Big Picture (see Textbook or Class Homepage)

- Issue 1: computers, CSE, computer architecture? (4 주)
  - Fundamental concepts and principles
- Issue 2: ISA (HW-SW interface) design (5 주)
  - Ch. 1: computer performance
  - Ch. 2: language of computer; ISA
  - Ch. 3: data representation and ALU
- Issue 3: implementation of ISA (internal design) (5주)
  - Ch. 4: processor (data path, control, pipelining)
  - Ch. 5: memory system (cache memory)
- Short introduction to parallel processors

Class Topics (클래스 홈페이지 참조)

- ❑ Part 1: Fundamental concepts and principles
- ❑ Part 2: 빠른 컴퓨터를 위한 ISA design
  - Topic 1 Computer performance and ISA design (Ch. 1)
  - Topic 2 RISC (MIPS) instruction set (Chapter 2)
    - 2-1 ALU and data transfer instructions
    - 2-2 Branch instructions
    - 2-3 Supporting program execution
  - Topic 3 Computer arithmetic and ALU (Chapter 3)
- ❑ Part 3: ISA 의 효율적인 구현 (pipelining, cache memory)

Chapter 2

Instructions: Language of the Computer

Part 1:

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

Some of authors' slides are modified

Instruction Set

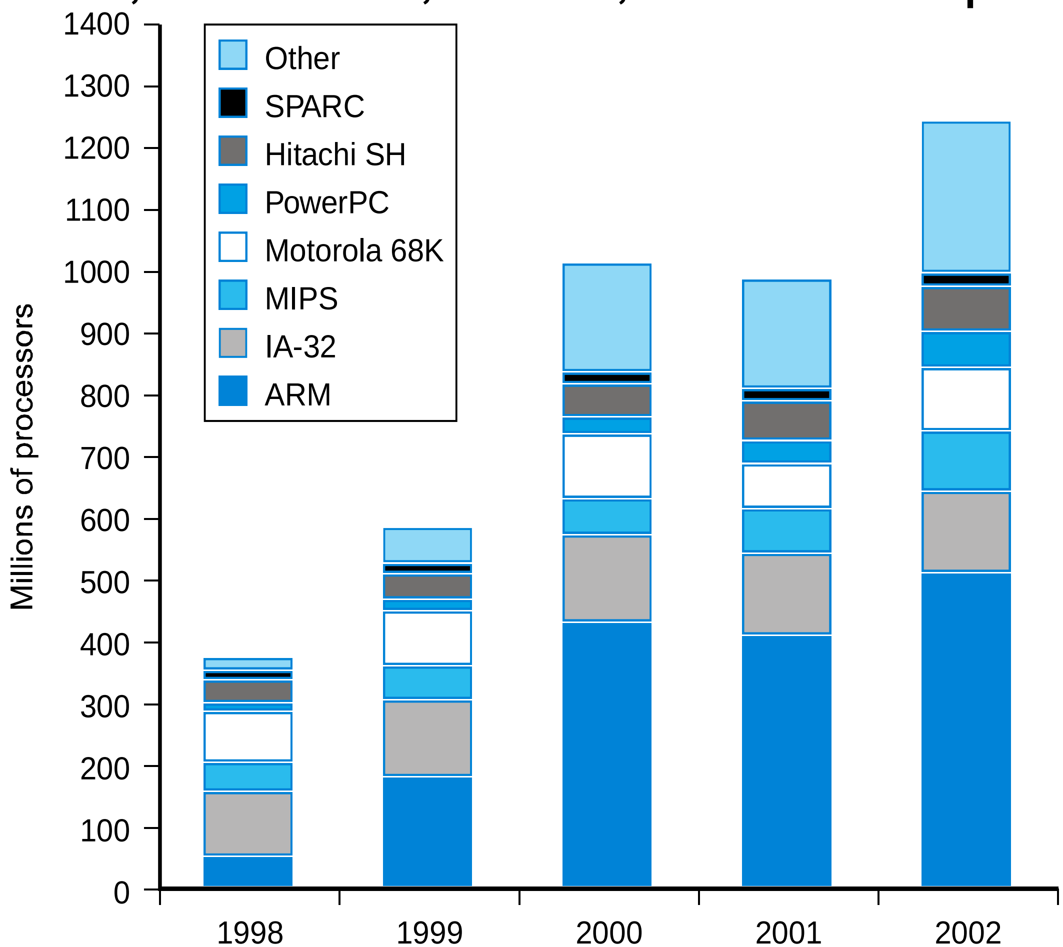
- The repertoire of instructions of a computer
- Early computers had very simple instruction sets
  - Simplified implementation
- CISC
- Many modern computers also have simple instruction sets
- Different computers have different instruction sets (독점성)
  - But with many aspects in common

