

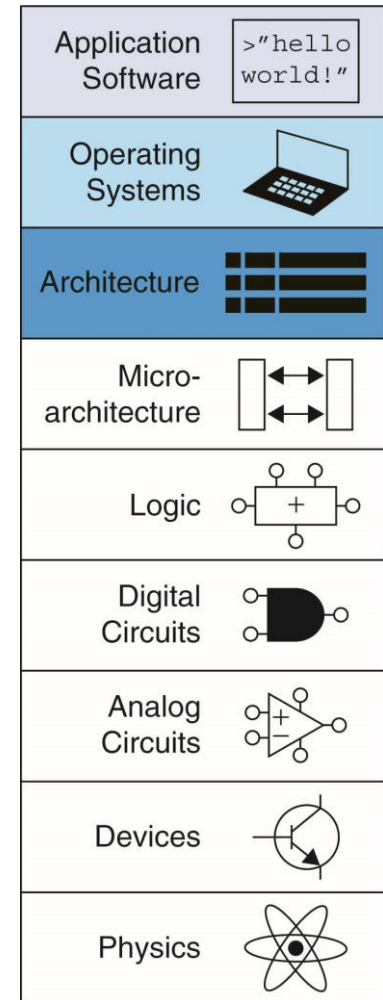
Chapter 6

Digital Design and Computer Architecture, 2nd Edition

David Money Harris and Sarah L. Harris

Chapter 6 :: Topics

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



Introduction

- Jumping up a few levels of abstraction
- **Architecture:** (Chapter 6) programmer's view of computer
 - Defined by instructions & 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



Assembly Language

- **Instructions:** commands in a computer's language
 - **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

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) with David Patterson
- Developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors had been sold



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**

Instructions: Addition

C Code

```
a = b + c;
```

MIPS assembly code

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

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

Instructions: Subtraction

- Similar to addition - only mnemonic changes

C Code

```
a = b - c;
```

MIPS assembly code

```
sub a, b, c
```

- **sub:** mnemonic
- **b, c:** source operands
- **a:** destination operand

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

Multiple Instructions

- More complex code is handled by multiple MIPS instructions.

C Code

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

MIPS assembly code

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

Design Principle 2

Make the common case fast

- MIPS includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) are 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 x86, are *complex instruction set computers (CISC)*

Operands

- Operand location: physical location in computer
 - Register operands
 - Memory operands
 - Immediate operands (located in the instruction itself (used for constants))

Operands: Registers

- MIPS has 32 32-bit registers
- Registers are faster than memory
- MIPS called “32-bit architecture” because it operates on 32-bit data

Design Principle 3

Smaller is Faster

- MIPS includes only a small number of registers
- MIPS includes only a small number of instructions (*reduced instruction set*)

MIPS Register Set

Name	Register Number	Usage
\$0	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	Function return values
\$a0-\$a3	4-7	Function 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	Function return address

Operands: Registers

- Registers:
 - \$ before name
 - Example: \$0, “register zero”, “dollar zero”
- Registers used for specific purposes:
 - \$0 always holds the constant value 0.
 - the *saved registers*, \$s0–\$s7, used to hold variables
 - the *temporary registers*, \$t0 - \$t9, used to hold intermediate values during a larger computation
 - Discuss others later

Instructions with Registers

- Revisit add instruction

C Code

```
a = b + c
```

MIPS assembly code

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

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables kept in registers

(Example of principle #2: Make the common case fast)

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

Reading Word-Addressable Memory

- Memory read called *load*
- **Mnemonic:** *load word* (lw)
- **Format:**
 $lw \$s0, 5(\$t1)$
- **Address calculation:**
 - add *base address* ($\$t1$) to the *offset* (5)
 - $address = (\$t1 + 5)$
- **Result:**
 - $\$s0$ holds the value at address $(\$t1 + 5)$

Any register may be used as base address

Reading Word-Addressable Memory

- **Example:** read a word of data at memory address 1 into \$s3
 - address = ($\$0 + 1$) = 1
 - \$s3 = 0xF2F1AC07 after load

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

Writing Word-Addressable Memory

- Memory write are called *store*
- **Mnemonic:** *store word* (sw)

Writing Word-Addressable Memory

• **Example:** Write (store) the value in $\$t4$ into memory address 7

- add the base address ($\$0$) to the offset ($0x7$)
- address: $(\$0 + 0x7) = 7$

Offset can be written in decimal (default) or hexadecimal

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

Byte-Addressable Memory

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

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)
- MIPS is byte-addressed, not word-addressed


Reading Byte-Addressable Memory

- Example:** Load a word of data at memory address 4 into \$s3.
- \$s3 holds the value 0xF2F1AC07 after load

