CSE305 Computer Architecture

Performance

Daehoon Kim
Department of EECS, DGIST

Defining (Speed) Performance

- **To maximize performance, need to minimize execution time**

$$\text{performance}_X = 1 / \text{execution\_time}_X$$

If X is n times faster than Y, then

$$\frac{\text{performance}_X}{\text{performance}_Y} = \frac{\text{execution\_time}_Y}{\text{execution\_time}_X} = n$$

Relative Performance Example

- **If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?**

We know that A is n times faster than B if

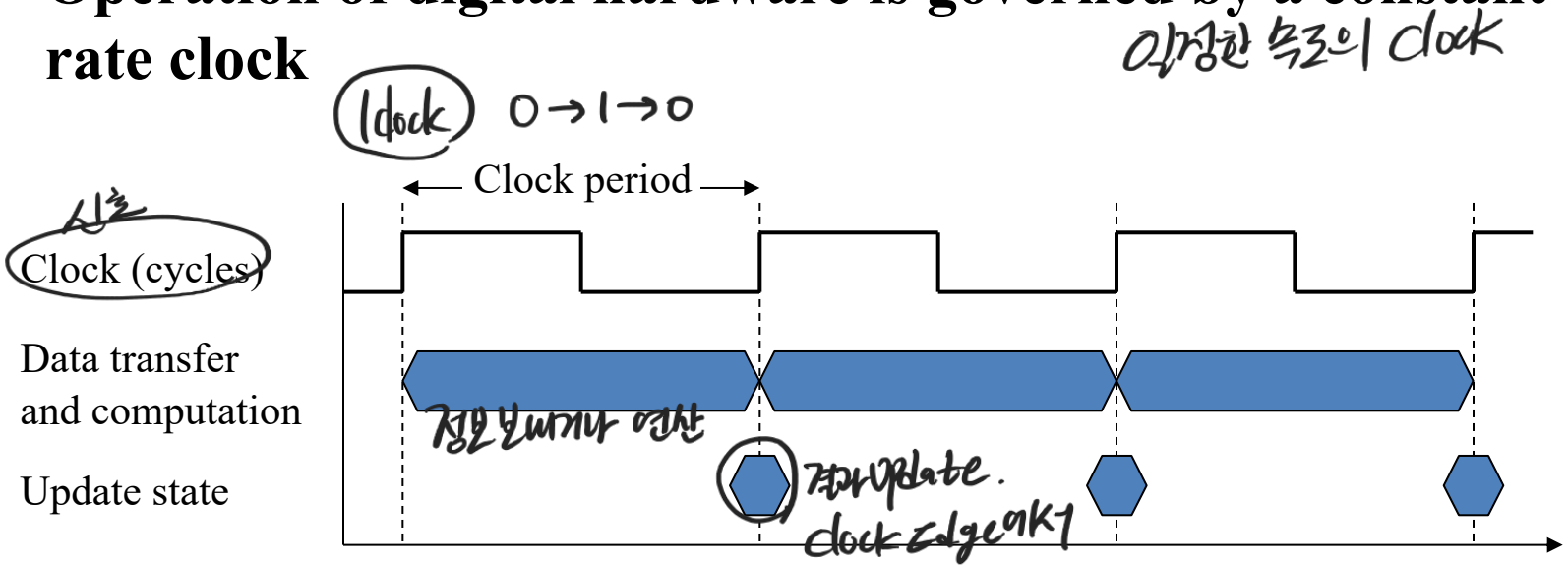
$$\frac{\text{performance}_A}{\text{performance}_B} = \frac{\text{execution\_time}_B}{\text{execution\_time}_A} = n$$

The performance ratio is $\frac{15}{10} = 1.5$

So A is 1.5 times faster than B (i.e., speedup is 1.5)

CPU Clocking

- **Operation of digital hardware is governed by a constant-rate clock**

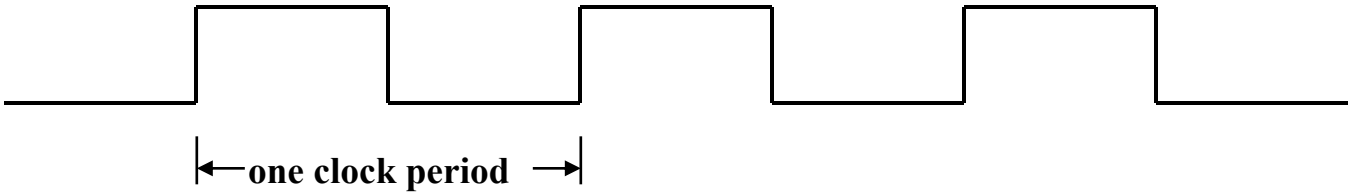


  - Clock frequency (rate): cycles per second
    - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz
  - Clock period (clock cycle time): duration of a clock cycle
    - e.g., 250ps = 0.25ns = 250×10^{-12}s

Machine Clock Rate

- **Clock rate (clock cycles per second in MHz or GHz) is inverse of clock cycle time (clock period)**

$$CC = 1 / CR$$ 역수관계



10 nsec clock cycle => 100 MHz clock rate

5 nsec clock cycle => 200 MHz clock rate

2 nsec clock cycle => 500 MHz clock rate

1 nsec (10^{-9}) clock cycle => 1 GHz (10^{9}) clock rate

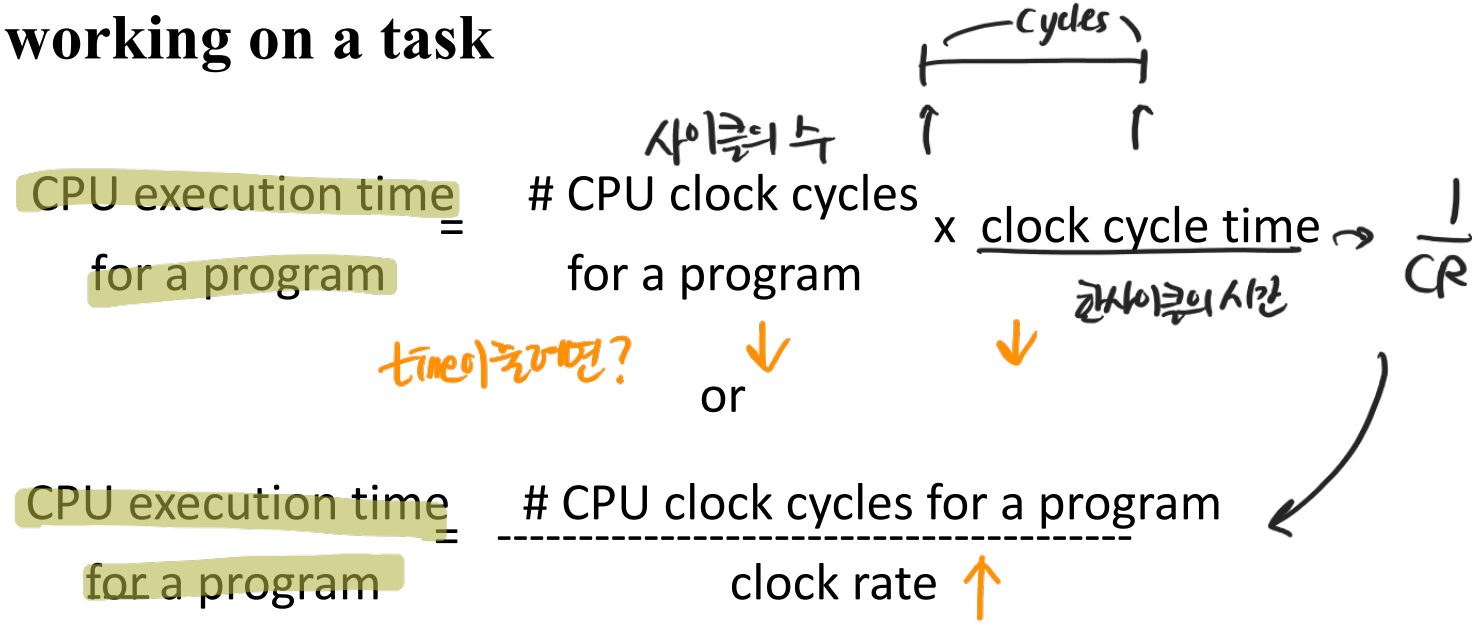
500 psec clock cycle => 2 GHz clock rate

250 psec clock cycle => 4 GHz clock rate

200 psec clock cycle => 5 GHz clock rate

Performance Factors

- **CPU execution time (CPU time) : time the CPU spends working on a task**



- **How to improve performance?**
  - **Reducing either *the length of the clock cycle (or increasing clock rate) or the number of clock cycles***
  - **Hardware designer must often trade off clock rate against cycle count**

clock rate VS cycle count
trade off 관계다.

Improving Performance Example

- **A program runs on computer A with a 2 GHz clock in 10 seconds. What clock rate must computer B run at to run this program in 6 seconds? Unfortunately, to accomplish this, computer B will require 1.2 times as many clock cycles as computer A to run the program.**

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{clock rate}_A}$$

$$\text{CPU clock cycles}_A = 10 \text{ sec} \times 2 \times 10^9 \text{ cycles/sec} = 20 \times 10^9 \text{ cycles}$$

$$\text{CPU time}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{\text{clock rate}_B}$$

$$\text{clock rate}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = 4 \text{ GHz}$$

Clock Cycles per Instruction

- **Performance equation w/ the number of instructions?**
  - **The compiler generates the instructions to execute**
  - **Execution time must depend on the number of instructions in a program**

cycle period × execute time

$$\text{\# CPU clock cycles for a program} = \text{\# Instructions for a program} \times \text{Average clock cycles per instruction}$$

instruction 개수

instruction 당 clock cycle의 평균

- ***Clock cycles per instruction (CPI)*** **– the average number of clock cycles each instruction takes to execute**
  - **A way to compare two different implementations of the same ISA**

x86 - Arm

The Performance Equation

- **Our basic performance equation is then**

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{clock period}$$

or

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{clock rate}}$$

- **Calculate avg. CPI?**
  - **Can measure the CPU execution time by running the program**
  - **The clock rate is usually given**
  - **Can measure overall instruction count by using profilers or simulators without knowing all of the implementation details**
  - **Modern processors offer HW performance counter measuring CPI or IPC**
- **CPI varies by instruction type and ISA implementation**

Using the Performance Equation

- **Computers A and B implement the same ISA. Computer A has a clock cycle time of 250 ps and an average CPI of 2.0 for a program and computer B has a clock cycle time of 500 ps and an average CPI of 1.2 for the same program. Which computer is faster and by how much?**

Each computer executes the same number of instructions, *I*, so

$$\text{CPU time}_A = I \times 2.0 \times 250 \text{ ps} = 500 \times I \text{ ps}$$

$$\text{CPU time}_B = I \times 1.2 \times 500 \text{ ps} = 600 \times I \text{ ps}$$

Clearly, A is faster … by the ratio of execution times

$$\frac{\text{performance}_A}{\text{performance}_B} = \frac{\text{execution\_time}_B}{\text{execution\_time}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

Effective CPI

- **Computing the overall effective CPI is done by looking at the different types of instructions and their individual cycle counts and averaging**

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (CPI_i \times IC_i)$$

  - **Where IC_i is the count (percentage) of the number of instructions of class i executed**
  - **CPI_i is the (average) number of clock cycles per instruction for that instruction class**
  - **n is the number of instruction classes**

- **The overall effective CPI varies by instruction mix**

A Simple Example

Op	Freq	CPI_i	Freq x CPI_i
ALU	50%	1	.
Load	20%	5	
Store	10%	3	
Branch	20%	2	
			Σ =

- **How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?**
- **How does this compare with using branch prediction to shave a cycle off the branch time?**
- **What if two ALU instructions could be executed at once?**

A Simple Example

Op	Freq	CPI_i	Freq x CPI_i	A	B	C
ALU	50%	1	.5	.5	.5	.25
Load	20%	5	1.0	.4	1.0	1.0
Store	10%	3	.3	.3	.3	.3
Branch	20%	2	.4	.4	.2	.4
			$\sum$ = 2.2	1.6	2.0	1.95

A. **How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?**

CPU time new = 1.6 x IC x CC so 2.2/1.6 means 37.5% faster

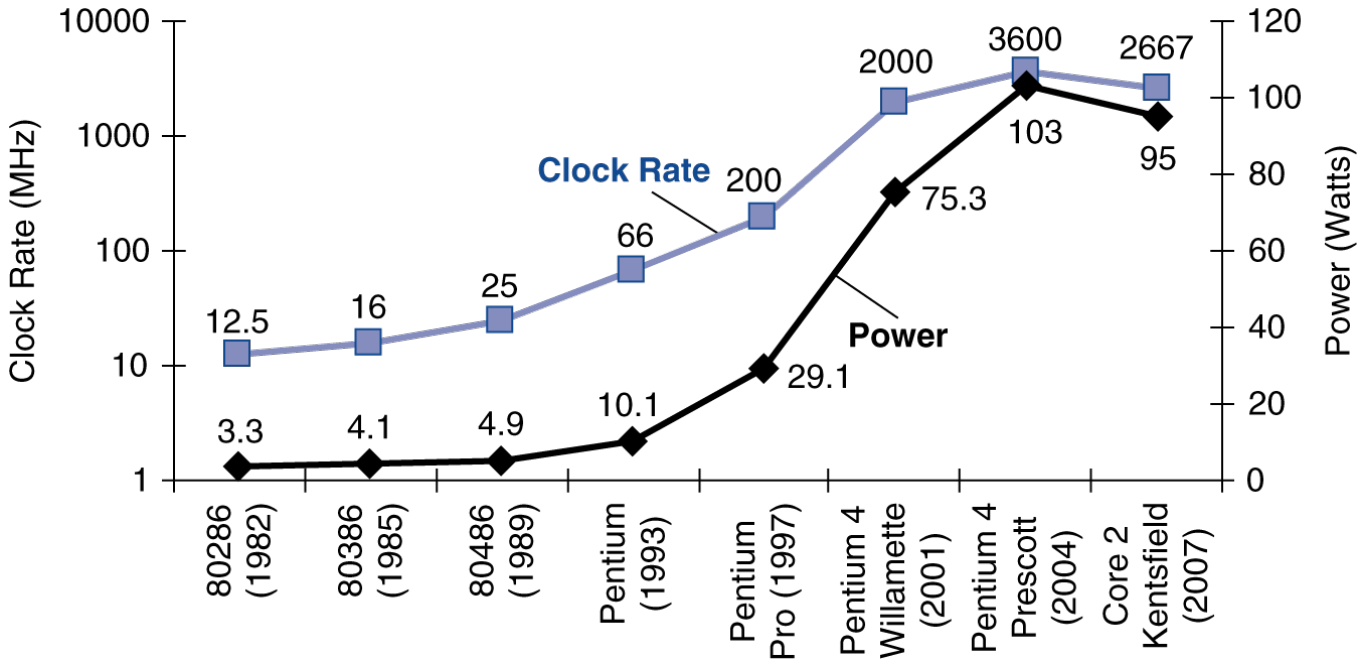
B. **How does this compare with using branch prediction to shave a cycle off the branch time?**

CPU time new = 2.0 x IC x CC so 2.2/2.0 means 10% faster

C. **What if two ALU instructions could be executed at once?**

CPU time new = 1.95 x IC x CC so 2.2/1.95 means 12.8% faster

Power Trends



Heat power budget 전력

- **In CMOS IC technology**

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

clock 높이면 V↑

clock rate

Reducing Power

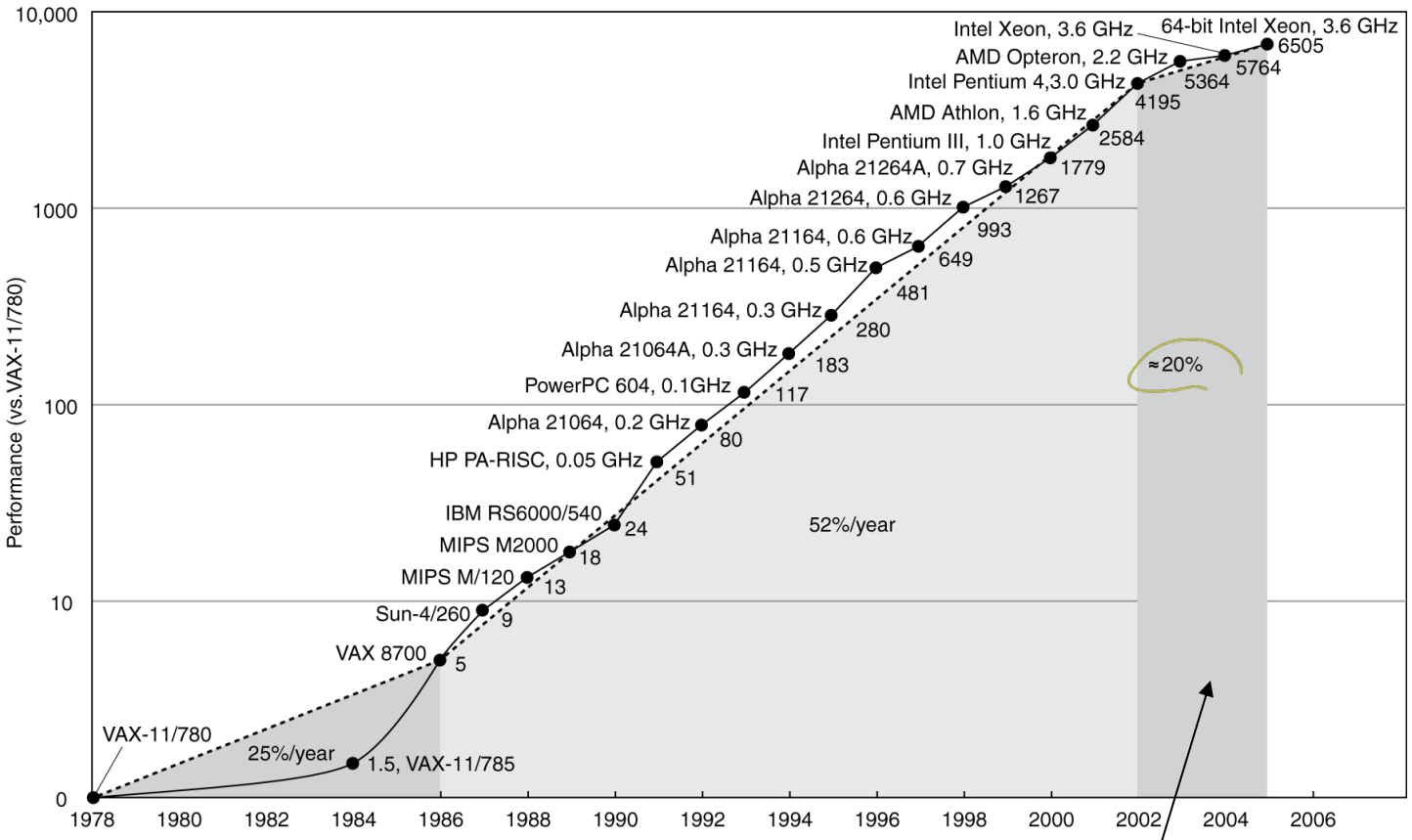
- **Suppose a new CPU has**
  - **15% capacitive load, 15% voltage, and 15% frequency reduction**

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}^{\ 2} \times F_{old}} = 0.85^4 = 0.52$$

- **How to overcome the power wall**
  - **Reduce frequency -> reduce performance also**
  - **Reduce voltage further -> leading to more leakage power**

- **How else can we improve performance?**

Uniprocessor Performance



Constrained by power, instruction-level parallelism, memory latency

Multi-core Processors

- **Multi-core processors**
  - **More than one core per chip**
- **Requires explicitly parallel programming**
  - **Compare with instruction level parallelism**
    - **Hardware executes multiple instructions at once**
    - **Hidden from the programmer**
- **Hard to do**
  - **Programming for performance**
  - **Load balancing**
  - **Optimizing communication and synchronization**

Amdahl's Law

- **Improving an aspect of a computer and expecting a proportional improvement in overall performance**

$$T_{improved} = \frac{T_{affected}}{improvement\ factor} + T_{unaffected}$$

- **Example: multiplication accounts for 80s/100s**
  - **How much improvement in multiplication performance to get 5× overall?**

$$20 = \frac{80}{n} + 20$$

- Can't be done!

- ***Make the common case fast!***

CSE305 Computer Architecture

Instruction Set Architecture I

Daehoon Kim
Department of EECS, DGIST

Instruction Set Architecture

- **Contract between programmer and the hardware**
  - **Defines operations of instructions**
  - **Defines visible state of the system**
  - **Defines how state changes in response to instructions**
- **Programmer: ISA is the model of how a program will execute**
- **Hardware Designer: ISA is the formal definition of the correct way to execute a program**
- **ISA specification**
  - **All architecturally visible states: registers**
  - **Instruction formats and behaviors**
  - **Memory model (paging, segmented memory etc)**
  - **I/O (input/output of disk, network), interrupts**

The MIPS Instruction Set

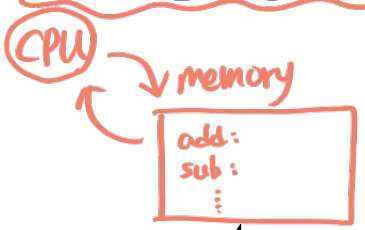
- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Two Key Principles of Machine Design

1. **Instructions are represented as binary numbers (machine code) (e.g., add r1, r2, r3 -> 01000110..)**

보통다 ISA이다.

2. **Programs (i.e., machine code) are stored in memory to be read or written, just like data – *stored-program concept (e.g., Von Neumann Architecture)***



- ***Binary compatibility issue:*** **computers cannot execute binary files for different ISA – leads industry to align around a small number of ISAs**

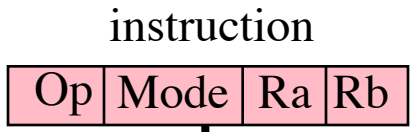
호환성

Memory

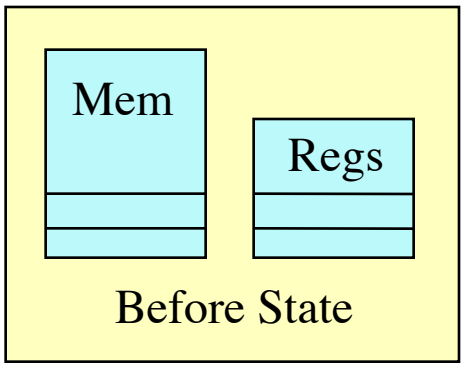
Accounting program (machine code)
C compiler (machine code)
Payroll data
Source code in C for accounting program

ISA Basics

Instruction formats
Instruction types
Addressing modes

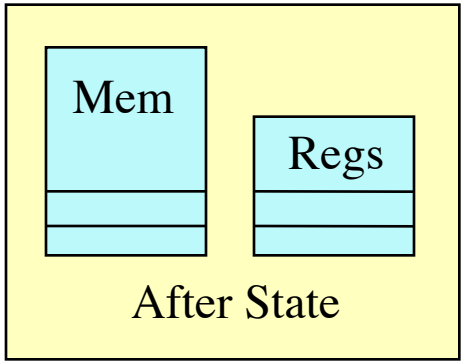


machine



머신상태바뀜.