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 (교육)
  - 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, …



32-bit 이상

Chapter 2

- Illustrate MIPS instructions
  - ALU instructions (part 1)
  - Data transfer instructions
  - Branch instructions (part 2)
  - Stored program concept
- Also illustrate
  - Supporting procedure calls (part 3)
  - Linking and running programs
  - ARM, IA-32 architectures (optional part 4)
- Can start thinking of HW-SW interactions

ISA 감상: 생각의 초점

- RISC ISA 는 어떻게 생겼나? 왜 그렇게 생겼나?
  - Commonly-used (i.e., simple) operations 지원
    - 자주 나오는 것을 single instruction 으로 (빠르게)
- RISC ISA 는 program execution 을 어떻게 지원하나?
  - Statement 들을 어떻게 지원하나 (Topics 2-1 and 2-2)
  - Function 들을 어떻게 지원하나 (Topic 2-3)

- Caution: *we go down to lower level of abstraction*
  - Machine instruction level

ALU instructions

(이미 상당히 많이 이야기 했음)

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c    // a gets b + c
```

// destination first in assembly

- 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
```

- Assembler names for R0, …, R31
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables

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
```

- ALU instructions: register-based
  - Register addressing mode (어떤 operands, 어떻게 사용)
- What if `g, h, i, j` are in memory?

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file (R0 – R31)
  - Store data for executing program
  - Used by compilers or assembly programmers
- Why only 32 general-purpose registers?

opcode	R(5)	R(5)	R(5)	

  - What if we use 64 or 16 registers? Which is better?
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

Data Transfer Instructions

- Load and store

(이미 상당히 많이 이야기 했음)

Memory Organization (반복)

- Viewed as a large, single-dimension array
  - A memory address is an index into the array
- Byte addressing: 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 (반복)

- 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
  - What are the least 2 significant bits of word address?

Memory Operands (복습)

- 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
- MIPS is Big Endian
  - Most-significant byte at least address of a word

Memory Operand Example 1

- C code: `g = h + A[8];` // array in memory
  - 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

- Base addressing mode (어떤 operands, 어떻게 사용)

Memory Operand Example 2

- C code: `A[12] = h + A[8];`
  - h in $s2, base address of A in $s3
- Compiled MIPS code:

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

opcode (lw)	R(5)	R(5)	offset

? large offset

Quiz

- C code: `A[12] = h + A[8];`
- Compiled MIPS code: (반복)

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

- Why not use these?