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



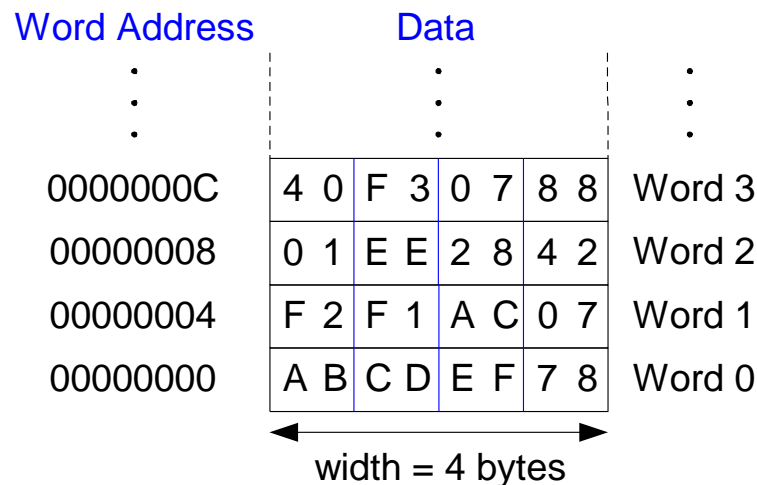
width = 4 bytes

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
```



Big-Endian & Little-Endian Memory

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

Big-Endian

Byte Address			
⋮			
C	D	E	F
8	9	A	B
4	5	6	7
0	1	2	3
MSB		LSB	

Word Address
⋮
C
8
4
0

Little-Endian

Byte Address			
⋮			
F	E	D	C
B	A	9	8
7	6	5	4
3	2	1	0
MSB		LSB	

Big-Endian & Little-Endian Memory

- Jonathan Swift's *Gulliver's Travels*: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- It doesn't really matter which addressing type used – except when the two systems need to share data!

Big-Endian

Byte Address			
⋮			
C	D	E	F
8	9	A	B
4	5	6	7
0	1	2	3
MSB		LSB	

Little-Endian

Byte Address			
⋮			
F	E	D	C
B	A	9	8
7	6	5	4
3	2	1	0
MSB		LSB	

Big-Endian & Little-Endian Example

- Suppose `$t0` initially contains `0x23456789`
- After following code runs on big-endian system, what value is `$s0`?

```
sw $t0, 0($0)
```

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

- In a little-endian system, what is `$s0`?

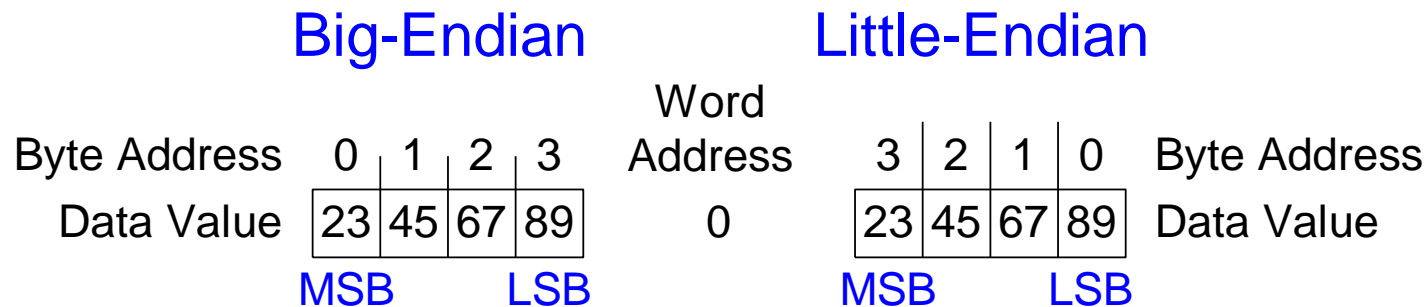
Big-Endian & Little-Endian Example

- Suppose `$t0` initially contains `0x23456789`
- After following code runs on big-endian system, what value is `$s0`?

```
sw $t0, 0($0)
```

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

- In a little-endian system, what is `$s0`?
- Big-endian: `0x00000045`
- Little-endian: `0x00000067`



Operands: Constants/Immediates

- `lw` and `sw` use constants or *immediates* e.g. `sw $t1, 48($0)`
- Value is *immediately* available from instruction
- Constant is a 16-bit 2's complement number
- `addi`: add immediate
- Is subtract immediate (`subi`) necessary?

C Code

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

MIPS assembly code

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

Design Principle 4

Good design demands good compromises

- Number of instruction formats should be minimum
 - to adhere to Design Principles 1 and 3 (simplicity favors regularity, and smaller is faster).
- Multiple instruction formats allow flexibility
 - Constants are very frequent in code (so “make the common case fast”, Design Principle #2)
 - add, sub: use 3 register operands
 - lw, sw, addi: use 2 register operands and a constant
- The compromise is to have 3 instruction formats

Machine Language

- Binary representation of instructions
- Computers only understand 1's and 0's
- 32-bit instructions
 - Simplicity favors regularity: 32-bit data & instructions, all instructions are 32-bits
- 3 instruction formats:
 - **R-Type**: register operands
 - **I-Type**: immediate operand
 - **J-Type**: for jumping (discuss later)

R-Type

- *Register-type*
- Uses 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 code*. When `op=0`, `funct` tells the processor which R-type 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

R-Type Examples

Assembly Code

