

Chapter 2

Instructions: Language of the Computer

Part 1:

- **ALU instructions**
- **Data transfer instructions**

Instruction Set

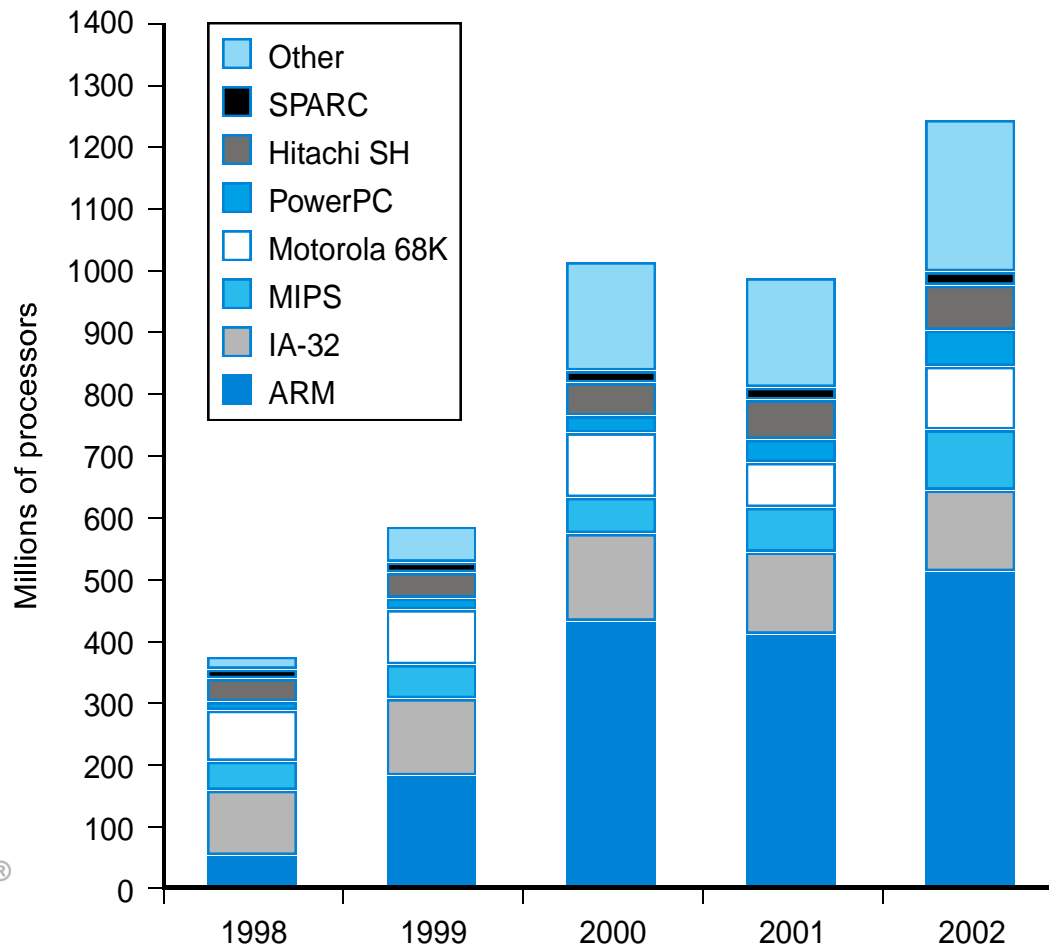
- The repertoire of instructions of a computer
- Different computers have different instruction sets
 - But with many aspects in common
- Early computers had very simple instruction sets
 - Simplified implementation
- Many modern computers also have simple instruction sets

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
 - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern ISAs
 - See MIPS Reference Data tear-out card, and Appendixes B and E

MIPS Instruction Set Architecture:

- similar to other architectures developed since the 1980's
- almost 100 million MIPS processors manufactured in 2002
- used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, ...



