Chapter 2

Instructions: Language of the Computer

Part 2 (Control Instructions):

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

Control

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:

```
bne  $t0, $t1, Label
beq  $t0, $t1, Label
```

- Example: if (i==j) h = i + j;

```
        bne  $s0, $s1, Label
        add  $s3, $s0, $s1
Label: ....
```

Conditional Branch

- How to specify destination address?
  - Compiler has 32-bit destination address
- PC-relative addressing mode
  - Destination = PC + displacement (or offset)
    - Usually, destination is near current instruction
    - Pack offset bits in single instruction
  - **beq** r1, r2, offset; **bne** r1, r2, offset
    - Why not "**beq** r1, r2, r3"? // make common case fast
  - Position independence
    - Eliminate work in linking

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Control

- MIPS unconditional branch instructions:

```
j  label
```

- Example:

```
if (i != j)                       beq $s4, $s5, Lab1
    h = i + j;                    add $s3, $s4, $s5
else                              j Lab2
    h = i - j;             Lab1:  sub $s3, $s4, $s5
                           Lab2:  ...
```

- Why new instruction?
  – Procedure calls: often jump more than 16 –bit distance

Jump Addressing

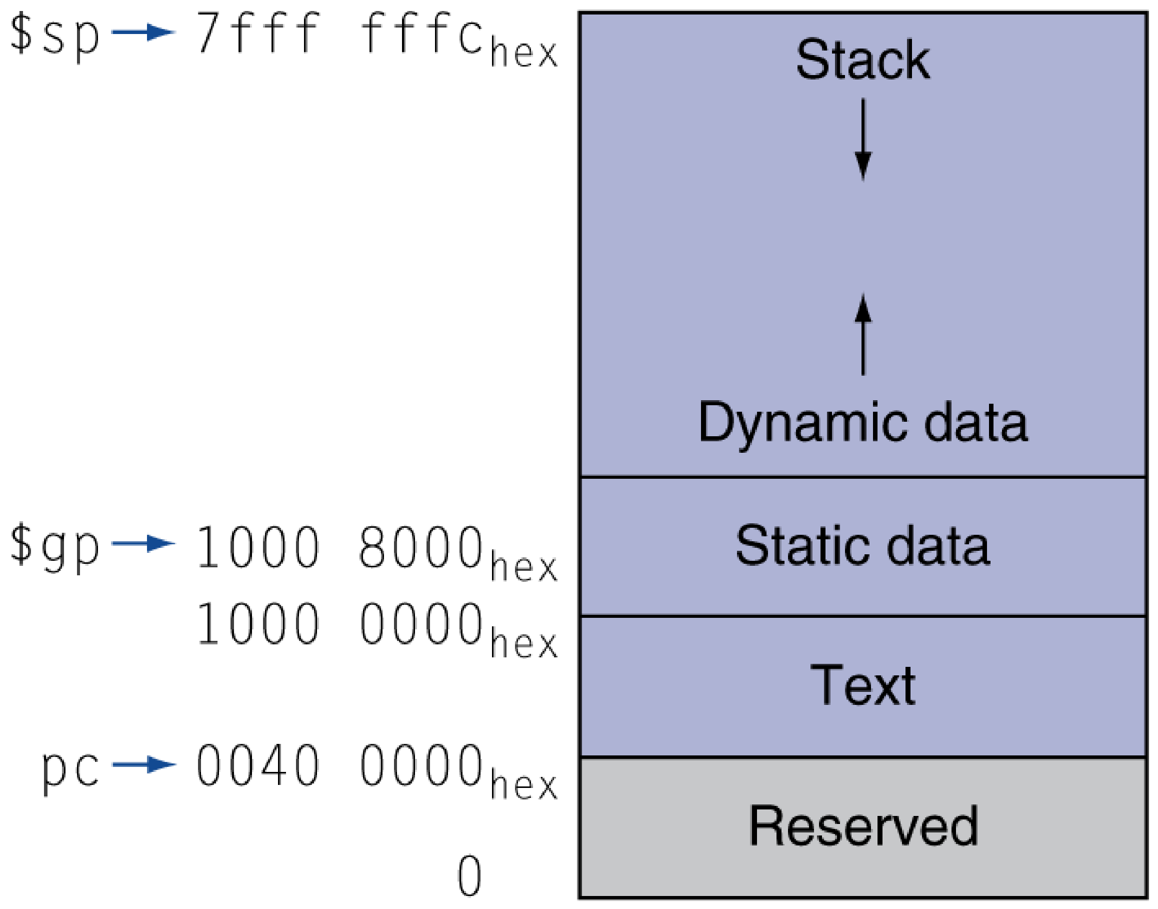
- Jump (**j** and **jal**) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits

- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address × 4)
  - Linker/loader: note the 256MB boundary

Memory layout (미리 보기)

- MIPS memory allocation for program and data
  - Static, automatic (runtime stack), dynamic (heap)



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`

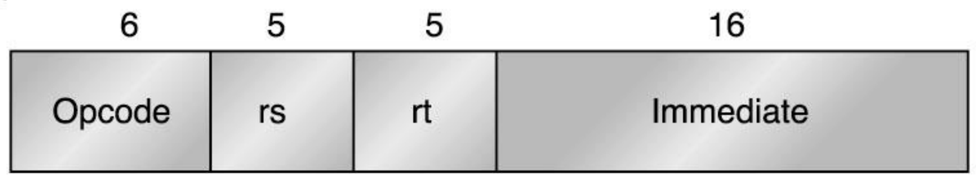
- Formats:

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

Overview of MIPS

(반복)

I-type instruction

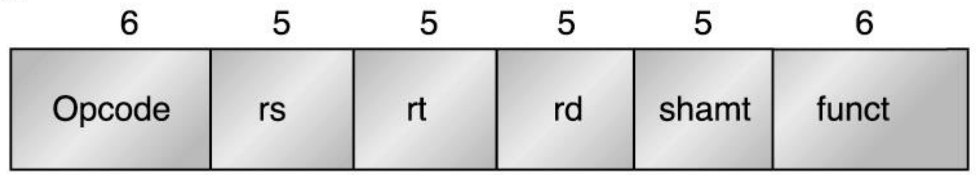


Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

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

lw/sw (Displacement mode)
beq (PC-relative mode)
addi (Immediate mode)

