
CMPEN431

Computer Architecture

Fall 2017

Instructions:

Language of the Computer

Mahmut Taylan Kandemir (www.cse.psu.edu/~kandemir)

[Adapted from *Computer Organization and Design, 5th Edition*,
Patterson & Hennessy, © 2014, MK]

Course Administration

- ❑ Instructor: Mahmut Taylan Kandemir (kandemir@cse.psu.edu)
W321, Westgate Bldg
Office Hours: Tue-Thu 2PM-3PM
- ❑ TA: Huaipan Jiang
Office Hours: posted on Canvas
- ❑ URL: Canvas
- ❑ Text: **Required:** *Computer Org and Design*, 5rd Ed.,
Patterson & Hennessy, ©2014
- ❑ Slides: pdf on Canvas after the lecture
- ❑ ACK: Profs. Mary Jane Irwin, John Sampson

Evaluating ISAs

❑ Design-time metrics

- Can it be implemented, at what cost (design, fabrication, test, packaging), with what power, with what reliability?
- Can it be programmed? Ease of compilation?

❑ Static Metrics

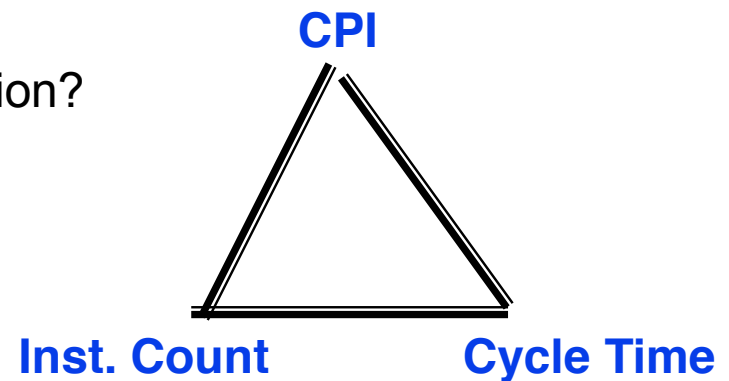
- How many bytes does the program occupy in memory?

❑ Dynamic Metrics

- How many instructions are executed? How many bytes does the processor fetch to execute the program?
- How many clocks are required per instruction?
- How "lean" (fast) a clock is practical?

Best Metric: Time to execute the program!

depends on the instructions set, the processor organization, and compilation techniques.



Below the Program

❑ High-level language program (in C)

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

one-to-many

C compiler

❑ Assembly language program (for MIPS)

```
swap:  sll    $2, $5, 2
        add    $2, $4, $2
        lw     $15, 0($2)
        lw     $16, 4($2)
        sw     $16, 0($2)
        sw     $15, 4($2)
        jr     $31
```