Chapter 2

- Illustrate MIPS instructions
 - ALU instructions (part 1)
 - Data transfer instructions
 - Branch instructions (part 2)
 - ✦ Associated addressing modes, operands, formats
- Also illustrate
 - Stored program concept
 - Supporting procedure calls (part 3)
 - Linking and running programs
 - ARM, IA-32 architectures (optional part 4)



ALU instructions

Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination
add a, b, c # a gets b + c
- All arithmetic operations have this form
- *Design Principle 1*
 - Simplicity enables higher performance at lower cost



Arithmetic Example

- C code:

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

- Compiled MIPS code:

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```


Register Operand Example

- C code:

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

- `f, ..., j` in `$s0, ..., $s4`

- Compiled MIPS code:

`add $t0, $s1, $s2`

`add $t1, $s3, $s4`

`sub $s0, $t0, $t1`

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32×32 -bit register file
 - Use for frequently accessed data
 - Numbered 0 to 31
 - 32-bit data called a “word”
- Assembler names
 - \$t0, \$t1, ..., \$t9 for temporary values
 - \$s0, \$s1, ..., \$s7 for saved variables
- *Design Principle 2: Smaller is faster*
 - c.f. main memory: millions of locations





Data Transfer instructions

Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" mean that the index points to a byte of memory

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
...	

Memory Organization


- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
...	

Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$
- Words are aligned
i.e., what are the least 2 significant bits of word address?

Memory Operands

- Main memory used for composite data
 - Arrays, structures, dynamic data
 - To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
 - Memory is byte addressed
 - Each address identifies an 8-bit byte
 - Words are aligned in memory
 - Address must be a multiple of 4
 - MIPS is Big Endian
 - Most-significant byte at least address of a word
-  [®] c.f. Little Endian: least-significant byte at least address

Memory Operand Example 1

- C code:

`g = h + A[8];`

- `g` in `$s1`, `h` in `$s2`, base address of `A` in `$s3`

- Compiled MIPS code:

- Index 8 requires offset of 32

- 4 bytes per word

```
lw    $t0, 32($s3)    # load word
add   $s1, $s2, $t0
```

offset

base register

Memory Operand Example 2

- C code:

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

- `h` in `$s2`, base address of `A` in `$s3`

- Compiled MIPS code:

- Index 8 requires offset of 32

```
lw    $t0, 32($s3)    # load word
add   $t0, $s2, $t0
sw    $t0, 48($s3)    # store word
```


So far we've learned:

MIPS

- Loading words but addressing bytes
- Arithmetic on registers only

Instruction

Meaning

add \$s1, \$s2, \$s3

$\$s1 = \$s2 + \$s3$

sub \$s1, \$s2, \$s3

$\$s1 = \$s2 - \$s3$

lw \$s1, 100(\$s2)

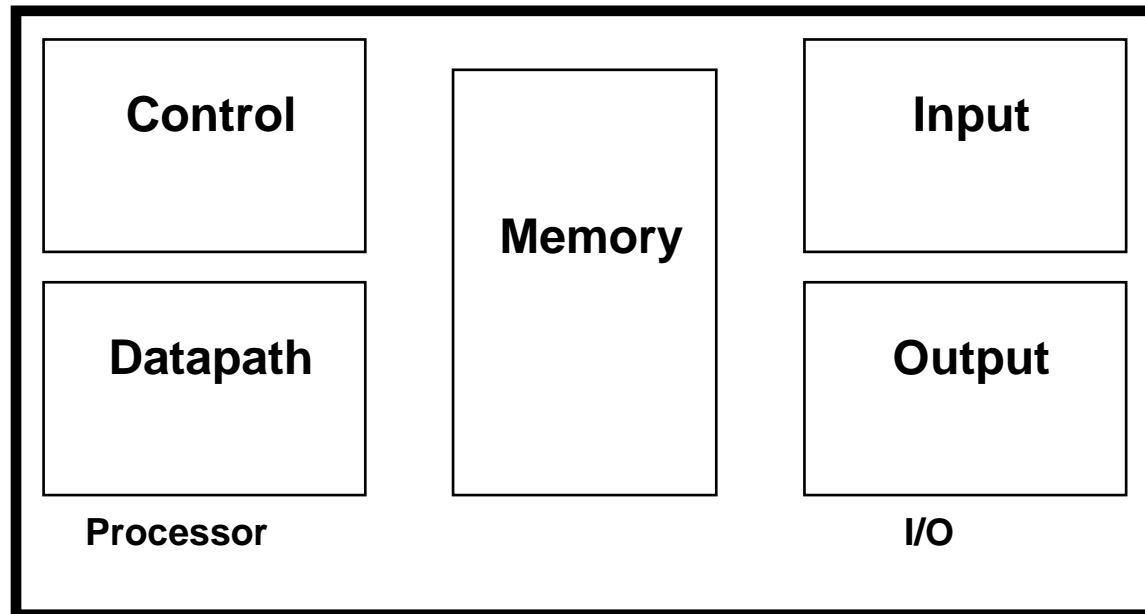
$\$s1 = \text{Memory}[\$s2 + 100]$

sw \$s1, 100(\$s2)

$\text{Memory}[\$s2 + 100] = \$s1$

Registers vs. Memory

- Arithmetic instructions operands must be registers,
— only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables



Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

Quiz

- Executing n instructions in load-store arch.

```
lw    $t0, 32($s3)    # load word
add   $t0, $s2, $t0
sw    $t0, 48($s3)    # store word
.
.
```

- Number of memory access?
 - Number of instruction access?
 - Number of data access?
 - Number of memory reads?
 - Number of memory writes?