Data types
Operations
Interrupts/Events

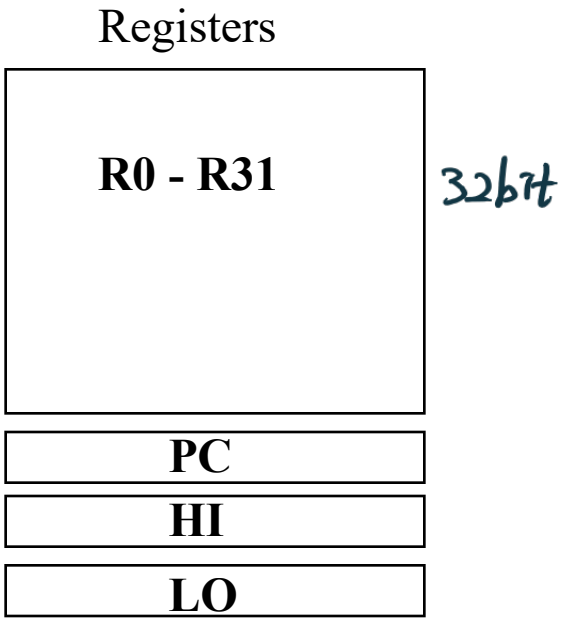


Machine state
Memory organization
Register organization

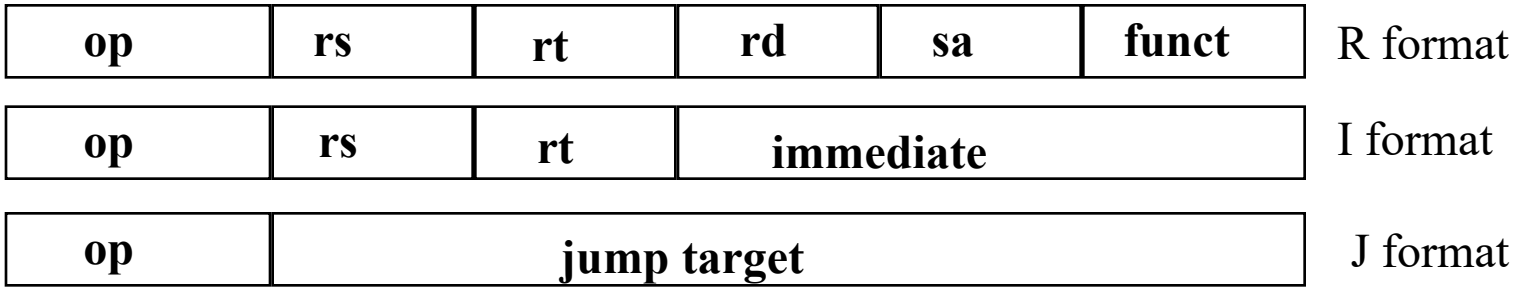
Memory가 변하거나 레지스터가 변하거나

MIPS-32 ISA

- **Instruction Categories**
  - **Computational** 계산
  - **Load/Store** 메모리값 읽고쓰기
  - **Jump and Branch** if, else, elif
  - **Etc.**

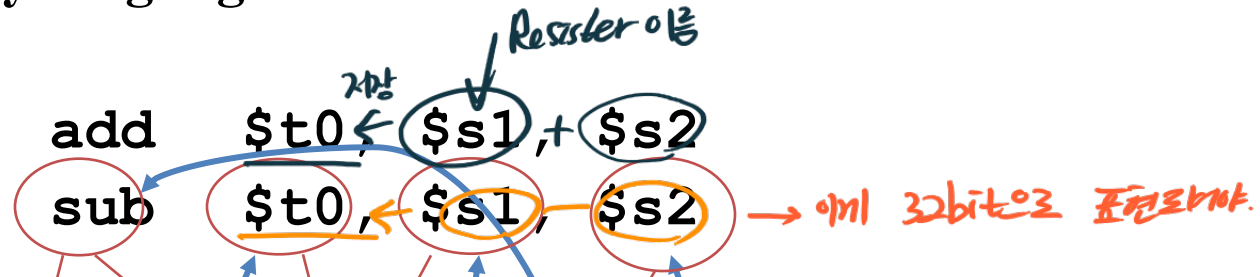


3 Instruction Formats: all 32 bits wide



MIPS Arithmetic Instructions

- **MIPS assembly language arithmetic statement**



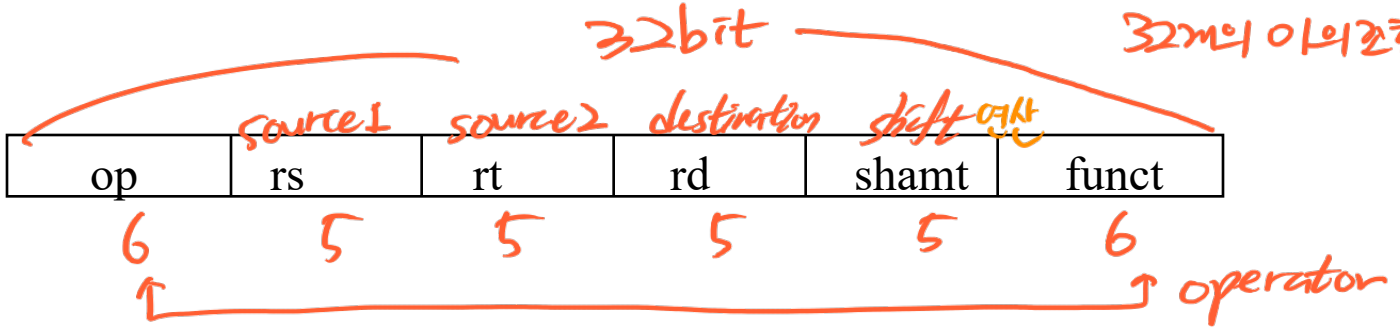
- Each arithmetic instruction performs one operation
- Each specifies exactly three operands that are all contained in the register file ($t0,$s1,$s2)

destination ← source1 op source2

- Instruction Format (R format)

MIPS Instruction Fields

- **MIPS fields are given names to make them easier to refer to**



규약

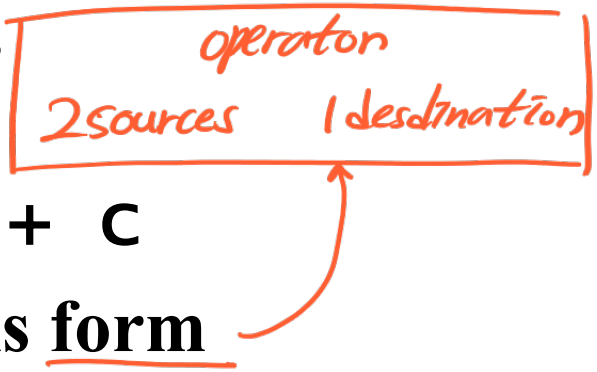
op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result's destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

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 favors regularity**
  - **Regularity makes implementation simpler**
  - **Simplicity enables higher performance at lower cost**



Arithmetic Example

- **C code:**

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

- **Compiled MIPS code:**

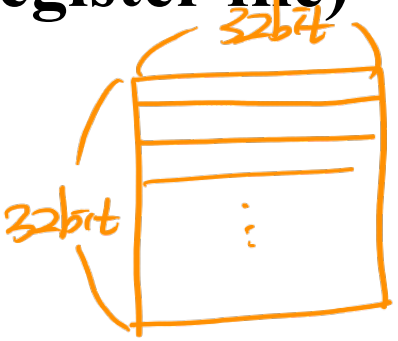
```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

Register Operands

- **Arithmetic instructions use register operands**

add r1 r2 r3

- **MIPS has a 32 × 32-bit register (i.e., register file)**
  - **Use for frequently accessed data**
  - **Numbered 0 to 31** → 32
  - ***32-bit data called a "word"*** **(1 byte = 8 bits, thus 4 bytes)**

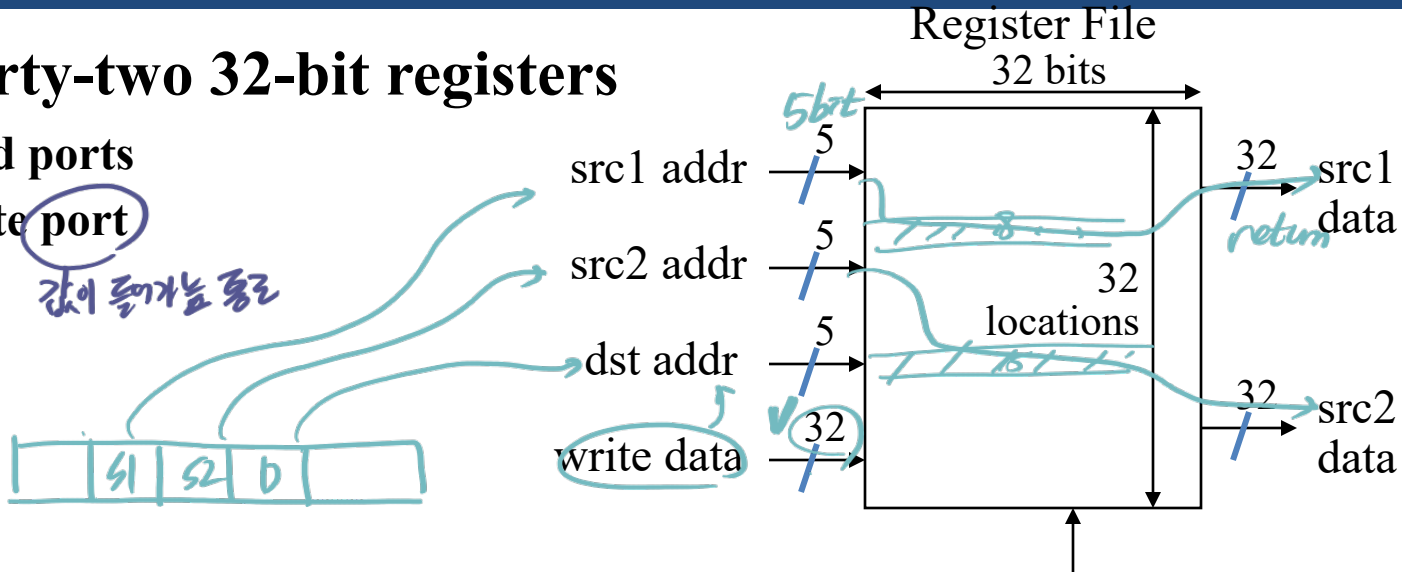


- **Register names**
  - **$t0, $t1, …, $t9 for temporary values**
  - **$s0, $s1, …, $s7 for saved variables**
- ***Design Principle 2:*** **Smaller is faster**
  - **Limit of 32 registers**
  - **Register files with more locations are slower**

↳ access latency ↑

MIPS Register (i.e., Register File)

- **Holds thirty-two 32-bit registers**
  - **Two read ports**
  - **One write port**



- **Registers are**
  - **Faster than other memories (e.g., cache, memory)**
  - **Easier for a compiler to use**
    - **e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack**
  - **Can hold variables so that**
    - **Code density improves (since register are named with fewer bits than a memory location)**

Register Operand Example

- **C code:**

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

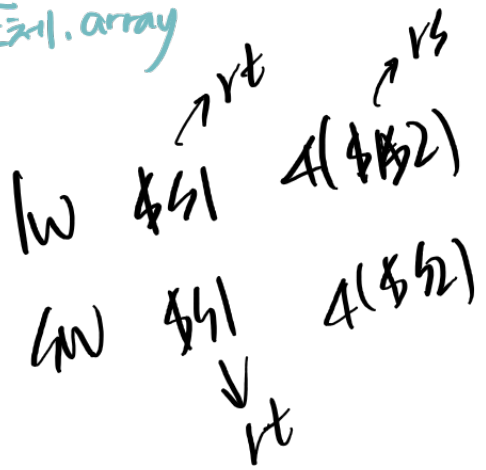
  - **f, …, j in $s0, …, $s4**

- **Compiled MIPS code:**

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

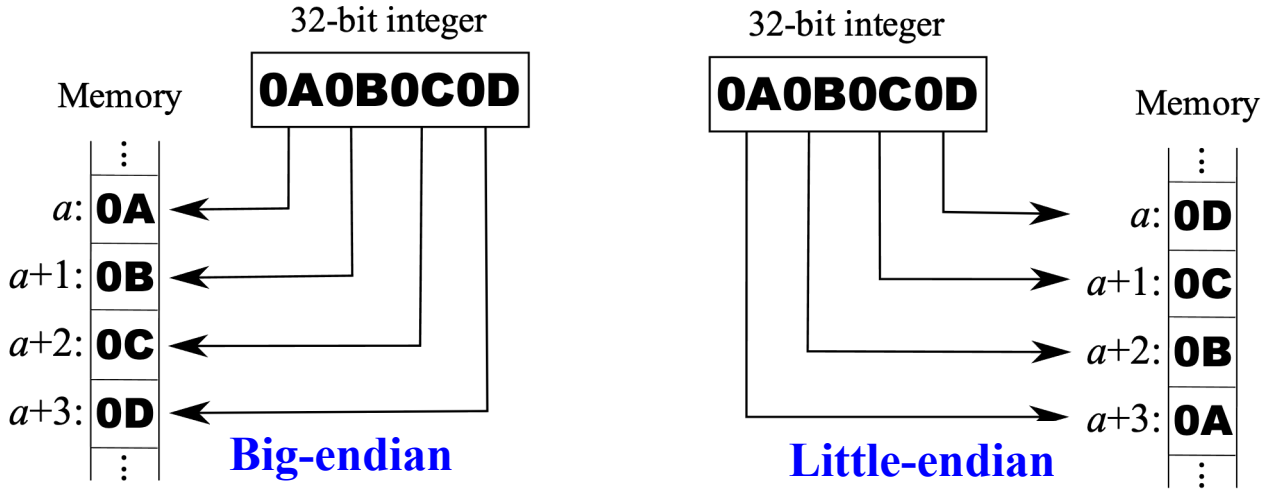
Memory Operands

- **Main memory used for composite data**
  - **Arrays, structures, etc.**
- **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**



Byte Addresses

- Most architectures address individual bytes in memory
  - Alignment restriction - the memory address of a word must be on natural word boundaries (a multiple of 4 in MIPS-32)
- Big Endian: MSB (most significant byte) in the lowest address
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
  - Good for comparison
- Little Endian: LSB (least significant byte)in the lowest address
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)
  - Good for calculation

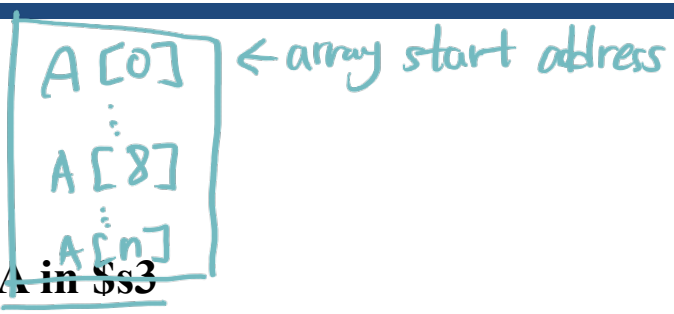


Memory Operand Example 1

- **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 since the size of each element of the array is 4 bytes

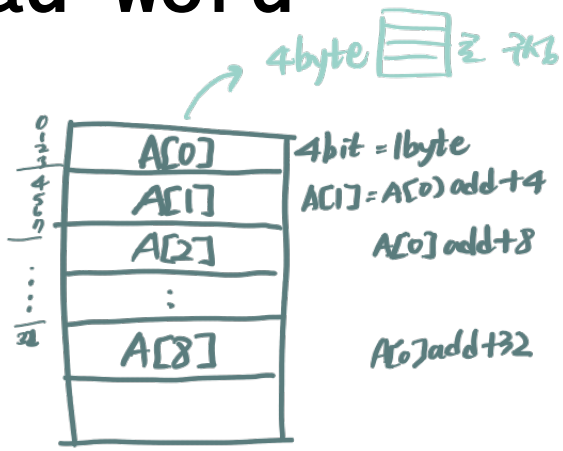
Load word

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

offset

base register

base address of A



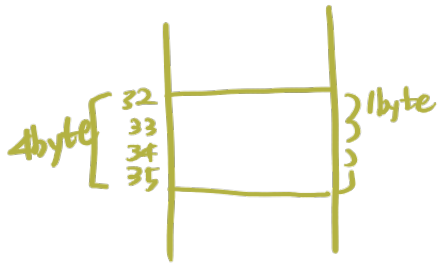
Memory Operand Example 2

- 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



base address 가 들어있다 가정

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

Store word

sw rt offset(rs)

rt: 저장

| op | rs | rt | offset |

lw rt offset(rs)

rt: data 불러서 rt에

load 에서만 rt가 write로 쓰인다.

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!

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*

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
```

move instruction ← add + $zero

Unsigned Binary Integers

- **Given an n-bit number**

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$ (n bit)
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- **Given an n-bit number**

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- **Bit 31 is sign bit (32nd bit)**
  - **1 for negative numbers**
  - **0 for non-negative numbers**
- **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 → 0, 0 → 1**

$$x + \bar{x} = 1111\ldots111_2 = -1$$

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

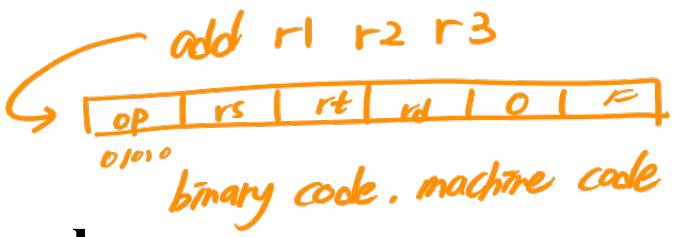
- **Example: negate +2**
  - **+2 = 0000 0000 … 0010_2**
  - **–2 = 1111 1111 … 1101_2 + 1**
    = 1111 1111 … 1110_2

Sign Extension

- Representing the same number using more bits
- In MIPS instruction set
  - e.g., `addi $s1, $s2, 100:` extend 16-bits "100" to 32-bits "100"
- 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

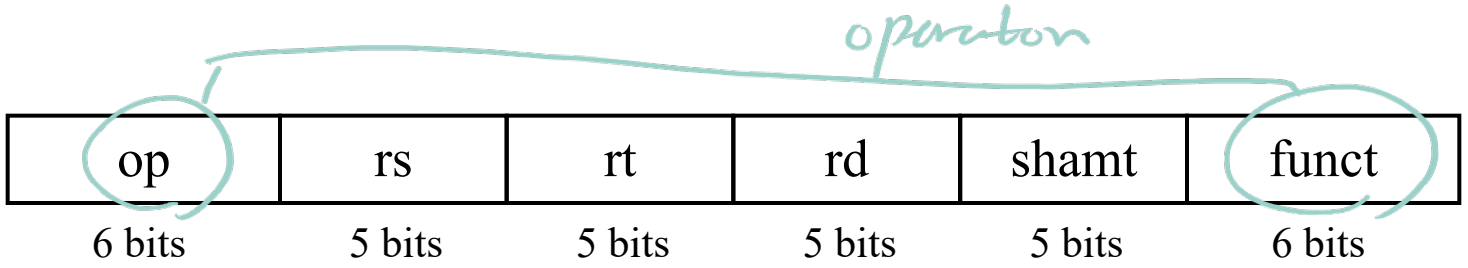
Representing Instructions

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



for decoding

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

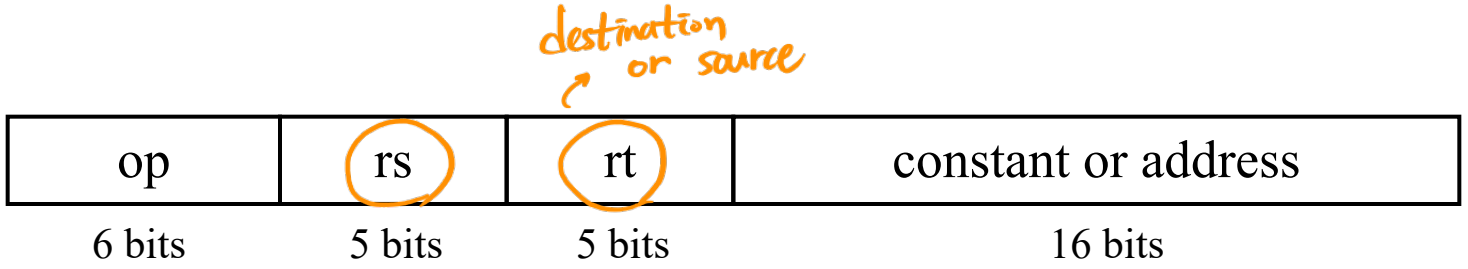
R-type은 이부분이 중요

encoding

encoding

$$00000010001100100100000000100000_2 = 02324020_{16}$$

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**

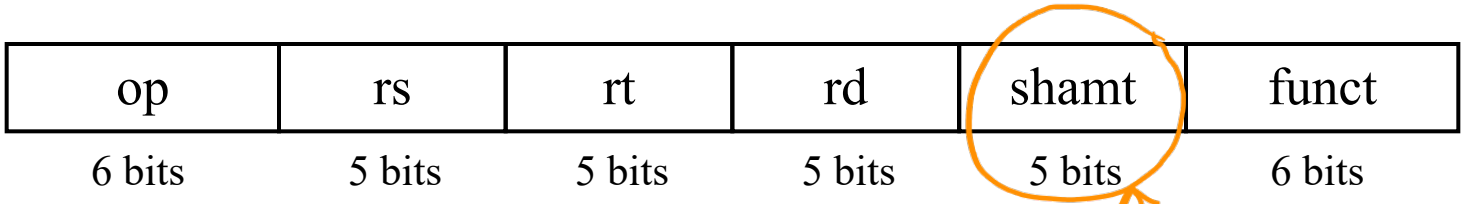
Logical Operations

- **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



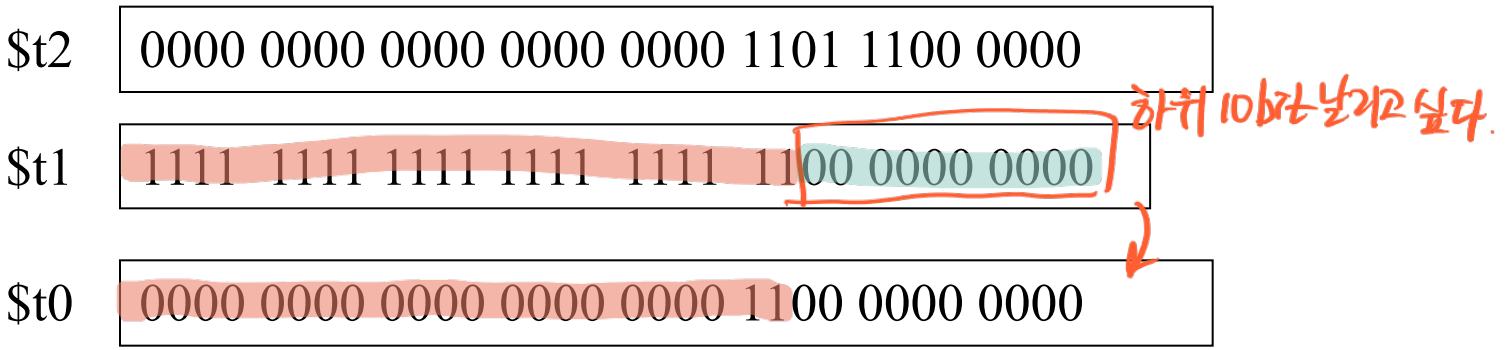
- **shamt: how many positions to shift**
- **Shift left logical**
  - **Shift left and fill with 0 bits**
  - **`sll` by *i* bits multiplies by 2^i**
- **Shift right logical**
  - **Shift right and fill with 0 bits**
  - **`srl` by *i* bits divides by 2^i (unsigned only)**

AND Operations

bit masking 할때 사용

- **Useful to mask bits in a word**
  - **Select some bits, clear others to 0**
  - **e.g, clear the last 10 bits**

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



하위 10비트 남기고 싶다.

OR Operations

- **Useful to include bits in a word**
  - **Set some bits to 1, leave others unchanged**
  - **e.g., set 11st~14th bit**

set → 1
clear → 0

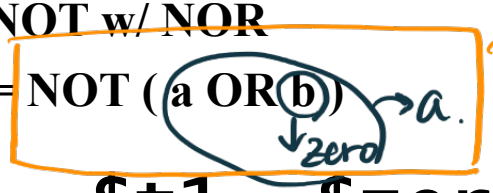
```
or $t0, $t1, $t2
```

$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

want to set

NOT Operations

- **Useful to invert bits in a word**
  - **Change 0 to 1, and 1 to 0**
- **MIPS already has NOR 3-operand instruction**
  - **Represent NOT w/ NOR**
  - **a NOR b == NOT (a OR b)**




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

Register 0: always read as zero

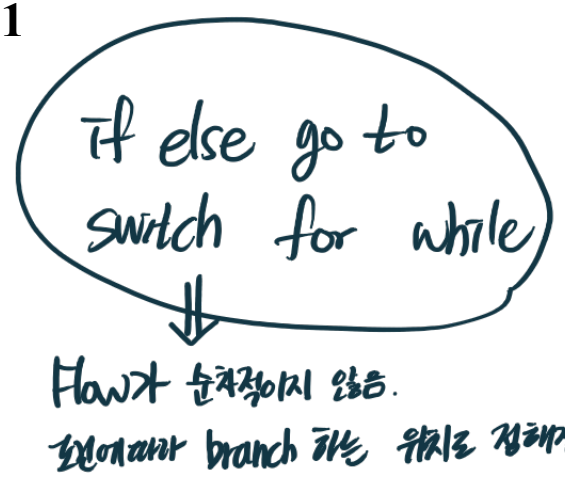
t1 → Not

reverse

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

Conditional Operations

- **Branch to a labeled instruction if a condition is true**
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

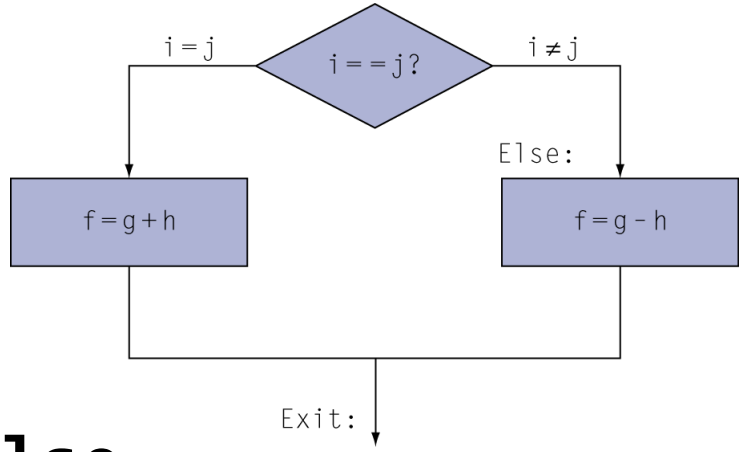


Compiling If Statements

- C code:

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

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j   Exit
Else:   sub $s0, $s1, $s2
Exit:   …
```

Assembler calculates addresses

Compiling Loop Statements

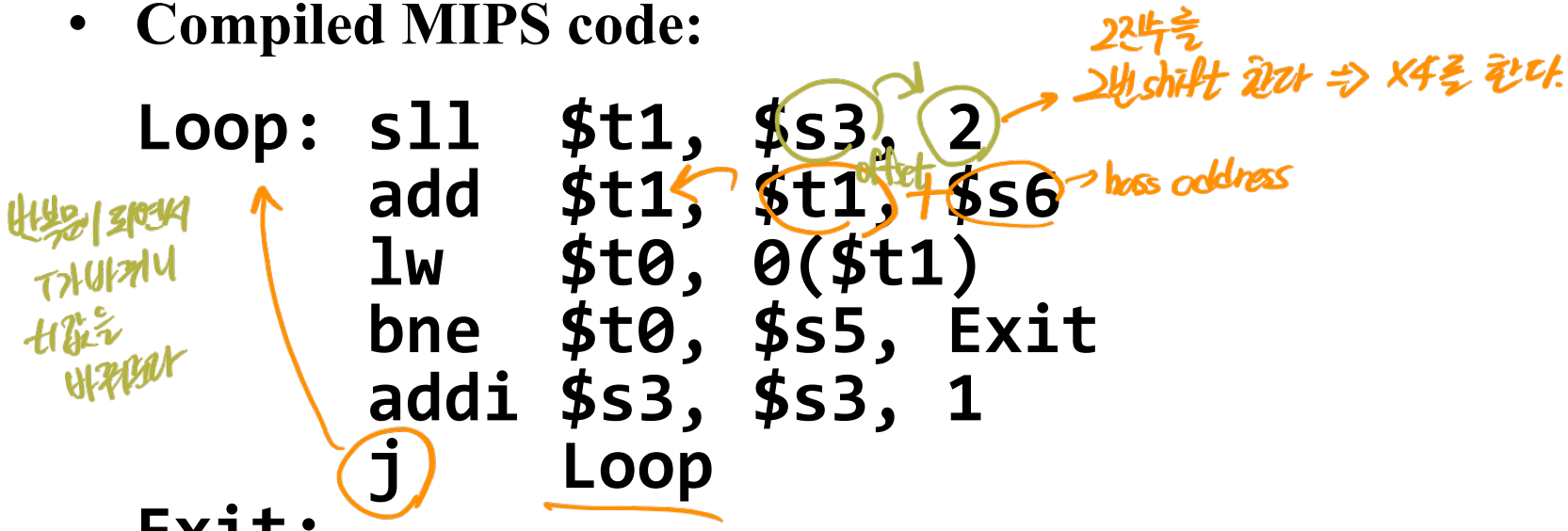
- **C code:**

```
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: …
```



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
```

Branch Instruction Design

- Why not `blt`, `bgt`, 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

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 ⇒ $t0 = 1`
  - `sltu $t0, $s0, $s1  # unsigned`
    - `+4,294,967,295 > +1 ⇒ $t0 = 0`

MIPS Design Principles

- **Simplicity favors regularity**
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- **Smaller is faster**
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- **Make the common case fast**
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands
- **Good design demands good compromises**
  - three instruction formats

CSE305 Computer Architecture

Instruction Set Architecture II

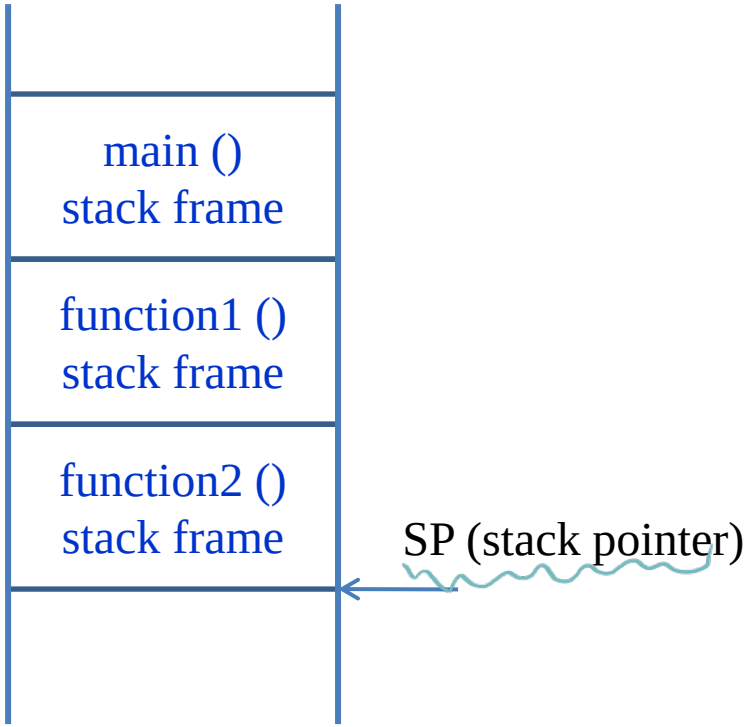
Daehoon Kim
Department of EECS, DGIST

Procedure Calls

- **Stack**
  - **Part of memory to store local variables in function**
  - **Each procedure call creates a stack frame**
  - **On function exit, remove the stack frame by moving SP (stack pointer)**

```
void main () {
        …
        fuction1()
        …
}

void function1 ()
        …
        function2 ()
        …
}
```



Six Steps in Execution of a Procedure

1. Main routine (caller) places parameters in a place where the procedure (callee) can access them
   - `$a0` - `$a3`: four argument registers
2. Caller transfers control to the callee
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
   - `$v0` - `$v1`: two value registers for result values
6. Callee returns control to the caller
   - `$ra`: one return address register to return to the point of origin

MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

Instructions for Accessing Procedures

- **MIPS procedure call instruction:**

```
jal   ProcedureAddress      #jump and link
```

  - **Saves PC+4 in register $ra to have a link to the next instruction for the procedure return**
  - **Jumps to target address**
  - **Machine format (J format):**

function name

26bit로 변환됨.

