, etc.
- Double-precision:
 - cop = 17 (010001₂)
 - add.d, sub.d, div.d, neg.d, abs.d, etc.
- 3 register operands:
 - fs, ft: source operands
 - fd: destination operands

F-Type



Copyright © 2007 Elsevier

6-<118>



Floating-Point Branches

- Set/clear condition flag: fpcond
 - Equality: c.seq.s, c.seq.d
 - Less than: c.lt.s, c.lt.d
 - Less than or equal: c.le.s, c.le.d
- Conditional branch
 - bclf: branches if fpcond is FALSE
 - bclt: branches if fpcond is TRUE
- Loads and stores
 - lwc1: lwc1 \$ft1, 42(\$s1)
 - swc1: swc1 \$fs2, 17(\$sp)

Copyright © 2007 Elsevier

6-<119>



Looking Ahead

- Microarchitecture – building MIPS processor in hardware!

Bring colored pencils

Copyright © 2007 Elsevier

6-<120>