one-to-one

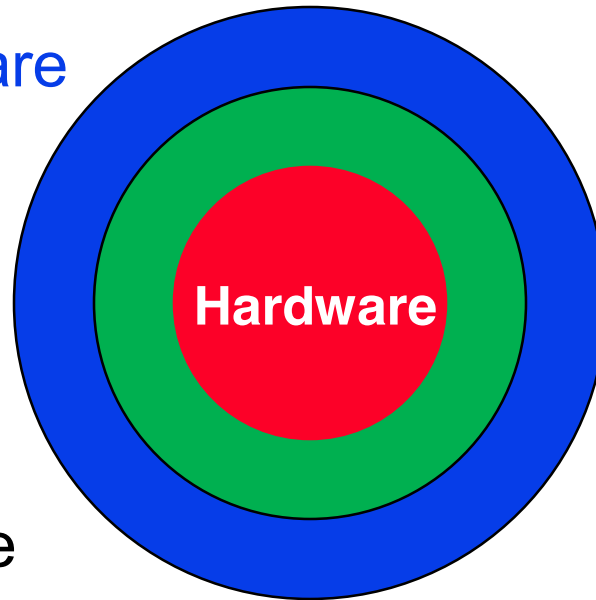
assembler

❑ Machine (object, binary) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
. . .
```

Below the Program, Con't

Applications software



Systems software

❑ System software

- Compiler – translate programs written in a high-level language (HLL, e.g., C) to machine code – CmpSc 471
 - Performs various code and data optimizations
- Operating system – supervising program that interfaces the user's program with the hardware (e.g., Linux, MacOS, Windows) – CmpSc 473
 - Handles basic input and output operations
 - Manages storage (disk) and memory (virtual memory)
 - Schedules tasks and provides for protected sharing of hardware resources (OS memory space)

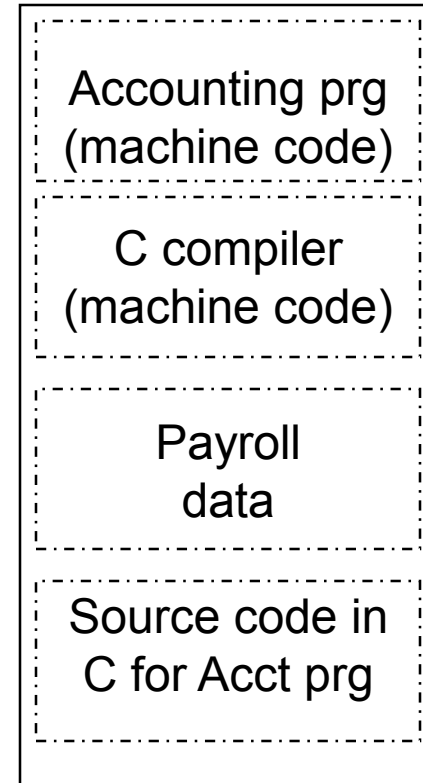
Two Key Principles of Machine Design

1. Instructions are represented as numbers and, as such, are indistinguishable from data
2. Programs are stored in alterable memory (that can be read or written to) just like data

Memory

□ Stored-program concept

- Programs can be shipped as files of binary numbers – **binary compatibility**
- Computers can inherit ready-made software provided they are compatible with an existing ISA – this has led the industry to align around a **small** number of ISAs



RISC vs CISC

- ❑ RISC = Reduced Instruction Set Computer
 - MIPS, SPARC, PowerPC, ARM (Cortex), etc.
- ❑ CISC = Complex Instruction Set Computer
 - X86 is the only surviving example
- ❑ Goals in the 1980s – reduce design time, faster/smaller implementation, ISA processor/compiler co-design
- ❑ ISAs are measured by how well compilers use them, not by how well or how easily assembly language programmers use them
- ❑ There are (or, at least, it's believed there are) many old and useful programs that only exist as machine code, so supporting old ISAs has economic value

MIPS (RISC) Design Principles – Part 1

简单源于规律性

□ Simplicity favors regularity

- Regularity makes implementation simpler
- Simplicity enables higher performance at lower cost
 - fixed size instructions (32-bits (now 64-bits)), small number of instruction formats (three for MIPS), opcode in a fixed location (the first 6 bits for MIPS), etc.

□ Smaller is faster

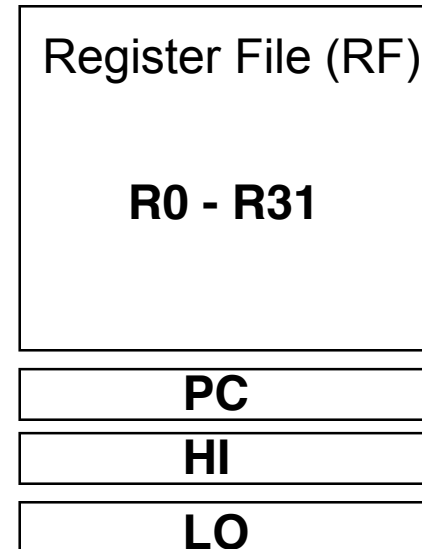
- Smaller ISA reduces design and implementation costs (and power?), chip sizes, etc.
- Faster
 - limited instruction set and formats, **load-store architecture**
 - <http://www.arm.com/products/processors/instruction-set-architectures/index.php>
 - limited number of registers in the register file (RF)
 - limited number of memory addressing modes
 - Memory address = register value + constant

MIPS-32 ISA

❑ Instruction Categories

- Computational
- Load/Store
- Jump and Branch
- Floating Point
 - coprocessor
- Memory Management
- Special

Registers



3 Instruction Formats: **all 32 bits wide**



Review: The Fetch/Execute Cycle

- ❑ Memory stores both instruction and data (object code and data bits ... just bits)

1. Instruction is fetched from memory at the address indicated by the Program Counter (PC)
2. Control unit decodes the instruction, generates signals to other components so the instruction can be executed
 1. Data is read from the RF or, if necessary, from memory
 2. Datapath executes the instruction as directed by the Control
 3. Data is written to the RF of, if necessary, to memory
3. Control updates the PC which specifies the next instruction to fetch and then execute

MIPS Arithmetic Instructions

- ❑ MIPS assembly language arithmetic statement

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