Immediate Operands (미리 보기)

- Constant data specified in an instruction
`addi $s3, $s3, 4`
- No subtract immediate instruction
 - Just use a negative constant
`addi $s2, $s1, -1`
- *Design Principle 3: Make the common case fast*
 - Small constants are common
 - Immediate operand avoids a load instruction

Our First Example (미리 보기)

- Can we figure out the code?

```
swap(int v[], int k);  
{ int temp;  
  temp = v[k]  
  v[k] = v[k+1];  
  v[k+1] = temp;  
}
```

➡ swap:

```
      muli $2, $5, 4 // k in $5  
      add $2, $4, $2 // v in $4  
      lw $15, 0($2)  
      lw $16, 4($2)  
      sw $16, 0($2)  
      sw $15, 4($2)  
      jr $31 // return
```

The Constant Zero

- MIPS register 0 (\$zero) is the constant 0
 - Cannot be overwritten
- Useful for common operations
 - E.g., move between registers
`add $t2, $s1, $zero`



Review: Representation of Numbers

8-bit Binary Numbers (복습)

Unsigned	0	0000 0000	0	Signed
	1	0000 0001	1	
	.	.	.	
	.	.	.	
	127	0111 1111	127	
	128	1000 0000	-128	
	129	1000 0001	-127	
	130	1000 0010	-126	
	.	.	.	
	.	.	.	
0 ~ 128 (0 ~ 2^n-1)	254	1111 1110	-2	-128 ~ 127 ($-2^n \sim 2^n-1$)
	255	1111 1111	-1	

Unsigned Binary Integers (skip)

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $+2^n - 1$

- Example

- $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Using 32 bits

- 0 to +4,294,967,295

2s-Complement Signed Integers

(skip)

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$

- Example

- $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$

- Using 32 bits

- $-2,147,483,648$ to $+2,147,483,647$

MIPS (복습)

□ 32 bit signed numbers:

0000	0000	0000	0000	0000	0000	0000	0000	$_{\text{two}}$	=	0_{ten}	
0000	0000	0000	0000	0000	0000	0000	0001	$_{\text{two}}$	=	$+ 1_{\text{ten}}$	
0000	0000	0000	0000	0000	0000	0000	0010	$_{\text{two}}$	=	$+ 2_{\text{ten}}$	
...											
0111	1111	1111	1111	1111	1111	1111	1110	$_{\text{two}}$	=	$+ 2,147,483,646_{\text{ten}}$	<i>maxint</i>
0111	1111	1111	1111	1111	1111	1111	1111	$_{\text{two}}$	=	$+ 2,147,483,647_{\text{ten}}$	
1000	0000	0000	0000	0000	0000	0000	0000	$_{\text{two}}$	=	$- 2,147,483,648_{\text{ten}}$	<i>minint</i>
1000	0000	0000	0000	0000	0000	0000	0001	$_{\text{two}}$	=	$- 2,147,483,647_{\text{ten}}$	
1000	0000	0000	0000	0000	0000	0000	0010	$_{\text{two}}$	=	$- 2,147,483,646_{\text{ten}}$	
...											
1111	1111	1111	1111	1111	1111	1111	1101	$_{\text{two}}$	=	$- 3_{\text{ten}}$	
1111	1111	1111	1111	1111	1111	1111	1110	$_{\text{two}}$	=	$- 2_{\text{ten}}$	
1111	1111	1111	1111	1111	1111	1111	1111	$_{\text{two}}$	=	$- 1_{\text{ten}}$	

2s-Complement Signed Integers

(복습)

- Bit 31 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- $-(-2^n - 1)$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

Signed Negation

- Complement and add 1
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111 \dots 111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
 - $+2 = 0000 \ 0000 \dots 0010_2$
 - $-2 = 1111 \ 1111 \dots 1101_2 + 1$
 $= 1111 \ 1111 \dots 1110_2$