Jal func_ 이후
jr ra 명령어쓰지만 더라로
자동으로 Jal값+4 (PC+4)에 저장한다.
(PC→현재 실행하고 있는 instruction의 주소)

0x03	26 bit address

- **Then can do procedure return with a**

```
jr    $ra      #return
```

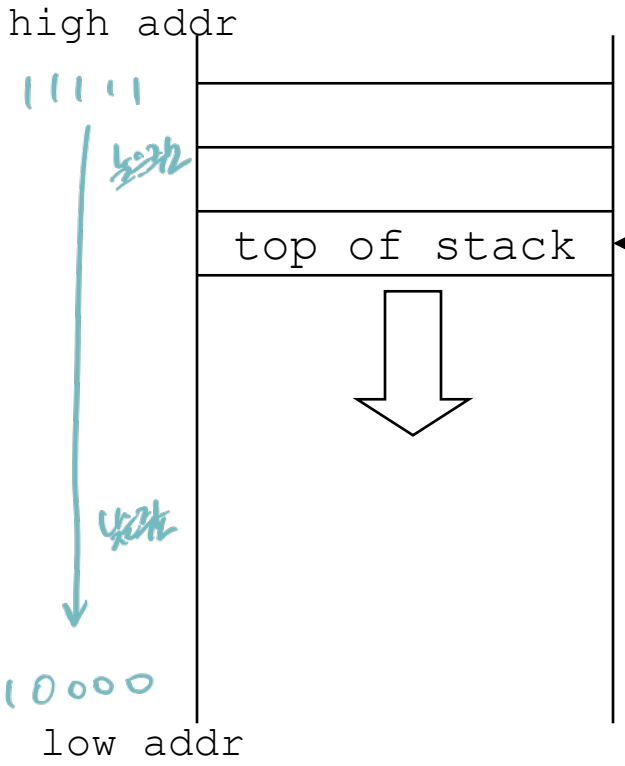
ra라는 register에 저장됨.

  - **Copies $ra to program counter (PC)**
  - **Can also be used for computed jumps (switch statement)**
  - **Instruction format (R format):**

0	31				0x08

Aside: Spilling Registers

- **What if the callee needs to use more registers than allocated to argument and return values?**
  - **callee uses a stack – a last-in-first-out queue**



- One of the general registers, `$sp` (`$29`), is used to address the stack (which "grows" from high address to low address)

add data onto the stack – push

`$sp = $sp – 4`

data on stack at new $sp

remove data from the stack – pop

data from stack at $sp

`$sp = $sp + 4`

Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

Leaf Procedure Example

- **MIPS code:**

`leaf_example:`	
`addi $sp, $sp, -4` `sw $s0, 0($sp)`	Save $s0 on stack

`add $t0, $a0, $a1`
`add $t1, $a2, $a3`
`sub $s0, $t0, $t1`
	Procedure body

`add $v0, $s0, $zero`	Result
`lw $s0, 0($sp)` `addi $sp, $sp, 4`	Restore $s0
`jr $ra`	Return

꼴에대한이이 ← Non-Leaf Procedures

- **Procedures that call other procedures**
- **For nested call, caller needs to save on the stack:**
  - **Its return address**
  - **Any arguments and temporaries needed after the call**
- **Restore from the stack after the call**

func1() ← non-leaf procedure → 여기서쓰는 ra, S0 S1 … 레지스터

다르므로 저장을 해주어야함.

fun2() ← leaf procedure

{ return → 여기서쓰는 ra, S0부터 레지스터

return

Non-Leaf Procedure Example

- **C code:**

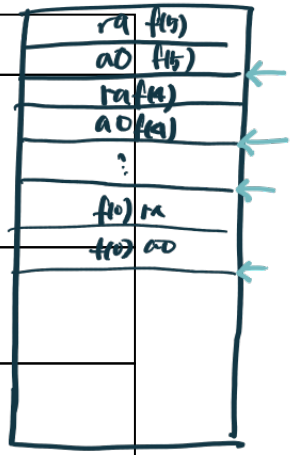
```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

  - **Argument n in $a0**
  - **Result in $v0**

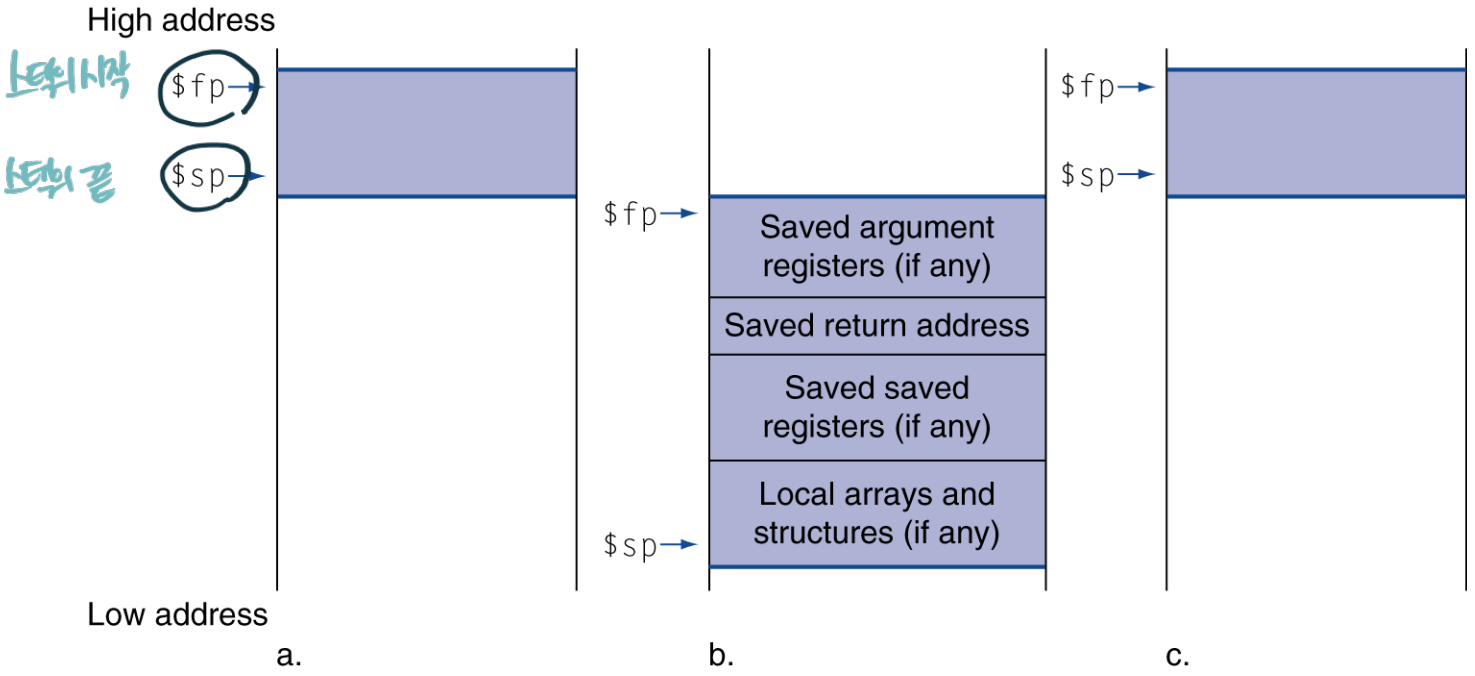
Non-Leaf Procedure Example

- **MIPS code:**

```
fact:
    addi $sp, $sp, -8       # adjust stack for 2 items
    sw   $ra, 4($sp)        # save return address
    sw   $a0, 0($sp)        # save argument
    slti $t0, $a0, 1        # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1      # if so, result is 1
    addi $sp, $sp, 8        #   pop 2 items from stack
    jr   $ra                #   and return
L1: addi $a0, $a0, -1       # else decrement n
    jal  fact               # recursive call
    lw   $a0, 0($sp)        # restore original n
    lw   $ra, 4($sp)        #   and return address
    addi $sp, $sp, 8        # pop 2 items from stack
    mul  $v0, $a0, $v0      # multiply to get result
    jr   $ra                # and return
```



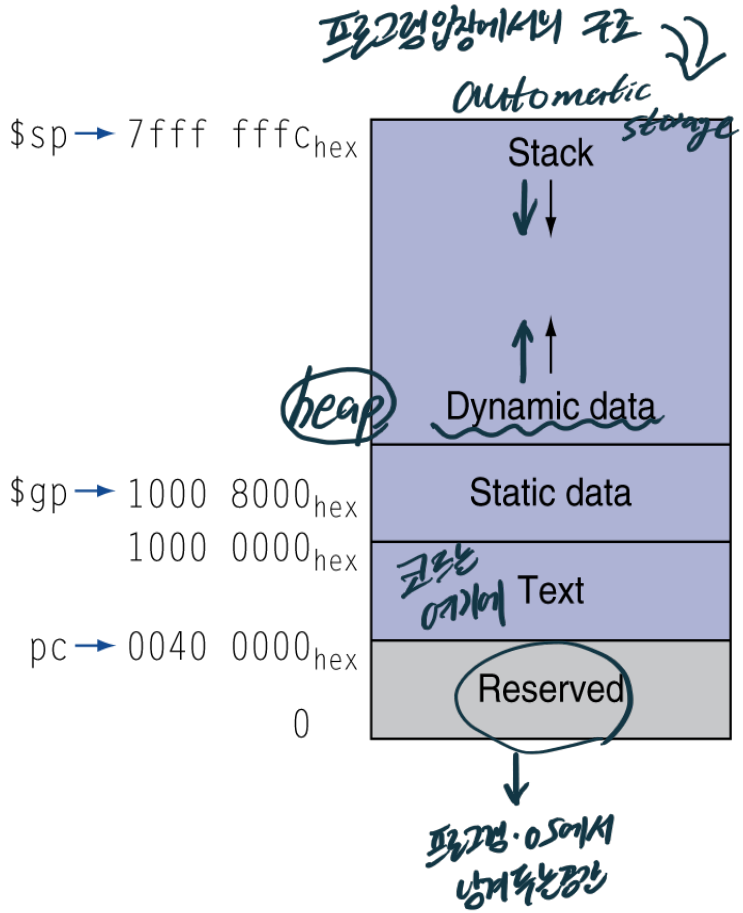
Local Data on the Stack



- **Stack frame saves local data allocated by callee**
  - **e.g., C automatic variables**
- **$fp: frame pointer**
  - **Point top of stack frame**

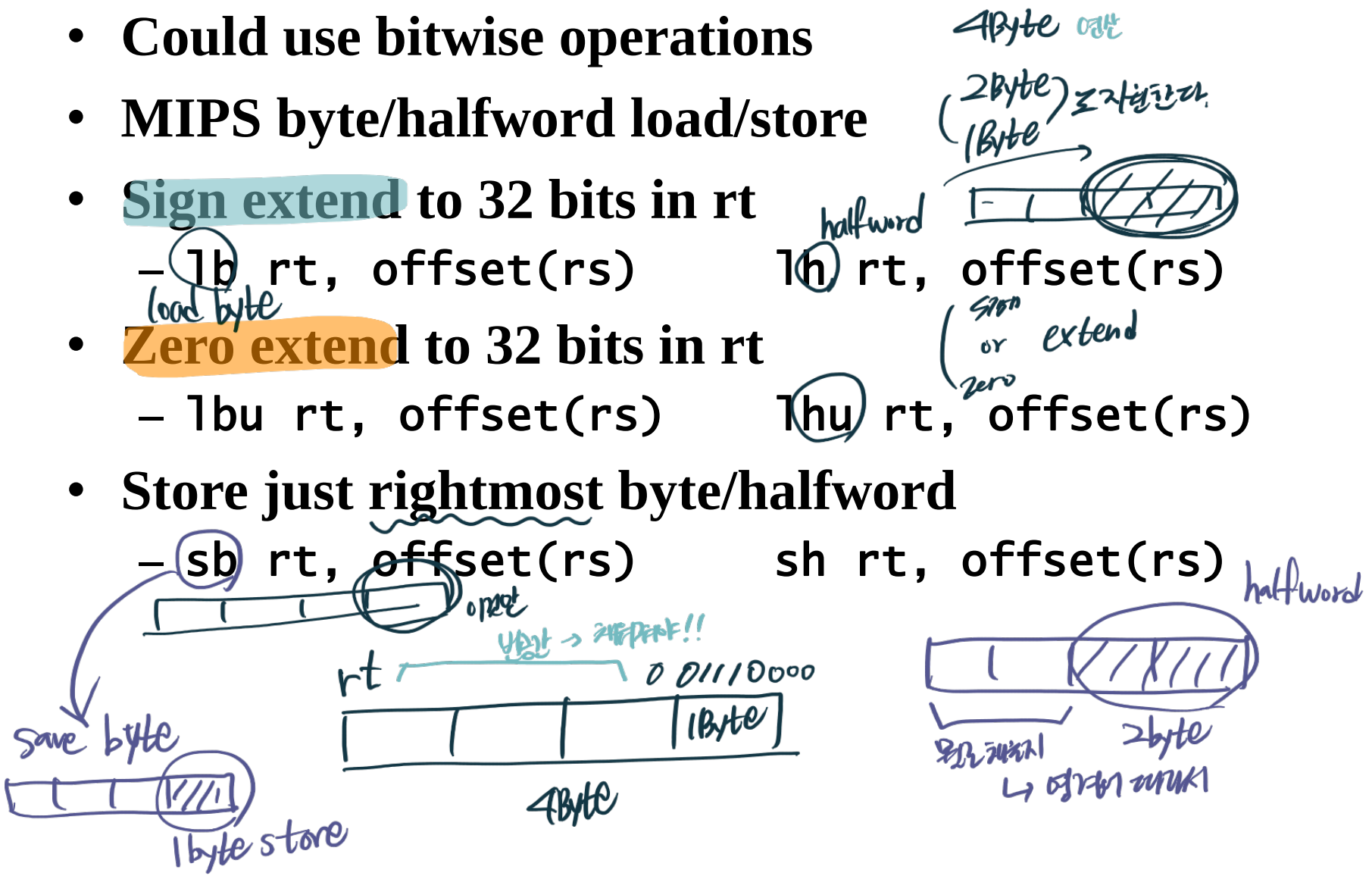
Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±16-bits offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Byte/Halfword Operations

- **Could use bitwise operations**
- **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)`
- **Store just rightmost byte/halfword**
  - `sb rt, offset(rs)` `sh rt, offset(rs)`



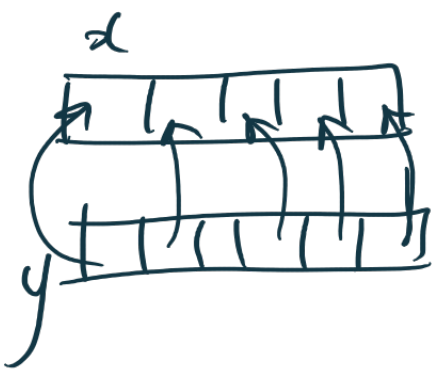
Memory Addressing Modes

- **Various Memory Addressing Modes**
  - **Immediate: R4 ← Mem[Imm]**
  - **Register indirect: R4 ← Mem [R1]**
  - **Displacement: R4 ←Mem [R1+Disp]**
  - **Indexed: R4 ← Mem [R1 + R2]**
  - **Memory indirect: R4 ←Mem[Mem[R1]]**
  - **Auto increment: R4 ← Mem[R1], R1 ← R1 + d**
  - **Scaled: R4 ← Mem[R1 + R2 * Scale]**
- **MIPS: support only displacement mode**
- **X86: support displacement, indexed and scaled mode**

String Copy Example

- **C code (naïve):**
  - **Null-terminated string**

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```



  - **Addresses of x, y in $a0, $a1**
  - **i in $s0**

String Copy Example

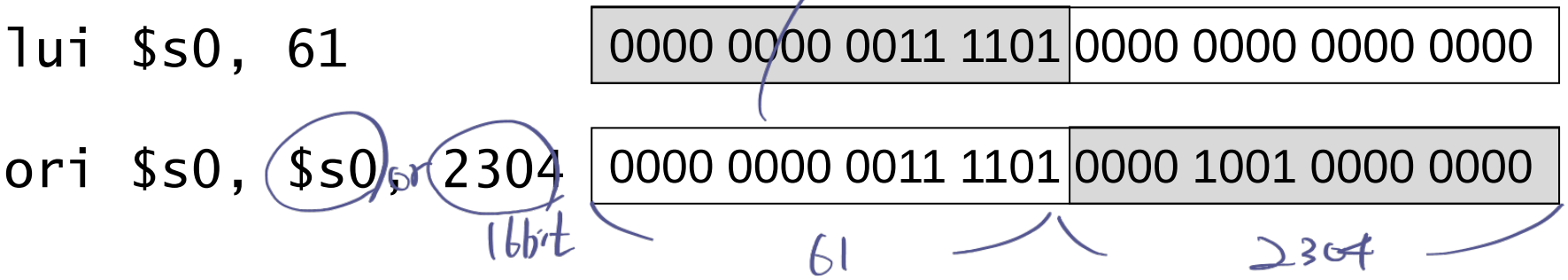
- **MIPS code:**

```
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw   $s0, 0($sp)        # save $s0
    add  $s0, $zero, $zero  # i = 0
L1: add  $t1, $s0, $a1      # addr of y[i] in $t1
    lbu  $t2, 0($t1)        # $t2 = y[i]
    add  $t3, $s0, $a0      # addr of x[i] in $t3
    sb   $t2, 0($t3)        # x[i] = y[i]
    beq  $t2, $zero, L2     # exit loop if y[i] == 0
    addi $s0, $s0, 1        # i = i + 1
    j    L1                 # next iteration of loop
L2: lw   $s0, 0($sp)        # restore saved $s0
    addi $sp, $sp, 4        # pop 1 item from stack
    jr   $ra                # and return
```

32-bit Constants

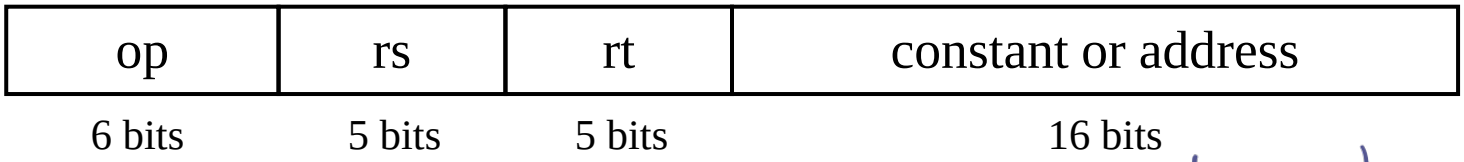
- **Most constants are small**
  - **16-bit immediate is sufficient**
- **For the occasional 32-bit constant**

  `lui rt, constant`
  - **Copies 16-bit constant to left 16 bits of rt**
  - **Clears right 16 bits of rt to 0**



Branch Addressing

- **Branch instructions specify (e.g., beq)**
  - **Opcode, two registers, target address**
- **Most branch targets are near branch**
  - **Forward or backward**



- **PC-relative addressing**
  - **Target address = PC + offset * 4 (offset: 16bits constant or address)**
  - **PC already incremented by 4**

Jump Addressing

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

- **(Pseudo) Direct jump addressing**
  - **Target address = $PC_{31...28}$: (address × 4)**

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: …
```

	rs	rt	rd		
80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000 × 4				
80024						

*For "bne", target address = PC + offset * 4 = 80024*

For "j", target address = $PC_{31…28}$: (address × 4)

C Sort Example

- **Illustrates use of assembly instructions for a C bubble sort function**
- **Swap procedure (leaf)**

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

  - **v in $a0, k in $a1, temp in $t0**

The Procedure Swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                         #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

The Sort Procedure in C

- **Non-leaf (calls swap)**

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i - 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v in $a0, n in $a1, i in $s0, j in $s1

function call 호출시 preservation 필요 레지스터.

