

COMPUTER ORGANIZATION AND ARCHITECTURE (IT 2202)

Lecture 8

Instruction Set Architecture

- MIPS architecture:
 - Developed by John Hennessy and his colleagues at Stanford University in the 1980's.
 - MIPS are popular RISC processor.
 - Used in many commercial systems, including NEC, Silicon Graphics, Nintendo, and Cisco.
- Real architecture but easy to understand

MIPS Arithmetic

- All Instructions have 3 operands
- Operand order is fixed
 - Destination appears first

Example

C code $A=B+C$

MIPS Code: `add $s0, $s1,$s2`

Instructions

- MIPS includes only simple, commonly used instructions. Same number of operands (two sources and one destination)
- Hardware to decode and execute the instruction is 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.

Arithmetic 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

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

Instructions: More Complex Code

- More complex code is handled by multiple MIPS instructions.

High-level code

`a = b + c - d;`

MIPS assembly code

`add t, b, c # t = b + c`

`sub a, t, d # a = t - d`

Operands

- **A computer needs a physical location from which it retrieves binary operands**
- **A computer retrieves operands from:**
 - **Registers**
 - **Memory**
 - **Constants (also called *immediates*)**
- **Scalars mapped to registers**
- **Structures, arrays etc in memory**

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 the 32-bit version.)

MIPS Register Set

Name	Register Number	Usage
\$0	0	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

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.
 - *saved registers*, \$s0-\$s7, are used to hold variables
 - *temporary registers*, \$t0 - \$t9, are used to hold intermediate values during a larger computation.

Instructions with registers

High-level code

$a = b + c$

MIPS assembly code

\$s0 = a, \$s1 = b, \$s2 = c
add \$s0, \$s1, \$s2

Expressions need to be broken

C code

$A = B + C + D;$

$E = F - A;$

MIPS code

add \$t0, \$s1, \$s2

add \$s0, \$t0, \$s3

sub \$s4, \$s5, \$s0

Instructions with registers

Translate the following high-level code into assembly language. Assume variables a–c are held in registers \$s0–\$s2 and f–j are in \$s3–\$s7.

$a = b - c;$

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

`# MIPS assembly code`

`# $s0 = a, $s1 = b, $s2 = c, $s3 = f, $s4 = g, $s5 = h,`

`# $s6 = i, $s7 = j`

`sub $s0, $s1, $s2 # a = b - c`

`add $t0, $s4, $s5 # $t0 = g + h`

`add $t1, $s6, $s7 # $t1 = i + j`

`sub $s3, $t0, $t1 # f = (g + h) - (i + j)`

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, so it can hold a large number 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

Word-Addressable Memory

- Each 32-bit data word has a unique address

W o r d A d d r e s s	D a t a	
⋮	⋮	⋮
0 0 0 0 0 0 0 3	4 0 F 3 0 7 8 8	W o r d 3
0 0 0 0 0 0 0 2	0 1 E E 2 8 4 2	W o r d 2
0 0 0 0 0 0 0 1	F 2 F 1 A C 0 7	W o r d 1
0 0 0 0 0 0 0 0	A B C D E F 7 8	W o r d 0

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
- Hexadecimal constants are written with the prefix 0x

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 writes are called *stores*

Mnemonic: *store word* (sw)

Example: Write (store) the value held in \$t4 into memory address 3

Offset can be written in decimal (default) or hexadecimal

Memory address calculation:

- add the base address (\$0) to the offset (0x3)
- address: (\$0 + 0x3) = 3

Any register may be used to store the base address

Assembly code

sw \$t4, 0x3(\$0) # write the value
of \$t4 to memory word 3

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 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 →

Reading Byte-Addressable Memory

Address of a memory word must now be multiplied by 4.

For example,

- address of memory word 2 is $2 \times 4 = 8$
- address of memory word 10 is $10 \times 4 = 40$ (0x28)

Load a word of data at memory address 4 into \$s3.

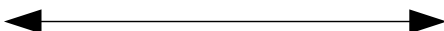
\$s3 holds the value 0xF2F1AC07 after the instruction completes.