Sign Extension

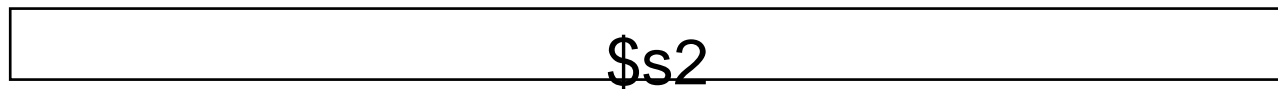
- Representing a number using more bits
 - Preserve the numeric value
- In MIPS instruction set (나중에 나옴)
 - `addi`: extend immediate value
 - `lb`, `lh`: extend loaded byte/halfword
 - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
 - `+2`: 0000 0010 => 0000 0000 0000 0010
 - `-2`: 1111 1110 => 1111 1111 1111 1110

Sign Extension

- Example: `lw $t0, 32($s2)`



- ALU: add after sign extension to get memory address



+



Hexadecimal (skip)

- Base 16
 - Compact representation of bit strings
 - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

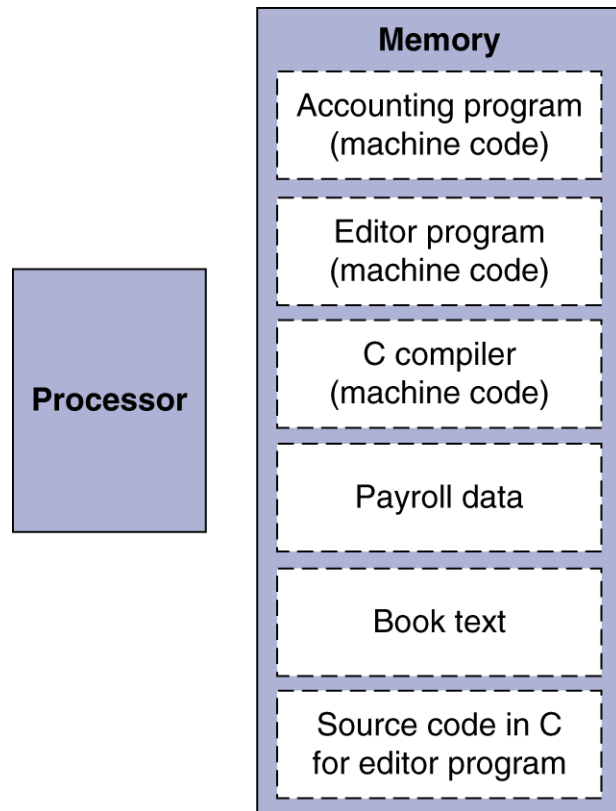
- Example: eca8 6420
 - 1110 1100 1010 1000 0110 0100 0010 0000



Representation of Instructions

Stored Program Computers

The BIG Picture



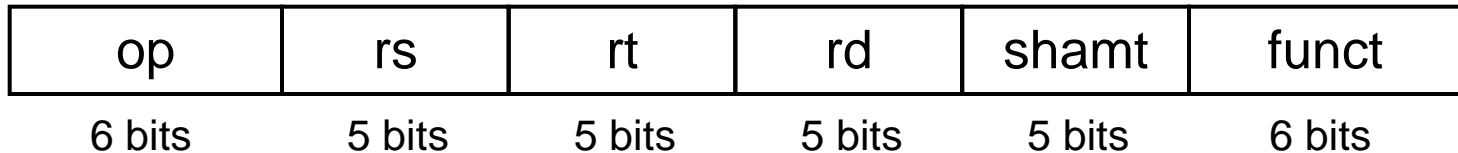
- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
 - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
 - Standardized ISAs

Representing Instructions

- Instructions are encoded in binary
 - Called machine code
- MIPS instructions
 - Encoded as 32-bit instruction words
 - Small number of formats encoding operation code (opcode), register numbers, ...
 - Regularity!
- Register numbers
 - \$t0 – \$t7 are reg's 8 – 15
 - \$t8 – \$t9 are reg's 24 – 25
 - \$s0 – \$s7 are reg's 16 – 23



MIPS R-format Instructions



■ Instruction fields

- op: operation code (opcode)
- rs: first source register number
- rt: second source register number
- rd: destination register number
- shamt: shift amount (00000 for now)
- funct: function code (extends opcode)

R-format Example

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

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

special	\$s1	\$s2	\$t0	0	add
---------	------	------	------	---	-----

0	17	18	8	0	32
---	----	----	---	---	----

000000	10001	10010	01000	00000	100000
--------	-------	-------	-------	-------	--------