The Procedure Body

```
        move $s2, $a0          # save $a0 into $s2
        move $s3, $a1          # save $a1 into $s3
        move $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $s3    # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
        beq  $t0, $zero, exit1 # go to exit1 if $s0 ≥ $s3 (i ≥ n)
        addi $s1, $s0, -1      # j = i - 1
for2tst: slti $t0, $s1, 0      # $t0 = 1 if $s1 < 0 (j < 0)
        bne  $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
        sll  $t1, $s1, 2       # $t1 = j * 4
        add  $t2, $s2, $t1     # $t2 = v + (j * 4)
        lw   $t3, 0($t2)       # $t3 = v[j]
        lw   $t4, 4($t2)       # $t4 = v[j + 1]
        slt  $t0, $t4, $t3     # $t0 = 0 if $t4 ≥ $t3
        beq  $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
        move $a0, $s2          # 1st param of swap is v (old $a0)
        move $a1, $s1          # 2nd param of swap is j
        jal  swap              # call swap procedure
        addi $s1, $s1, -1      # j -= 1
        j    for2tst           # jump to test of inner loop
exit2:  addi $s0, $s0, 1       # i += 1
        j    for1tst           # jump to test of outer loop
exit1:
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

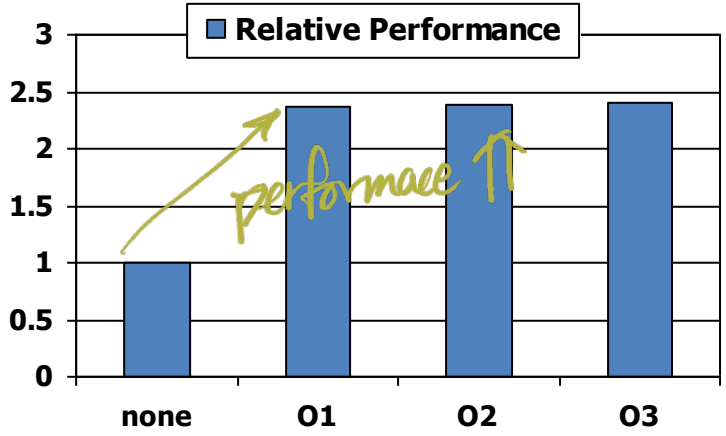
The Full Procedure

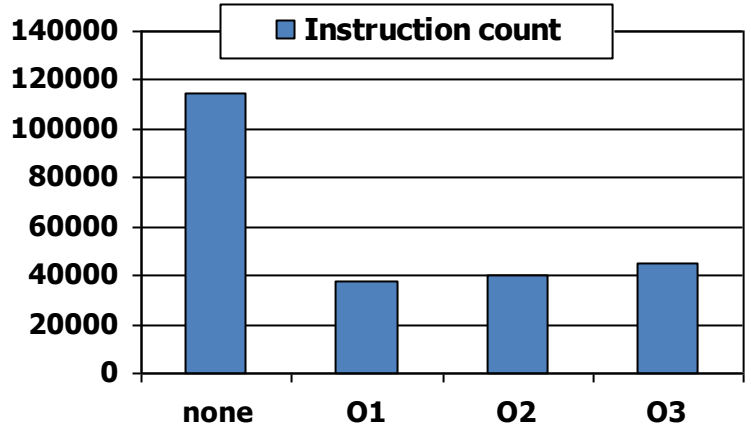
```
sort:   addi $sp,$sp, -20      # make room on stack for 5 registers
        sw $ra, 16($sp)        # save $ra on stack
        sw $s3,12($sp)         # save $s3 on stack
        sw $s2, 8($sp)         # save $s2 on stack
        sw $s1, 4($sp)         # save $s1 on stack
        sw $s0, 0($sp)         # save $s0 on stack
        …                      # procedure body
        …
        exit1: lw $s0, 0($sp)  # restore $s0 from stack
        lw $s1, 4($sp)         # restore $s1 from stack
        lw $s2, 8($sp)         # restore $s2 from stack
        lw $s3,12($sp)         # restore $s3 from stack
        lw $ra,16($sp)         # restore $ra from stack
        addi $sp,$sp, 20       # restore stack pointer
        jr $ra                 # return to calling routine
```

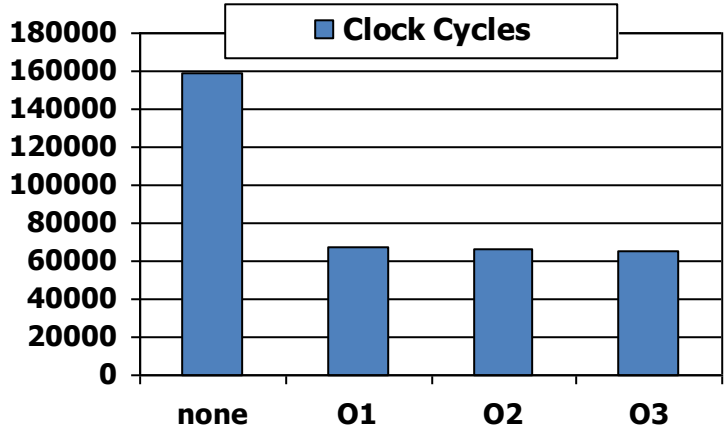
Effect of Compiler Optimization

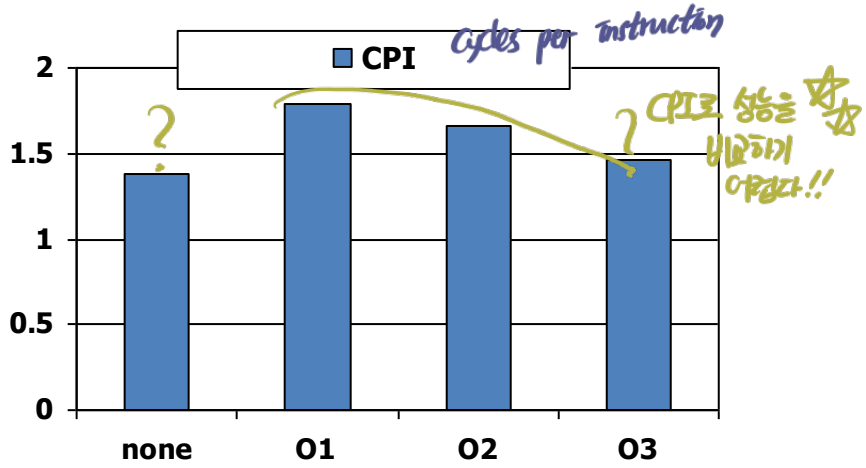
Compiled with gcc for Pentium 4 under Linux

같은 코드라도 Compile 옵션에 따라 성능이 다름.
→ Instruction이 다르게 생성된다!

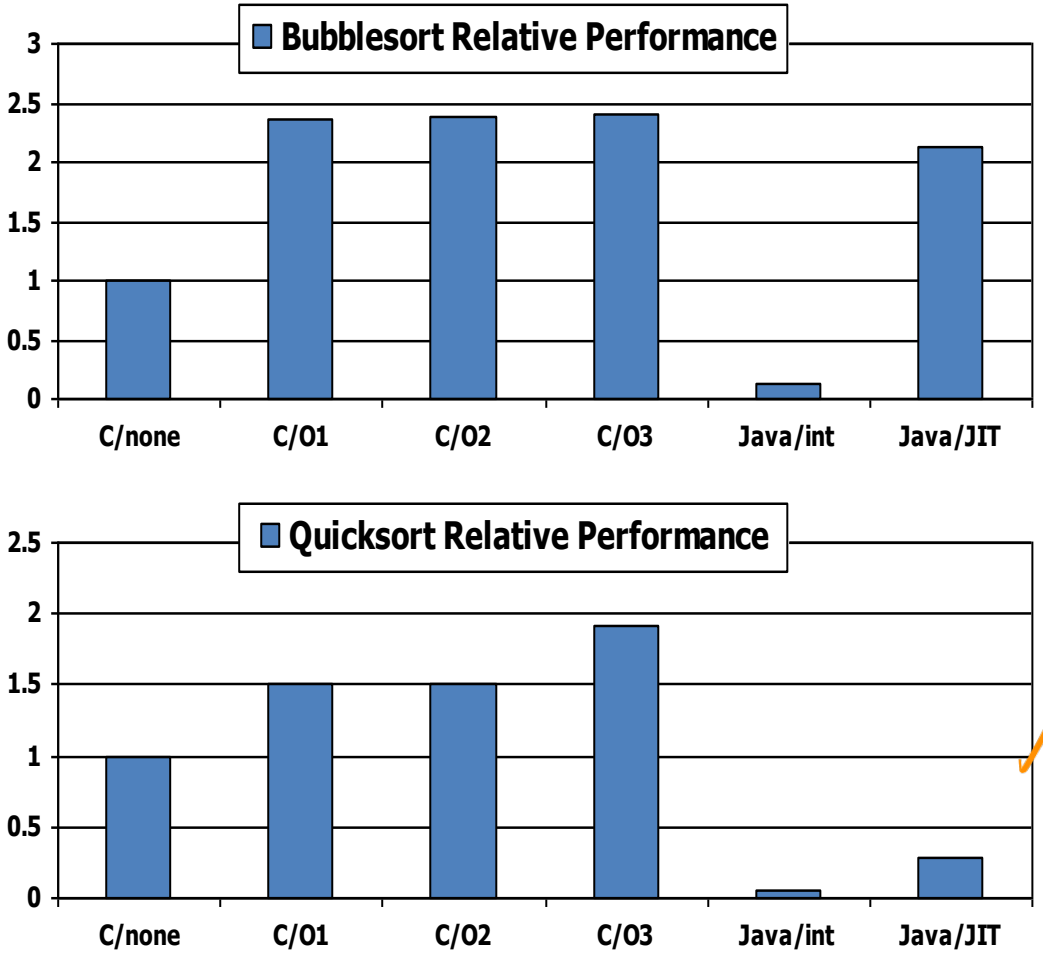








Effect of Language and Algorithm



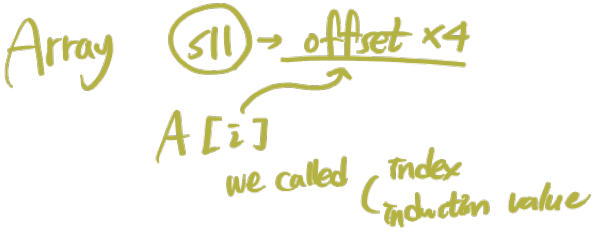
why?
instruction이 다르게 생성된다.

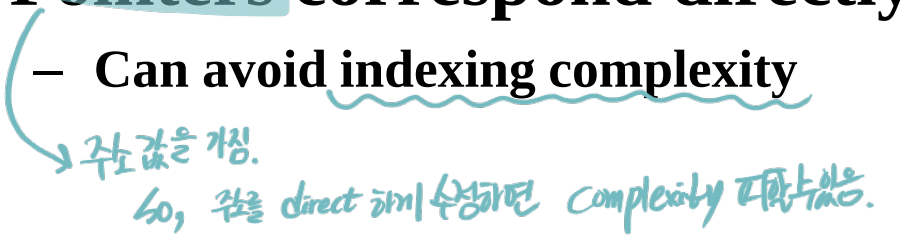
Lessons Learnt

- *Instruction count and CPI are not good performance indicators*
- **Compiler optimizations are sensitive to the algorithm**
- **Java/JIT compiled code is significantly faster than JVM interpreted**
  - **Comparable to optimized C in some cases**
- **Nothing can fix a dumb algorithm!**

Arrays vs. Pointers

- **Array indexing involves**
  - **Multiplying index by element size**
  - **Adding to array base address**
- **Pointers correspond directly to memory addresses**
  - **Can avoid indexing complexity**





Example: Clearing and Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
      move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2   # $t1 = i * 4
      add $t2,$a0,$t1  # $t2 =
                       #   &array[i]
      sw $zero, 0($t2) # array[i] = 0
      addi $t0,$t0,1   # i = i + 1
      slt $t3,$t0,$a1  # $t3 =
                       #   (i < size)
      bne $t3,$zero,loop1 # if (…)
                          # goto loop1
```

```
      move $t0,$a0     # p = & array[0]
      sll $t1,$a1,2    # $t1 = size * 4
      add $t2,$a0,$t1  # $t2 =
                       #   &array[size]
loop2: sw $zero,0($t0) # Memory[p] = 0
      addi $t0,$t0,4   # p = p + 4
      slt $t3,$t0,$t2  # $t3 =
                       #(p<&array[size])
      bne $t3,$zero,loop2 # if (…)
                          # goto loop2
```

Comparison of Array vs. Ptr

- **Array version requires shift to be inside loop**
  - **Part of index calculation for incremented i**
  - **c.f. incrementing pointer**
- **Compiler can achieve same effect as manual use of pointers by eliminating induction variable**

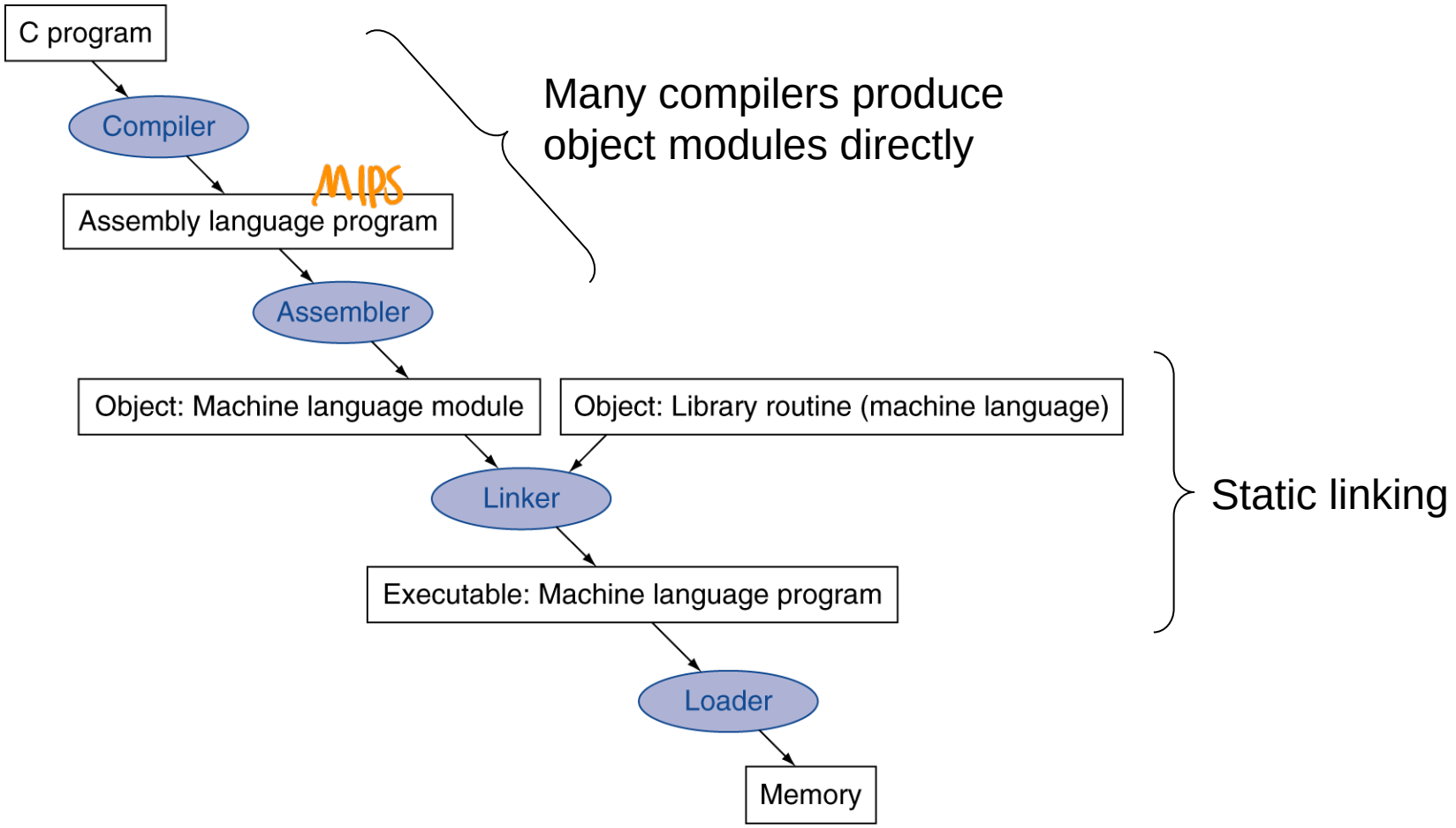
Assembler Pseudoinstructions

- **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**

Translation and Startup



Producing an Object Module

- **Assembler translates program into machine instructions**
- **Provides information for building a complete program from the pieces**
  - **Header: described contents of object module (e.g., size, name)**
  - **Text segment: translated instructions (i.e., machine code)**
  - **Data segment: data allocated for the life of the program**
  - **Relocation info: instructions and data that depend on absolute addresses**
  - **Symbol table: mapping between labels and addresses**
  - **Debug info: for associating with source code**

다양한 정보들

gcc g++

Linking Object Modules

- **Produces an executable image**
  1. **Place code and data modules symbolically in memory**
  2. **Determine the addresses of data and instructions labels using relocation info & symbol table**
  3. **Patch both the internal and external references**

Loading a Program

- **Load from image file on disk into memory**
  1. **Read header to determine segment sizes**
  2. **Create virtual address space**
  3. **Copy text and initialized data into memory**
     - **Or set page table entries so they can be faulted in**
  4. **Set up arguments (for main function) on stack**
  5. **Initialize registers (including $sp, $fp, $gp)**
  6. **Jump to startup routine**
     - **Copies arguments to $a0, … and calls main**
     - **When main returns, do exit syscall**

Dynamic Linking

- **Only link/load library procedure when it is called**
  - **Requires procedure code to be relocatable**
  - **Avoids image bloat caused by static linking of all referenced libraries**
  - **Automatically picks up new library versions**

CSE305 Computer Architecture

Processor I : Datapath and Control

Daehoon Kim
Department of EECS, DGIST

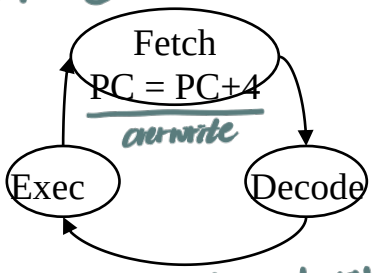
Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`

The Processor: Datapath & Control

- **Our implementation of the MIPS is simplified**
  - **memory-reference instructions:** `lw, sw`
  - **arithmetic-logical instructions:** `add, sub, and, or, slt`
  - **control flow instructions:** `beq, j`
- **Generic implementation**
  - **use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)**
  - **decode the instruction (and read registers)**
  - **execute the instruction**

register : 명령어레지스터 Instruction의 주소 담고있음.



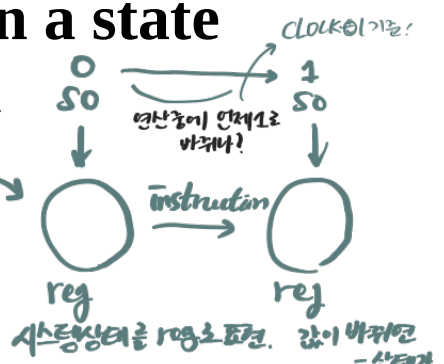
fetch : PC주소에서 Instruction 가져오는 것

- **What do we need to consider to design a processor?**
  - **Need ALU (arithmetic logic unit)**
  - **All instructions (except `j`) use the ALU after reading the registers**
  - **Memory references use the ALU to compute addresses (e.g., lw, sw)**
  - **Arithmetic instructions use the ALU to do the required arithmetic (e.g., add, sub)**
  - **Control instructions use the ALU to compute branch conditions (e.g., beq, bne)**

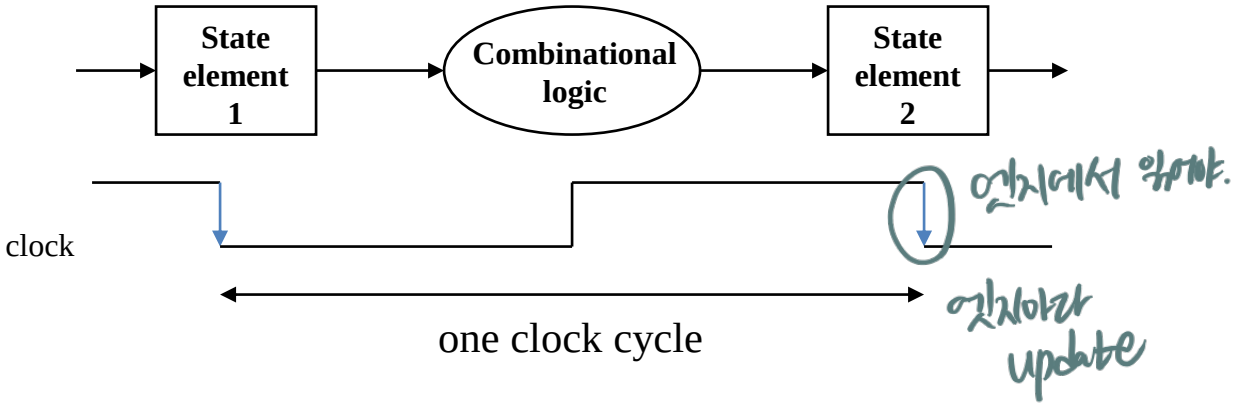
target address

Clocking Methodologies

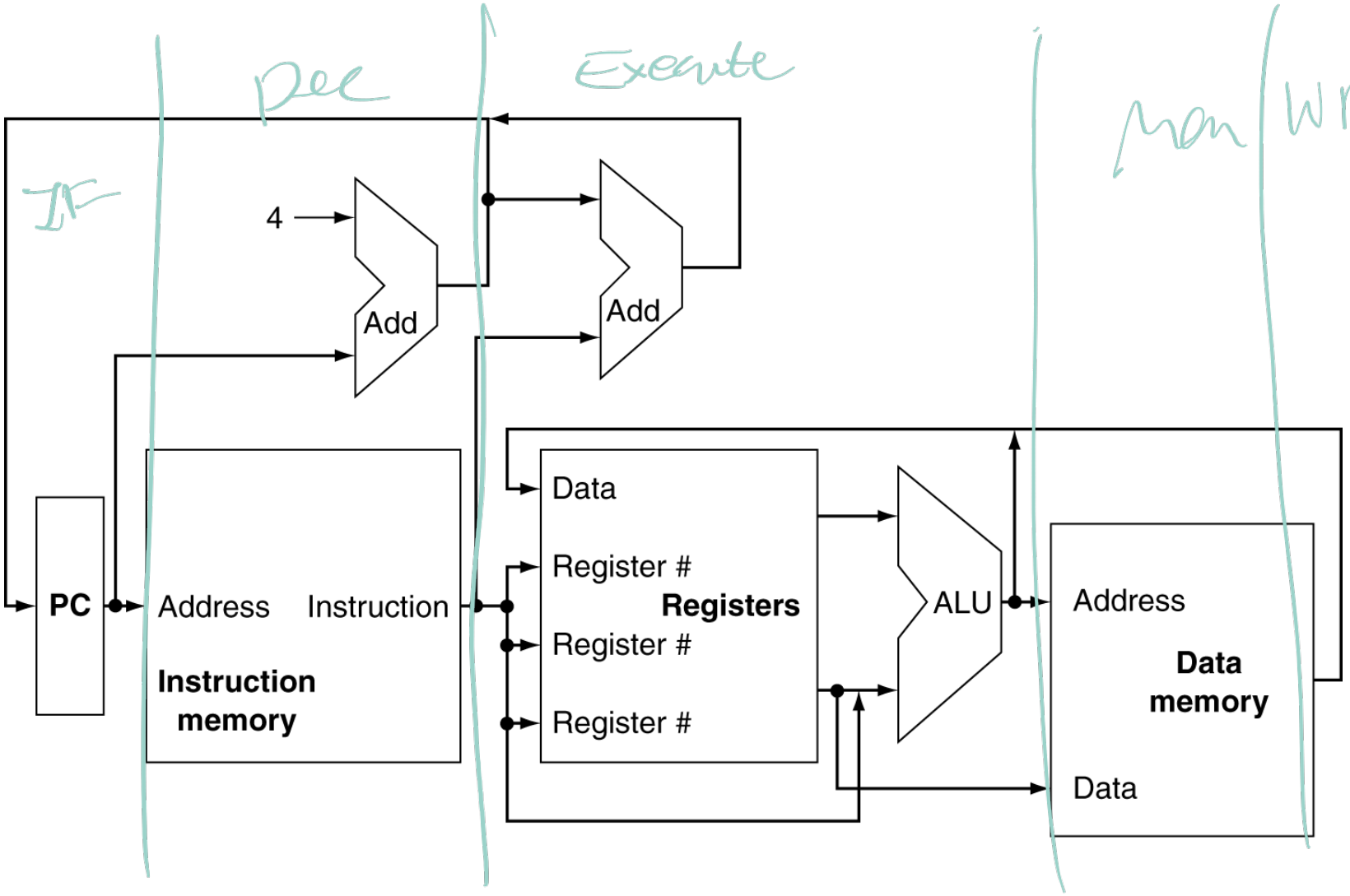
- **The clocking methodology defines when data in a state element is valid and stable relative to the clock**
  - **State elements - a memory element such as a register**
  - **Edge-triggered - all state changes occur on a clock edge**



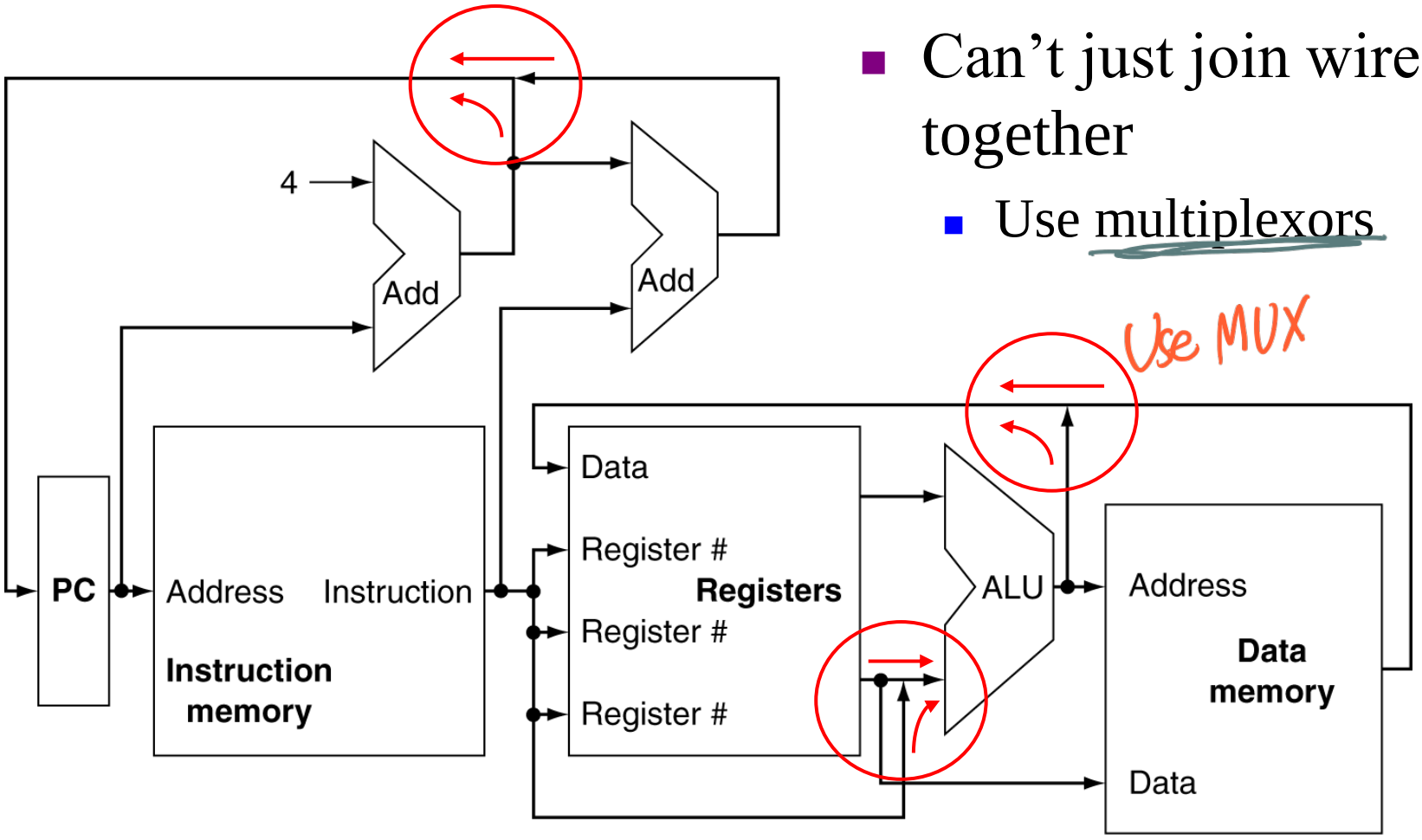
- **Typical execution**
  - **read contents of state elements → send values through combinational logic → write results to one or more state elements**