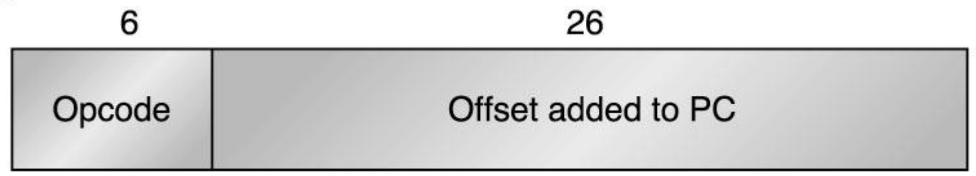
R-type instruction



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

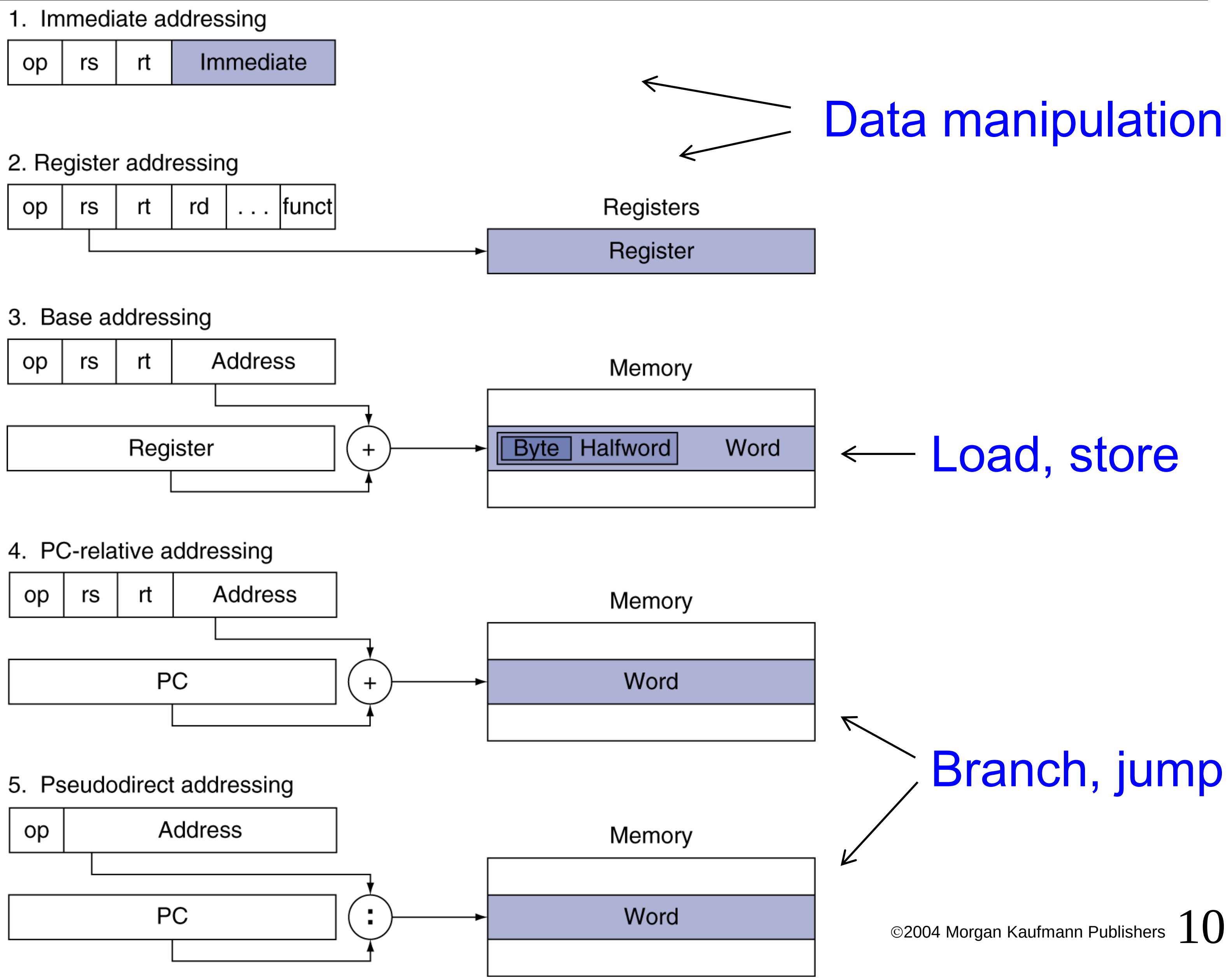
J-type instruction



Jump and jump and link
Trap and return from exception

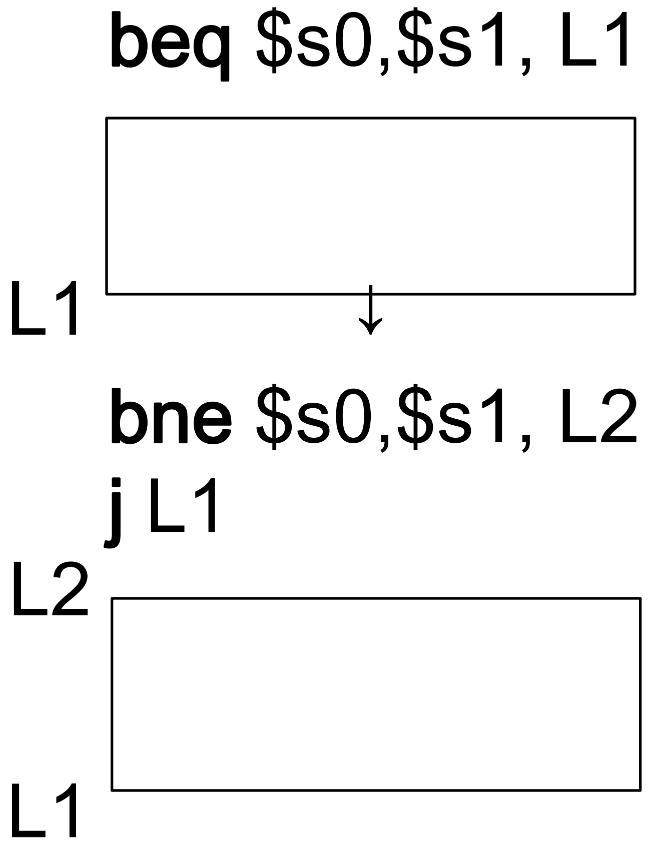
jump (Pseudo-direct mode)