```
add $s0, $s1, $s2
```

```
sub $t0, $t3, $t5
```

Field Values

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	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

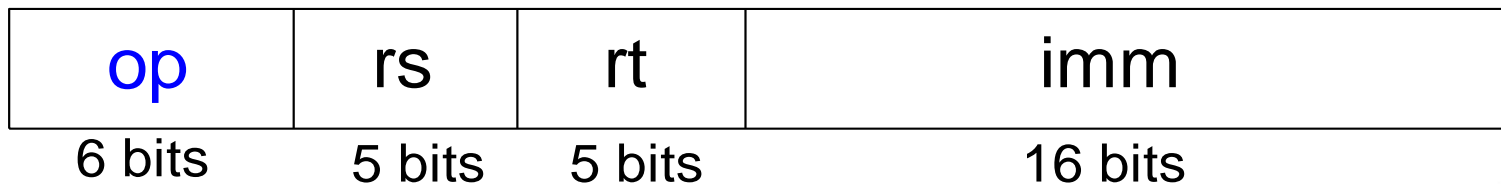
Note the order of registers in the assembly code:

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


I-Type

- *Immediate-type*
- 3 operands:
 - rs, rt: register operands
 - imm: 16-bit 2's complement immediate
- Other fields:
 - op: the opcode ($\neq 0$)
 - Simplicity favors regularity: all instructions have opcode
 - Operation is completely determined by opcode

I-Type



I-Type Examples

Assembly Code

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

Field Values

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

Machine Code

Note the differing order of registers in 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	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)

6 bits 5 bits 5 bits 16 bits



Machine Language: J-Type

- *Jump-type*
- 26-bit address operand (addr)
- Used for jump instructions (j : op=2, jal : op=3)

J-Type



Review: Instruction Formats

R-Type

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

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

J-Type

op	addr
6 bits	26 bits



Power of the Stored Program

- 32-bit instructions & data stored in memory
- Sequence of instructions: only difference between two applications
- To run a new program:
 - No rewiring required
 - Simply store new program in memory
- Program Execution:
 - Processor *fetches* (reads) instructions from memory in sequence (Von Neumann architecture)
 - Processor performs the specified operation

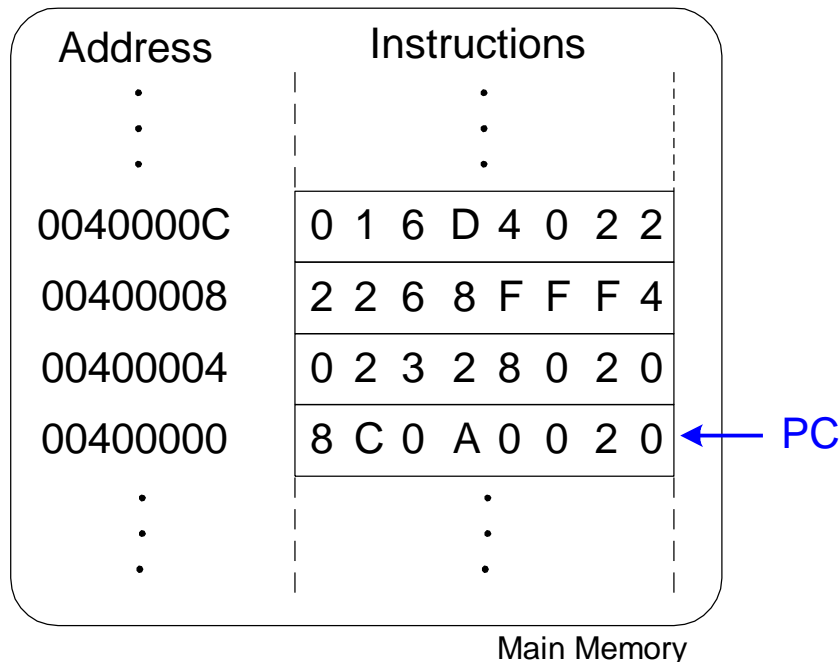
The Stored Program

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



Program Counter (PC): keeps track of current instruction

Interpreting Machine Code

- Start with opcode: tells how to parse rest
- If opcode all 0's
 - R-type instruction
 - Function bits tell operation
- Otherwise
 - opcode tells operation

Machine Code

(0x2237FFF1)

op	rs	rt	imm
001000	10001	10111	1111 1111 1111 0001
2	2	3	7 F F F 1

Field Values

8

17

23

-15

addi \$s7, \$s1, -15

(0x02F34022)

op	rs	rt	rd	shamt	funct
000000	10111	10011	01000	00000	100010
0	2	F	3	4	0 2 2

0

23

19

8

0

34

sub \$t0, \$s7, \$s3

Programming

- High-level languages:
 - e.g., C, Java, Python
 - Written at higher level of abstraction
- Common high-level software constructs:
 - if/else statements
 - for loops
 - while loops
 - arrays
 - function calls

Ada Lovelace, 1815-1852

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



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:
 - $A \text{ NOR } \$0 = \text{NOT } A$
- `andi`, `ori`, `xori`
 - 16-bit immediate is zero-extended (*not* sign-extended)
 - `nor`i not needed

Logical Instructions Example 1

Source Registers

\$s1	1111	1111	1111	1111	0000	0000	0000	0000
\$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code

```
and $s3, $s1, $s2  
or  $s4, $s1, $s2  
xor $s5, $s1, $s2  
nor $s6, $s1, $s2
```

Result

\$s3								
\$s4								
\$s5								
\$s6								

Logical Instructions Example 1

Source Registers

\$s1	1111	1111	1111	1111	0000	0000	0000	0000
\$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Result

\$s3	0100	0110	1010	0001	0000	0000	0000	0000
\$s4	1111	1111	1111	1111	1111	0000	1011	0111
\$s5	1011	1001	0101	1110	1111	0000	1011	0111
\$s6	0000	0000	0000	0000	0000	1111	0100	1000

Logical Instructions Example 2

Assembly Code

```
andi $s2, $s1, 0xFA34 $s2
ori  $s3, $s1, 0xFA34 $s3
xori $s4, $s1, 0xFA34 $s4
```

Source Values

\$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100
	← zero-extended →							

Result

\$s2								
\$s3								
\$s4								

Logical Instructions Example 2

Assembly Code

```
andi $s2, $s1, 0xFA34
ori  $s3, $s1, 0xFA34
xori $s4, $s1, 0xFA34
```

\$s1

0000	0000	0000	0000	0000	0000	1111	1111
------	------	------	------	------	------	------	------

imm

0000	0000	0000	0000	1111	1010	0011	0100
------	------	------	------	------	------	------	------

← zero-extended →

Source Values

Result

\$s2

0000	0000	0000	0000	0000	0000	0011	0100
------	------	------	------	------	------	------	------

\$s3

0000	0000	0000	0000	1111	1010	1111	1111
------	------	------	------	------	------	------	------

\$s4

0000	0000	0000	0000	1111	1010	1100	1011
------	------	------	------	------	------	------	------

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`

Shift Instructions

Assembly Code

Field Values

	op	rs	rt	rd	shamt	funct
sll \$t0, \$s1, 2	0	0	17	8	2	0
srl \$s2, \$s1, 2	0	0	17	18	2	2
sra \$s3, \$s1, 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	



Generating Constants

- Make 16-bit constants using `addi`:

C Code

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

MIPS assembly code

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

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

C Code