```
lw  $t0, 32-bit address    // more than 32 bits
lw  $t0, $t1($s3)    // compiler: 상수값 알고 있음
```

What we learned about MIPS:

- Loading words but addressing bytes
- Arithmetic on registers only

- Instruction / Meaning

Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 – $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100] = $s1`

† Register addressing mode (ALU)

† Base addressing Mode (memory access)

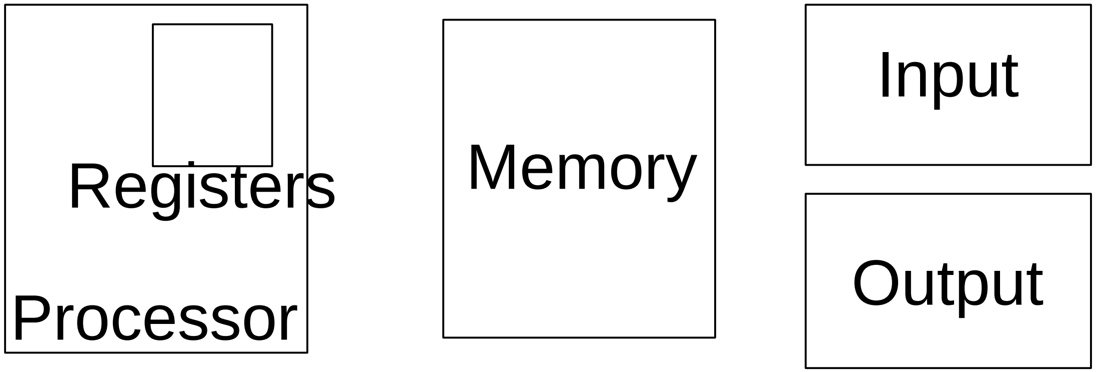
❖ Addressing mode: 어떤 operands 나오며 어떻게 사용하나

Registers vs. Memory

(이미 상당히 많이 이야기 했음)

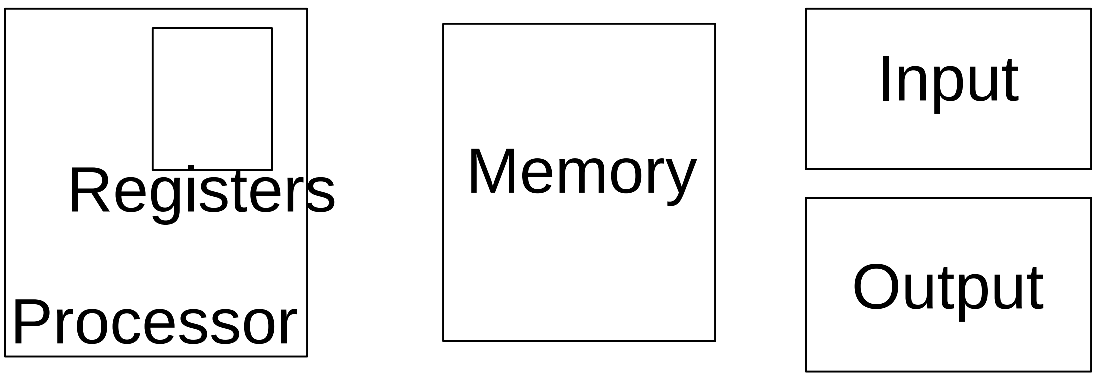
Registers vs. Memory (복습)

- Arithmetic instructions operands must be registers
  - Only 32 registers provided
- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed (IC ↑)



Register Allocation and Spill

- Compiler associates variables with registers (register allocation – compilers or assembly programmers)
  - What about programs with lots of variables
- Compiler uses registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!



Quiz on Load-Store Architecture

- Executing *3* instructions in load-store architecture

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

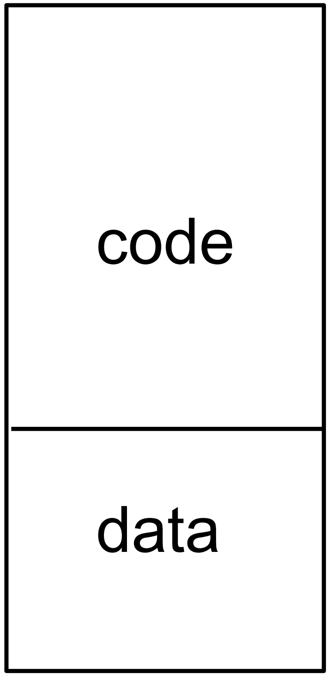
- Number of memory accesses?
  - Number of instruction accesses?
  - Number of data accesses?
  - Number of memory reads?
  - Number of memory writes?

Load-Store Architecture

- Memory access during instruction execution

	Fetch/decode	Execute
Instruction	√	
Instruction	√	
Instruction (load)	√	√ (read)
Instruction	√	
Instruction (store)	√	√ (write)
Instruction	√	
	Code (read)	Data (read/write)

Memory



The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - Move between registers

    ```
    add $t2, $s1, $zero
    ```

  - Clear register

    ```
    add $t2, $zero, $zero
    ```

  - Jump if equal zero (beqz) // not studied yet

    ```
    beq $t2, $zero, destination
    ```

Review: Representation of Numbers

- 2's Complement (음의 정수)

8-bit Binary Numbers (복습)

2의 보수

Unsigned		Binary	Signed	
	0	0000 0000	0	
	1	0000 0001	1	
		.		
		.		
Unsigned	127	0111 1111	127	Signed
	128	1000 0000	-128	
0 ~ 255	129	1000 0001	-127	-128 ~ 127
($0 \sim 2^8-1$)	130	1000 0010	-126	$-2^7 \sim (2^7-1)$
		.		
		.		
	254	1111 1110	-2	
	255	1111 1111	-1	

2's-Complement Signed Integers (복습)

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

2's-Complement Signed Integers (복습)

- 32 bit signed numbers: $-2^{31} \sim (2^{31} - 1)$