```
sub    $t0, $s1, $s2
```

- ❑ Each arithmetic instruction performs **one** operation

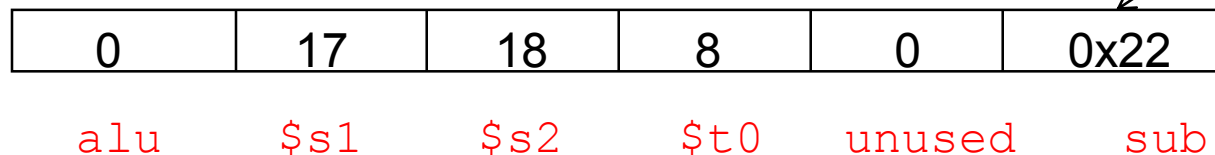
操作数

- ❑ Each specifies exactly **three** operands that are all contained in the datapath's RF (\$t0, \$s1, \$s2)

destination ← source1 **op** source2

- ❑ Instruction Format (**R** format)

hexidecimal (4-bits
per hex digit (0 to f))



MIPS Instruction Fields

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

op	rs	rt	rd	shamt	funct
----	----	----	----	-------	-------

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

MIPS (RISC) Design Principles – Part 2



❑ Make the common case fast

- Find the biggest impact on performance
 - E.g., accessing registers is fast, memory is slow
- Which are the “common cases”? Are they the same for all programs? Will they be the same in the future?
 - arithmetic operands in the RF (load-store machine)
 - allow instructions to contain immediate operands (small constants), otherwise have to bring the constants in from memory, store them in the RF, and access them from there

❑ Good design demands good compromises

- Evaluate the many options, determine their impact on performance (IPC?, IC?, clock rate?), make a reasonable choice that does *not* limit future extensions
 - three instruction formats, as similar as possible
 - only two branch instructions (`beq`, `bne`) with a way to do many more with the `slt` “set up” instruction

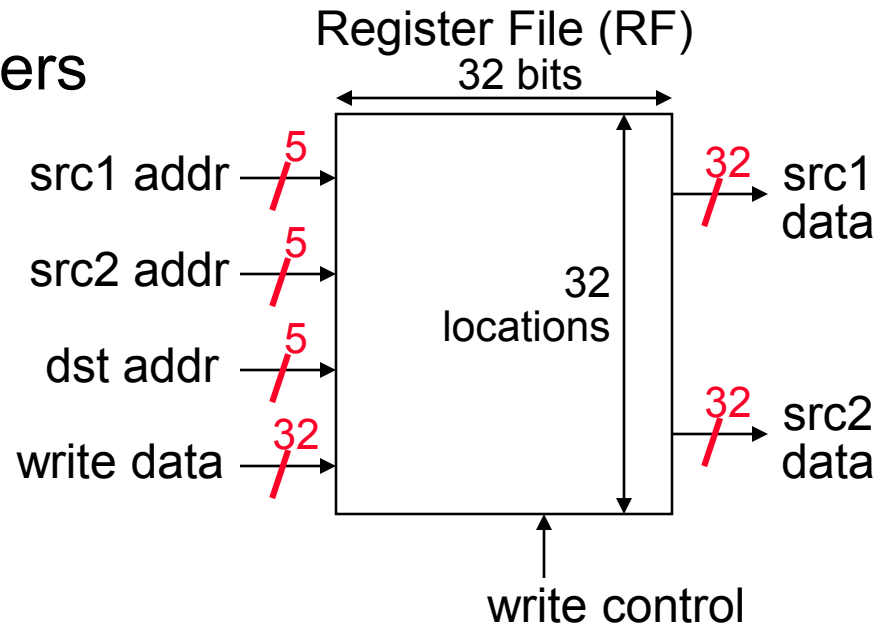
MIPS Register File (RF)

❑ Holds thirty-two 32-bit registers

- Two read ports and
- One write port

❑ Registers are

- **Faster** than main memory
 - But RFs with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
 - Increasing number of read/write ports impacts speed quadratically
 - Large RFs consume more power
- Improves code density (a register is named with fewer bits than a memory location)
- Easier for a compiler to use