```
int a = 0xFEDC8765;
```

MIPS assembly code

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

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, remainder
 - `div $s0, $s1`
 - Quotient in `lo`
 - Remainder in `hi`
- Moves from `lo/hi` special registers
 - `mflo $s2`
 - `mfhi $s3`

Branching

- Execute instructions out of sequence
- Types of branches:
 - **Conditional**
 - branch if equal (beq)
 - branch if not equal (bne)
 - **Unconditional**
 - jump (j)
 - jump register (jr)
 - jump and link (jal)

Review: The Stored Program

Assembly Code

```
lw    $t2, 32($0)
add   $s0, $s1, $s2
addi  $t0, $s3, -12
sub   $t0, $t3, $t5
```

Machine Code

```
0x8C0A0020
0x02328020
0x2268FFF4
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

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 location. They can't be reserved words and must be followed by colon (:)

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

Unconditional Branching (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
```


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)

jr is an **R-type** instruction.

High-Level Code Constructs

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

If Statement

C Code

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

f = f - i;
```

MIPS assembly code

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

If Statement

C Code

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

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
        bne $s3, $s4, L1
        add $s0, $s1, $s2

L1:     sub $s0, $s0, $s3
```

Assembly tests opposite case ($i \neq j$) of high-level code ($i == j$)

If/Else Statement

C Code

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

MIPS assembly code

If/Else Statement

C Code

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

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
    bne $s3, $s4, L1
    add $s0, $s1, $s2
    j   done
L1:   sub $s0, $s0, $s3
done:
```

While Loops

C 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

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

While Loops

C 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

        addi $s0, $0, 1
        add  $s1, $0, $0
        addi $t0, $0, 128
while:   beq  $s0, $t0, done
        sll  $s0, $s0, 1
        addi $s1, $s1, 1
        j    while
done:
```

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

For Loops

for (initialization; condition; loop operation)
statement

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

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 Loops

C 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

For Loops

C 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
        addi $s1, $0, 0
        add  $s0, $0, $0
        addi $t0, $0, 10
for:     beq  $s0, $t0, done
        add  $s1, $s1, $s0
        addi $s0, $s0, 1
        j    for
done:
```

Less Than Comparison

C 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

Less Than Comparison

C 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
        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:    slt   $t1, $s0, $t0
        beq   $t1, $0, done
        add   $s1, $s1, $s0
        sll   $s0, $s0, 1
        j     loop
done:
```

**slt: \$t1 = 1 if i < 101,
\$t1 = 0 otherwise**

Arrays

- Access large amounts of similar data
- **Index**: access each element
- **Size**: number of elements

Arrays

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

0x12340010	array[4]
0x1234800C	array[3]
0x12348008	array[2]
0x12348004	array[1]
0x12348000	array[0]

Accessing Arrays

// C Code

```
int array[5];  
array[0] = array[0] * 2;  
array[1] = array[1] * 2;
```

Accessing Arrays

// C Code

```
int array[5];  
array[0] = array[0] * 2;  
array[1] = array[1] * 2;
```

MIPS assembly code

array base address = \$s0

```
lui    $s0, 0x1234          # 0x1234 in upper half of $s0  
ori    $s0, $s0, 0x8000     # 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
```

Arrays using For Loops

// C 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
```

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:



ASCII Code

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

ASCII 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		

Moving Characters

- ASCII chars are 8-bits (used in C)
- So MIPS offers lb, lbu and sb
 - lbu \$s1, 3(\$0) # \$s1 \leftarrow Mem[3] (zero-extended byte)
 - lb \$s2, 1(\$0) # \$s2 \leftarrow Mem[1] (sign-extended byte)
 - sb \$s3, 5(\$0) # Mem[5] \leftarrow LSByte of \$s3
- Standard UNICODE chars are 16-bit (used in Java)
- So MIPS offers lh, lhu and sh
- See www.unicode.org for much more!

ASCII Strings

- ASCII strings are arrays of chars, stored in consecutive bytes of memory
- ASCII strings have variable (i.e. unknown) length
- Must be terminated w/ NULL character (NULL's code = 0x00)
- E.g. "Hello!" is 0x48 65 6C 6C 6F 21 00 (7 bytes, not 6!)

Manipulating ASCII strings

// C Code

```
char chararray[10];  
int i;
```

```
for (i=0; chararray[i] != 0; i = i + 1)  
    chararray[i] = chararray[i] - 32;
```

MIPS assembly code

uses direct addressing (i.e. pointer to the string)

\$s0 = base address of chararray

\$s1 is not used: no need for index variable i !