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

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
00000008	0	1	E	E	2	8	4	2
00000004	F	2	F	1	A	C	0	7
00000000	A	B	C	D	E	F	7	8

width = 4 bytes

Load / Store example

C code: $A[8] = h + A[8];$

MIPS code: lw \$t0, 32(\$s3)
 add \$t0, \$s2, \$t0
 sw \$t0, 32(\$s3)

Words and Bytes

- 2^{32} bytes : byte addresses from 0 to $2^{32}-1$
- 2^{30} words : byte addresses 0, 4, 8, ... $2^{32}-4$

Big-Endian and Little-Endian Memory

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

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 Addressing

- With Big Endian addressing, the byte binary address

$x \dots x00$

is in the most significant position (big end) of a 32 bit word (IBM, Motorola, Sun, HP).

M S B		L S B	
0	1	2	3
4	5	6	7

Little Endian Addressing

- With Little Endian addressing, the byte binary address
x . . . x00
is in the least significant position (little end) of a 32 bit word (DEC, Intel).

M S B			L S B		
3	2	1	0		
7	6	5	4		

- Programmers/protocols should be careful when transferring binary data between Big Endian and Little Endian machines

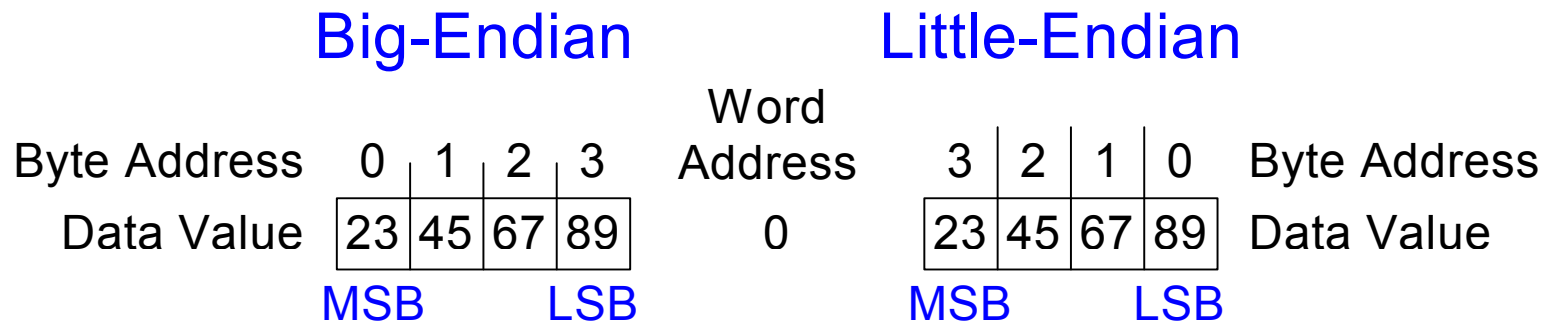
Big- and Little-Endian Example

- 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



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.
- Add immediate (addi) instruction adds an immediate to a variable (held in a register).
- An immediate is a 16-bit two's complement number.

High-level code

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

MIPS assembly code

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

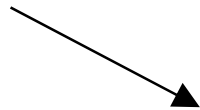
Constants in MIPS instructions

- `addi $29, $29, 4` 'i' is for 'immediate'
- `slti $8, $18, 10`
 `andi $29, $29, 6`
 `ori $29, $29, 4`

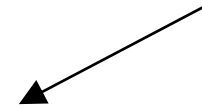
How about larger constants?

