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 2 (Control Instructions):

- Conditional branch
- Unconditional branch (jump)
- Procedure call and return

Some of authors' slides are modified

Control Instructions

- ❑ Alter control flow (or program execution flow)
  - Change PC value (or "next" instruction to be executed)
  - About 20% of all instructions (common case)
- ❑ MIPS conditional branch (or decision making) instructions:

```
bne   $t0,  $t1,  target-address      // branch if not equal
beq   $t0,  $t1,  target-address      // branch if equal
```

- ❑ C code: if (i == j) h = i + j;

```
            bne   $s0,  $s1,  target-address
            add   $s3,  $s0,  $s1
  Target:  ....
```

Conditional Branch

- ❑ How to specify target (or destination) address?
  - Compiler has 32-bit destination address
  - Need to specify destination in a fewer number of bits
- ❑ PC-relative addressing mode (어떤 operands, 어떻게 사용)

  bne r1, r2, jump-distance;

  - PC (destination) ← current PC + jump-distance
    - – Usually, destination is near current instruction
    - – Pack jump distance in a single instruction
      - † Make the common case fast

Conditional Branch

- ❑ Conditional branch instructions use I-format
  - Keep format as similar as possible

beq r1, r2, offset; **bne** r1, r2, offset

op	rs	rt	offset (or jump distance)
6 bits	5 bits	5 bits	16 bits

- ❑ Forward or backward, branch taken or untaken
  - Two’s complement offset : $- 2^{15} \sim (2^{15} – 1)$
- ❑ Performance question: is 16-bit offset adequate?
  - HW-SW interactions (benchmark programs)

Branch Offset in Words

beq r1, r2, offset; **bne** r1, r2, offset

op	rs	rt	16-bit offset

- Byte offset
  - Jump one instruction: 0000 0000 0000 0100
  - Jump two instructions: 0000 0000 0000 1000
- Word offset (표현 범위: 18-bit)
  - Jump one instruction: 0000 0000 0000 0001
  - Jump two instructions: 0000 0000 0000 0010
- PC-relative addressing
  - PC (destination) ← current PC + offset × 4

Conditional Branch

beq r1, r2, offset; **bne** r1, r2, offset

- ❑ Why not “**beq** r1, r2, r3” ?
- ❑ Relative jump is position-independent
  - Eliminate work in linking (e.g., dynamic library)

```
C code:     if (i == j) h = i + j;     (반복)

            bne   $s0,  $s1,  1 (jump in words)
            add   $s3,  $s0,  $s1
   Target:  ....
```

Chapter 2

Part 2 (Control Instructions):

- Conditional branch
- Unconditional branch (jump)
- Procedure call/return

Unconditional Branch

□ MIPS unconditional branch instruction: j target-address

□ C code:

```
if  (i != j)                       beq  $s4,  $s5,  Label1
      h = i + j;                   add  $s3,  $s4,  $s5
else                                j  Label2
      h = i - j;         Label1:   sub  $s3,  $s4,  $s5
                         Label2:   ...
```

□ Why new instruction? Why not “beq r1, r1, offset” ?

- Need to jump more than 16 bits
    - Procedure calls (or long GOTO)

Jump Addressing

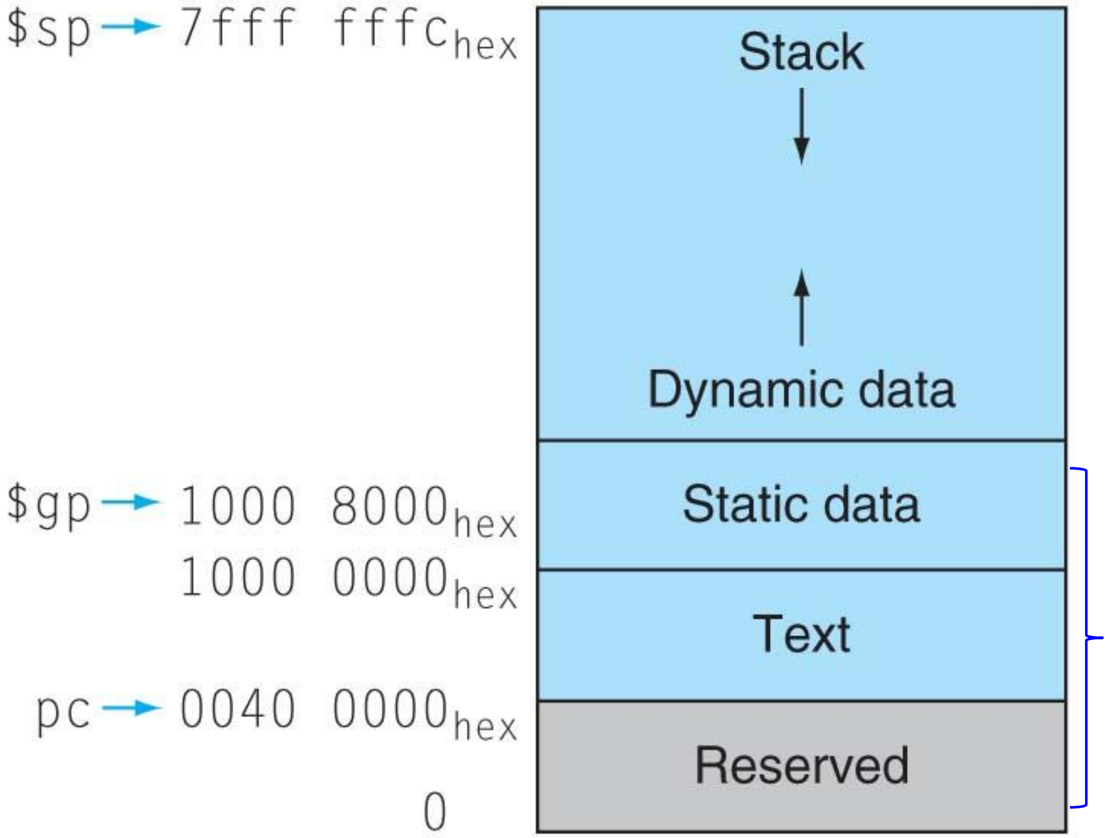
- J format

op	destination address (in words)
6 bits	26 bits

- How to specify 32-bit destination address?
  - Have 28-bit byte address
  - Remaining 4 bits?