CPU Overview

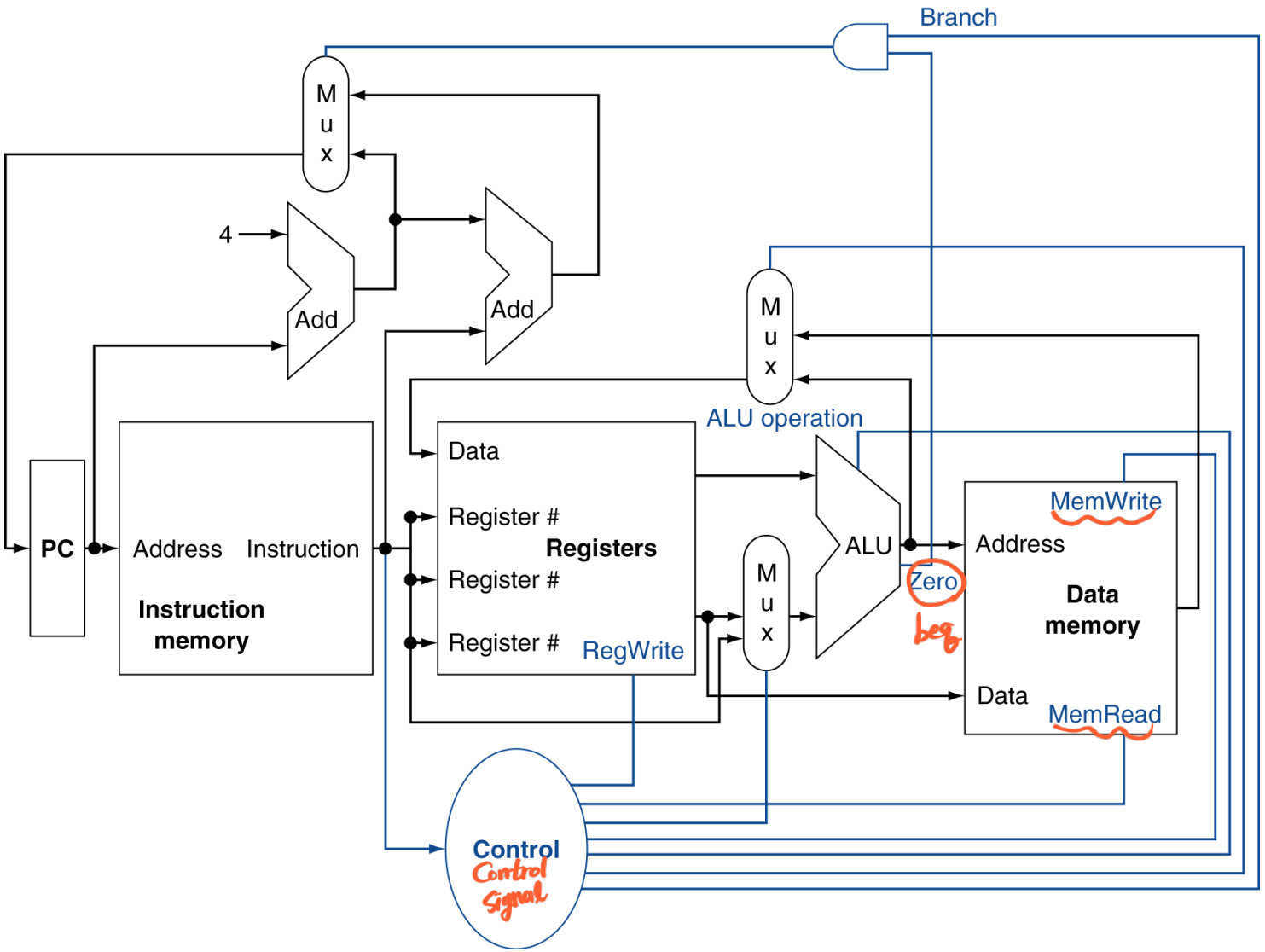


Multiplexers



- Can't just join wires together
  - Use multiplexors

Control



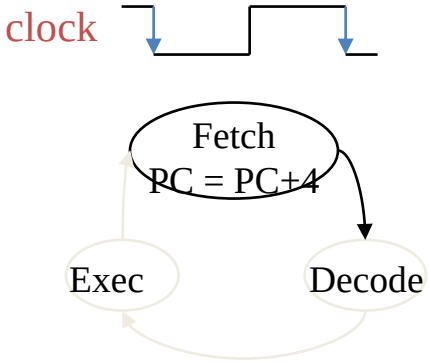
Memory
Register
읽기/쓰기

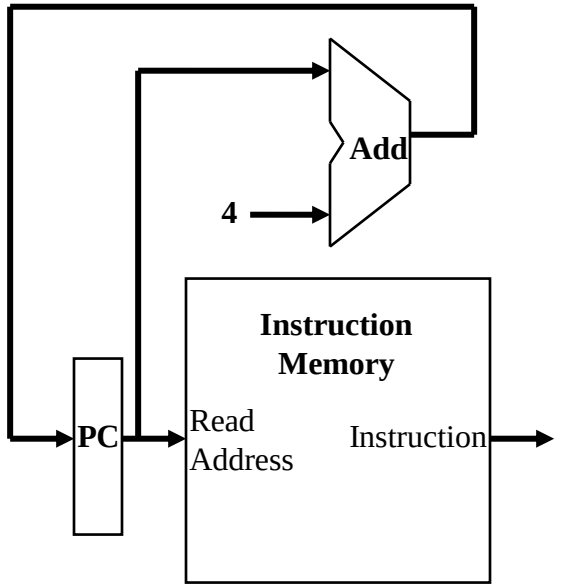
Building a Datapath

- **Datapath**
  - **Elements that process data and addresses in the CPU**
    - **Registers, ALUs, mux's, memories, …**
- **We will build a MIPS datapath incrementally**
  - **Refining the overview design**

Fetching Instructions

- **Fetching instructions involves**
  - **reading the instruction from the Instruction Memory**
  - **updating the PC value to be the address of the next (sequential) instruction**



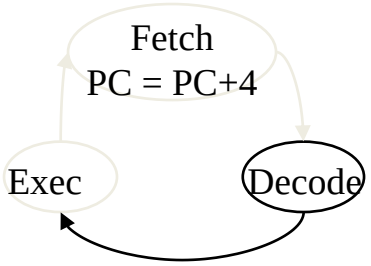


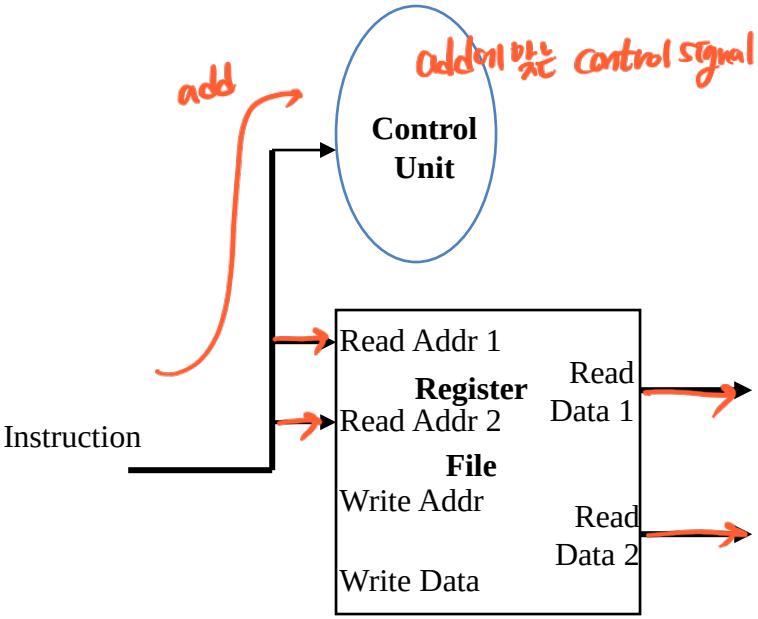
- **PC is updated every clock cycle, so it does not need an explicit write control signal just a clock signal**
- **Reading from the Instruction Memory doesn't need an explicit read control signal**

매 클럭마다 진행되는 것들!

Decoding Instructions

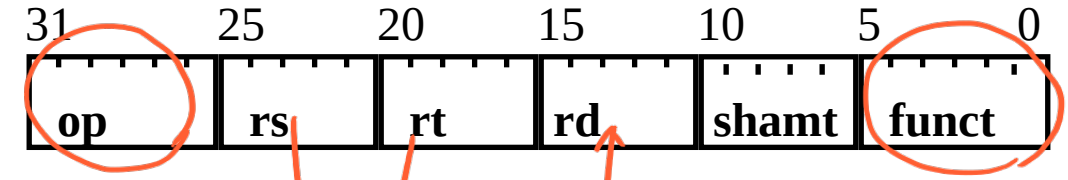
- **Decoding instructions involves**
  - **sending the fetched instruction's opcode and function field bits to the control unit**
  - **reading two values from the Register File (Register File addresses are contained in the instruction)**



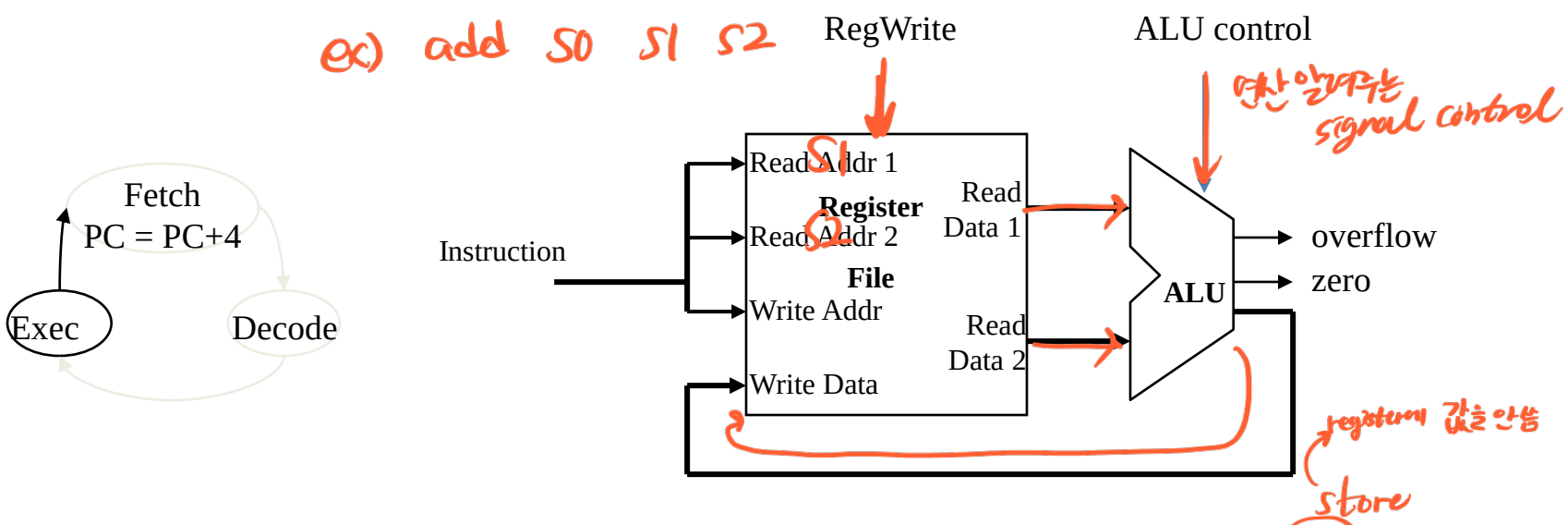


Executing R Format Operations

- **R format operations (add, sub, slt, and, or)**



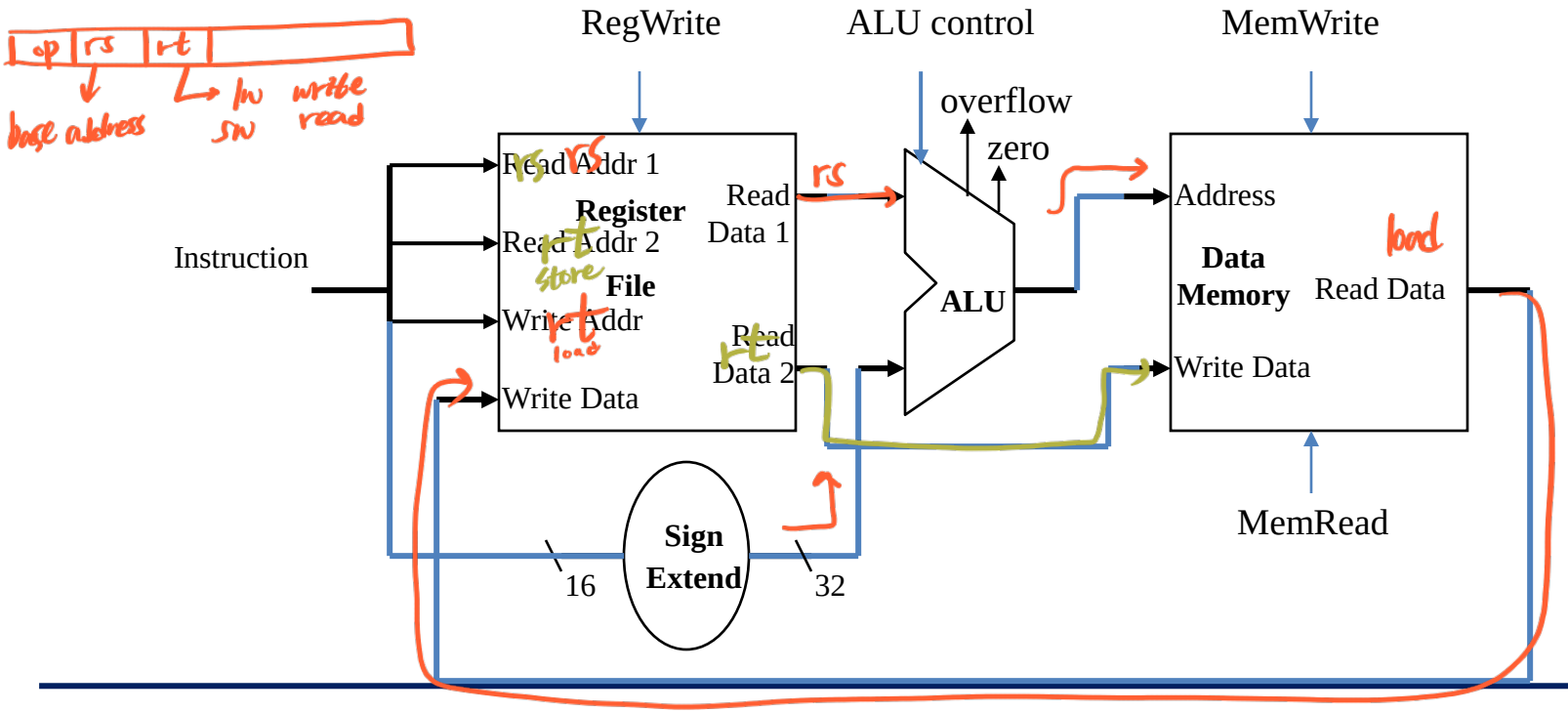
  - **perform operation (op and funct) on values in rs and rt**
  - **store the result back into the Register File (into location rd)**



Note that Register File is not written every cycle (e.g. **sw**), so we need an explicit write control signal for the Register File

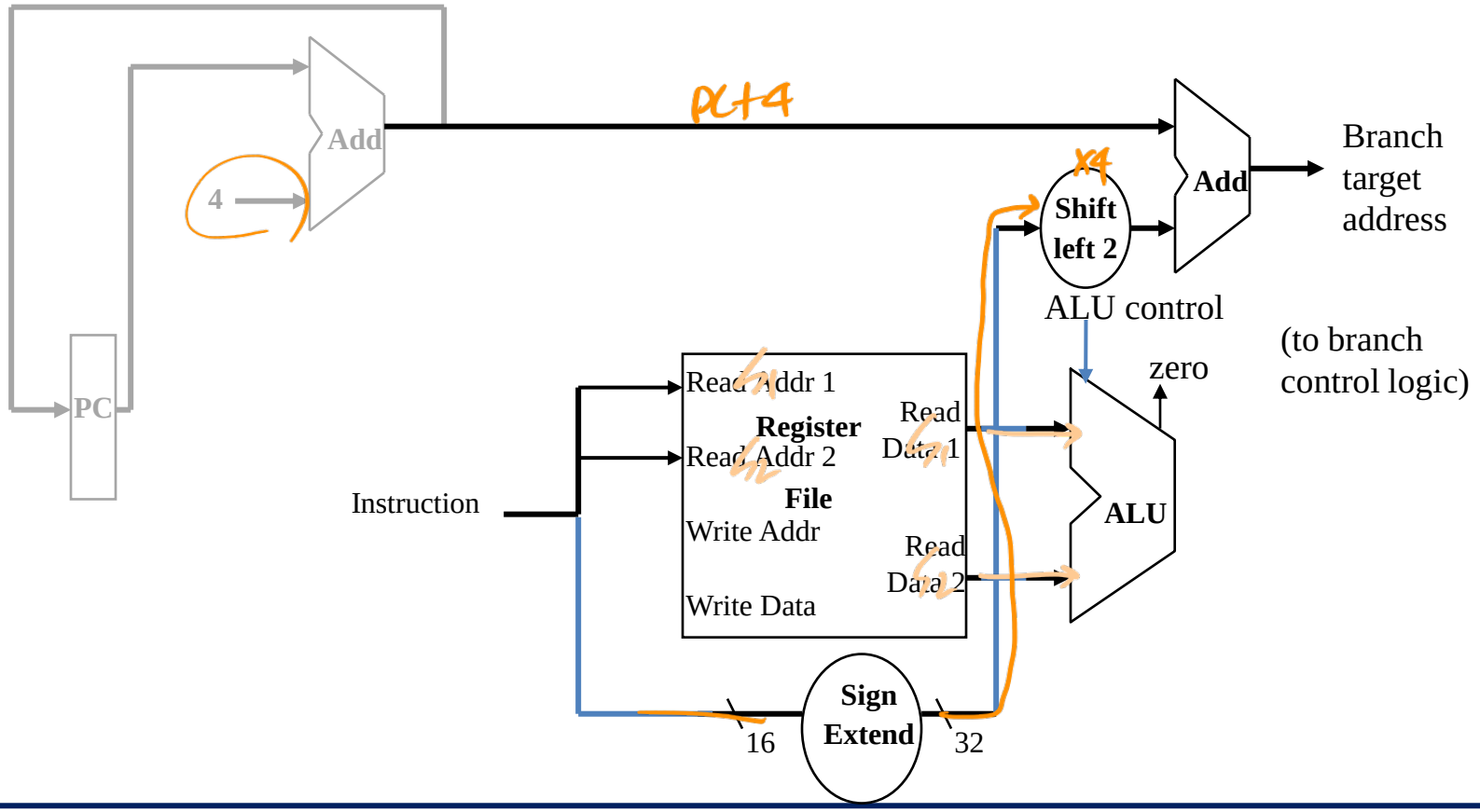
Executing Load and Store Operations

- **Load and store operations involve**
  - **compute memory address by adding the base register (read from the Register File during decode) to the 16-bit signed-extended offset field in the instruction**
  - **store value (read from the Register File during decode) written to the Data Memory**
  - **load value, read from the Data Memory, written to the Register File**



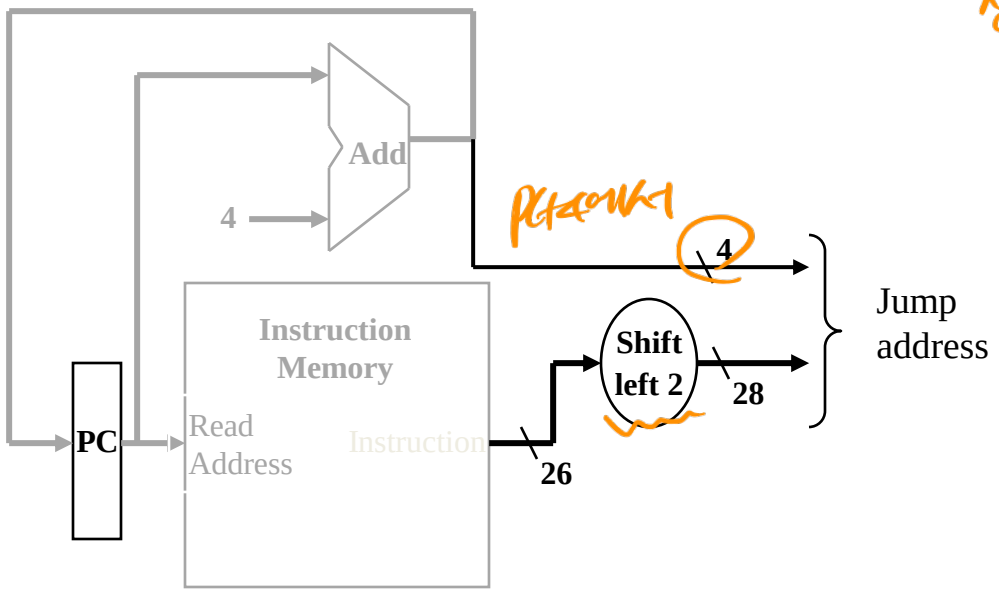
Executing Branch Operations

- **Branch operations involve**
  - **compare the operands read from the Register File during decode for equality (`zero` ALU output)**
  - **compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instructions**



Executing Jump Operations

- **Jump operation involves**
  - **replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits**



Creating a Single Datapath

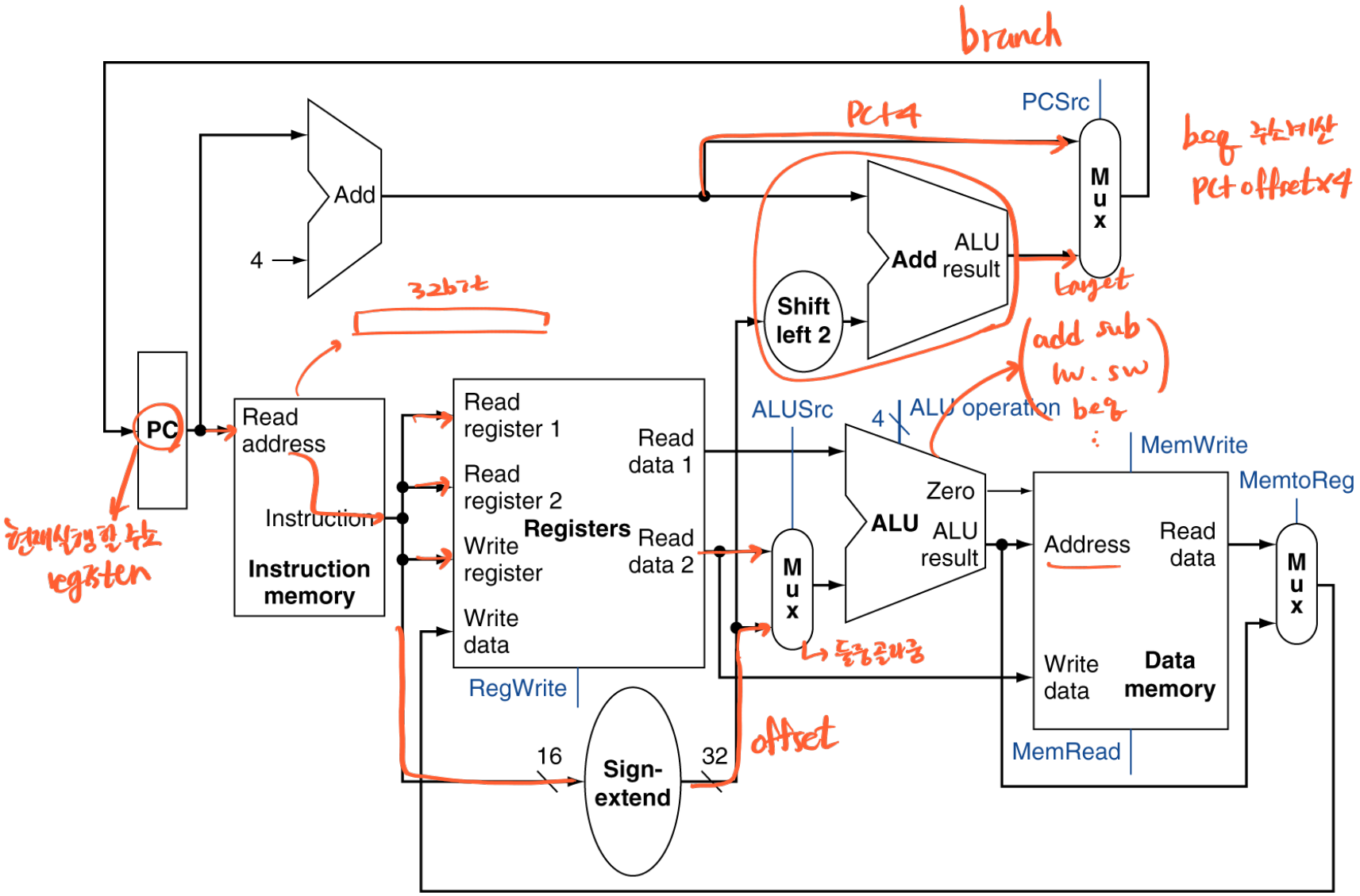
- **Assemble the datapath segments and add control lines and multiplexors as needed**
- **Single cycle design – fetch, decode and execute each instructions in one clock cycle**
  - **no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)**
  - **multiplexors needed at the input of shared elements with control lines to do the selection**
  - **write signals to control writing to the Register File and Data Memory**
- **Cycle time is determined by length of the longest path**