To load a 32 bit constant into a register, we use two instructions

one instruction fills this



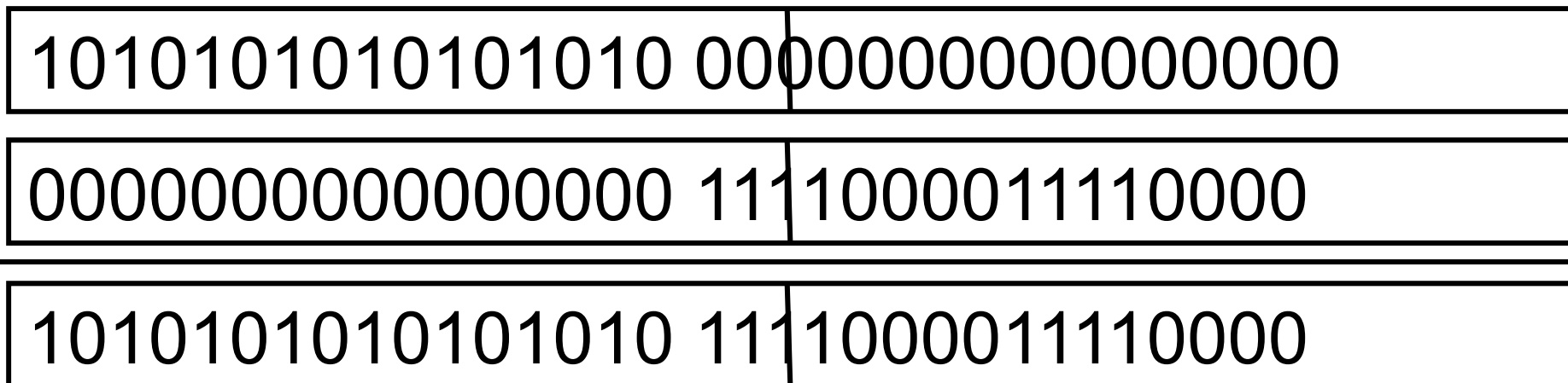
one instruction fills this



1010101010101010 111	1000011110000
----------------------	---------------

Loading larger constants

- "load upper immediate" instruction
 lui \$t0, 1010101010101010
- then get the lower order bits right, i.e.,
 ori \$t0, \$t0, 1111000011110000



Instruction Formats

R - T y p e

o p	r s	r t	r d	s h a m t	f u n c t
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I - T y p e

o p	r s	r t	i m m
6 bits	5 bits	5 bits	16 bits

J - T y p e

o p	a d d r
6 bits	26 bits

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 be performed
 - sham t: the *shift amount* for shift instructions, otherwise it's 0

R-Type



R-Type Examples

Assembly Code

ld \$s0, \$s1, \$s2

lb \$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

Order of registers in the assembly code:

add rd, rs, rt

Instructions are 32 bits long

registers have numbers 0 .. 31,

e.g., \$t0=8, \$t1=9, \$s0=16, \$s1=17 etc.

R-Type Examples

Assembly Code

Field Values

	o p	r s	r t	r d	s h a m t	f u n c t
add \$s0, \$s1, \$s2	0	17	18	16	0	32
add \$t0, \$t3, \$t5	0	11	13	8	0	34
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Order of registers in the assembly code:

add rd, rs, rt

Instructions are 32 bits long

registers have numbers 0 .. 31,

e.g., \$t0=8, \$t1=9, \$s0=16, \$s1=17 etc.

-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	

Order of registers in the assembly code:

add rd, rs, rt

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 - T y p e

o p	r s	r t	i m m
6 bits	5 bits	5 bits	16 bits

-Type Examples

Assembly Code

Field Values

	op	rs	rt	imm
addi \$s0, \$s1, 5	8	17	16	5
addi \$t0, \$s3, -12	8	19	8	-12
lw \$t2, 32(\$0)	35	0	10	32
sw \$s1, 4(\$t1)	43	9	17	4
	6 bits	5 bits	5 bits	16 bits

Differing order of registers in the assembly and machine codes:

addi rt, rs, imm

lw rt, imm(rs)

sw rt, imm(rs)

Type Examples

Assembly Code

Field Values

	op	rs	rt	imm
<code>addi \$s0, \$s1, 5</code>	8	17	16	5
<code>addi \$t0, \$s3, -12</code>	8	19	8	-12
<code>lw \$t2, 32(\$0)</code>	35	0	10	32
<code>sw \$s1, 4(\$t1)</code>	43	9	17	4
	6 bits	5 bits	5 bits	16 bits

Reversing order of registers in the assembly and machine codes:

`addi rt, rs, imm`

`addi rt, imm(rs)`

`addi rt, imm(rs)`

Machine Code

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x2230000)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A002)
101011	01001	10001	0000 0000 0000 0100	(0xAD31000)
6 bits	5 bits	5 bits	16 bits	

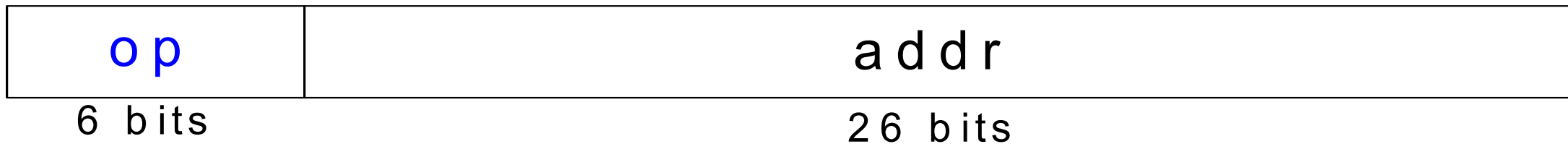
2's complement of (-12)=1111 1111 1111 0100