$00000010001100100100000000100000_2 = 02324020_{16}$

MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
 - rt: destination or source register number
 - Constant: -2^{15} to $+2^{15} - 1$
 - Address: offset added to base address in rs
- *Design Principle 4: Good design demands good compromises*
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible

Machine Language

- Consider the load-word and store-word instructions,
 - What would the regularity principle have us do?
 - New principle: Good design demands a compromise
- Introduce a new type of instruction format
 - I-type for data transfer instructions
 - other format was R-type for register
- Example: `lw $t0, 32($s2)`

35	18	8	32
op	rs	rt	16 bit number

- Where's the compromise?

What we covered:

- **Register Addr. Mode (Arith.)**
- **Displacement Ad. Mode (ld/st)**

Back to ALU Instructions

- **Immediate Addr. Mode**

Revisit ALU instructions

- Small constants are used quite frequently
 - e.g., $A = A + 5;$
 $B = B + 1;$
 $C = C - 18;$
- Solutions? Why not?
 - put 'typical constants' in memory and load them.
 - create hard-wired registers (like \$zero) for constants
- MIPS Instructions (in about 25% of ALU instructions) :
 - `addi $29, $29, 4`
 - `slti $8, $18, 10`
 - `andi $29, $29, 6`
 - `ori $29, $29, 4`

Design Principle: Make the common case fast. *Which format?*



Immediate Operands

- Constant data specified in an instruction
`addi $s3, $s3, 4`
- No subtract immediate instruction
 - Just use a negative constant
`addi $s2, $s1, -1`
- *Design Principle 3: Make the common case fast*
 - Small constants are common
 - Immediate operand avoids a load instruction

32-bit Constants

- Most constants are small
 - 16-bit immediate is sufficient
- For the occasional 32-bit constant
 - lui rt, constant
 - Copies 16-bit constant to left 16 bits of rt
 - Clears right 16 bits of rt to 0

lui \$s0, 61	\$s0	0000 0000 0111 1101	0000 0000 0000 0000
ori \$s0, \$s0, 2304		0000 0000 0000 0000	0000 1001 0000 0000
	\$s0	0000 0000 0111 1101 0000 1001 0000 0000	



32-bit Constants

- Can handle the following situations?

ld \$t1, 2²¹(\$s2) // data transfer

addi \$t1, \$t2, 2²¹ // ALU instructions

// also in branch instructions later

- What we can do:

lhi \$t0, upper 16 bits

ori \$t0, \$t0, lower 16 bits

add \$t0, \$t0, \$s2 // add \$t1, \$t2, \$t0

ld \$t1, 0(\$t0)





Back to ALU Instructions

- Logical Instructions

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor

- Useful for extracting and inserting groups of bits in a word

Shift Operations

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

- shamt: how many positions to shift
- Shift left logical
 - Shift left and fill with 0 bits
 - sll by i bits multiplies by 2^i
- Shift right logical
 - Shift right and fill with 0 bits
 - srl by i bits divides by 2^i (unsigned only)

Logical Operations

Shifts

- Left/right, logical/arithmetic/rotational

`sll $t2, $s0, 4`

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

- Multiplication/division

Bitwise AND

- Apply bit pattern (usually called “mask”) to force 0s

Bitwise OR

- Force 1s

Constants are useful in AND and OR: `andi`, `ori`

AND Operations

- Useful to mask bits in a word
 - Select some bits, clear others to 0
- and \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
 - Set some bits to 1, leave others unchanged
- or \$t0, \$t1, \$t2

\$t2 0000 0000 0000 0000 0000 1101 1100 0000

\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 0000 0000 0000 0000 0011 1101 1100 0000

NOT Operations

- Useful to invert bits in a word
 - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
 - $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

`nor $t0, $t1, $zero` ←

Register 0: always
read as zero

\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 1111 1111 1111 1111 1100 0011 1111 1111

Operand Types

- Word**
- Half word**
- Byte**

Character Data

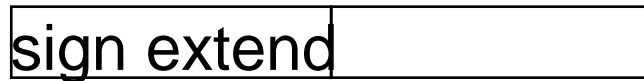
- Byte-encoded character sets
 - ASCII: 128 characters
 - 95 graphic, 33 control
 - Latin-1: 256 characters
 - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
 - Used in Java, C++ wide characters, ...
 - Most of the world's alphabets, plus symbols
 - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