But. It is constant. 1 cycle → 1 instruction

ex) add 10ps

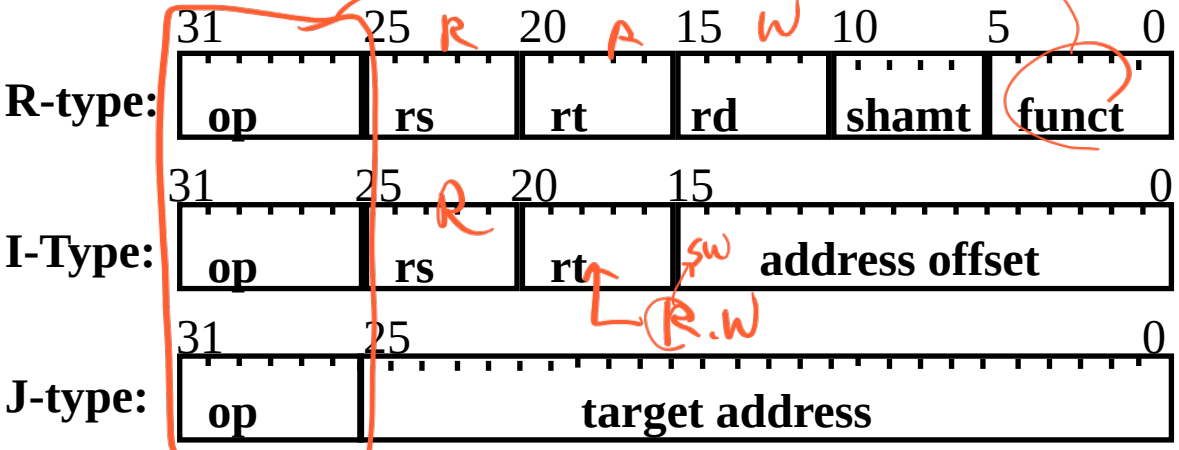
lw 30ps → cycle time

Full Datapath



Adding the Control

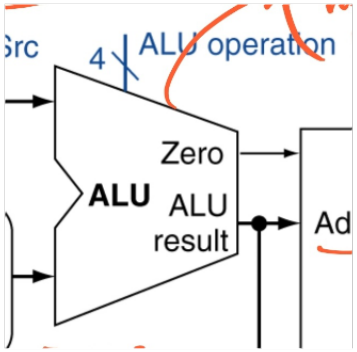
- **Selecting the operations to perform (ALU, Register File and Memory read/write)**
- **Controlling the flow of data (multiplexor inputs)**



- **Observations**
  - **op field always in bits 31-26**
  - **addr. of registers to be read are always specified by the rs field (bit 25-21) and rt field (bit 20-16); for lw and sw, rs is the base register**
  - **addr. of register to be written is in one of two places – in rt (bits 20-16) for lw; in rd (bits 15-11) for R-type instructions**
  - **offset for beq, lw, and sw always in bits 15-0**

ALU Control

- **ALU used for**
  - **Load/Store: "add" -> calculate a target memory address**
  - **Branch: "subtract" -> check the result is 0 or not**
  - **R-type: depends on funct field**



ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

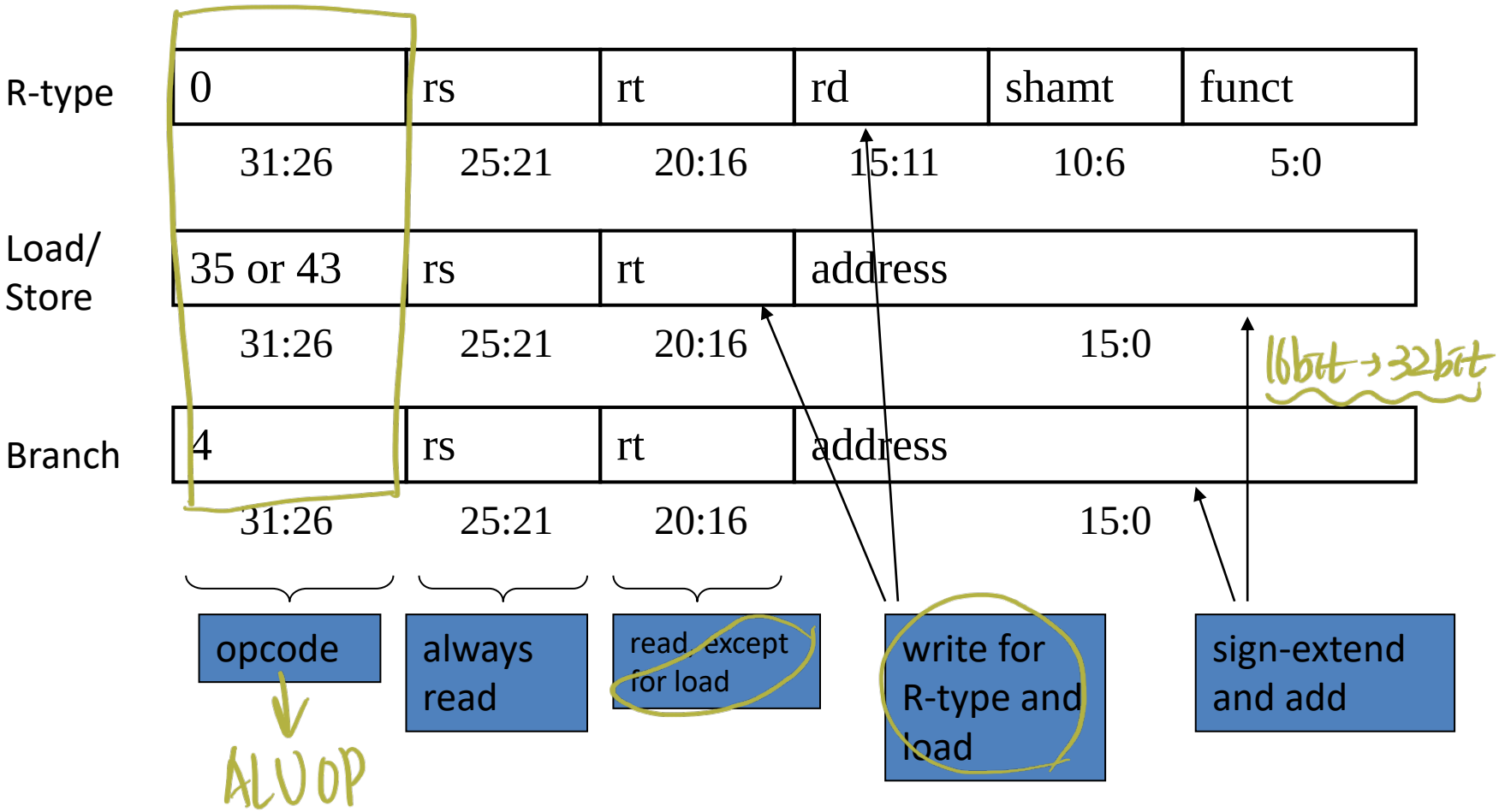
ALU Control

- **Assume 2-bit ALUOp derived from opcode**
  - **Combinational logic derives ALU control** ALUOP
- **ALUOp**
  - **Control field for ALU control**
  - **Indicate whether the operation to be performed should be add for lw&sw, subtract for beq, or determined by the operation encoded in the funct field**

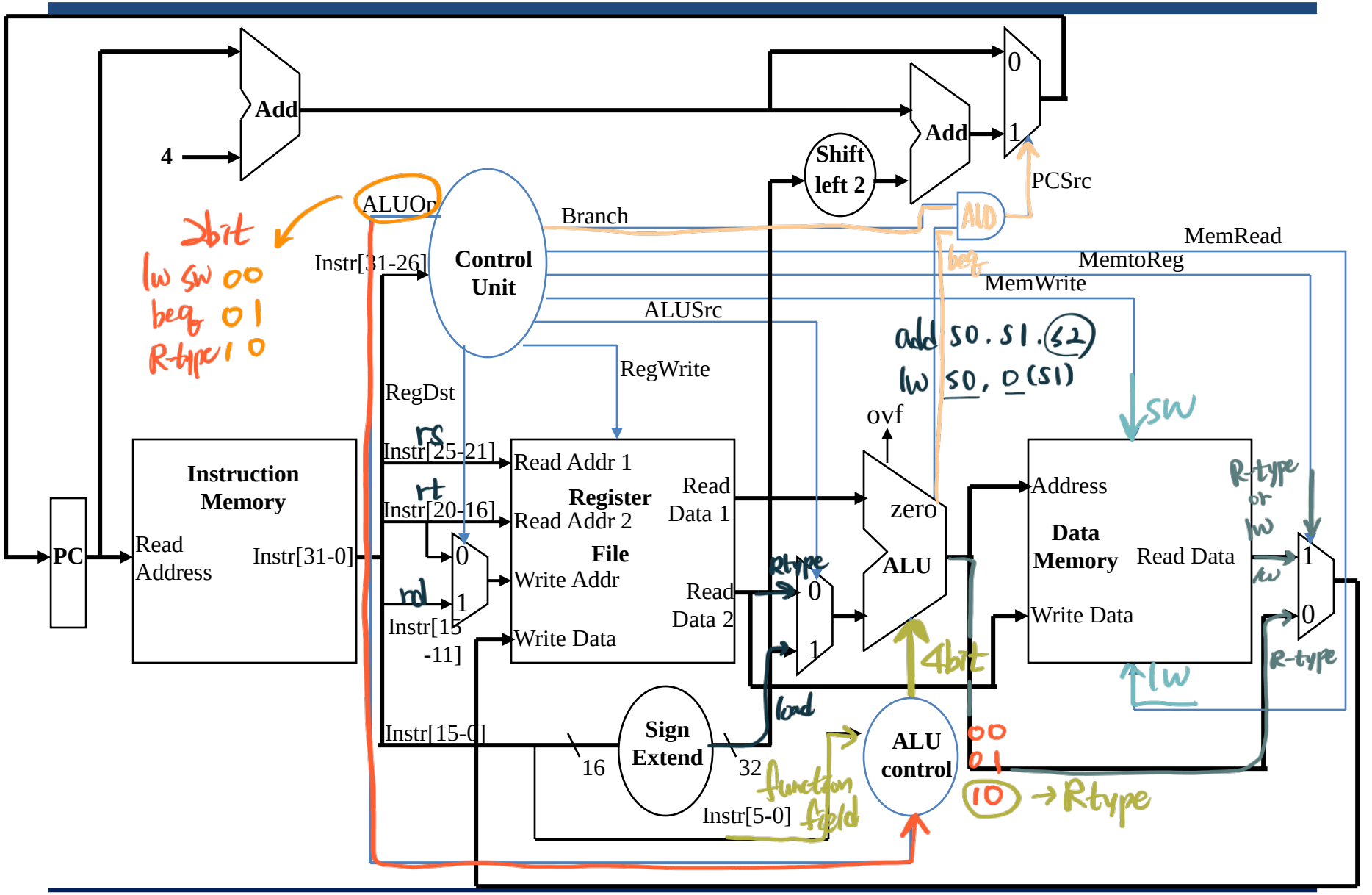
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The Main Control Unit

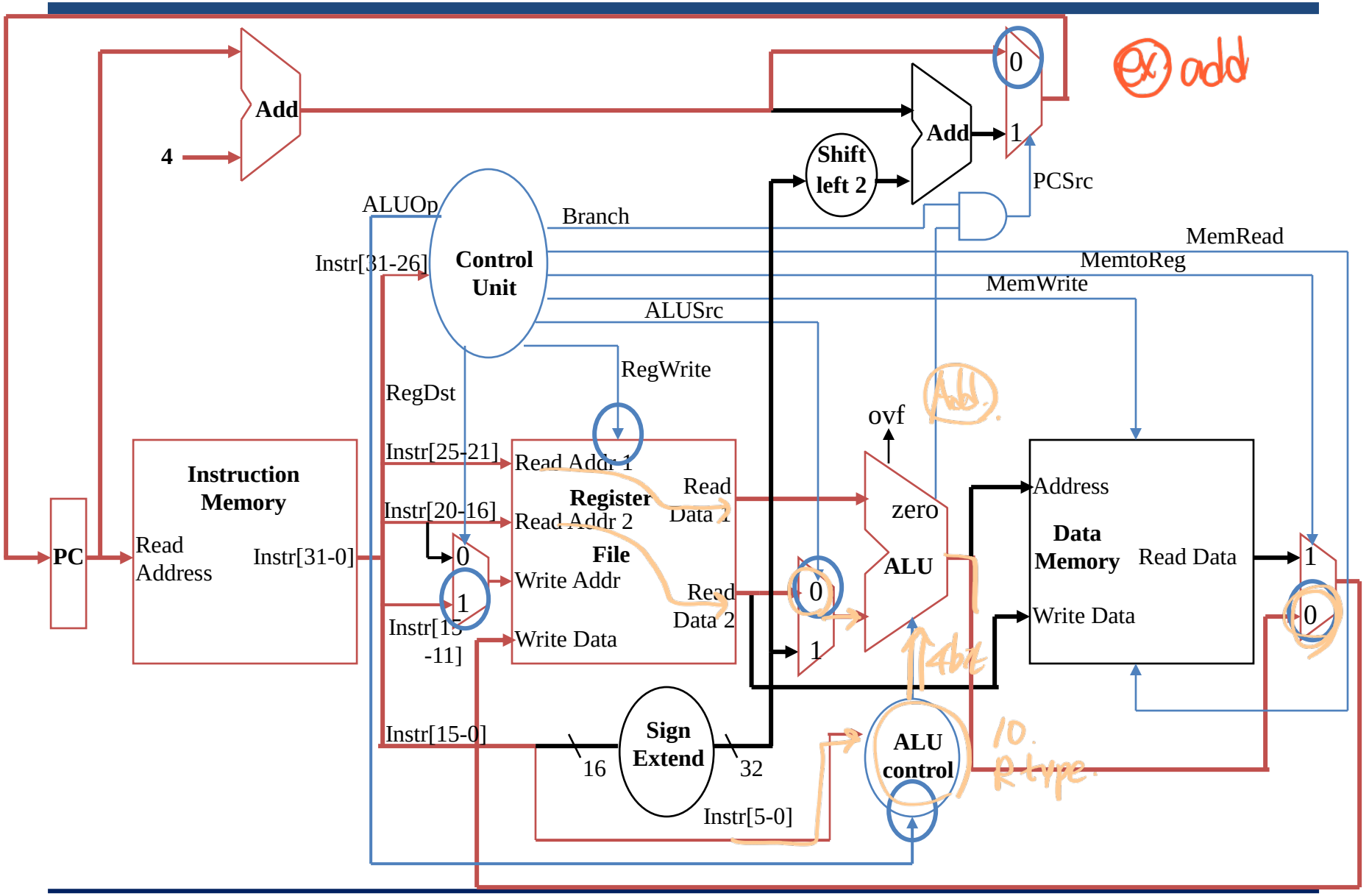
- **Control signals derived from instruction**