Memory Layout (미리보기)

□ Figure 2.13 MIPS memory allocation for program and data



Process
virtual address space

Executable

static code:

0000 0000 ~ 0FFF FFFF

(2^{28} = 256 MB)

Jump Addressing

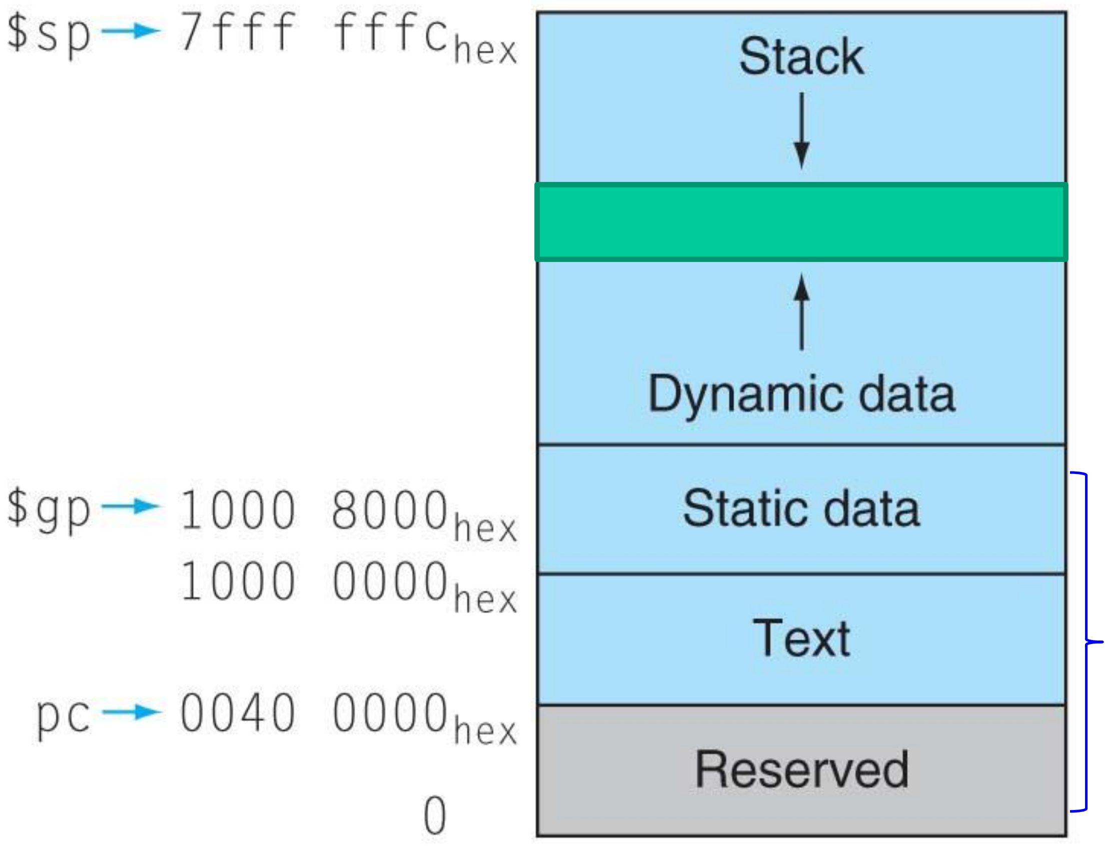
- J format

op	destination address (in words)
6 bits	26 bits

- Jump (j) destination could be anywhere in text segment
  - Note the 256MB boundary (28-bit)
  - Destination: “0000 : 26-bit destination in words : 00”
- Pseudo-direct addressing (어떤 operands, 어떻게 사용)
  - PC ← $PC_{31…28}$: (destination address × 4)
    - Why “$PC_{31…28}$” instead of “0000” ?

Memory Layout (미리보기)

❑ Figure 2.13 MIPS memory allocation for program and data



dynamic library:

3000 0000 ~ 3FFF FFFF

(2^{28} = 256 MB)

?

Executable

static code:

0000 0000 ~ 0FFF FFFF

(2^{28} = 256 MB)

ISA Design

- ❑ Computer system design
  - Architecture, OS, compiler
- ❑ Input for ISA design
  - OS vendors, application designers
- ❑ ISA design
  - Many SW-HW interactions

So Far:

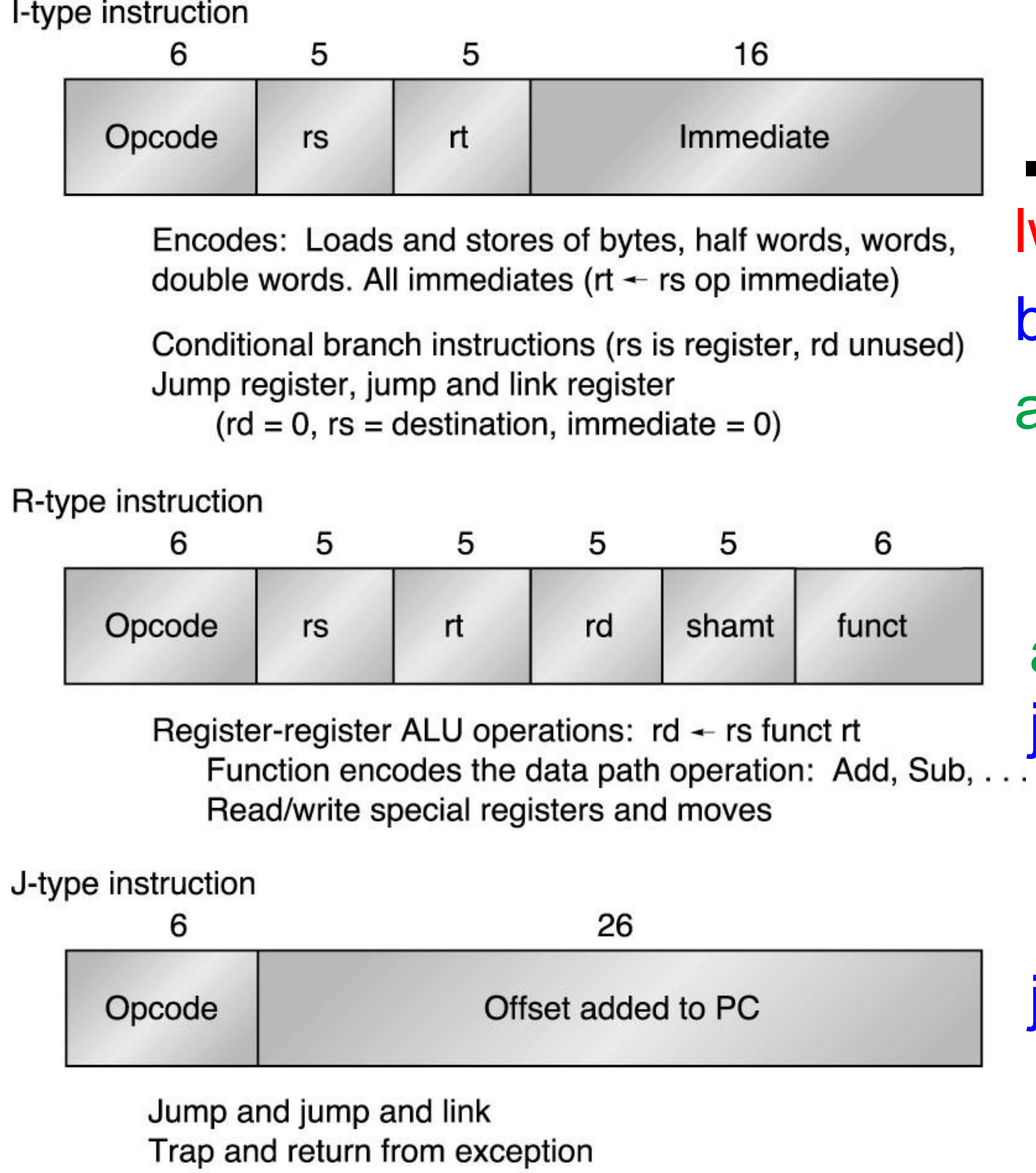
Instruction	Meaning
`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`
`lw $s1,100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1,100($s2)`	`Memory[$s2+100] = $s1`
`bne $s4,$s5,L`	`Next instr. is at Label if $s4 ≠ $s5`
`beq $s4,$s5,L`	`Next instr. is at Label if $s4 = $s5`
`j Label`	`Next instr. is at Label`