```
0000 0000 0000 0000 0000 0000 0000 0000two = 0ten
0000 0000 0000 0000 0000 0000 0000 0001two = + 1ten
0000 0000 0000 0000 0000 0000 0000 0010two = + 2ten
...
0111 1111 1111 1111 1111 1111 1111 1110two = + 2,147,483,646ten
0111 1111 1111 1111 1111 1111 1111 1111two = + 2,147,483,647ten     maxint
1000 0000 0000 0000 0000 0000 0000 0000two = – 2,147,483,648ten     minint
1000 0000 0000 0000 0000 0000 0000 0001two = – 2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0010two = – 2,147,483,646ten
...
1111 1111 1111 1111 1111 1111 1111 1101two = – 3ten
1111 1111 1111 1111 1111 1111 1111 1110two = – 2ten
1111 1111 1111 1111 1111 1111 1111 1111two = – 1ten
```

Signed Negation (2's Complement)

- Complement (1 → 0, 0 → 1) and add 1
- Example: negate +2
  - $+2 = 0000\ 0000 \ldots 0010_2$
  - $-2 = 1111\ 1111 \ldots 1101_2 + 1$

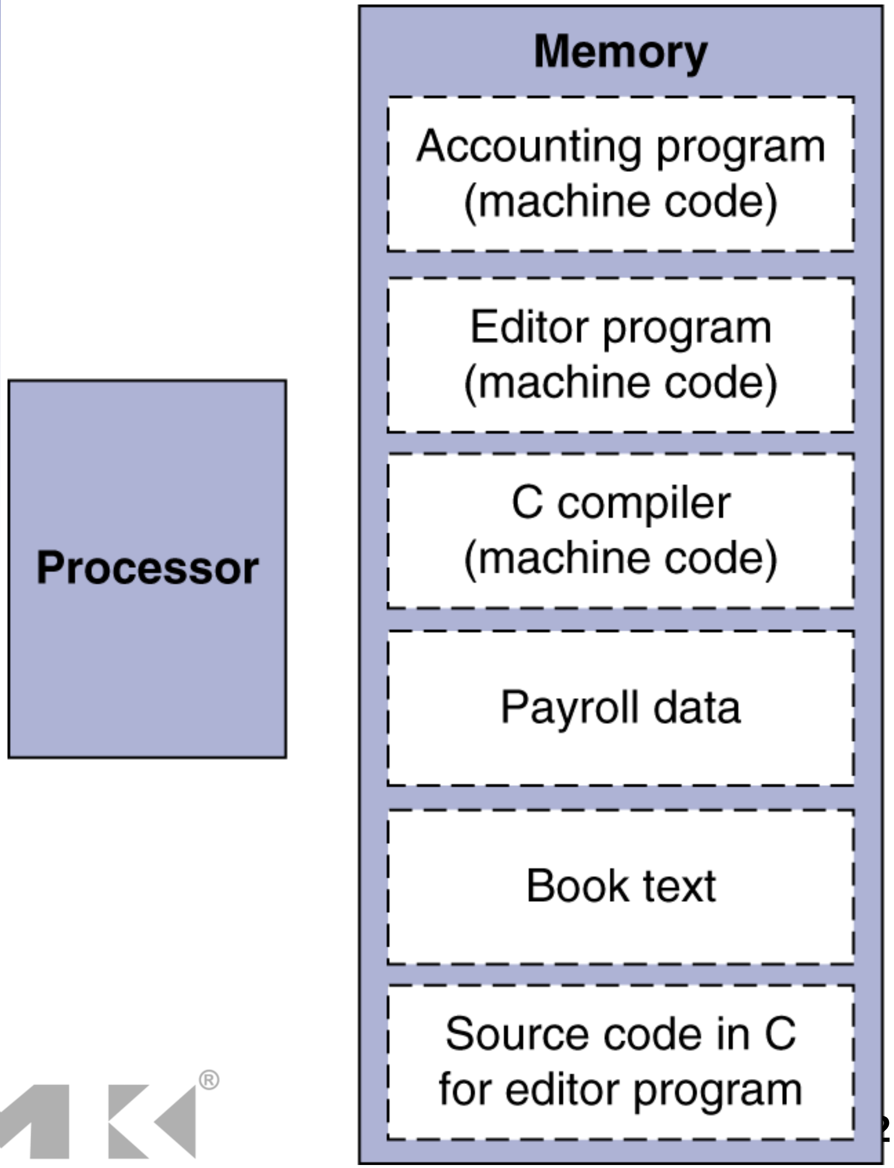
    $= 1111\ 1111 \ldots 1110_2$
- 왜 2의 보수를 사용하나?

$$x + \overline{x} = 1111...111_2 = -1$$ // not good for ALU

$$\overline{x} + 1 = -x$$ // better; 2's complement

Representation of Instructions

Stored Program Computers



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …

Representing Instructions

- (Assembly) 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, … (more later)
- Register names (암기 대상이 아님)
  - $t0 – $t7 are registers 8 – 15
  - $t8 – $t9 are registers 24 – 25
  - $s0 – $s7 are registers 16 – 23

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` // destination first in assembly

special	\$s1	\$s2	\$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

MIPS R-format Instructions

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

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

MIPS R-format Instructions

- R-format instructions (register-based ALU instructions)
  - Opcode: 0, function code determines operation
- Why use function codes?
  - 6-bit opcode not sufficient
- Why not use 8-bit opcode?
  - Limit the space for operands (see next instruction also)

Machine Language

- Consider the load-word and store-word instructions,