Addressing Mode Summary (반복)



Quiz: Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example



```
      beq $s0,$s1, L1
L1
```

jump distance > 2^{15}

↓

```
      bne $s0,$s1, L2
      j L1
L2
L1
```

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

Target Addressing Example

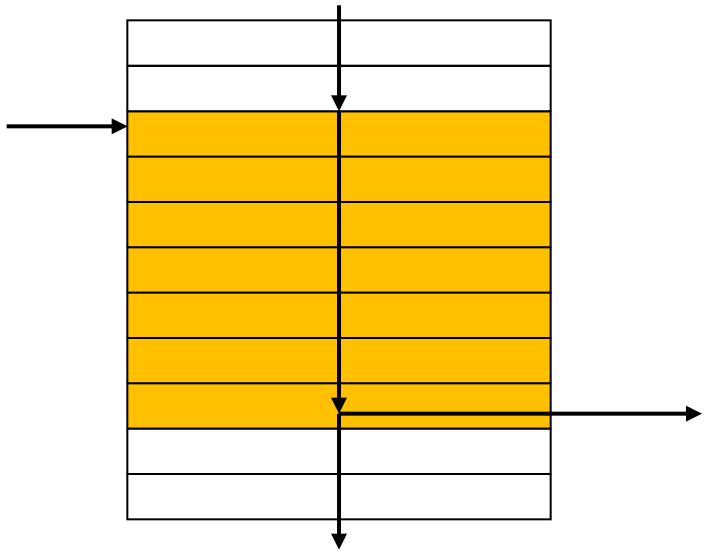
- Loop code from earlier example
  - Assume Loop at location 80000

```
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

Control Flow

- What about branch-if-less-then:

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

- Example:

```
                              slt $t0, $s4, $s5
      if (i < j)              beq $t0, $zero, Lab1
            h=i+j;            add $s3, $s4, $s5
      else                    j Lab2
            h=i-j;      Lab1: sub $s3, $s4, $s5
                        Lab2: ...
```

- Comparison instructions: **slt, slti, sgt, sge, sle**, ….
- Why not use "**blt** $s1, $s2, Label"? (*next slide*)

Branch Instruction Design

- Why not **blt, bge**, etc?
- Hardware for <, ≥, … slower than =, ≠
  - 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
  - Are you sure? Run benchmarks!

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- **slt** rd, rs, rt
  - if (rs < rt) rd = 1; else rd = 0;
- **slti** rt, rs, constant
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

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

Signed vs. Unsigned

- 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 $\Rightarrow$ $t0 = 1
  - **sltu** $t0, $s0, $s1 # unsigned
    - +4,294,967,295 > +1 $\Rightarrow$ $t0 = 0

More instructions

- jr (jump register)
- Procedure call and return
  - jal (jump and link)
  - jalr (jump and link register)

Additional Instruction Support

- What if target address unknown at compile time (runtime info.)
  - Case or switch statements
  - Virtual functions or methods in OOL like C++ or Java
  - Function pointers as arguments in C or C++
  - Dynamically shared libraries (or DLL)
    - † Runtime info., dynamic binding, jump table – use “jr”
  - Procedure return

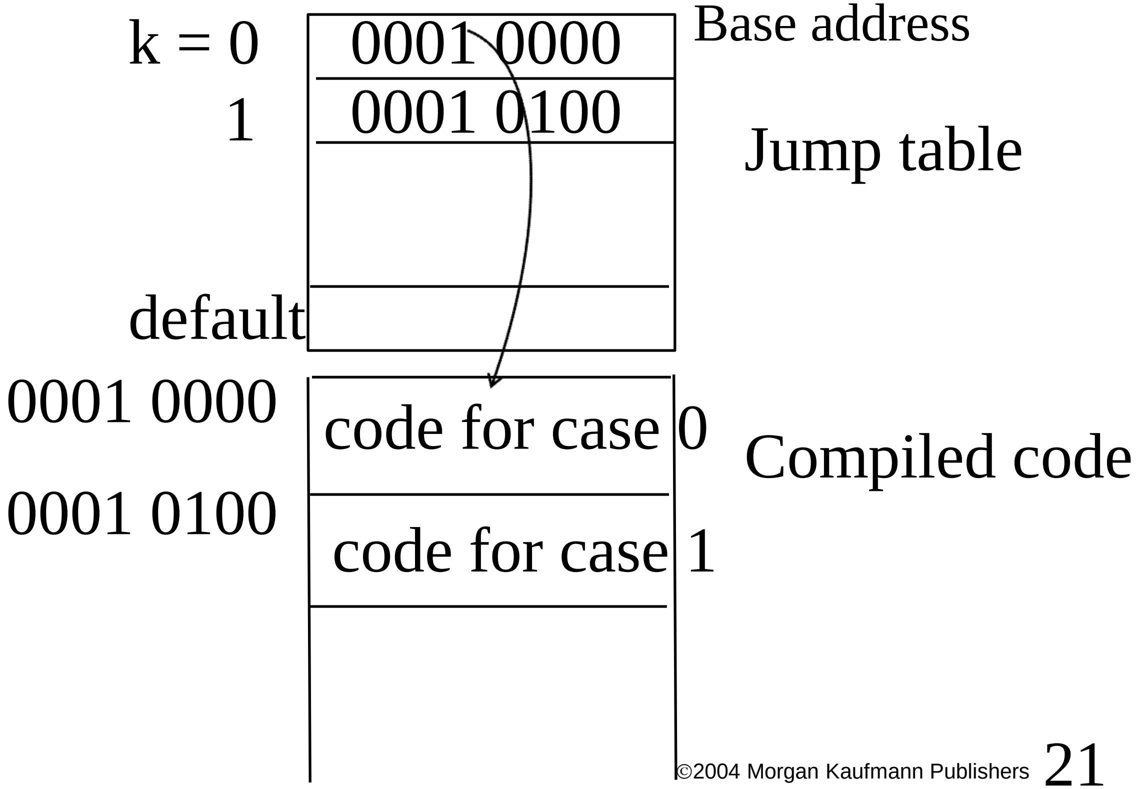
† Jump address is 32-bit

† Indirection is powerful method

Switch statement

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

```
$t0 = k * 4;
$t0 = $t0 + base address
                 of jump table
lw  $t1, 0($t0)
jr   $t1
```

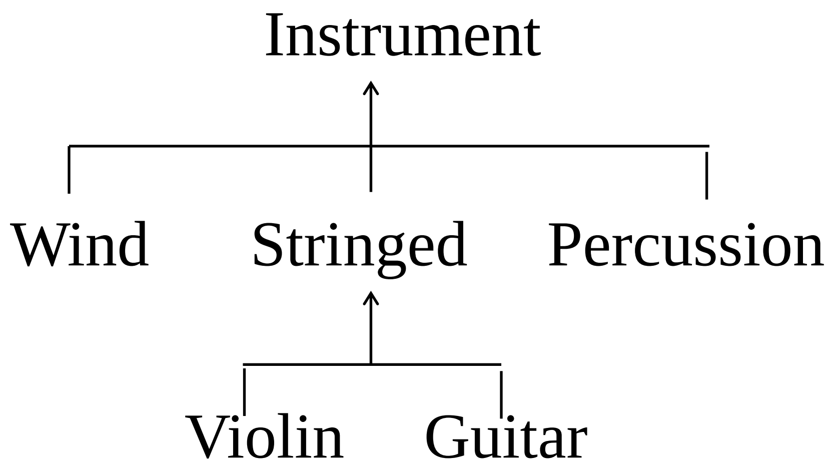


Jump Register (jr) Instruction in OOP

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

- Object-oriented programming (dynamic binding)