- Formats:

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Overview of MIPS

(반복)



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)

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

Jump and jump and link
Trap and return from exception

lw/sw (Base addr. mode)

beq (PC-relative mode)

addi (Immediate mode)

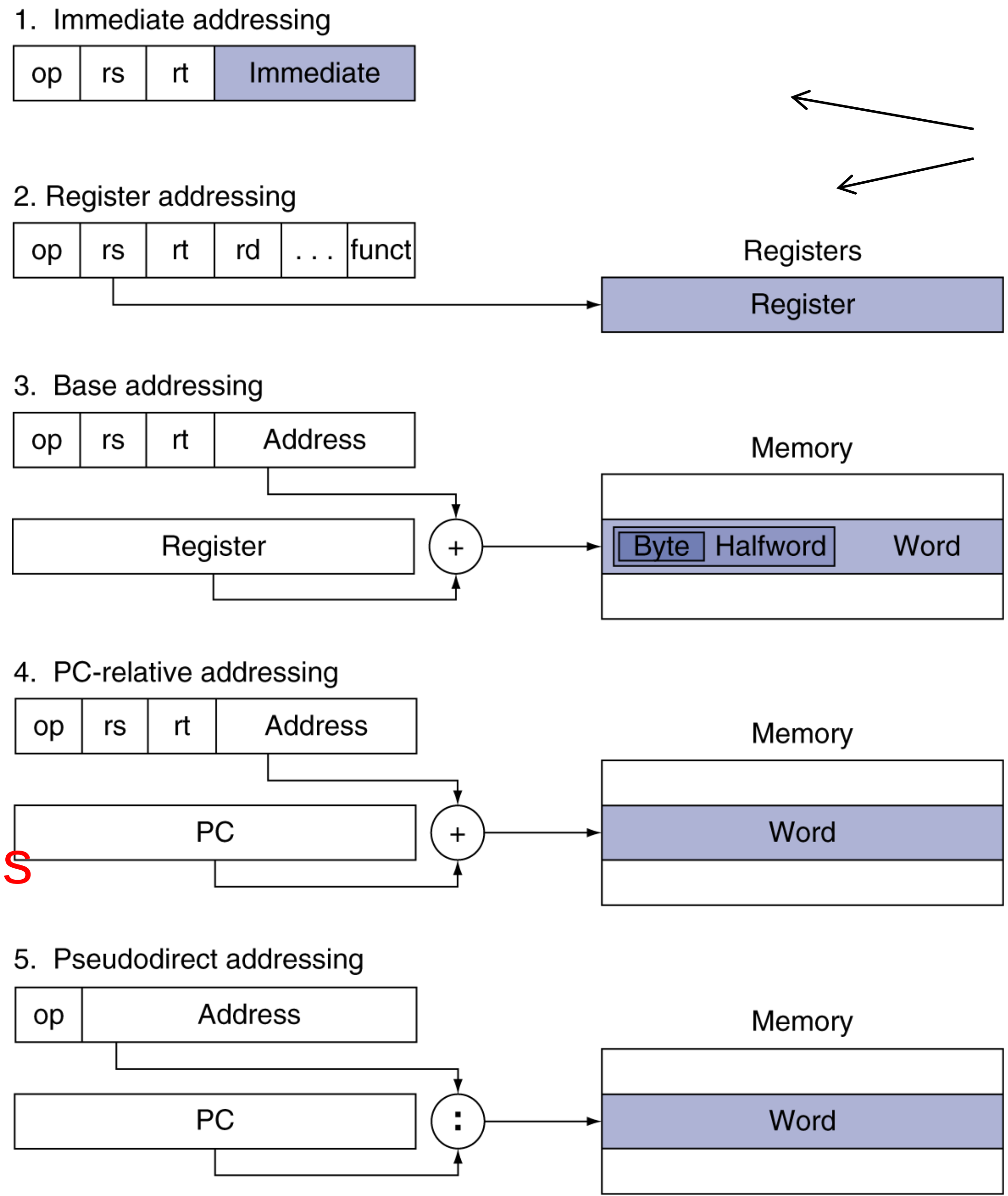
add (Register addr. mode)

jr

j (Pseudo-direct mode),

Only 3 (similar) formats;
easy to decode

Addressing Mode Summary (반복)



ALU (data manipulation)

Load, store

Branch, jump

Only 5 modes;

common operations

Chapter 2

Part 2 (Control Instructions):

More on beq, bne, j

Exercise: Compiling Loop Statements

- C code: while (save[i] == k) i += 1;
  - *i* in $s3, *k* in $s5, address of save in $s6

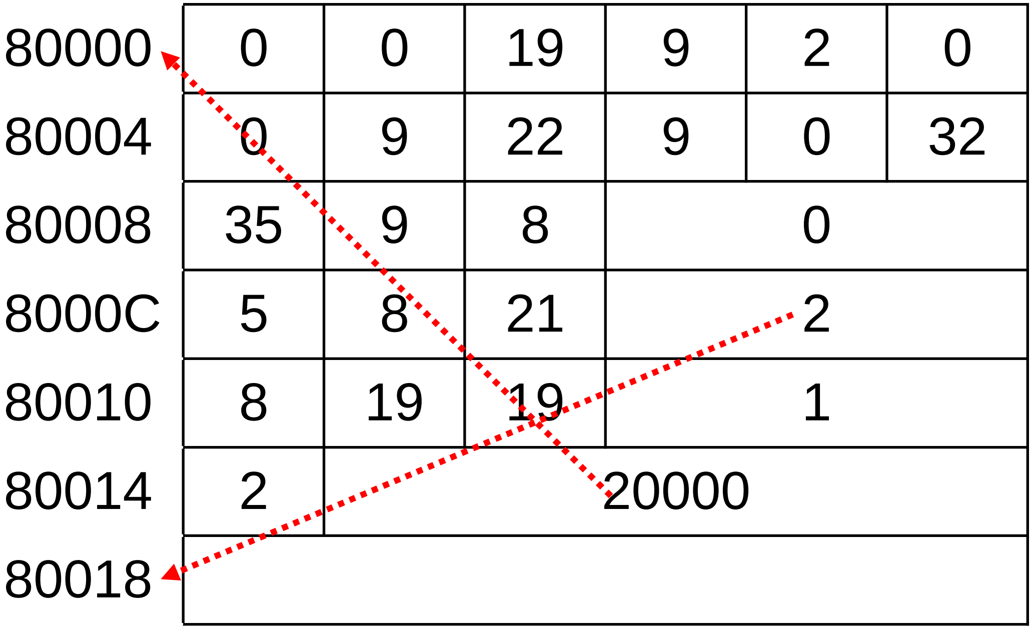
- Compiled MIPS code:

```
Loop: sll   $t1, $s3, 2         // 4 * i
      add   $t1, $t1, $s6       // address of save[i]
      lw    $t0, 0($t1)         // read in save[i]
      bne   $t0, $s5, Exit
      addi  $s3, $s3, 1         // i++
      j     Loop
Exit: …
```