```
lw $t0, 32($s2)
```

35	18	8	32

op	rs	rt	16 bit number

- Introduce a new type of instruction format
  - I-type for data transfer instructions (vs. R-type)
- What would the regularity principle have us do?
- New principle: Good design demands a compromise

MIPS I-format Instructions

```
lw $t0, 32($s2)
```

op: 35	rs	rt	constant or offset: 32
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number (lw, sw 다름)
  - Constant: -2^{15} to $+2^{15}-1$ (large constant?)
  - Address: offset added to base address in rs
- *Design Principle 3:* Good design demand good compromise
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

To Think about: R and I Formats

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add

0	17	18	8	0	32

```
lw  $t0,  32($s2)
```

35	18	8	32
6 bits	5 bits	5 bits	16 bits

- Two formats
  - What does that mean in terms of implementation?
  - Why does it help to keep formats similar?

To Think about: I-format Instructions

```
lw  $t0,  32($s2)
```

35	18	8	32
6 bits	5 bits	5 bits	16 bits

- Is 16-bit offset a good choice? How do we know?
  - HW-SW interactions (benchmark programs)
- Related issues: #registers, #bits for opcode
  - What about other choices?
    - 6-bit opcode, 16 registers, 18-bit offset
    - 8-bit opcode, 32 registers, 14 bit offset
- How do designers confirm that a design is good?

Sign Extension

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

35	18	8	32
op	rs	rt	16 bit number (2's complement)
	($s2)	($t0)	

- ALU: add after sign extension to get memory address

$s2	

+

	16 bit number (2's complement)

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value (examples: 8-bit to 16-bit)

    +2: 0000 0010 => 0000 0000 0000 0010

    –2: 1111 1110 => 1111 1111 1111 1110
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- In MIPS I-format instructions
  - lw, sw: extend the offset
  - `addi`: extend immediate value (more later)
  - `beq`, `bne`: extend the displacement (more later)

What we covered:

- Register addr. mode (Arithmetic)
- Base addressing mode (ld/st)

Back to ALU Instructions

- Immediate addressing mode

(총 5개 중에서 3번째 addressing mode)

Revisit ALU instructions

- Small constants used frequently (25% of ALU instructions)

  A = A + 5;   B = B + 1;   C = C - 18;

- Solutions? Why not?
  - Put 'typical constants' in memory and load them
  - More than one instruction for common case
- MIPS instructions:

```
addi $29, $29, 5
          //  add $29, $29, $27
andi $29, $29, 6
```

- Immediate addressing mode (어떤 operands, 어떻게 사용)

Immediate Operands

- Constant data instruction: `addi $29, $29, 4`
- Which format do we use?
  - I format: as many bits as possible for immediates
  - Keep formats similar

addi	29	29	#4
op	rs	rt	16 bit immediate operands

- Sign extension of immediate operands
- What about large immediate operands?

? `addi $s3, $s3,` 2^{20}

Immediate Operands

- *Design Principle 4:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction
- No subtract immediate instruction
  - Just use a negative constant

```
addi $s2, $s1, -1
```

To Think about: I-format Instructions (반복)

```
addi $29, $29, 4
```

addi	29	29	4
6 bits	5 bits	5 bits	16 bits

- Is 16-bit immediate a good choice? How do we know?
  - HW-SW interactions (benchmark programs)
- Related issues: #registers, #bits for opcode
  - What about other choices?
    - 6-bit opcode, 16 registers, 18-bit immediate
    - 8-bit opcode, 32 registers, 14 bit immediate
- How do designers confirm that a design is good?

Large Constants

- What if immediate operands require more than 16 bits?
  - I format immediate operands: $-2^{15} \sim (2^{15} - 1)$

    `? addi $s2, $s1,` $(8\times2^{16} + 128)$
- Instead of using “addi”
  - Load a large constant into a register (e.g., $s0)
    - This requires two machine instructions
  - Then use register add