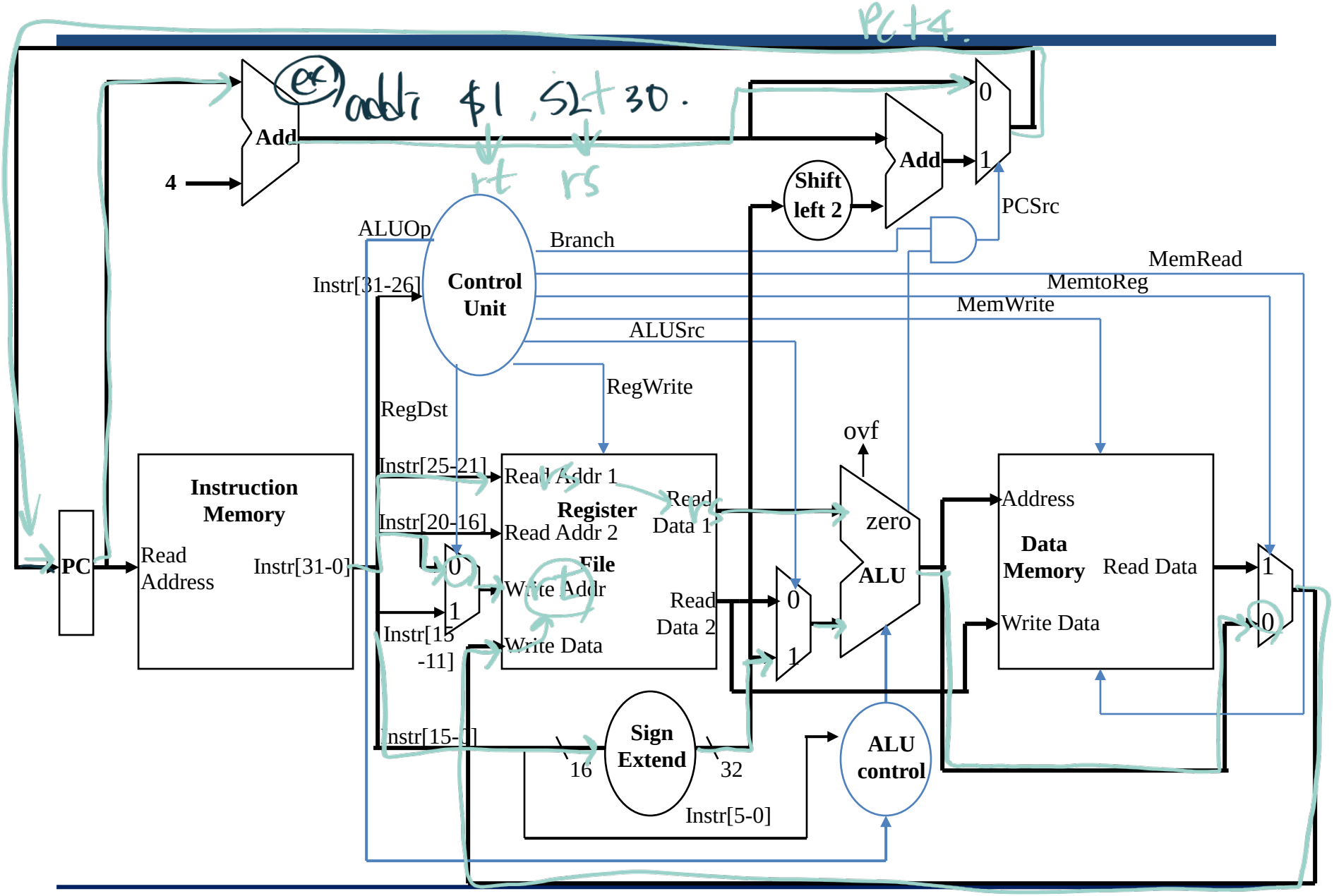
Single Cycle Datapath with Control Unit



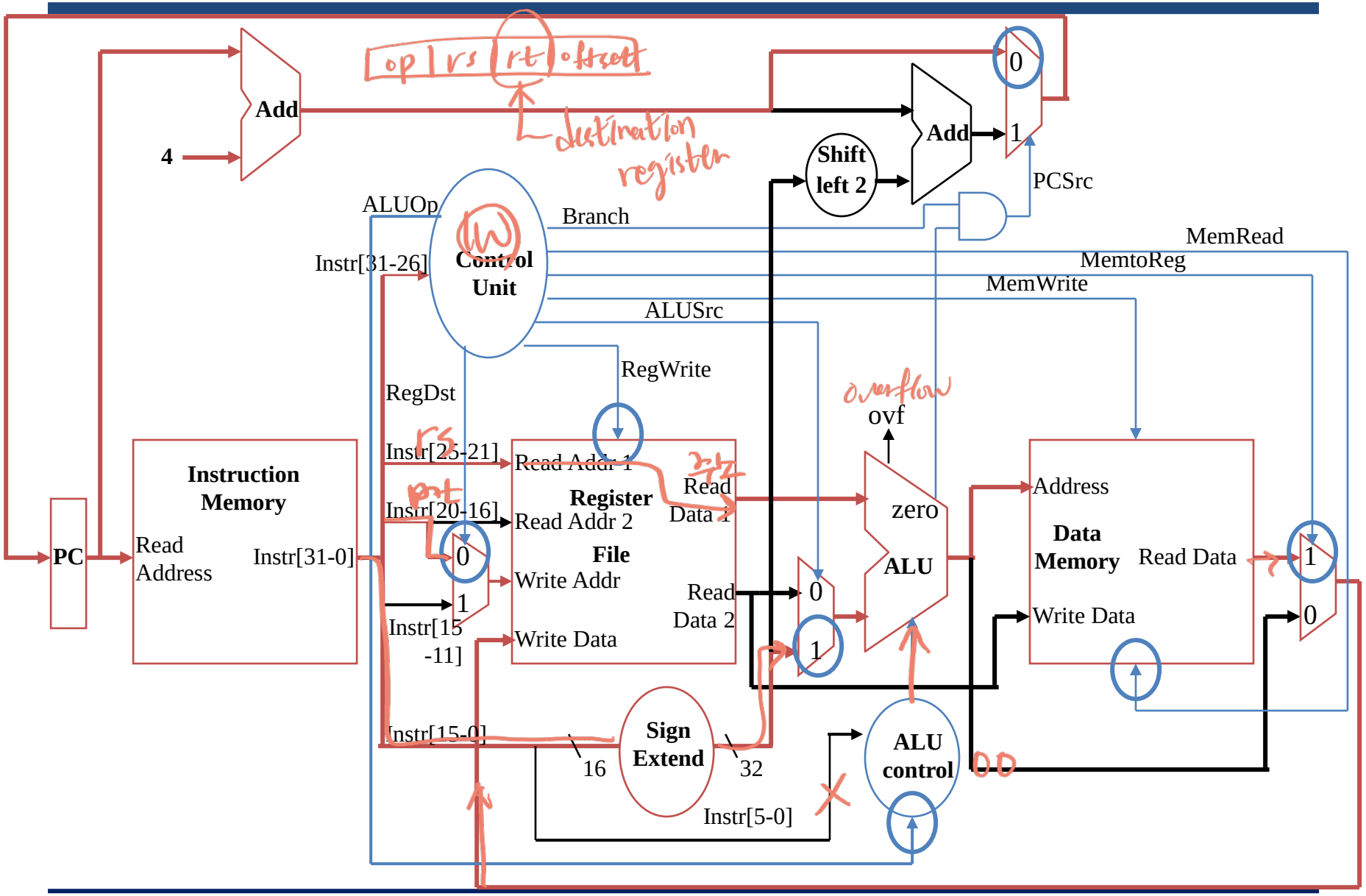
R-type Instruction Data/Control Flow



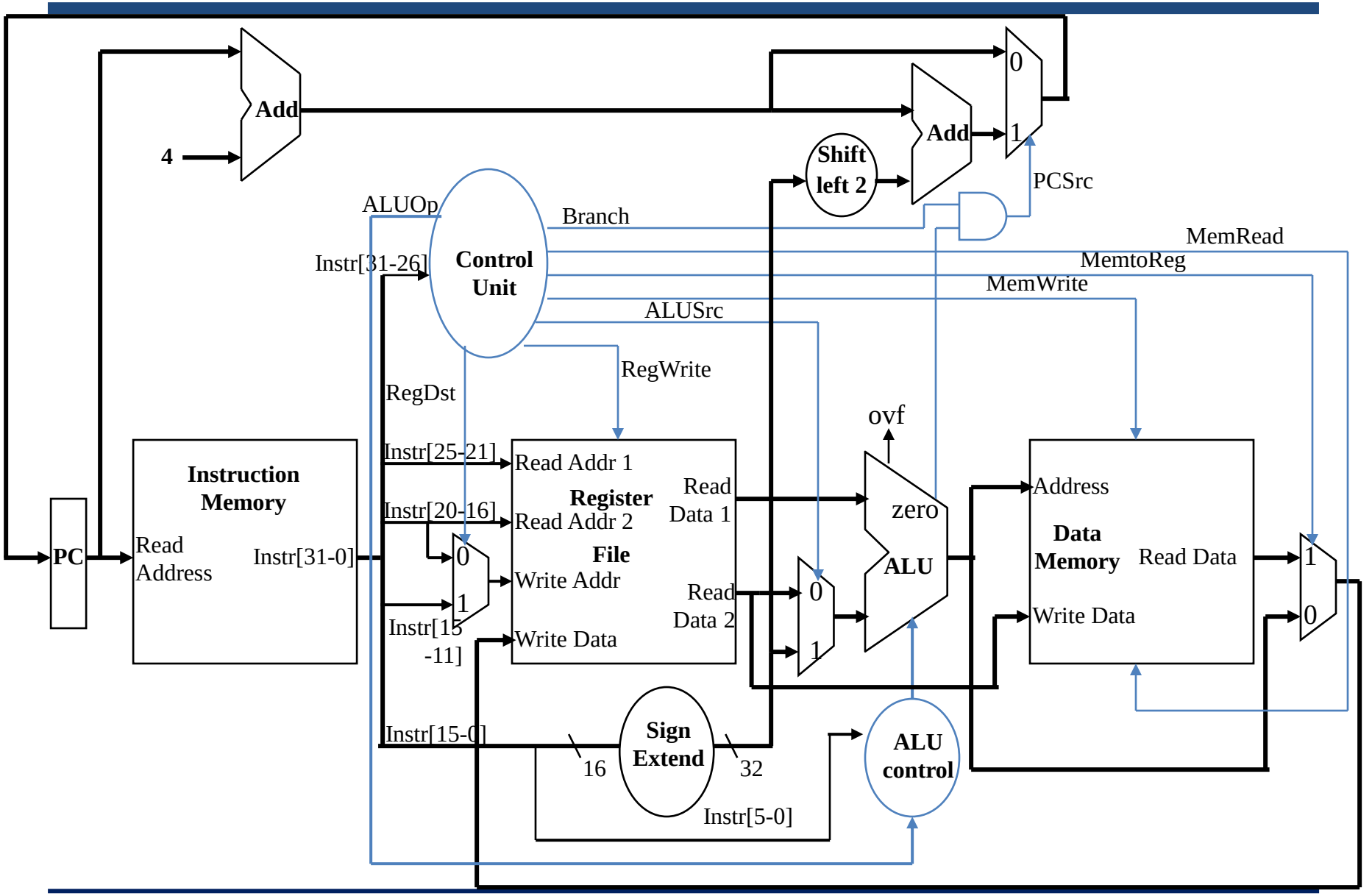
Load Word Data/Control Flow



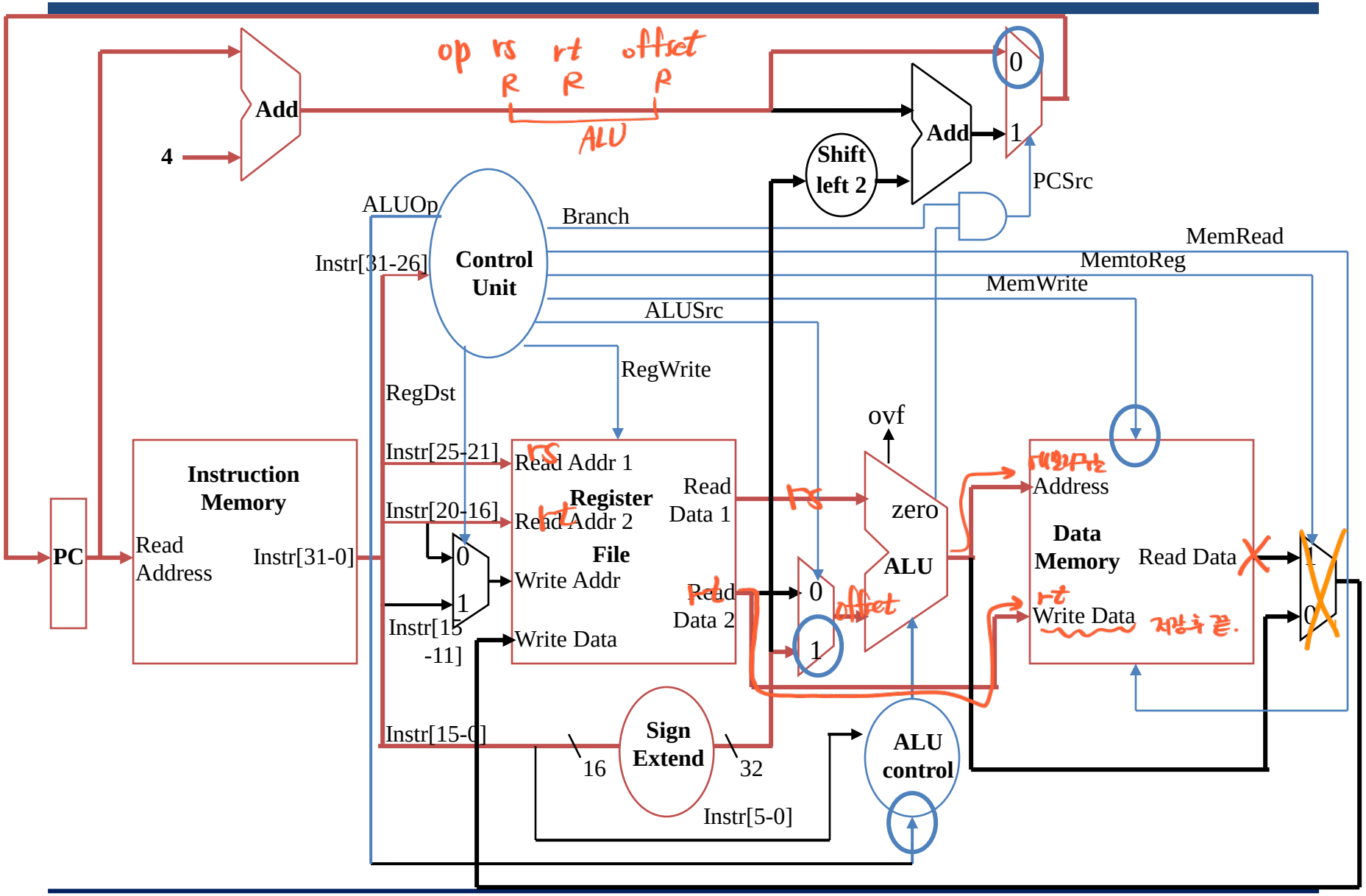
Load Word Data/Control Flow



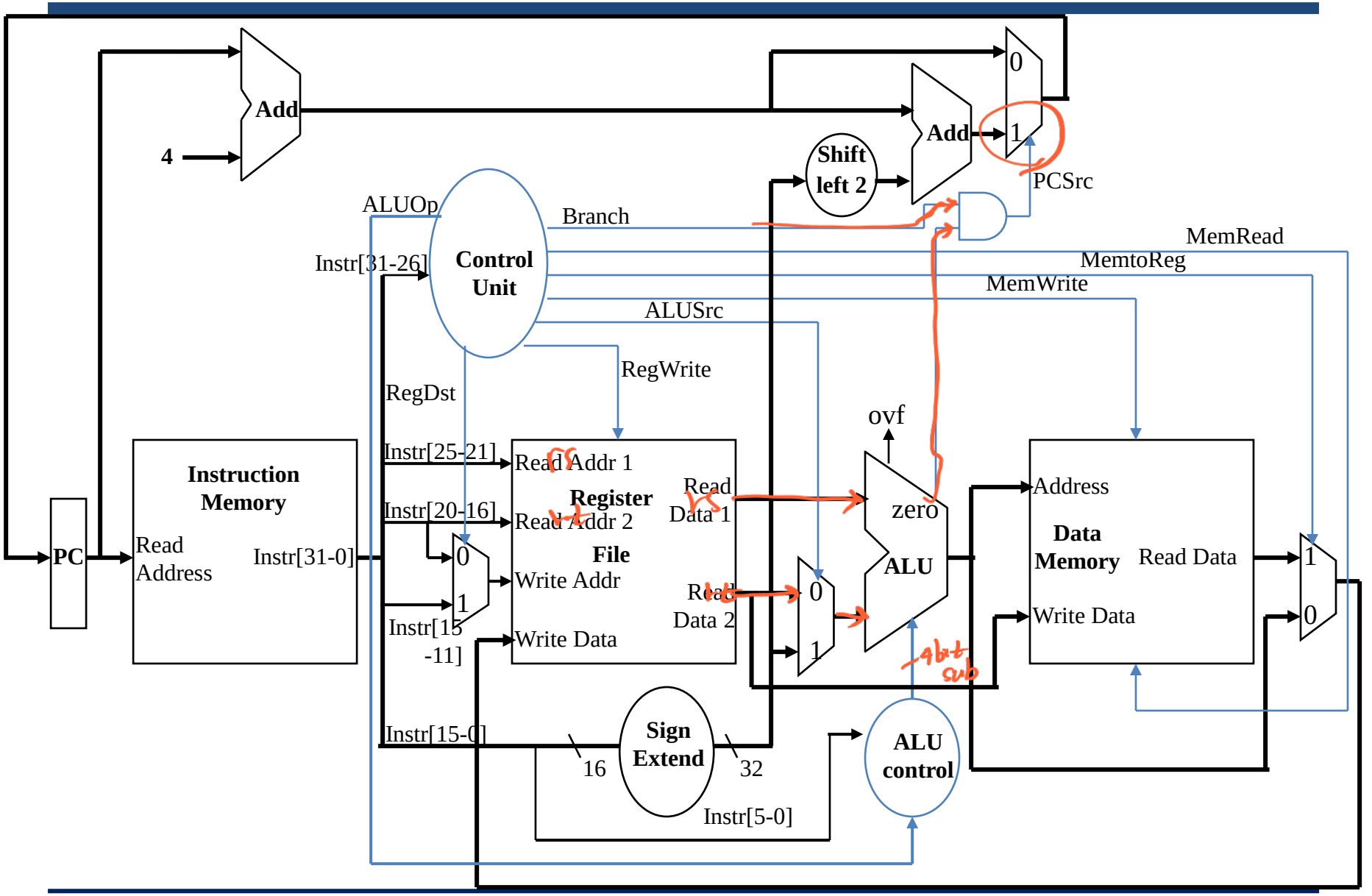
Store Word Data/Control Flow



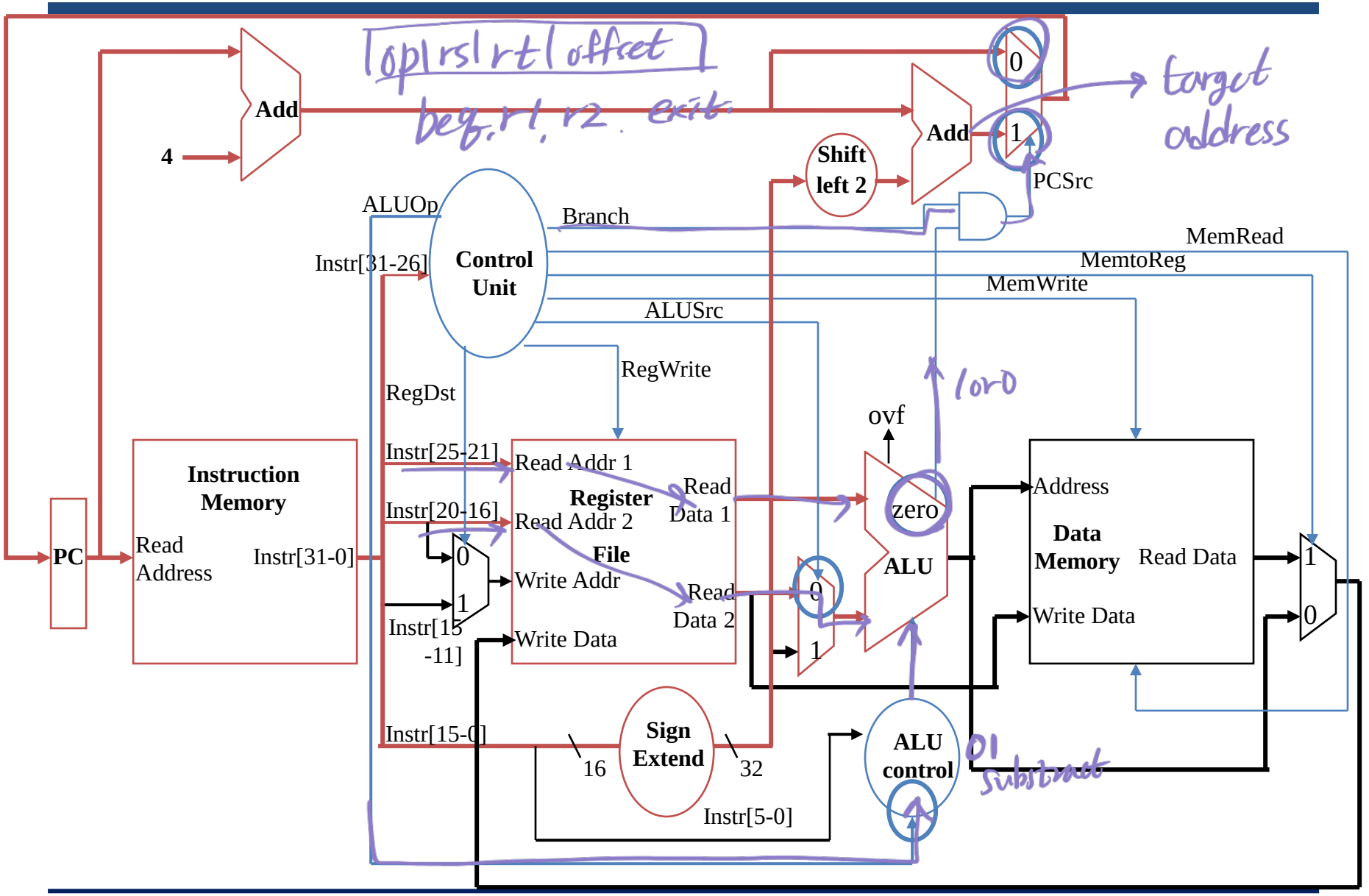
Store Word Data/Control Flow



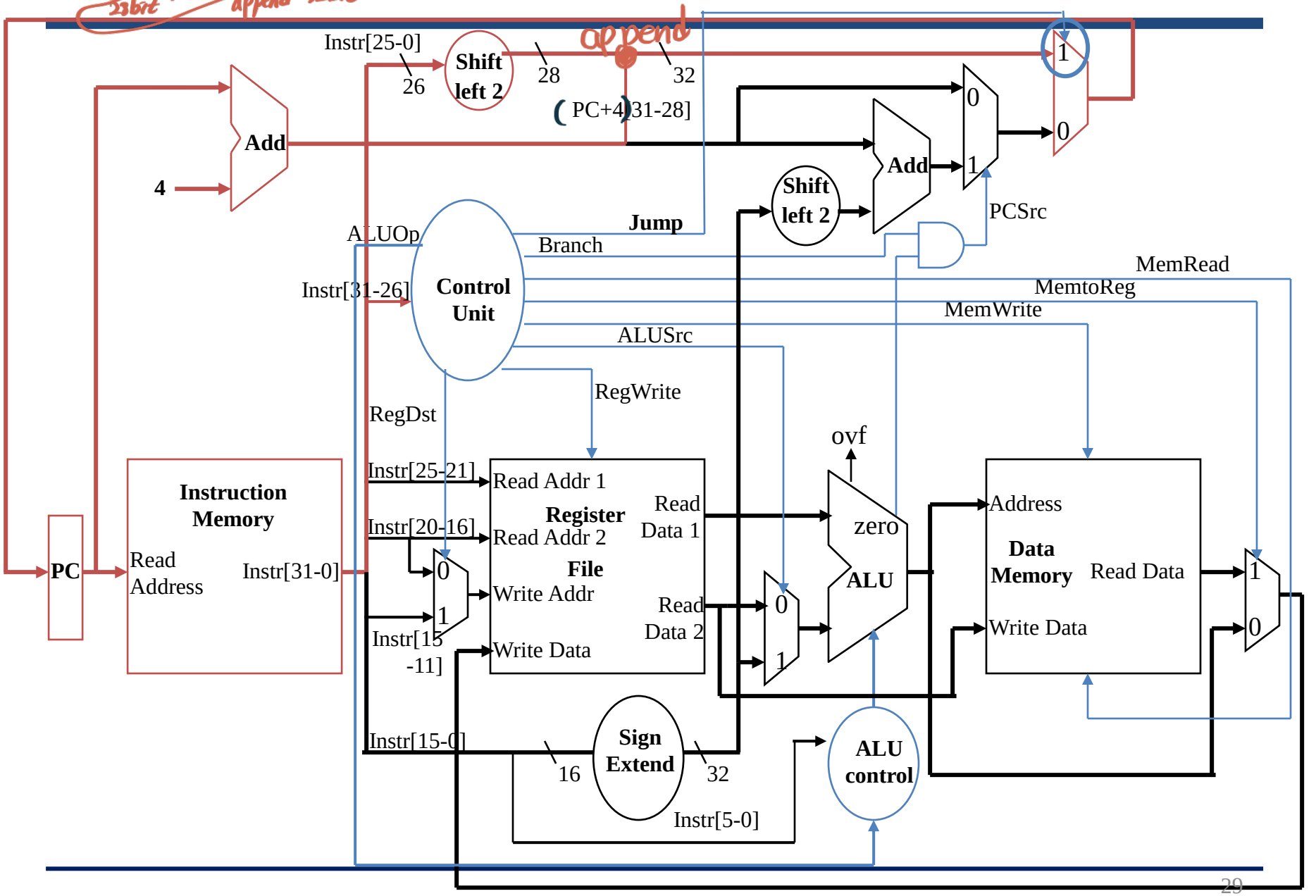
Branch Instruction Data/Control Flow



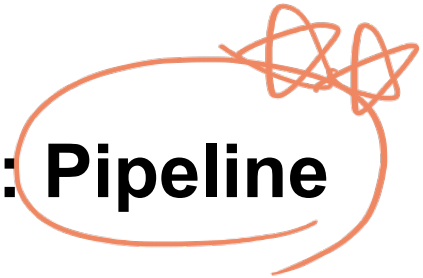
Branch Instruction Data/Control Flow



Adding the Jump Operation



CSE305 Computer Architecture



Processor II : Pipeline

Daehoon Kim

Department of EECS, DGIST

Instruction Critical Paths

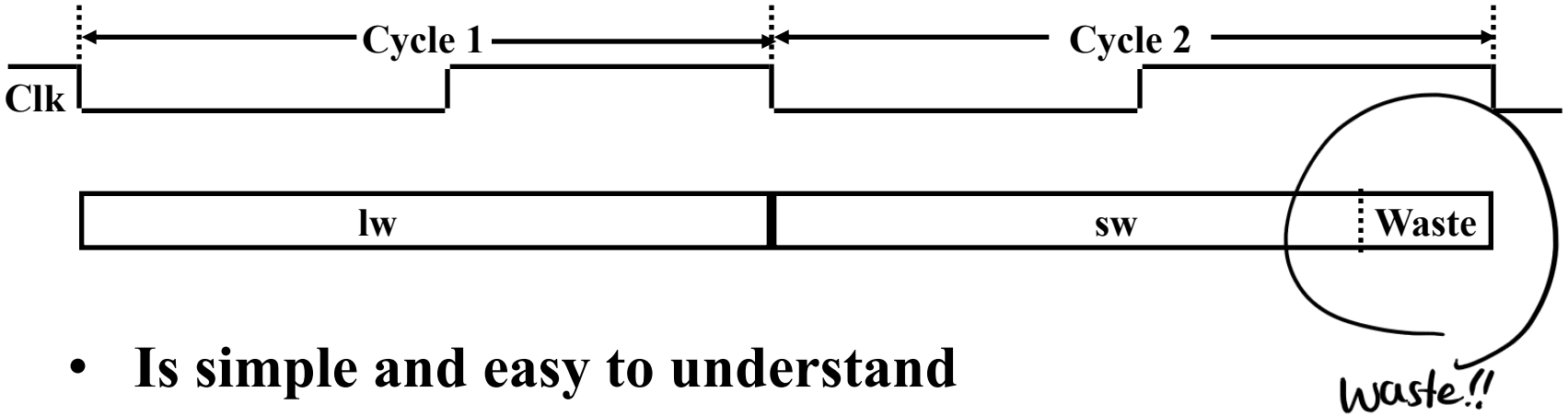
- **What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:**
  - **Instruction and data memory (200 ps)** 가정
  - **ALU and adders (200 ps)**
  - **Register File Access (reads or writes) (100 ps)**

cycle time 결정한 값은 800ps 임에 load까지 cover 가능.

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	200	100	200	X	100	600
load	200	100	200	200	100	800
store	200	100	200	200	X	700
beq	200	100	200	X	X	500
jump	200	X	X	✓	X	200

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - especially problematic for more complex instructions like floating point multiply



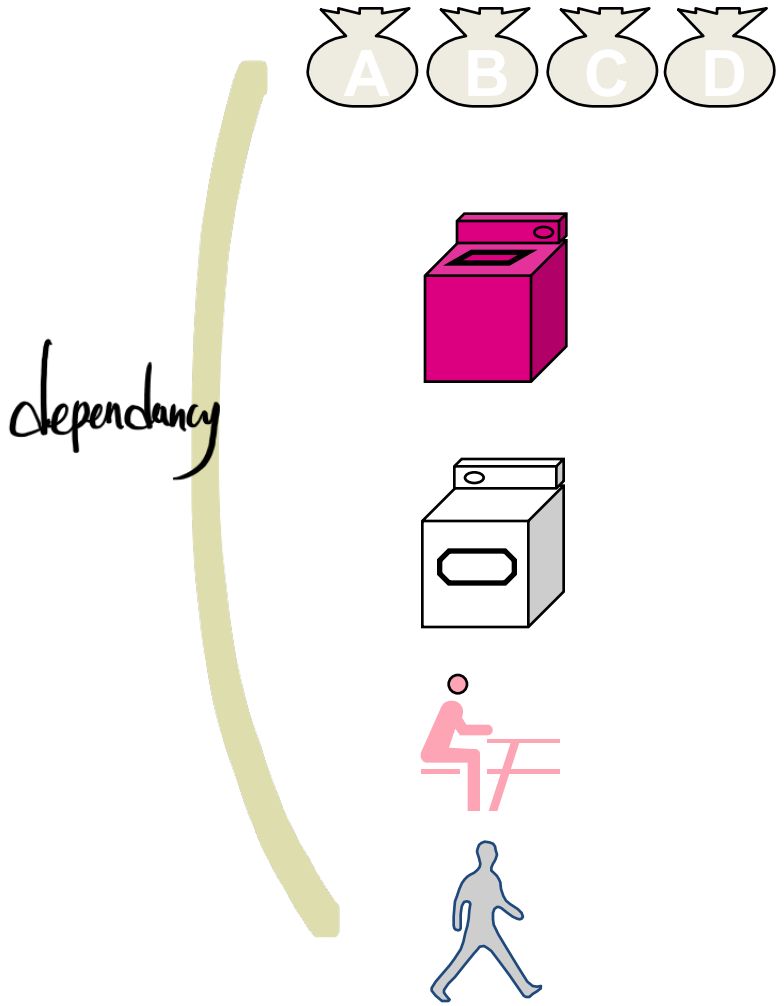
- Is simple and easy to understand

Performance Issues

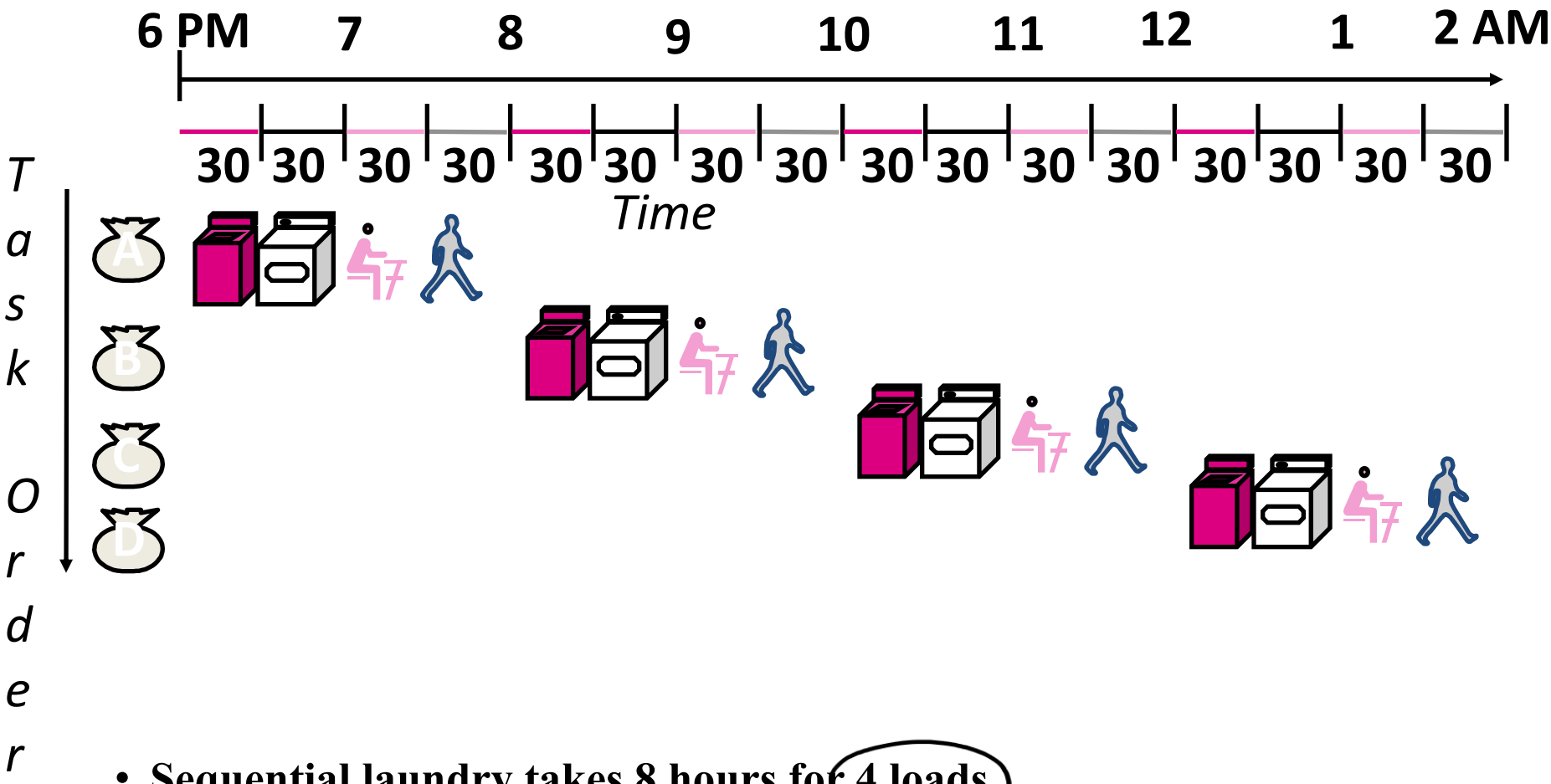
- **Longest delay determines clock period**
  - **e.g., lw determines clock period**
  - **Instruction memory → register file → ALU → data memory → register file**
- **Not feasible to vary period for different instructions**
- **We will improve performance by pipelining**

Pipelining is Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- "Folder" takes 30 minutes
- "Stasher" takes 30 minutes to put clothes into drawers