Target Addressing

- Loop code from previous example
    - Assume Loop at location 80000

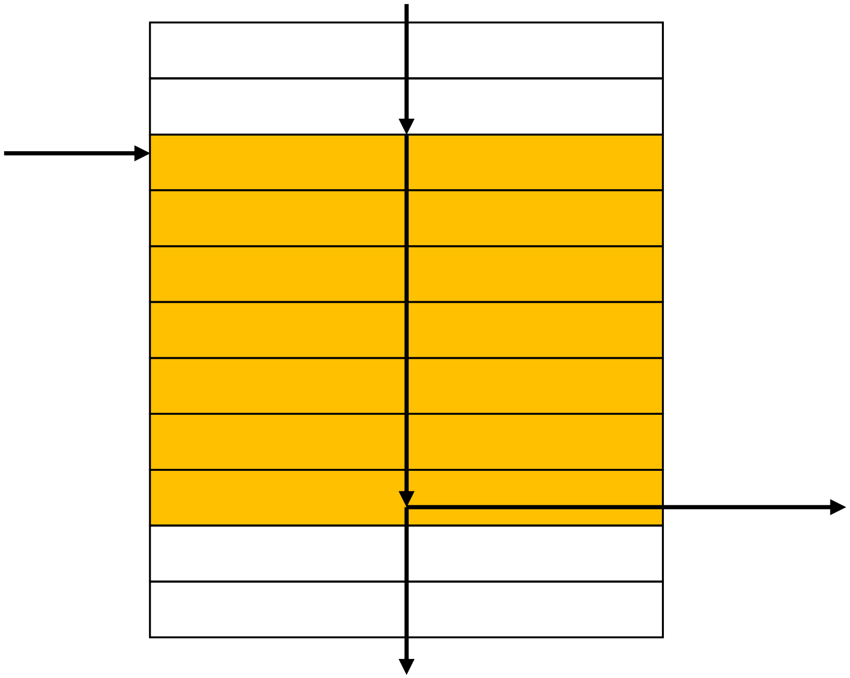
```
Loop: sll   $t1, $s3, 2
      add   $t1, $t1, $s6
      lw    $t0, 0($t1)
      bne   $t0, $s5, Exit
      addi  $s3, $s3, 1
      j     Loop
Exit: …
```



80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
8000C	5	8	21	2		
80010	8	19	19	1		
80014	2	20000				
80018						

Basic Blocks

- ❑ A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- ❑ A compiler identifies basic blocks for optimization
- ❑ An advanced processor can accelerate execution of basic blocks

Comparison for Branch

- ❑ What about branch-if-less-then:

```
                                   if  $s1  <  $s2  then
     slt  $t0,  $s1,  $s2                $t0 = 1
                                   else
                                         $t0 = 0
```

- ❑ C code:

```
                                  slt   $t0,  $s4,  $s5
     if (i < j)                   beq  $t0,  $zero,  Label1
        h = i + j;                add  $s3,  $s4,  $s5
     else                         j  Label2
        h = i - j;       Label1:  sub  $s3,  $s4,  $s5
                         Label2:       ...
```

Comparison for Branch

- ❑ **slt** rd, rs, rt (부연)
  - If (rs < rt) rd = 1; else rd = 0;
  - Use in combination with **beq**, **bne** (two instructions)

```
slt   $t0, $s1, $s2        #  if ($s1 < $s2)
bne $t0, $zero, L          #  branch to L
```

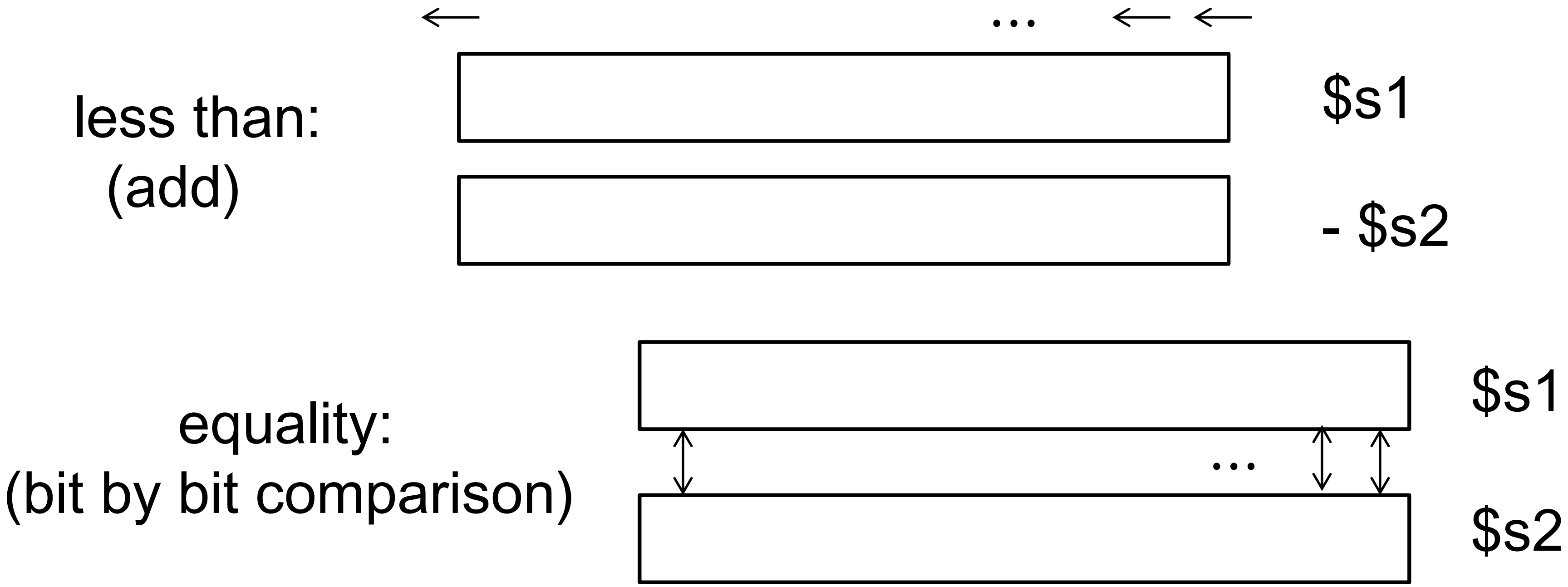
- ❑ Another common C code

```
if (i < 3) …
```

- ❑ **slti** rd, rs, constant (I-format)
  - If (rs < constant) rd = 1; else rd = 0;

Comparison for Branch

- ❑ Comparison instructions: **slt, sgt, sge, sle**, **slti,** ….
- ❑ Why not use "**blt** $s1, $s2, Label" in previous example?
  - Hardware for <, ≥, … slower than =, ≠   (Chapter 5)



less than:
(add)

$s1

- $s2

equality:
(bit by bit comparison)

$s1

$s2

Comparison for Branch

- ❑ Why not use "**blt** $s1, $s2, Label" in previous example?
  - • Hardware for <, ≥, … slower than =, ≠ (Chapter 5)
    - – Combining with branch involves more work per instruction, requiring a slower clock
    - – All instructions penalized!
- ❑ **beq** and **bne** are the common case
  - • This is a good design compromise (IC vs. cct)
    - – Are you sure? Run benchmarks!