```
add $s2, $s1, $s0
```
- A total of three instructions – but this is not common

32-bit Constants

$$a = b + (8 \times 2^{16} + 128);$$

Instruction	Register	Upper 16 bits	Lower 16 bits
`lui $s0, 8`	$s0	0000 0000 0000 1000	0000 0000 0000 0000
`ori $s0,$s0,128`		0000 0000 0000 0000	0000 0000 1000 0000
	$s0	0000 0000 0000 1000	0000 0000 1000 0000

- Instruction 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

32-bit Constants

- Can handle the following situations? (not common)

  addi $t1, $t2, 2^{21} // ALU instruction

  lw $t1, 2^{21}($s2) // data transfer

- What we can do:

```
lui  $t0, upper 16 bits          lui  $t0, upper 16 bits
ori  $t0, $t0, lower 16 bits     ori  $t0, $t0, lower 16 bits
add  $t1, $t2, $t0               add  $t0, $t0, $s2
                                 lw   $t1, 0($t0)
```

Example

- 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[0] in $4
        lw $15, 0($2)  // v[k]
        lw $16, 4($2)  // v[k+1]
        sw $16, 0($2)
        sw $15, 4($2)
        jr $31         // return
```

Back to ALU Instructions

- Logical Instructions

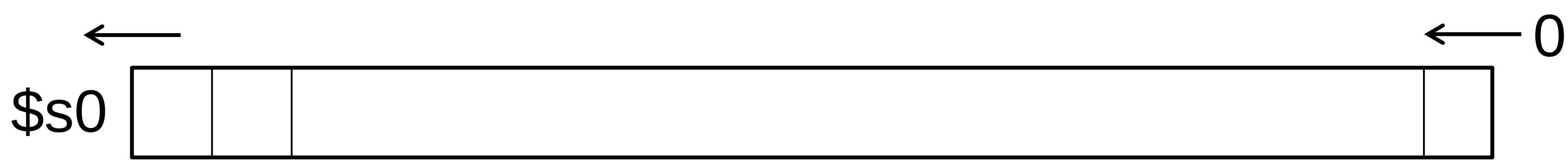
Logical Operations

- Instructions for bitwise manipulation
- Manipulate a part (group of bits) of a word
  - Move/extract bits, force 1's, force 0's

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

Shift Operations

- Shift left logical: shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i




```
sll    $t2,  $s0, 4      // why R-format?
```

0	0	16	10	4	0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Shift Operations

- Shift right logical: shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)
- Shift right arithmetic (`sra`)
  - Division for signed numbers
- Variable bits of shift (determined at runtime)

```
sllv $t0, $t1, $t2
srlv $t0, $t1, $t2
srav $t0, $t1, $t2
```

Bitwise AND Operations

- Select some bits, clear others to 0

```
and $t0, $t2, $t1
```

\$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

- `andi $t0, $t2,` "1111" $\times 2^{10}$ // immediate

Bitwise OR Operations

- Set some bits to 1, leave others unchanged

```
or $t0, $t2, $t1
```

\$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

- `ori $t0, $t2, "1111"` $\times 2^{10}$ // immediate

Bitwise NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

Logical Operations (반복)

- Instructions for bitwise manipulation
- Manipulate a part (group of bits) of a word
  - Move/extract bits, force 1's, force 0's

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

To Think about

- What about &&, ||, ! in if-statement? ("syntactic sugar")

```
if A
   if B
      "do the work"
```

```
if A
   "do the work"
   jump next statement
if B
   "do the work"
```

```
if A
   skip next block
"do the work"
```

Operand Types to Support

- Word
- Half word
- Byte

(Architect: 어떤 operands 지원하나?)

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

- Multimedia data since 1990s

Byte/Halfword Operations

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

	$t0

- `lh` `lhu`

sign extend	

zero extend	

- `lb` `lbu`

sign extend	

zero extend	

Byte/Halfword Operations (부연)

- Could use bitwise operations
  - But string processing is a common case
- MIPS byte/halfword load/store
  - Sign extend to 32 bits in rt