Manipulating ASCII strings

```
# MIPS assembly code
#     uses direct addressing (i.e. pointer to the string)
# $s0 = base address of chararray

    lbu    $t2, 0($s0)           # get 1st char: $t2= Mem[&chararray]
                                   #                               (= chararray[0])
loop:
    beq    $t2, $0, done         # if chararray[i]==NULL, exit loop
    addi   $t2, $t2, -32         # convert to upper case: t2=t2-32
    sb     $t2, 0($s0)          # store new value in array:
                                   # Mem[chararray[i] = $t2
    addi   $s0, $s0, 1          # address = address + 1
    lbu    $t2, 0($s0)          # get next char: $t2=Mem[&chararray]
                                   #                               (= chararray[i])
    j      loop                 # repeat
done:
```

Function Calls

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

C Code

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

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

Function Conventions

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

MIPS Function Conventions

- **Call Function:** jump and link (`j a1`)
- **Return from function:** jump register (`j r`)
- **Arguments:** `$a0` – `$a3`
- **Return values:** `$v0`, `$v1`

Function Calls

C 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

Function Calls

C 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: $PC \leftarrow 0x00401020$
stores return address: $\$ra \leftarrow PC + 4 = 0x00400204$

jr \$ra: jumps to address in \$ra: $PC \leftarrow \$ra = 0x00400204$

Input Arguments & Return Value

MIPS conventions:

- Argument values: $\$a0 - \$a3$
- Return values: $\$v0 - \$v1$

Input Arguments & Return Value

C 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
}
```

Input Arguments & Return Value

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 Function
```

```
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
```

Input Arguments & Return Value

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)
```

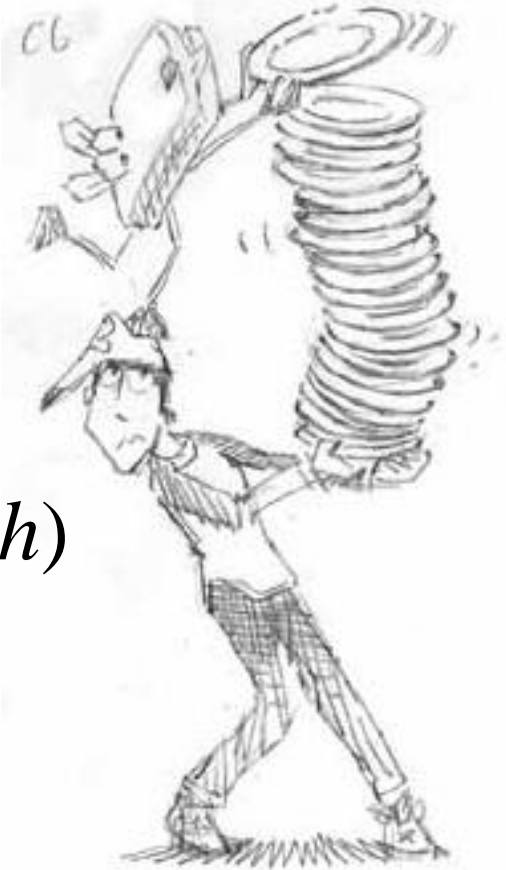
```
add $v0, $s0, $0     # put return value in $v0
```

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

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

The Stack

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



The Stack

- MIPS stack grows down (from higher to lower memory addresses)
- Stack pointer: `$sp` points to “top” of the stack, in MIPS the last full location (vs. first empty)

Address	Data	
7FFFFFFC	12345678	← <code>\$sp</code>
7FFFFFF8		
7FFFFFF4		
7FFFFFF0		
⋮	⋮	

Address	Data	
7FFFFFFC	12345678	
7FFFFFF8	AABBCCDD	
7FFFFFF4	11223344	← <code>\$sp</code>
7FFFFFF0		
⋮	⋮	



How Functions use the Stack

- Called functions must have no unintended side effects (such as overwriting other's data !)
- 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`

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

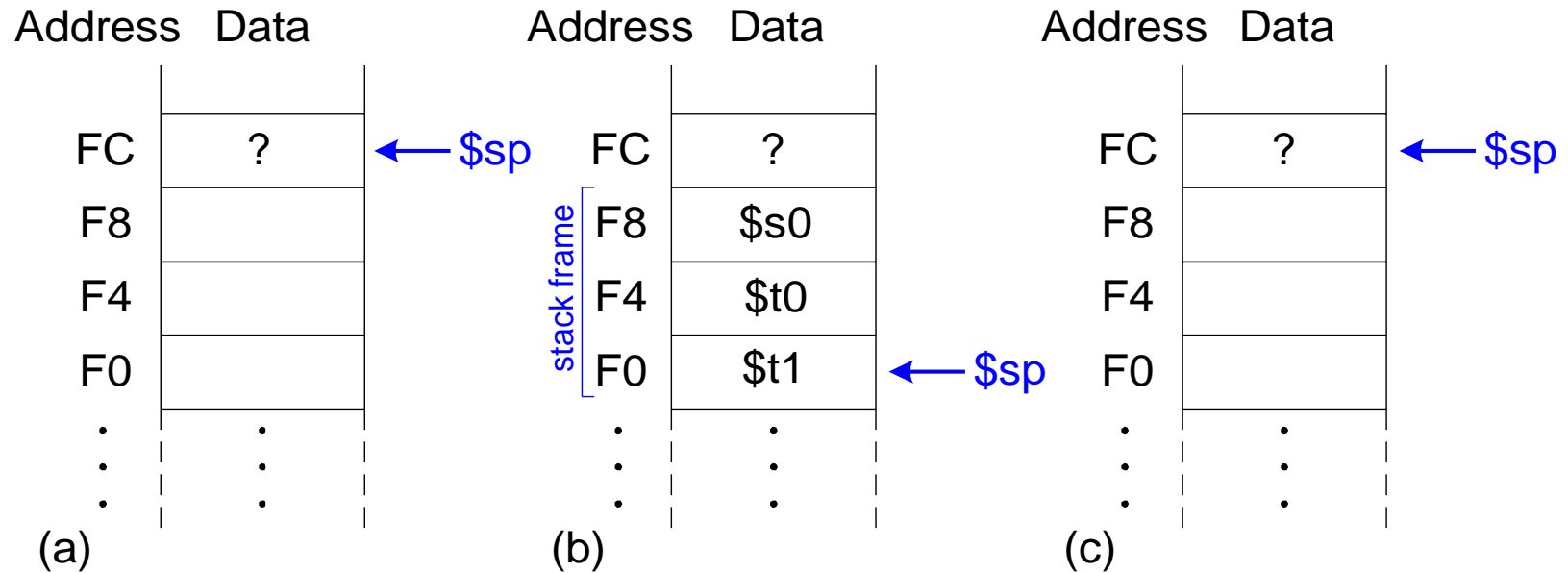
    add   $t0, $a0, $a1    # $t0 = f + g
    add   $t1, $a2, $a3    # $t1 = h + i
    sub   $s0, $t0, $t1    # result = (f + g) - (h + i)
    add   $v0, $s0, $0      # put return value in $v0

    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
```