Signed vs. Unsigned Comparison

- ❑ Signed comparison: **slt, slti**
- ❑ Unsigned comparison: **sltu, sltui**
- ❑ Example

$s0 = 1111 1111 1111 1111 1111 1111 1111 1111

$s1 = 0000 0000 0000 0000 0000 0000 0000 0001

- **slt** $t0, $s0, $s1 // signed

  –1 < +1 ⇒ $t0 = 1

- **sltu** $t0, $s0, $s1 // unsigned

  +4,294,967,295 > +1 ⇒ $t0 = 0

Chapter 2

Part 2 (Control Instructions):

- Unconditional branch:

jr (jump register) as well as j (jump)

Additional Instruction Support

- ❑ What if target address unknown at compile time (runtime info.)
  - J-format **j**

op	destination address
6 bits	26 bits

  - I-format **beq**

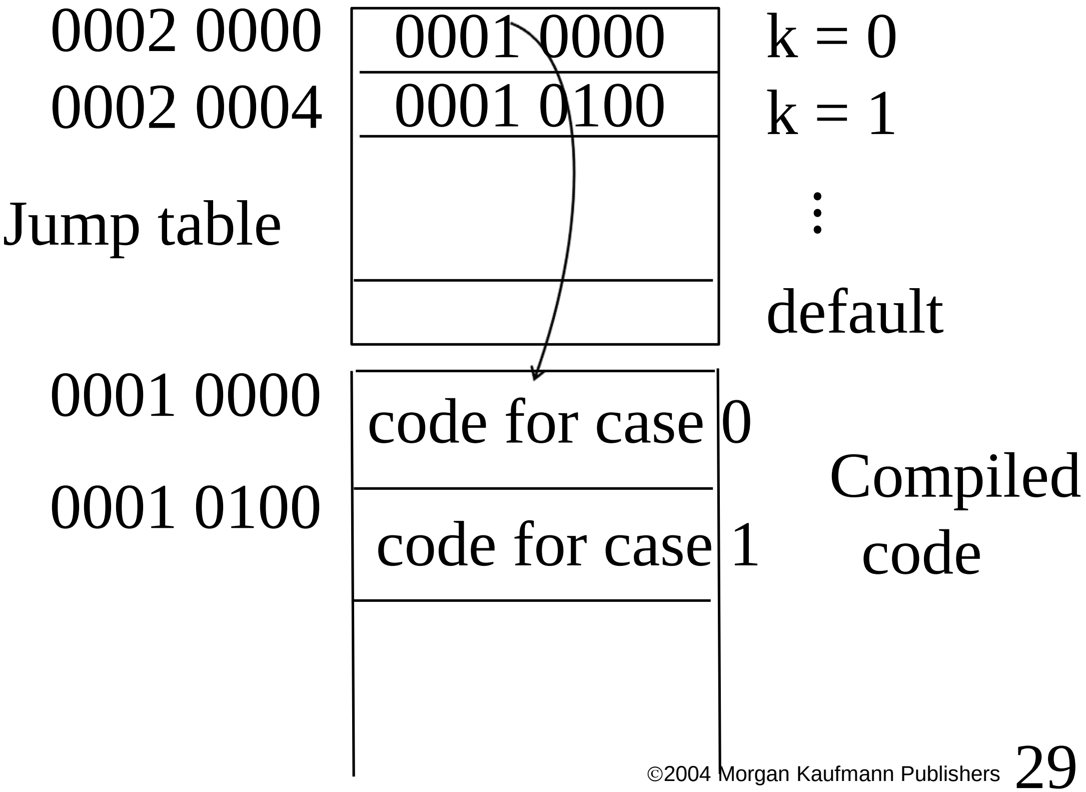
op	rs	rt	jump-distance
6 bits	5 bits	5 bits	16 bits

- ❑ Case or switch statements

Switch Statement and Jump Register (jr)

```
Switch (k) {
    case 0:  ---;
                break;
    case 1:  ---;
                break;
    .
    .
    case n:  ---;
                break;
    default:  ---;
```

```
$t0 = k * 4;
$t0 = $t0 + 0002 0000
lw  $t1, 0($t0)
jr   $t1                // R-format
```



0002 0000 | 0001 0000 | k = 0

0002 0004 | 0001 0100 | k = 1

⋮

Jump table

default

0001 0000 | code for case 0

0001 0100 | code for case 1

Compiled code

Switch Statement and Jump Register (jr)

- ❑ Switch statement example
  - Why not use “ if else” ?
  - What if cases are 1, 257, 10534, …?

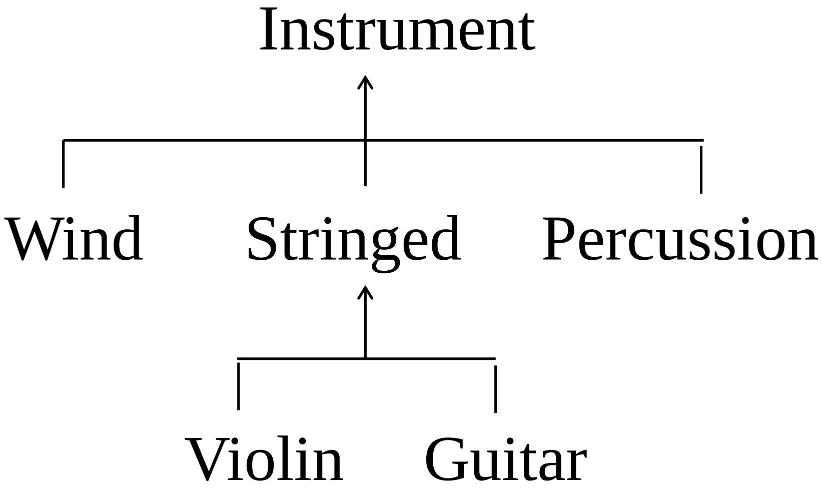
- ❑ Indirection is a powerful method

Jump Register (jr) Instruction

- What if target address unknown at compile time (runtime info.)
  - Case or switch statements
  - Virtual functions in OOP (polymorphism, dynamic binding)
  - Dynamically shared libraries (or DLL)
  - Return from procedure call, …
- **jr** (jump register) instruction // jr $s0 (R-format)
  - Dynamic binding: target address determined at runtime
  - Full 32-bit jump address