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

  - Zero extend to 32 bits in rt

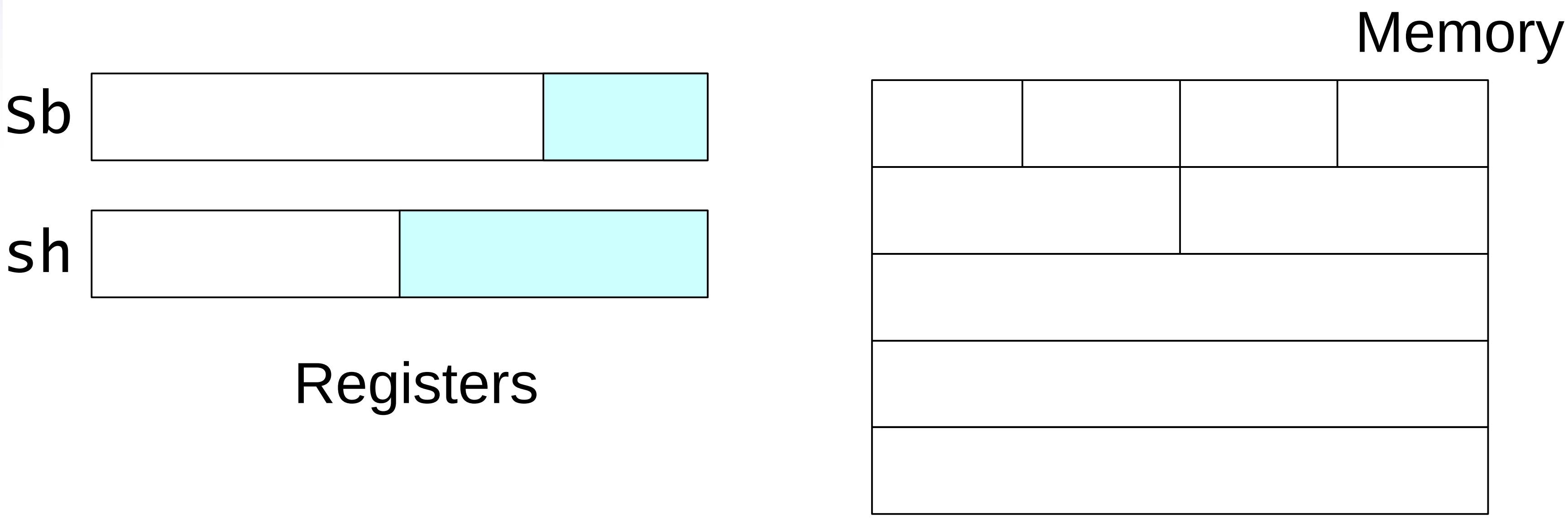
```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

Byte/Halfword Operations

- Store just rightmost byte/halfword

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

- Store 에서는 빈 공간 발생하지 않음



Where are we?

- End of (Core) ALU and Data Transfer Instructions

(Some of) ISA Design Issues (반복)

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

I-type instruction

6	5	5	16
Opcode	rs	rt	Immediate

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

lw/sw (Base addr. mode)
beq (PC-relative mode)
addi (Immediate mode)

R-type instruction

6	5	5	5	5	6
Opcode	rs	rt	rd	shamt	funct

Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

add (Register addr. mode)
jr

J-type instruction

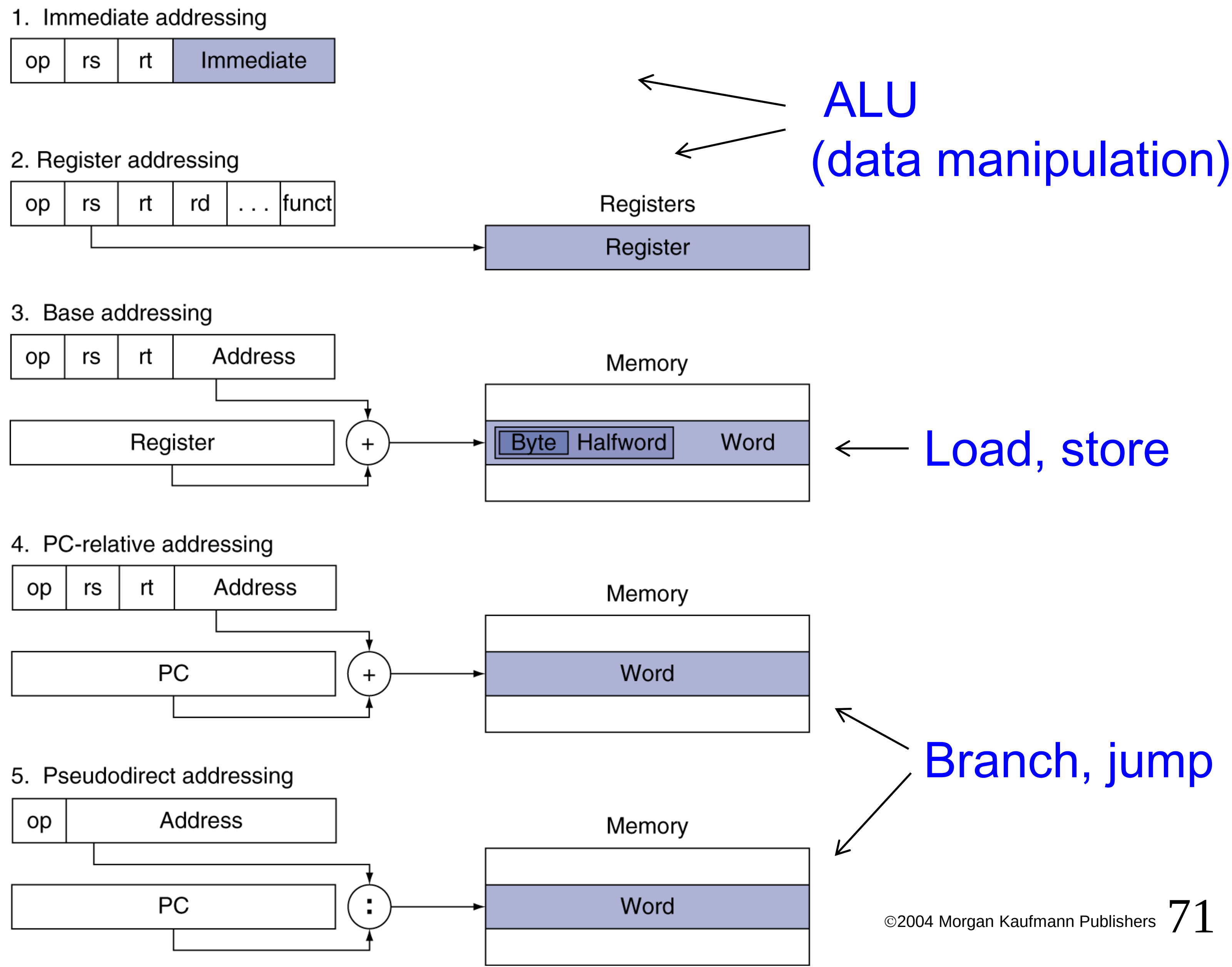
6	26
Opcode	Offset added to PC

Jump and jump and link
Trap and return from exception

jump (Pseudo-direct mode)

Overview of MIPS

Addressing Mode Summary

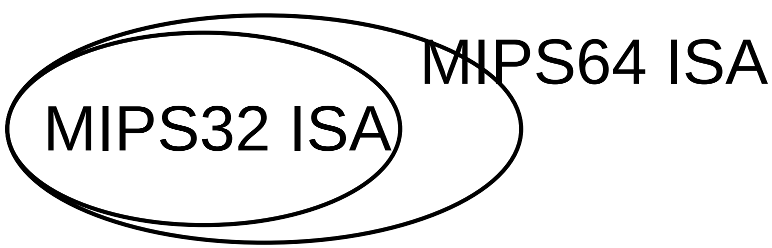


ISA 감상: 생각의 초점 (반복)

- RISC ISA 는 어떻게 생겼나? 왜 그렇게 생겼나?
  - Commonly-used (i.e., simple) operations 지원