Machine Language: J-Type

Jump-type

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

J - T y p e



Power of Stored Program

- 32-bit instructions and data stored in memory
- To run a new program:
 - No rewiring required
 - Simply store new program in memory
- Processor hardware executes the program:
 - *fetches* (reads) the instructions from memory in sequence
 - performs the specified operation
- Program counter (PC) keeps track of the current instruction
- In MIPS, programs typically start at memory address 0x00400000

Stored Program

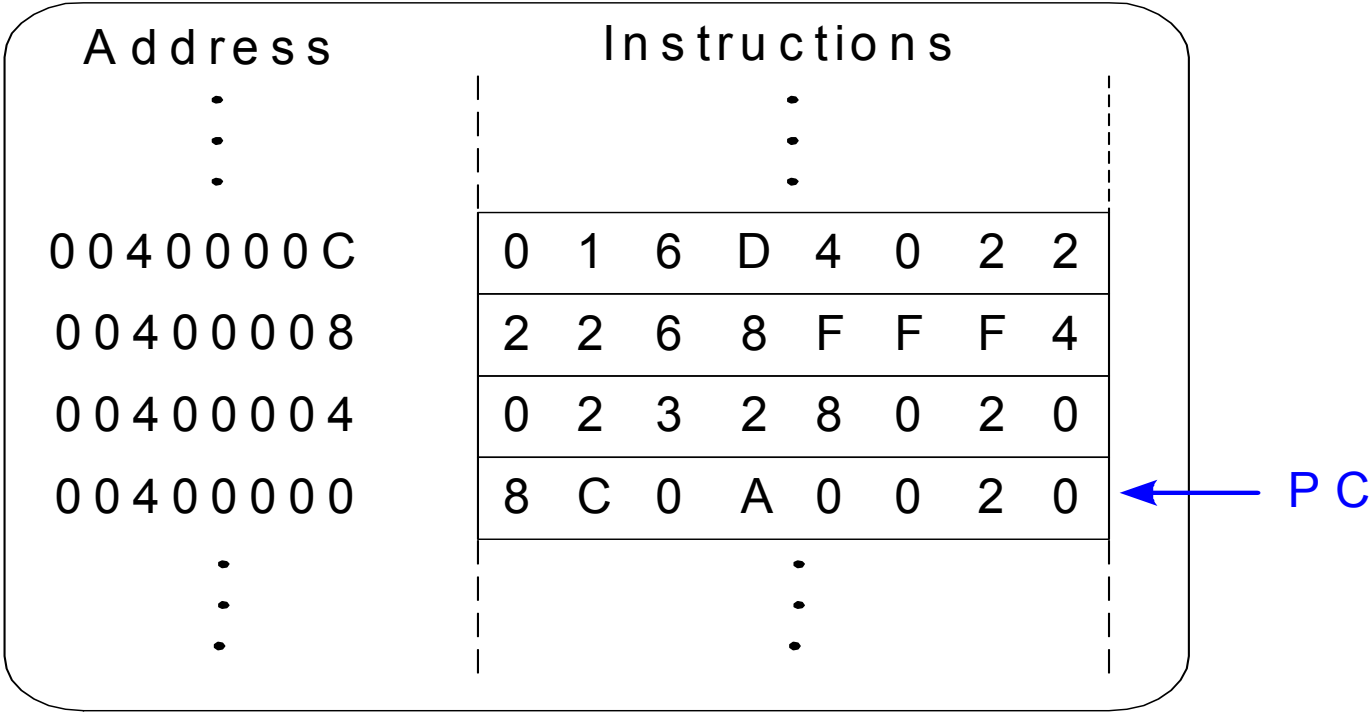
Assembly Code

```
l w      $ t 2 , 3 2 ( $ 0 )  
a d d    $ s 0 , $ s 1 , $ s 2  
a d d i  $ t 0 , $ s 3 , - 1 2  
s u b    $ t 0 , $ t 3 , $ t 5
```

Machine Code

```
0 x 8 C 0 A 0 0 2 0  
0 x 0 2 3 2 8 0 2 0  
0 x 2 2 6 8 F F F 4  
0 x 0 1 6 D 4 0 2 2
```