```
Instrument  p = new Violin();
p.printType();
```



* Container: ArrayList<Instrument>

* upcasting, is-a relationship

* dynamic binding

Additional Instruction Support (Part 3)

- Procedure call
  - "jump and link" and "jump and link register"
    - Save return address at known location (register)
  - Why not use two instructions?
- Return from procedure call
  - "jump register"

Control Flow

- **Example:**

```
if (i!=j)                 beq $s4, $s5, Lab1
        h=i+j;            add $s3, $s4, $s5
else                      j Lab2
        h=i-j;     Lab1:  sub $s3, $s4, $s5
                   Lab2:  ...
```

- **Instructions**

`bne $t4,$t5,Label`	**Next instruction is at Label if** `$t4 ° $t5`
`beq $t4,$t5,Label`	**Next instruction is at Label if** `$t4 = $t5`
`j Label`	**Next instruction is at Label**
`jal Label`	
`jr $s4`	

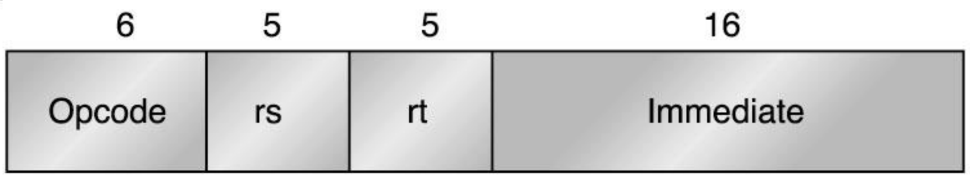
- **Format**

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

Overview of MIPS

(반복)

I-type instruction

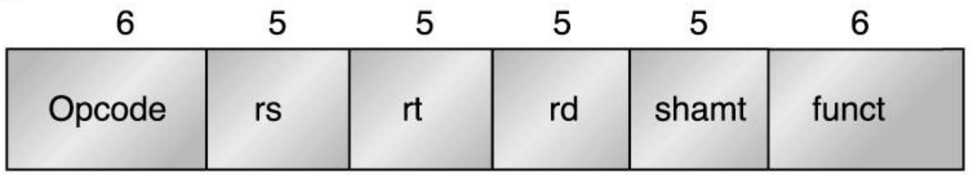


Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

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

lw/sw (Displacement mode)
beq (PC-relative mode)