    - 자주 나오는 것을 single instruction 으로 (빠르게)
- RISC ISA 는 program execution 을 어떻게 지원하나?
  - Statement 들을 어떻게 지원하나 (Topics 2-1 and 2-2)
  - Function 들을 어떻게 지원하나 (Topic 2-3)
- Caution: *we go down to lower level of abstraction*
  - Machine instruction level

64-Bit MIPS ISA

- We study 32-bit MIPS ISA, but I use 64-bit processor
- How do 64-bit MIPS instructions look like?
  - How do we utilize extra 32-bit?
    - Refer to MIPS64 ISA manual on Internet
- MIPS64 backward compatible to MIPS32



- Can fetch two instructions per memory access

Homework #7 (see Class Homepage)

1) Write a report summarizing the materials discussed in Topic 2-1

** 문장으로 써도 좋고 파워포인트 형태의 개조식 정리도 좋음

- Due: see Blackboard
  - Submit electronically to Blackboard

Class Topics (클래스 홈페이지 참조)

- Part 1: Fundamental concepts and principles
- Part 2: 빠른 컴퓨터를 위한 ISA design
  - Topic 1 Computer performance and ISA design (Ch. 1)
  - Topic 2 RISC (MIPS) instruction set (Chapter 2)
    - 2-1 ALU and data transfer instructions
    - 2-2 Branch instructions
    - 2-3 Supporting program execution
  - Topic 3 Computer arithmetic and ALU (Chapter 3)
- Part 3: ISA 의 효율적인 구현 (pipelining, cache memory)