The stack during `diffofsums` Call



Register saving

To share the burden of saving on the stack, the registers are divided into two groups:

Preserved <i>Callee-Saved</i>	Non-preserved <i>Caller-Saved</i>
\$s0-\$s7	\$t0-\$t9
\$ra	\$a0-\$a3
\$sp, \$fp, \$gp	\$v0-\$v1
stack above \$sp	stack below \$sp

Register saving

Callee:

- must save \$s-regs and \$ra,\$sp,\$fp,\$gp and stack above \$sp
- Save means either don't use, or push to stack and restore after use
- Free to change \$t, \$a & \$v-regs, stack below \$sp

Caller:

- needs to save things that may be changed by callee (\$t, \$a & \$v-regs, stack below \$sp) only if caller will need them *after the call and return*

Nested Function Calls

proc1:

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

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
```



Recursive Function Call

High-level code

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

Recursive Function Call

MIPS assembly code

```

0x90 factorial: addi $sp, $sp, -8    # make room for 2 items
0x94             sw  $a0, 4($sp)    # push $a0
0x98             sw  $ra, 0($sp)    # push $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:
                                # factorial(n-1)
0xBC             lw   $a0, 4($sp)    # pop $a0 (= saved n)
0xC0             mul  $v0, $a0, $v0  # n * factorial(n-1)
0xC4             lw   $ra, 0($sp)    # pop $ra
0xC8             addi $sp, $sp, 8    # restore $sp
0xCC             jr   $ra           # return

```



Stack During Recursive Call

Address Data

FC		← \$sp
F8		
F4		
F0		
EC		
E8		
E4		
E0		
DC		

Address Data

FC		← \$sp
F8	\$a0 (0x3)	
F4	\$ra	← \$sp
F0	\$a0 (0x2)	
EC	\$ra (0xBC)	← \$sp
E8	\$a0 (0x1)	
E4	\$ra (0xBC)	← \$sp
E0		
DC		

Address Data

FC		← \$sp	\$v0 = 6
F8	\$a0 (0x3)		
F4	\$ra	← \$sp	\$a0 = 3 \$v0 = 3 x 2
F0	\$a0 (0x2)		
EC	\$ra (0xBC)	← \$sp	\$a0 = 2 \$v0 = 2 x 1
E8	\$a0 (0x1)		
E4	\$ra (0xBC)	← \$sp	\$a0 = 1 \$v0 = 1 x 1
E0			
DC			

Function Call Summary

- **Caller**
 - Put arguments in $\$a0-\$a3$
 - Save any needed registers ($\$ra$, maybe $\$t0-\$t9$)
 - `jal callee`
 - Restore registers
 - Look for result in $\$v0$
- **Callee**
 - Save registers that would be changed ($\$s0-\$s7$, etc)
 - Perform function (using arguments in $\$a0-\$a3$)
 - Put result in $\$v0$
 - Restore registers that were saved
 - `jr $ra`

Addressing Modes

How do we address the operands?

- Register Only
- Immediate
- Base Addressing

How do we address the next instruction?

- (Default: $PC \leq PC + 4$)
- PC-Relative
- Pseudo-Direct

Addressing Modes

Register Only

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

Immediate

- 16-bit immediate (extended to 32-bits) used as an operand
 - **Example:** `addi $s4, $t5, -73` # sign-extend
 - **Example:** `ori $t3, $t7, 0xFF` # zero-extend

Addressing Modes

Base Addressing

- Address of operand is:
base address + sign-extended immediate
 - **Example:** `lw $s4, 72($0)`
 - $\text{address} = \$0 + 72$
 - **Example:** `sw $t2, -25($t1)`
 - $\text{address} = \$t1 - 25$

Addressing Modes

PC-Relative Addressing (used in beq and bne)