OOP and Dynamic Binding

- Inheritance



- Container: addresses of base class objects

ArrayList<Instrument>

violin	guitar	• •

  - Upcasting, “is-a” relationship

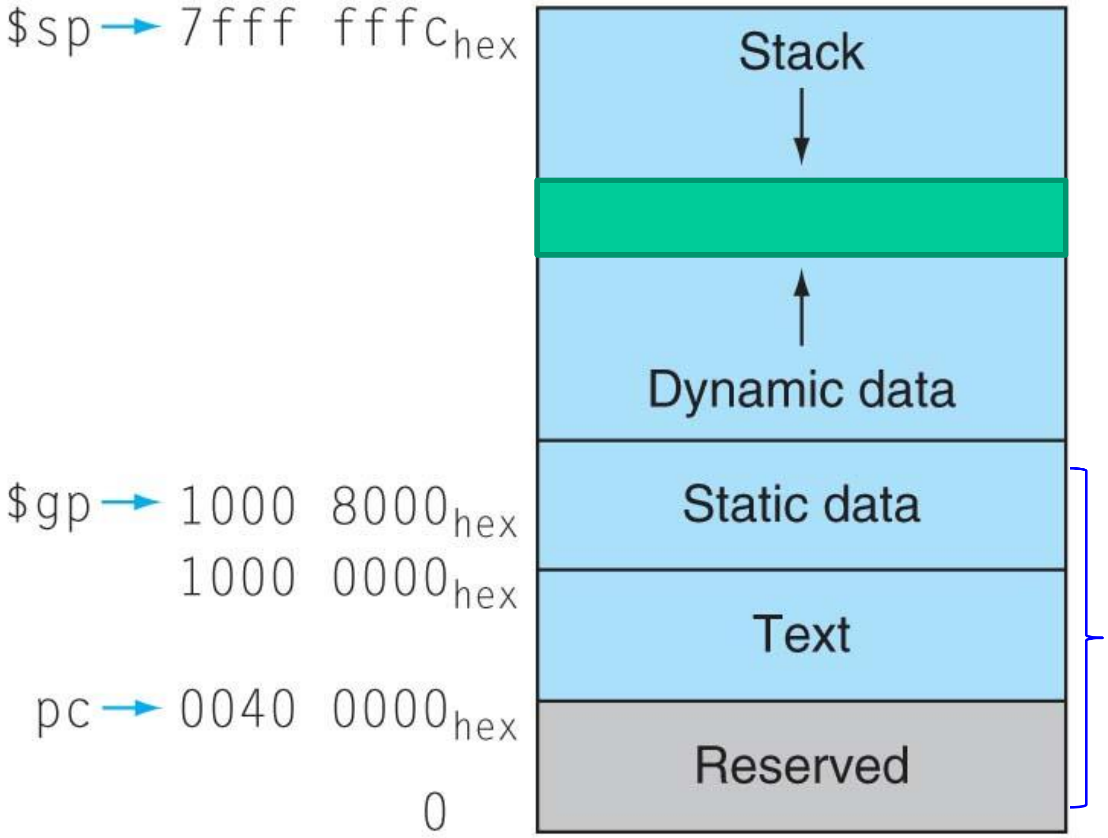
- OBJECT.print_type()
  - Dynamic (runtime) binding, polymorphism

Jump Register (jr) Instruction (반복)

- ❑ What if target address unknown at compile time (runtime info.)
  - Case or switch statements
  - Virtual functions in OOP (polymorphism, dynamic binding)
  - Dynamically shared libraries (or DLL)
  - Return from procedure call, …
- ❑ jr (jump register) instruction // jr $s0 (R-format)
  - Dynamic binding: target address determined at runtime
  - Full 32-bit jump address

Memory Layout (미리보기, 반복)

- Figure 2.13 MIPS memory allocation for program and data



dynamic library:
3000 0000 ~ 3FFF FFFF
(2^{28} = 256 MB)

?

Executable
static code:
0000 0000 ~ 0FFF FFFF
(2^{28} = 256 MB)

Chapter 2

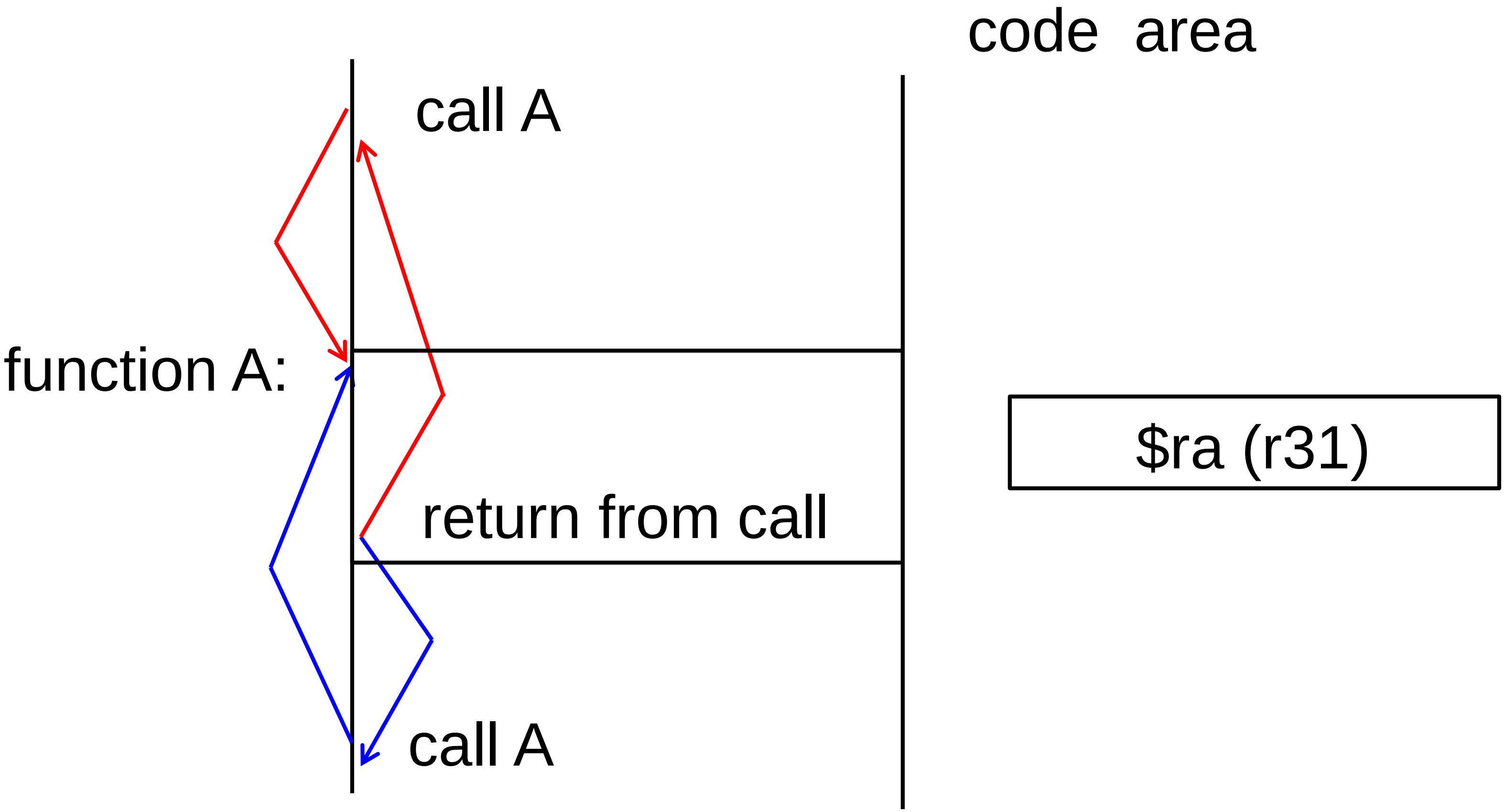
Part 2 (Control Instructions):