addi (Immediate mode)

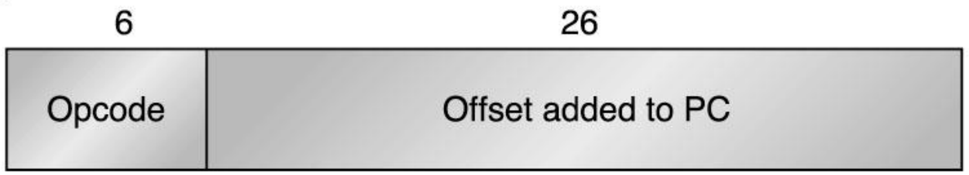
R-type instruction



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

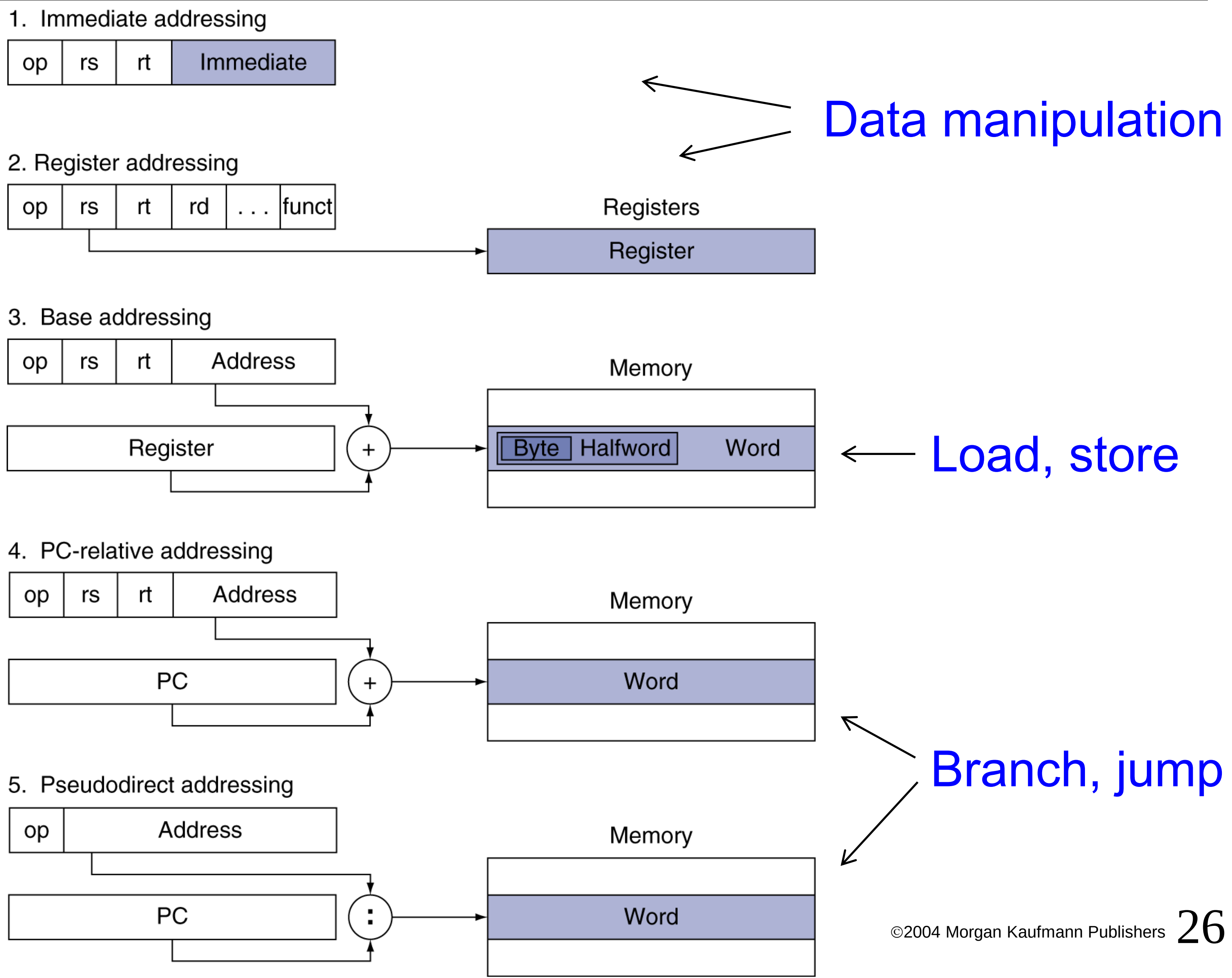
J-type instruction



Jump and jump and link
Trap and return from exception

jump (Pseudo-direct mode)

Addressing Mode Summary (반복)



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”
- When considering performance you should count real instructions

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler’s imagination

```
move $t0, $t1     →   add $t0, $zero, $t1
blt $t0, $t1, L   →   slt $at, $t0, $t1
                      bne $at, $zero, L
```

  – $at (register 1): assembler temporary

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 address**		
J	**op**	**26 bit address**				

- Rely on compiler to achieve performance

  — what are the compiler's goals?

To summarize:

MIPS operands

Name	**Example**	**Comments**
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Other Issues

- Part 3 of chapter 2:
  - Support for procedures
  - Linkers, loaders, memory layout
  - Stacks, frames, recursion
  - Dynamic linking, Java, program performance
- Part 4 of chapter 2
  - Alternative architectures: ARM, IA-32