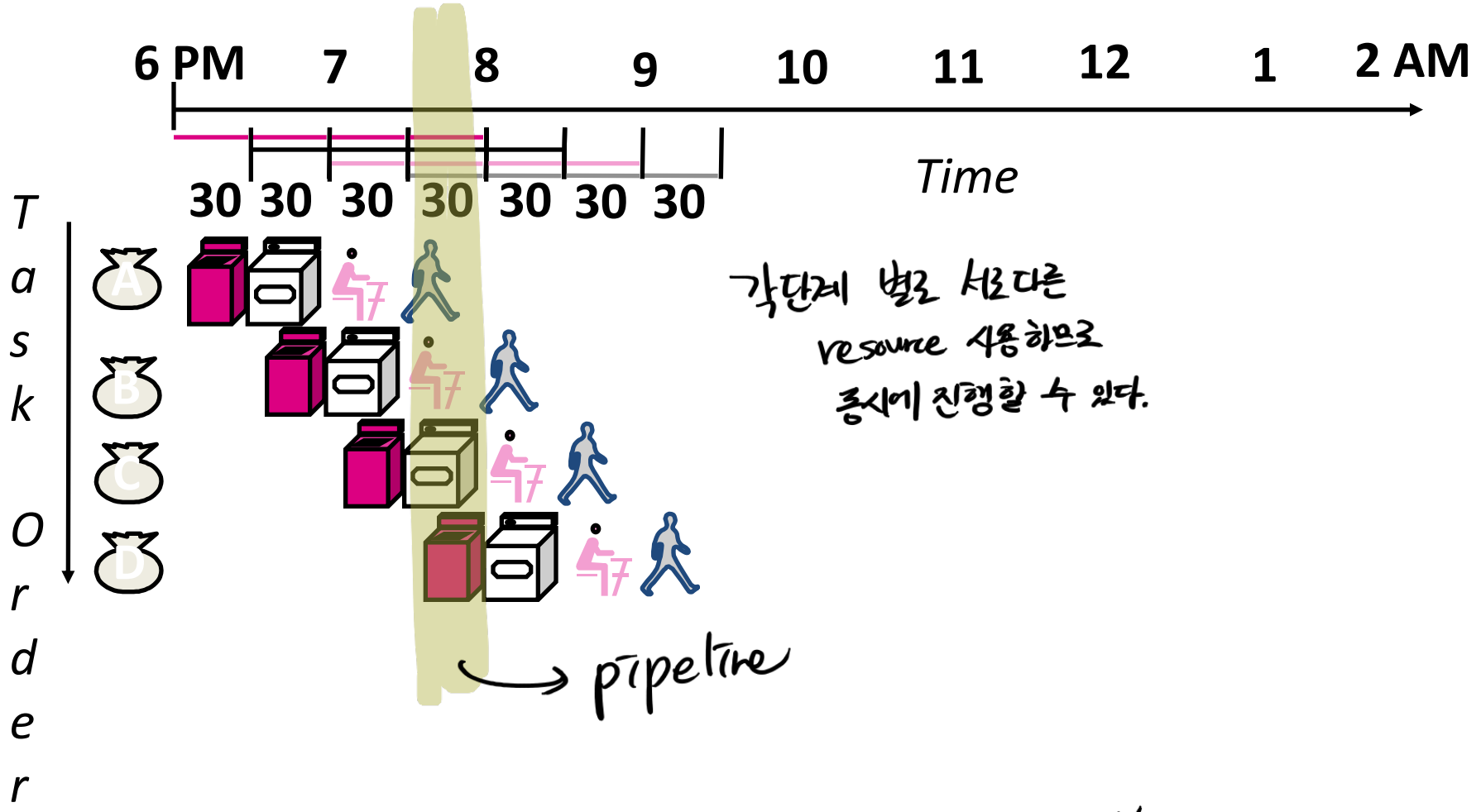
Sequential Laundry



- **Sequential laundry takes 8 hours for 4 loads**
- **If they learned pipelining, how long would laundry take?**

Slide courtesy of D. Patterson

Pipelined Laundry: Start work ASAP



- **Pipelined laundry takes 3.5 hours for 4 loads!**

Slide courtesy of D. Patterson

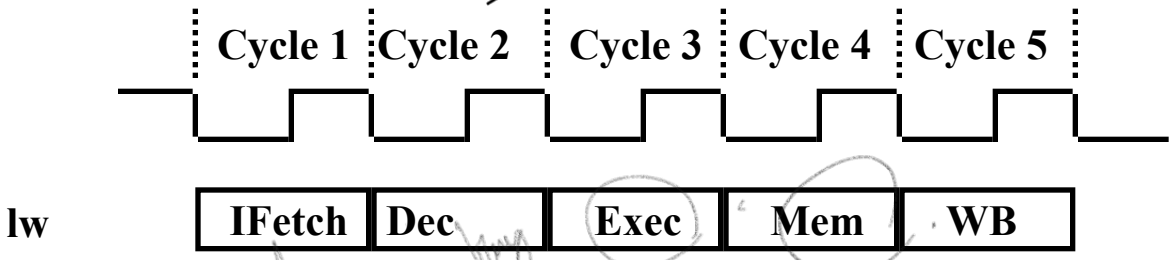
How Can We Make It Faster?

- Start fetching and executing the next instruction before the current one has completed
  - Pipelining – most of modern processors are pipelined for performance
  - Remember *the* performance equation:

$$\text{CPU time} = \text{CPI} * \text{CC} * \text{IC}$$

- Where does performance improvement of pipelining come from?
  - CPI w/ pipelining? almost same
  - IC w/ pipelining? same
  - CC w/ pipelining? N times faster!
- Under *ideal* conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages
  - A five stage pipeline is nearly five times faster because the CC is nearly five times faster

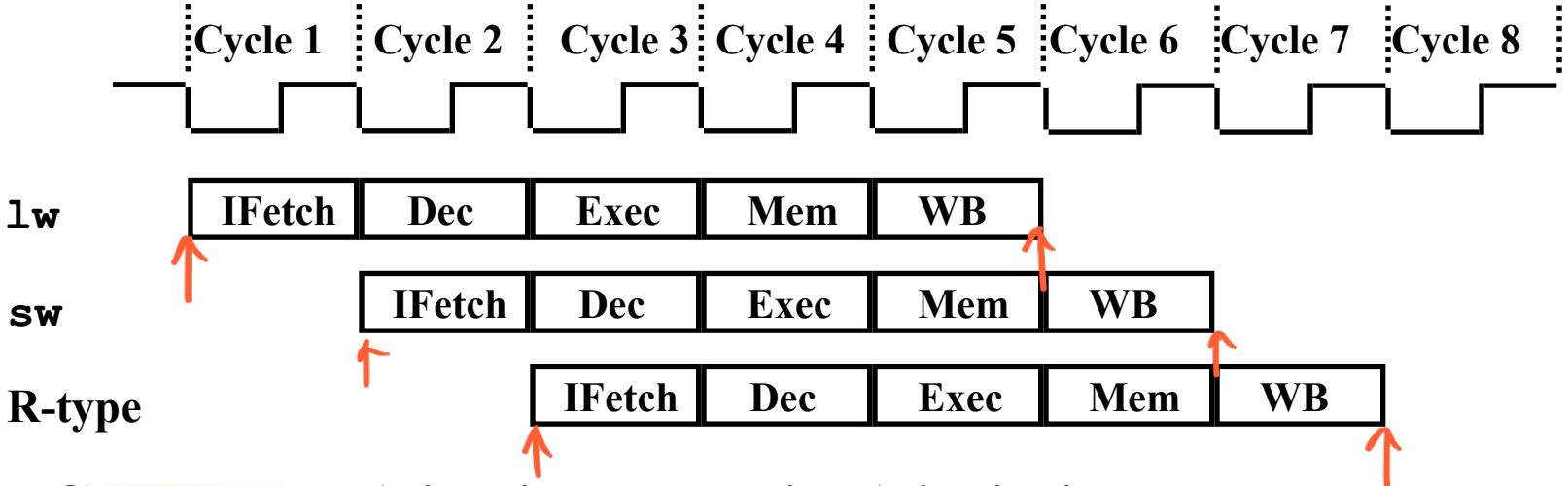
The Five Stages of Load Instruction



- **IFetch: Instruction Fetch and Update PC**
- **Dec: Registers Fetch and Instruction Decode**
- **Exec: Execute R-type; calculate memory address**
- **Mem: Read/write the data from/to the Data Memory**
- **WB: Write the result data into the register file**

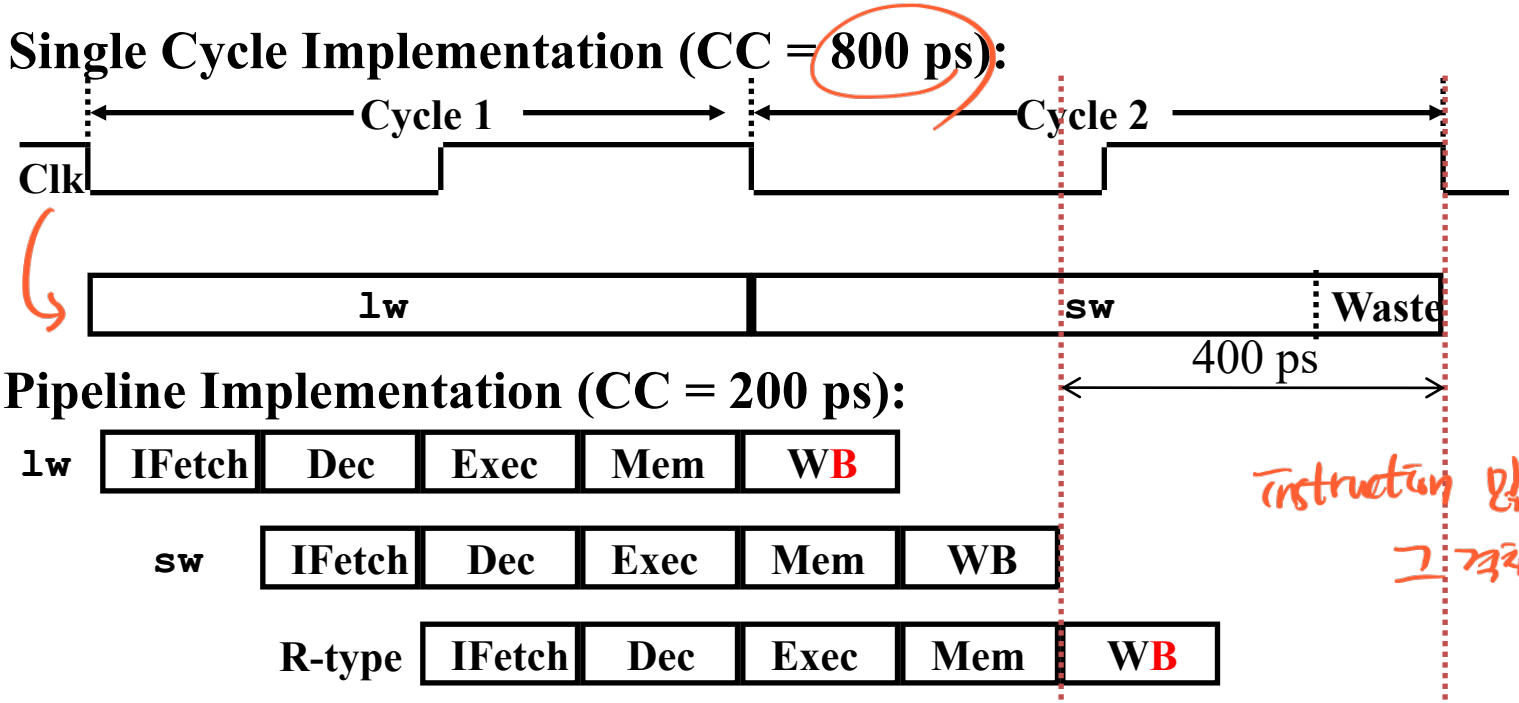
A Pipelined MIPS Processor

- Start the next instruction before the current one has completed
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced

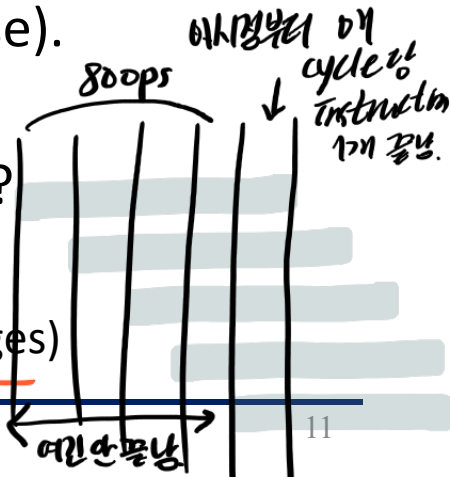


- Clock cycle (pipeline stage time) is limited by the slowest stage
  - for some stages don't need the whole clock cycle (e.g., WB)
  - for some instructions, some stages are wasted cycles (i.e., nothing is done during that cycle for that instruction)

Single Cycle versus Pipeline



- To complete an entire instruction in the pipelined case takes 1000 ps (as compared to 800 ps for the single cycle case). Why ? slowest stage time → 200ps
- How long does each take to complete 1,000,000 adds ?
  - Single cycle: 1,000,000 x 800 ps
  - Pipelined: 1,000,000 x 200 ps + 800 ps (to fill the pipeline stages)



Benefits of Pipelining

- **Before pipelining:**
  - **Throughput: 1 instruction per cycle**

$$t_{clk} = t_F + t_R + t_X + t_M + t_W$$

CPU TIME

=CPI + CC↓ + IC

  - **However, nearly 5x clock cycle time**

- **After pipelining (multiple instructions in pipe at one time)**
  - **Throughput: 1 instruction per cycle (*ideal pipeline*)**

$$t_{clk} = \max(t_F, t_R, t_X, t_M, t_W) + t_{latch}$$

latch에 값저장하는 시간

- **Suppose we execute 100 instructions**
  - **Single Cycle Machine: 45 ns/cycle x 1 CPI x 100 inst = 4500 ns**
  - **Ideal pipelined machine: 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns**

pipeline stage 값이 끝까지의 과정

Pipelining Lessons

- Pipelining doesn't help latency of single instruction, it helps *throughput* of entire workload
- Multiple instructions operating simultaneously using *different* resources
- *Potential speedup = number pipe stages*
- Time to "fill" pipeline reduces speedup

- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?

- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup

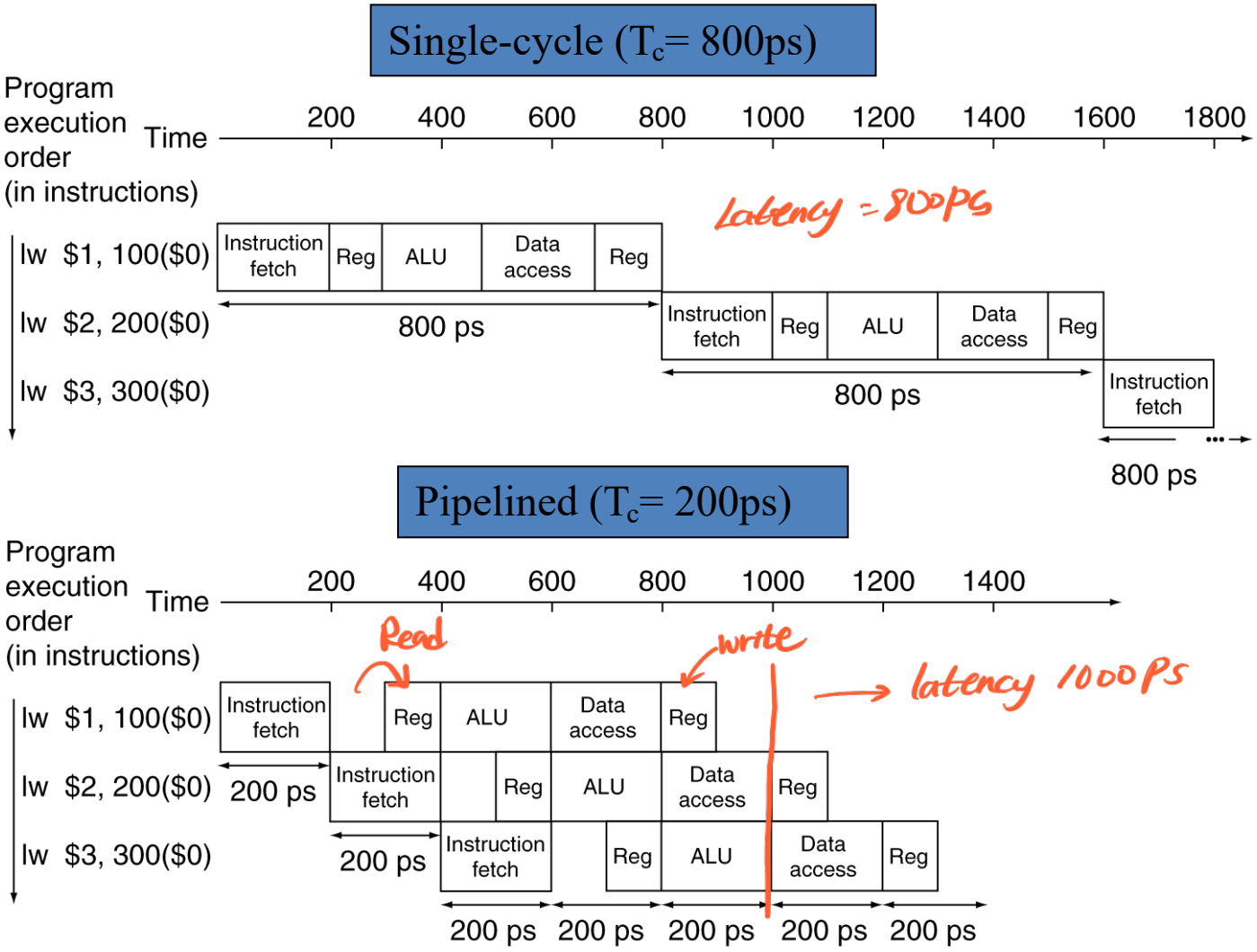
Pipeline Performance

- **Assume time for stages is**
  - **100ps for register read or write**
  - **200ps for other stages**
- **Compare pipelined datapath with single-cycle data path**

200ps

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps	X	700ps
R-format	200ps	100 ps	200ps	X	100 ps	600ps
beq	200ps	100 ps	200ps	X	X	500ps

Pipeline Performance

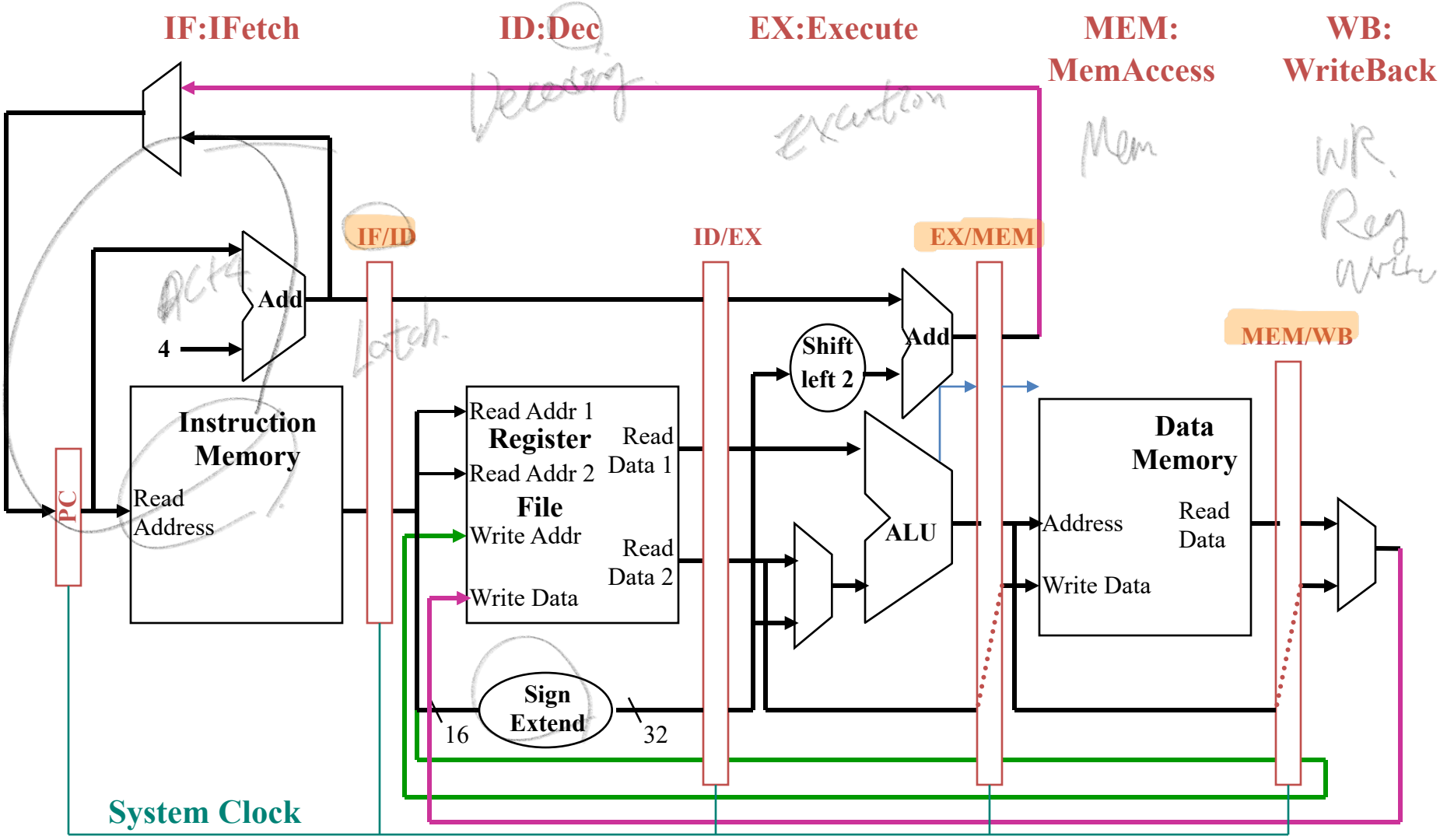


Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = Time between instructions$_{nonpipelined}$ / Number of stages
- If not balanced, speedup is less
- Speedup due to increased throughput (w/ reduced CC)
  - Latency (time for each instruction) does not decrease

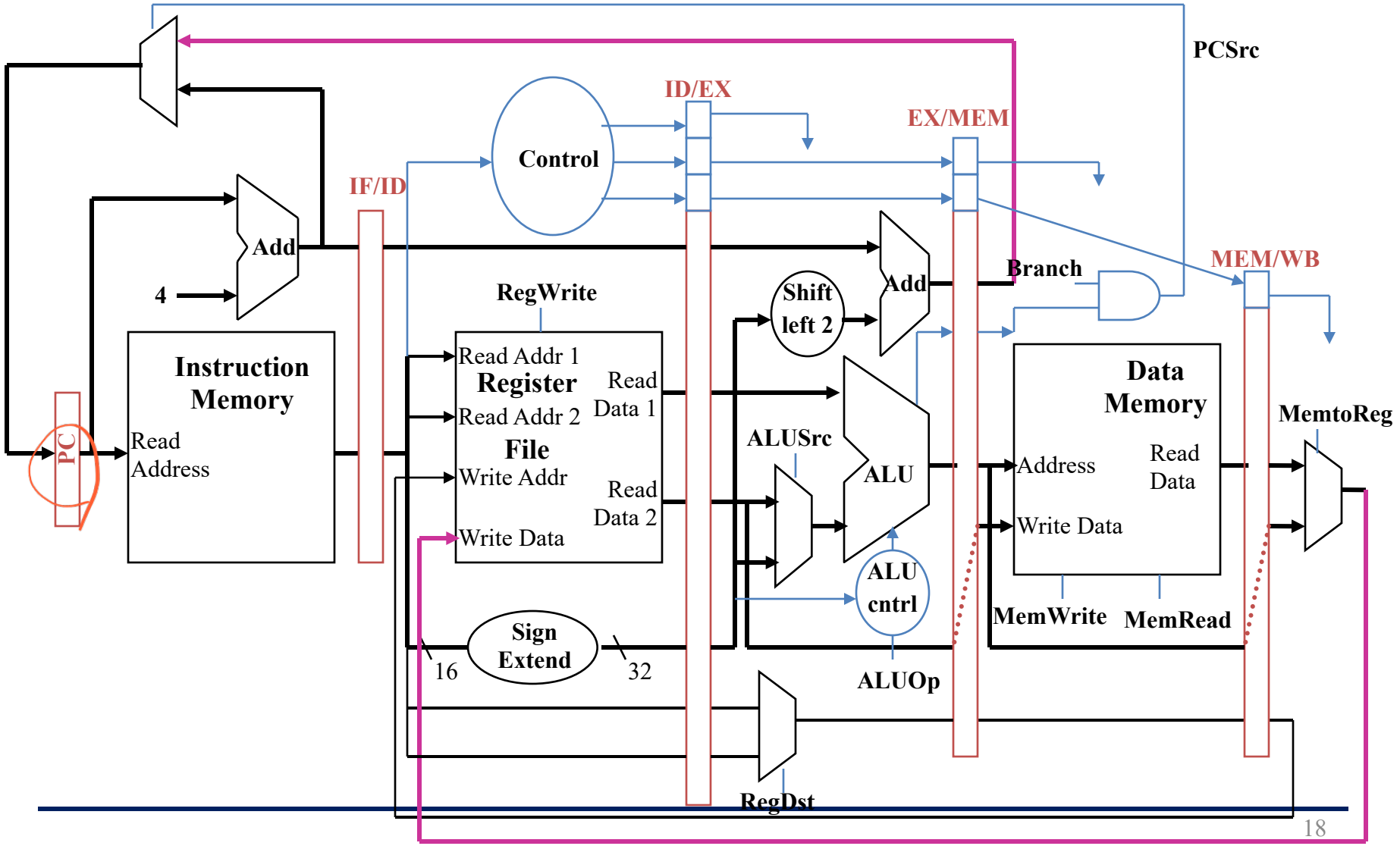
MIPS Pipeline Datapath Additions/Mods

- State registers between each pipeline stage to isolate them



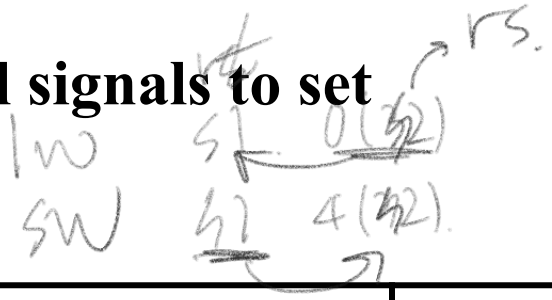
MIPS Pipeline Control Path Modifications

- All control signals can be determined during Decode
  - and held in the state registers between pipeline stages



Pipeline Control

- **IF Stage: read Instr Memory (always asserted) and write PC (on System Clock)**
- **ID Stage: no optional control signals to set**



	EX Stage				MEM Stage			WB Stage	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Brch	Mem Read	Mem Write	Reg Write	Mem toReg
R	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

ALUop

don't care

Why Pipeline? For Performance!