- Conditional branch
- Unconditional branch (jump)
- Procedure call and return

Return from Procedure Call

- ❑ Jump register (jr) instruction and return address register

```
jr  $ra
```



Procedural Call Instructions

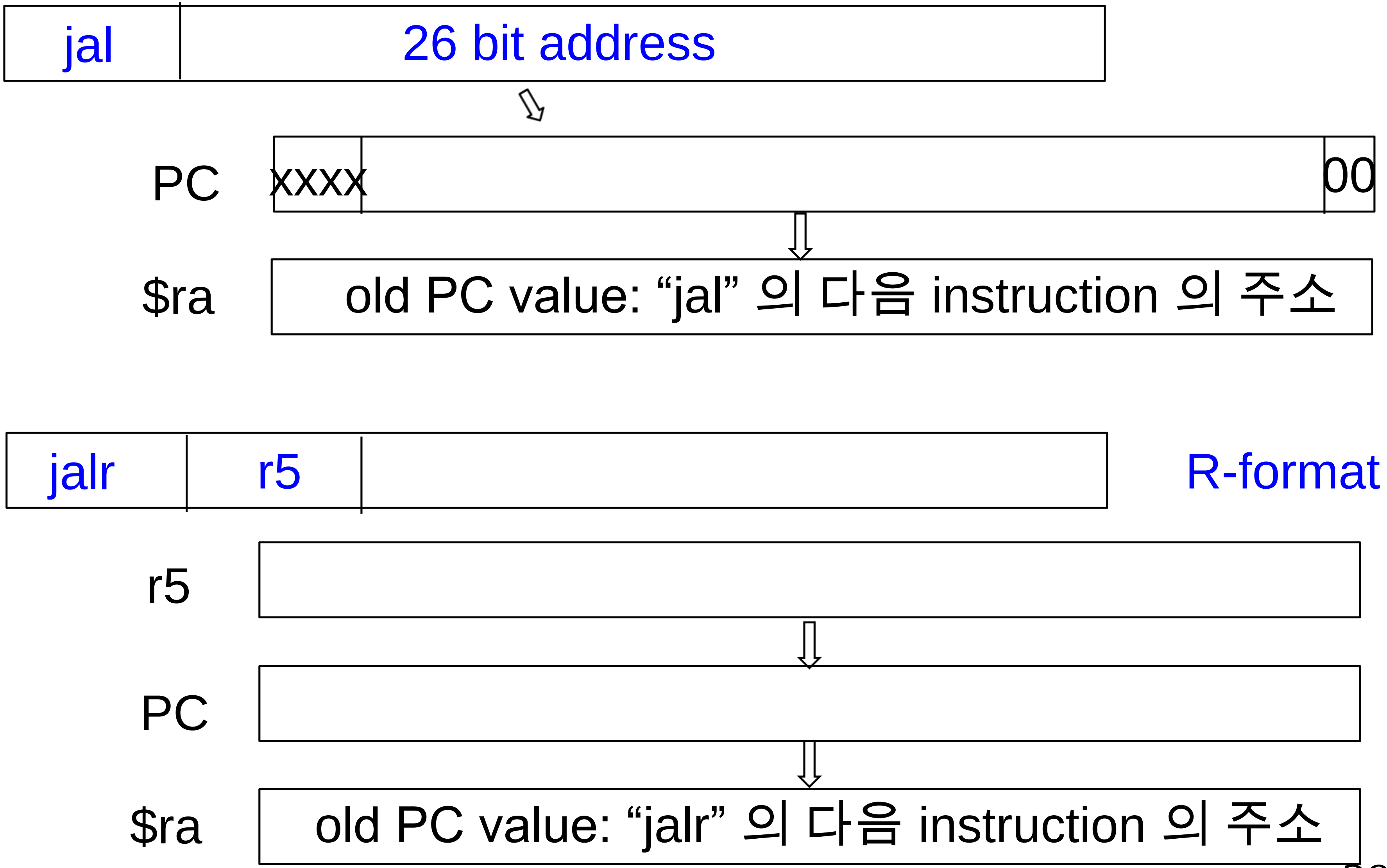
❑ Return from procedure call

- Compiler 는 누가 어디서 call 할 지 모름
- Caller 는 return address 를 r31에 저장
- Return from call: “**jr** r31”

Procedural Call Instructions

- ❑ Unconditional jump instructions
  - **j** (jump)
  - **jr** (jump register)
- ❑ Procedure call: unconditional jump plus return address
  - **jal** (jump and link)
  - **jalr** (jump and link register)
    - – Link: save return address at known location ($ra)
  - Why not use two instructions?

“jal” and “jalr”



jal | 26 bit address

PC | xxxx | 00

$ra | old PC value: “jal” 의 다음 instruction 의 주소

jalr | r5 | R-format

r5

PC

$ra | old PC value: “jalr” 의 다음 instruction 의 주소

Branch Instructions

- ❑ Instructions

```
bne $t4,$t5,16-bit-offset

beq $t4,$t5,16-bit-offset

j 26-bit-address

jal 26-bit-address

jr $s4

jalr $s4
```

- ❑ Format

<table>
<tr><td>R</td><td>op</td><td>rs</td><td>rt</td><td>rd</td><td>shamt</td><td>funct</td></tr>
<tr><td>I</td><td>op</td><td>rs</td><td>rt</td><td colspan="3">16 bit offset</td></tr>
<tr><td>J</td><td>op</td><td colspan="5">26 bit address</td></tr>
</table>

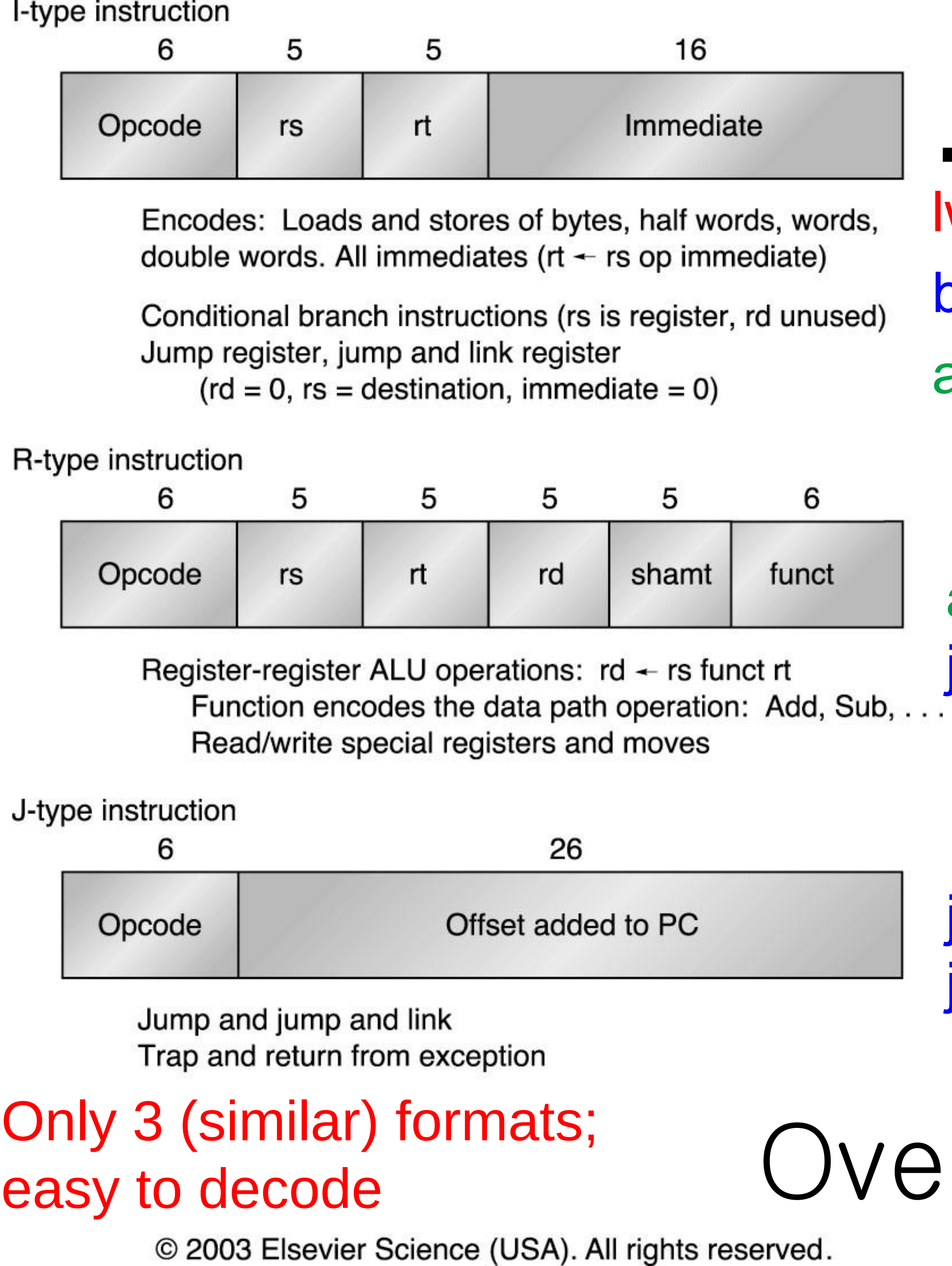
Chapter 2

Part 2 (Control Instructions):