 - e.g., $(A*B) - (C*D) - (E*F)$ can do multiplies in any order vs. stack



Aside: 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

MIPS Memory Access Instructions

- ❑ MIPS has two basic **data transfer** instructions for accessing memory

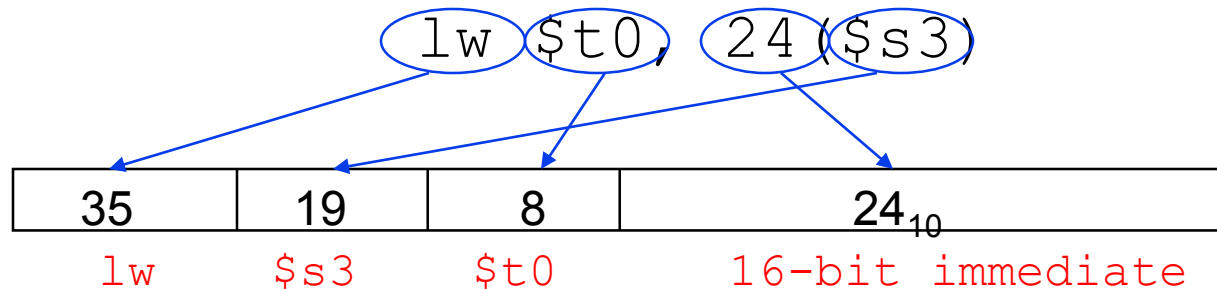
lw \$t0, 4 (\$s3) #load **word** from memory

sw \$t0, 8 (\$s3) #store **word** to memory

- ❑ The data is loaded into (**lw**) or stored from (**sw**) a register in the register file – a 5 bit address
- ❑ The memory address – a 32 bit address – is formed by adding the contents of the **base address register** to the sign-extended **offset** value
 - The offset is a 16-bit 2's complement number, so access is limited to memory locations within a region of $\pm 2^{13}$ (8,192) words or $\pm 2^{15}$ (32,768) bytes of the address in the base register

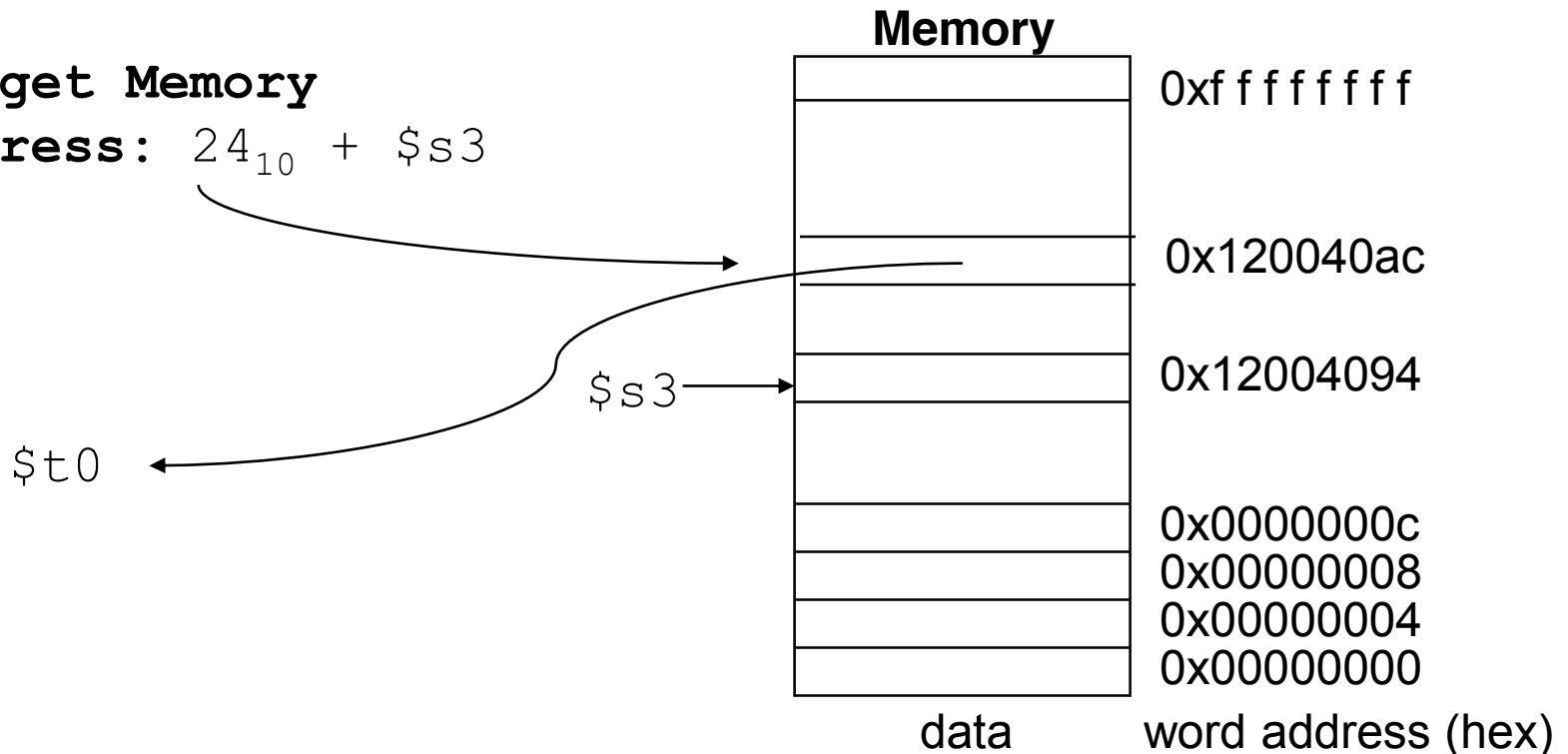
Machine Language - Load Instruction

❑ Load/Store Instruction Format (I format):



Target Memory

Address: $24_{10} + \$s3$



Aside: Loading and Storing Bytes

- ❑ MIPS provides special instructions to move bytes

`lb` `$t0, 1($s3)` #load `byte` from memory

`sb` `$t0, 6($s3)` #store `byte` to memory

<code>sb</code>	<code>\$s3</code>	<code>\$t0</code>	<code>6</code>
0x28	19	8	16 bit offset

- ❑ What 8 bits get loaded and stored?

- load byte places the byte from memory in the rightmost 8 bits of the destination register
 - what happens to the other bits in the register?
- store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
 - what happens to the other bits in the memory word?

MIPS Immediate Instructions

❑ Small constants are used often in typical code



❑ Possible approaches?

1. put “typical constants” in memory and load them into the RF
2. create hard-wired registers (like \$zero) for constants like 1
3. **have special instructions that contain constants !**