```

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

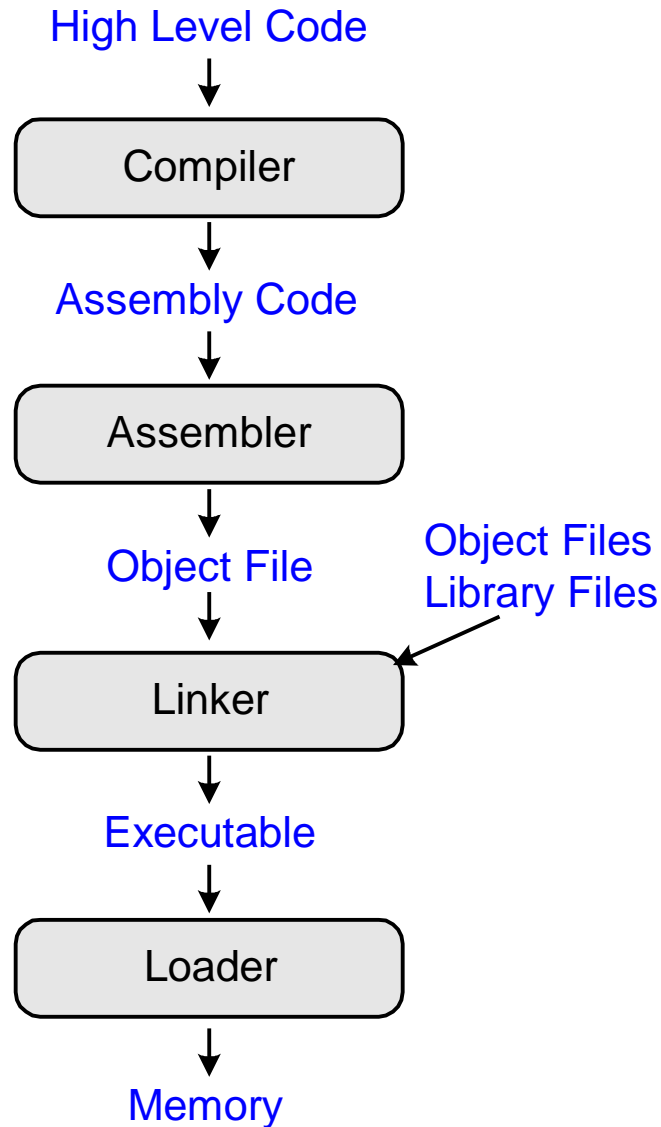
Field Values

	op	rs	rt	imm		
beq \$t0, \$0, else	4	8	0	3		
(beq \$t0, \$0, 3)	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Note: BTA (address of else:) = PC + 4 + (imm x 4)

Chapter 6 <113>

How to Compile & Run a Program



Grace Hopper, 1906-1992

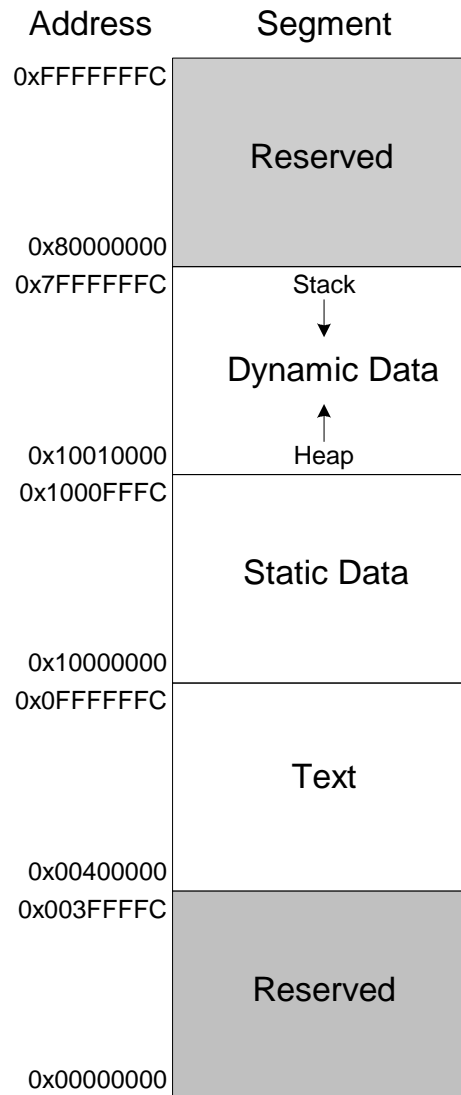
- Graduated from Yale University with a Ph.D. in mathematics
- Developed first compiler
- Helped develop the COBOL programming language
- Highly awarded naval officer
- Received World War II Victory Medal and National Defense Service Medal, among others



What is Stored in Memory?

- Instructions (in the *.text* segment)
- Data
 - Global/static: allocated before program begins (in the *.data* segment)
 - Dynamic: allocated within program (*heap* and *stack*)
- How big is memory?
 - At most $2^{32} = 4$ gigabytes (4 GB) in MIPS
 - From address 0x00000000 to 0xFFFFFFFF