Summary

Overview of MIPS

(반복)



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)

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

J-type instruction

6	26
Opcode	Offset added to PC

Jump and jump and link
Trap and return from exception

lw/sw (Base addr. mode)

beq (PC-relative mode)

addi (Immediate mode)

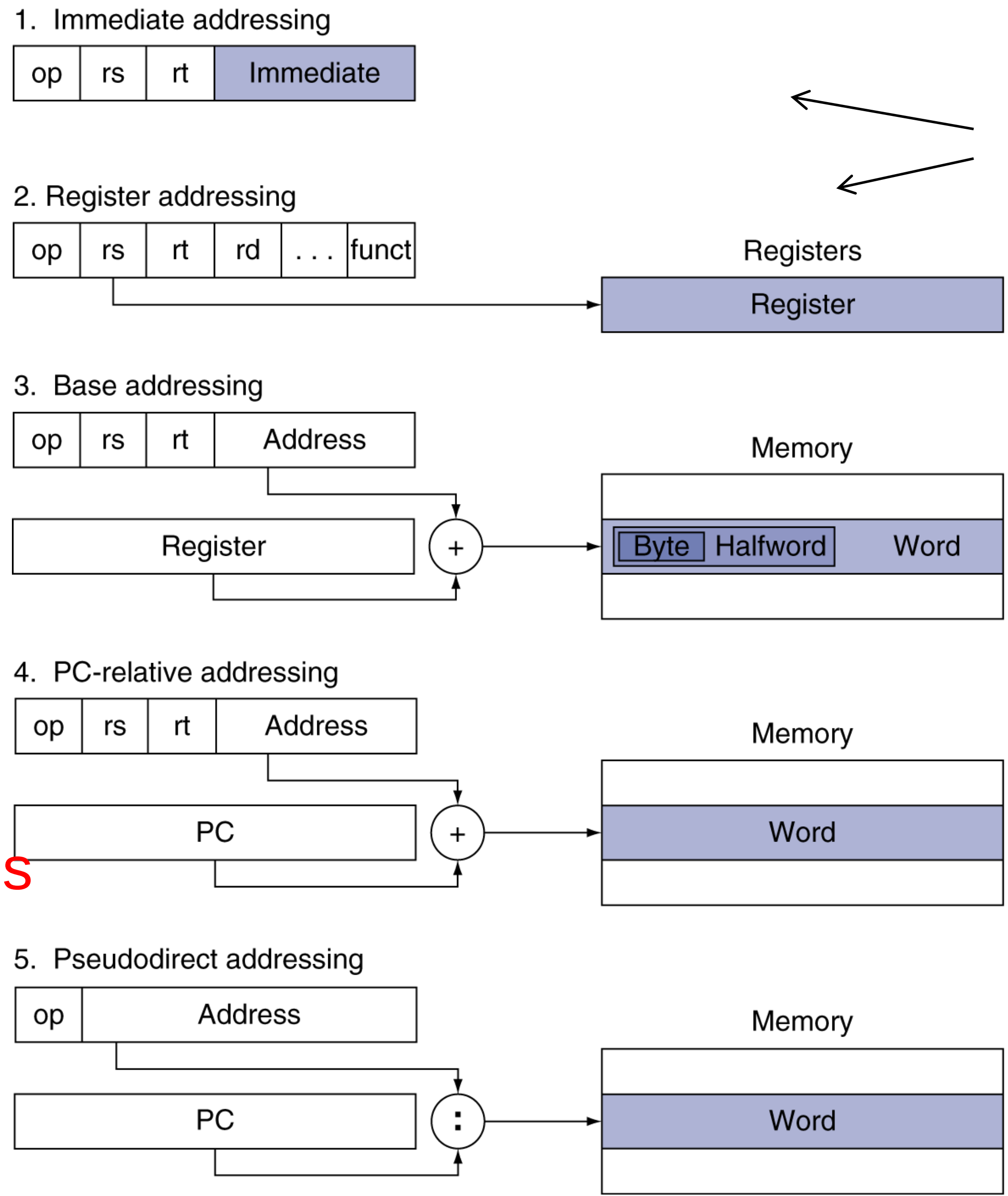
add (Register addr. mode)

jr, jalr

j (Pseudo-direct mode), jal

Only 3 (similar) formats; easy to decode

Addressing Mode Summary (반복)



ALU
(data manipulation)

Load, store

Branch, jump

Only 5 modes;

common operations

Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - Much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., “move $t0, $t1” exists only in Assembly
  - Would be implemented using “add $t0,$t1,$zero”

Assembler Pseudoinstructions

- □ Most assembler instructions represent machine instructions one-to-one
- □ Pseudoinstructions: figments of assembler's imagination

```
move  $t0, $t1      →   add  $t0,  $zero,  $t1

blt  $t0,  $t1,  4    →  slt   $at,  $t0,  $t1
                         bne  $at,  $zero, 4
```

- $at (register 1): assembler temporary

- □ When considering performance, you count real instructions

Overview of MIPS

- Simple instructions all 32 bits wide
- Very structured, no unnecessary baggage
- Only three instruction formats

R	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**
I	**op**	**rs**	**rt**	**16 bit offset**		
J	**op**	**26 bit address**				

- Rely on compiler to achieve performance
  - What are the compiler's goals?
    - IC ↓ (also CPI ↓)

ISA 감상: 생각의 초점 (반복)

- ❑ RISC ISA 는 어떻게 생겼나? 왜 그렇게 생겼나?
  - Commonly-used (i.e., simple) operations 지원
  - 자주 나오는 것을 single machine instruction 으로
    - 각 instruction format 및 addressing mode 의 필요성
  - ISA: collection of many SW-HW interactions
- ❑ RISC ISA 는 program execution 을 어떻게 지원하나
  - Statement 들을 어떻게 지원하나?
  - Function call and return 을 어떻게 지원하나? (Topic 2-3)

Homework #8 (see Class Homepage)

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

** 문장으로 써도 좋고 파워포인트 형태의 개조식 정리도 좋음

2) Solve Chapter 2 exercises 1, 2, 3, 4, 10, 11, 12, 18, 20, 24, 26, 39, 40, 41, 42, 47

** 문제의 수가 많다고 놀라지 마세요. 대부분 간단합니다. 그리고 일부만 풀어서 Homework #8 로 제출하고, 나머지 문제들은 다음 주에 Homework #9 와 함께 제출해도 됩니다.

❑ 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)