Stored Program



Main Memory

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

	Machine Code	Field Values	Assembly Code																														
(0x2237FFF1)	<table><tr><th>op</th><th>rs</th><th>rt</th><th>imm</th></tr><tr><td>001000</td><td>10001</td><td>10111</td><td>1111 1111 1111 0001</td></tr><tr><td>2</td><td>2</td><td>3</td><td>F F F 1</td></tr></table>	op	rs	rt	imm	001000	10001	10111	1111 1111 1111 0001	2	2	3	F F F 1	<table><tr><th>op</th><th>rs</th><th>rt</th><th>imm</th></tr><tr><td>8</td><td>17</td><td>23</td><td>-15</td></tr></table>	op	rs	rt	imm	8	17	23	-15	addi \$s7, \$s1, -15										
op	rs	rt	imm																														
001000	10001	10111	1111 1111 1111 0001																														
2	2	3	F F F 1																														
op	rs	rt	imm																														
8	17	23	-15																														
(0x02F34022)	<table><tr><th>op</th><th>rs</th><th>rt</th><th>rd</th><th>shamt</th><th>funct</th></tr><tr><td>000000</td><td>10111</td><td>10011</td><td>01000</td><td>00000</td><td>100010</td></tr><tr><td>0</td><td>2</td><td>F</td><td>3</td><td>4</td><td>0 2 2</td></tr></table>	op	rs	rt	rd	shamt	funct	000000	10111	10011	01000	00000	100010	0	2	F	3	4	0 2 2	<table><tr><th>op</th><th>rs</th><th>rt</th><th>rd</th><th>shamt</th><th>funct</th></tr><tr><td>0</td><td>23</td><td>19</td><td>8</td><td>0</td><td>34</td></tr></table>	op	rs	rt	rd	shamt	funct	0	23	19	8	0	34	sub \$t0, \$s7, \$s3 42
op	rs	rt	rd	shamt	funct																												
000000	10111	10011	01000	00000	100010																												
0	2	F	3	4	0 2 2																												
op	rs	rt	rd	shamt	funct																												
0	23	19	8	0	34																												

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)**
 - **nori not needed**

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

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

Result

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

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

```
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 Instruction Examples

Source Values

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

Assembly Code

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

Result

\$s2								
\$s3								
\$s4								

Logical Instruction Examples

Source Values

\$s1

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

imm

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

← zero-extended →

Assembly Code

Result

andi \$s2, \$s1, 0xFA34	\$s2	0000	0000	0000	0000	0000	0000	0011	0100
ori \$s3, \$s1, 0xFA34	\$s3	0000	0000	0000	0000	1111	1010	1111	1111
xori \$s4, \$s1, 0xFA34	\$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	

Shift Instructions

Source Values

\$s1	1111	0011	0000	0000	0000	0010	1010	1000
------	------	------	------	------	------	------	------	------

shamt

00100

Assembly Code

```
ll $t0, $s1, 4
```

\$t0

```
rl $s2, $s1, 4
```

\$s2

```
ra $s3, $s1, 4
```

\$s3

Result

0011	0000	0000	0000	0010	1010	1000	0000
0000	1111	0011	0000	0000	0000	0010	1010
1111	1111	0011	0000	0000	0000	0010	1010

Shift Instructions

Assembly Code

```
sllv $s3, $s1, $s2
srlv $s4, $s1, $s2
srav $s5, $s1, $s2
```

Field Values

op	rs	rt	rd	shamt	funct
0	18	17	19	0	4
0	18	17	20	0	6
0	18	17	21	0	7
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10010	10001	10011	00000	000100	(0x02519804)
000000	10010	10001	10100	00000	000110	(0x0251A006)
000000	10010	10001	10101	00000	000111	(0x0251A807)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Shift Instructions

Source Values

\$s1	1111	0011	0000	0100	0000	0010	1010	1000
\$s2	0000	0000	0000	0000	0000	0000	0000	1000

Assembly Code

```
sllv $s3, $s1, $s2  
srlv $s4, $s1, $s2  
srav $s5, $s1, $s2
```

Result

\$s3	0000	0100	0000	0010	1010	1000	0000	0000
\$s4	0000	0000	1111	0011	0000	0100	0000	0010
\$s5	1111	1111	1111	0011	0000	0100	0000	0010

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
```

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

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)

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    {0000 0000 0000 0001
addi $s1, $s1, 1         # not executed       0000 0000 0000 0100}
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 (:).

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      #  $\$s1 = 1 \ll 2 = 4$ 
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 / 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
```

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)**

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)**