■ `lw $t0, 32($s2)`



■ `lh`



`lhu`



`lb`



`lbu`



Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
 - String processing is a common case

lb rt, offset(rs) lh rt, offset(rs)

- Sign extend to 32 bits in rt

lbu rt, offset(rs) lhu rt, offset(rs)

- Zero extend to 32 bits in rt

sb rt, offset(rs) sh rt, offset(rs)

- Store just rightmost byte/halfword

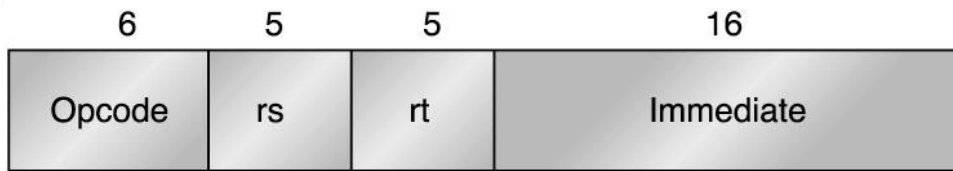


Where are we?

ISA Design Issues (Topic2-2, 반복)

- ❑ Operations (opcode)
 - How many, what types of instructions
 - ALU, data transfer, branch operations, others
- ❑ Operands
 - How many operands (in ALU instructions)?
 - Number of memory operands
 - How to specify the locations of operands
 - Addressing modes: register, direct, immediate, ...
 - Operand types (data types - more later)
- ❑ Instruction encoding
 - How to pack all in words

I-type instruction



Encodes: Loads and stores of bytes, half words, words, double words. All immediates ($rt \leftarrow rs \text{ op immediate}$)

Conditional branch instructions (rs is register, rd unused)

Jump register, jump and link register

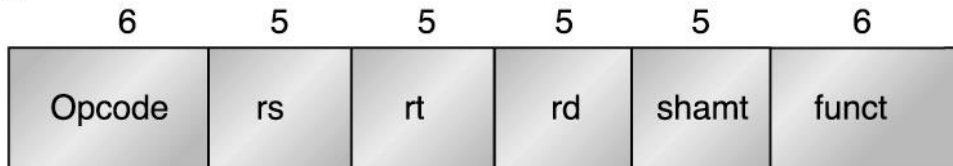
(rd = 0, rs = destination, immediate = 0)

lw/sw (Displacement mode)

beq (PC-relative mode)

addi (Immediate mode)

R-type instruction



Register-register ALU operations: $rd \leftarrow rs \text{ funct } rt$

Function encodes the data path operation: Add, Sub, . . .

Read/write special registers and moves

add (Register addr. mode)

jr

J-type instruction



Jump and jump and link

Trap and return from exception

jump (Pseudo-direct mode)

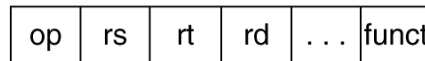
Overview of MIPS

Addressing Mode Summary

1. Immediate addressing



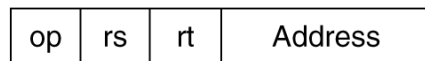
2. Register addressing



Registers

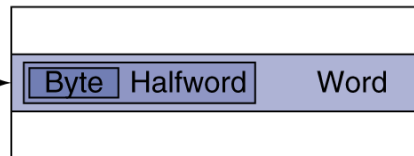
Register

3. Base addressing



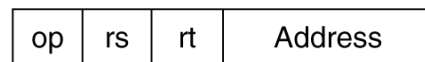
Memory

Register



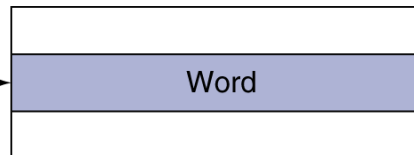
Load, store

4. PC-relative addressing



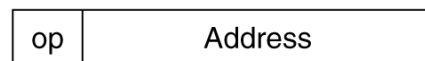
Memory

PC



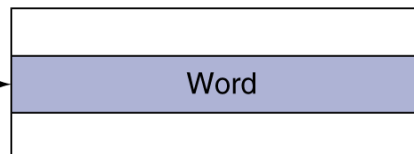
Branch, jump

5. Pseudodirect addressing



Memory

PC



Question on 64-Bit ISA

- ❑ We study 32-bit ISA, but I use 64-bit processor
- ❑ In 64-bit processor, instruction fetch bring in 64-bit
 - How do we utilize extra 32-bit?
 - Refer to MIPS64 ISA manual
 - † MIPS64 backward compatible to MIPS32