MIPS Memory Map



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);
```

```
}
```

Example Program: MIPS Assembly

```
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    # push 1 item
      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    # pop 1 item
      jr  $ra           # return to OS
sum:  add  $v0, $a0, $a1  # $v0 = a + b
      jr  $ra           # return
```

Example Program: Symbol Table

Symbol	Address

Example Program: Symbol Table

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

Example Program: Executable

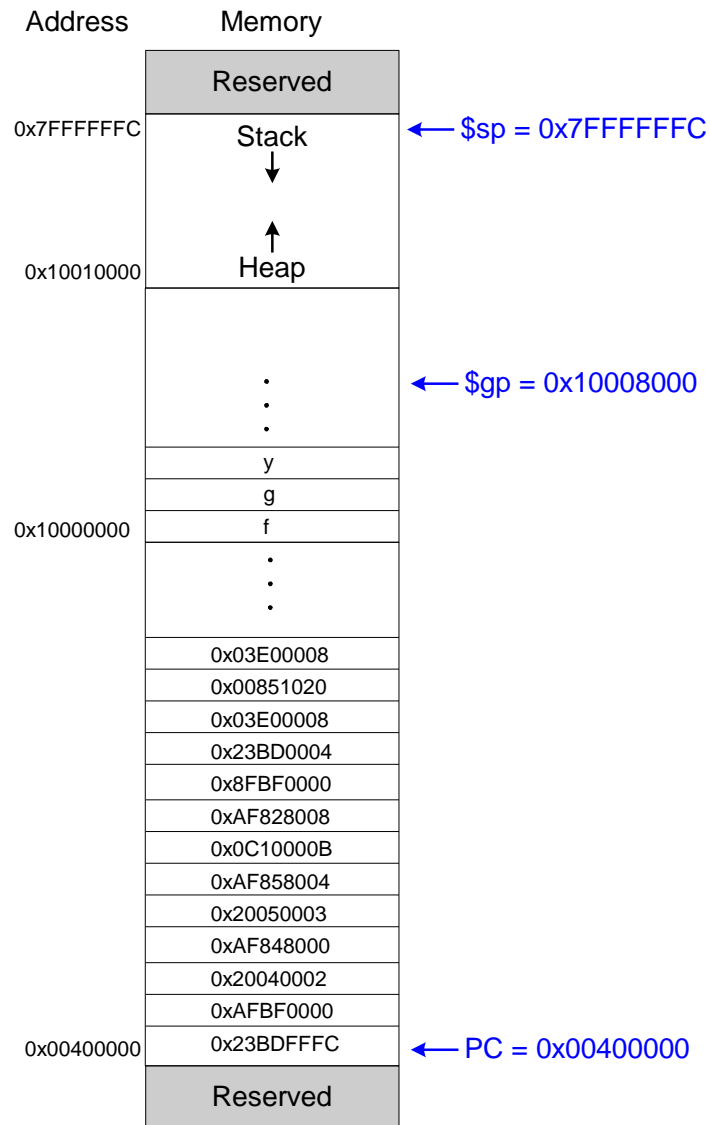
Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)
Text segment	Address	Instruction
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	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
sw  $v0, 0x8008 ($gp)
lw  $ra, 0 ($sp)
addi $sp, $sp, -4
jr   $ra
add  $v0, $a0, $a1
jr   $ra

```

Example Program: In Memory



Odds & Ends

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

Pseudo-instructions

Pseudo-instruction	MIPS Instructions
<code>li \$s0, 0x1234AA77</code>	<code>lui \$s0, 0x1234</code> <code>ori \$s0, 0xAA77</code>
<code>clear \$t0</code>	<code>add \$t0, \$0, \$0</code>
<code>move \$s1, \$s2</code>	<code>add \$s2, \$s1, \$0</code>
<code>nop</code>	<code>sll \$0, \$0, 0</code>

The assembler implements a virtual machine, by adding pseudo-instruction capability to the actual MIPS architecture. There are many possible pseudo-instructions (e.g. `beq $s1, 22, Branch_target`)

BUT: *the pseudo-instructions vary between assemblers, so using them could mean your code is no longer “portable”, it is assembler specific!*

Exceptions

- Unscheduled function call to *exception handler* (part of the OS)
- 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 exception handler (at instruction address 0x80000180 in MIPS)
 - Deals with the problem causing the exception
 - Returns to program

Exception Registers

- Not part of processor's register file
 - **Cause**: Records cause of exception
 - **EPC** (Exception PC): Records PC where exception occurred
- EPC and Cause: part of Coprocessor 0
- Move from Coprocessor 0
 - `mfc0 $t0, EPC`
 - Moves contents of EPC into `$t0`

Exception Causes

Exception	Cause
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Divide by 0	0x00000024
Undefined Instruction	0x00000028
Arithmetic Overflow	0x00000030
... (lots more—see Green Card)	...

Exception Flow

- Processor saves: $\text{Cause} \leftarrow$ code for exception type
and $\text{EPC} \leftarrow \text{PC}$ value that caused the exception
- Processor jumps to exception handler (0x80000180)
- Exception handler:
 - Saves registers on stack
 - Reads Cause register
 - Handles exception
 - Restores registers
 - Returns to program

```
mfc0 $t0, Cause
```

```
mfc0 $k0, EPC
```

```
jr $k0
```

Signed & Unsigned Instructions

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

Addition & Subtraction

- **Signed:** `add`, `addi`, `sub`
 - Same operation as unsigned versions
 - But processor causes exception on overflow
- **Unsigned:** `addu`, `addiu`, `subu`
 - Doesn't cause exception on overflow

Note: C language ignores exceptions; FORTRAN and others require the program be notified.

Beware: `addiu` sign-extends the immediate, before treating it as an unsigned number !

Multiplication & Division

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

$0xFFFFFFFF = -1$ (signed) or $= 2^{32} - 1$ (unsigned)

So

$0xFFFFFFFF \times 0xFFFFFFFF = ?$
= $0x0000000000000001$ w/ `mult` (signed)
= $0xFFFFFFFFFE00000001$ w/ `multu` (unsigned)

Set Less Than

- **Signed:** `slt`, `slti`
treats operands as signed
- **Unsigned:** `sltu`, `sltiu`
treats operands as unsigned

Beware: `sltiu` sign-extends the immediate, before comparing it to the register, treating both as unsigned

Loads

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

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.
 - Double-precision floating point registers: \$f0, \$f2, \$f4, etc.

Floating-Point Registers

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

F-Type Instruction Format

- opcode = 17 (010001_2)
- Single-precision:
 - cop = 16 (010000_2)
 - add.s, sub.s, div.s, neg.s, abs.s, etc.
- Double-precision:
 - cop = 17 (010001_2)
 - add.d, sub.d, div.d, neg.d, abs.d, etc.
- 3 register operands:
 - fs, ft: source operands
 - fd: destination operands

F-Type



Floating-Point Branches

- Set/clear condition flag: `fpcond`
 - Equality: `c.seq.s`, `c.seq.d`
 - Less than: `c.lt.s`, `c.lt.d`
 - L.T. 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 $ft1, 40($s1)`
 - `swc1: swc1 $fs2, 24($sp)`