```
addi $sp, $sp, 4      # $sp = $sp + 4
slti $t0, $s2, 15     # $t0 = 1 if $s2 < 15
                        # otherwise $t0 = 0
```

❑ Machine format (I format):

slti	\$s2	\$t0	15
0x0a	18	8	0x0f

❑ The constant is kept **inside** the instruction itself!

- Immediate format **limits** values to the range $+2^{15}-1$ to -2^{15}

Aside: How About Larger Constants?

- ❑ We'd also like to be able to load a 32 bit constant into a register, for this we must use two instructions
- ❑ a new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

16	0	8	1010101010101010 ₂
----	---	---	-------------------------------

- ❑ Then must get the lower order bits right, use

```
ori $t0, $t0, 1010101010101010
```

1010101010101010	0000000000000000
------------------	------------------

0000000000000000	1010101010101010
------------------	------------------

1010101010101010	1010101010101010
------------------	------------------

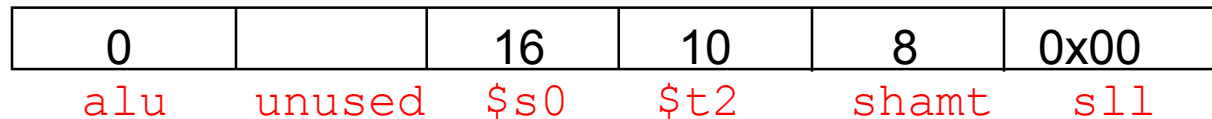
MIPS Shift Operations

- ❑ Need operations to **pack** and **unpack** 8-bit characters into 32-bit words
- ❑ Shifts move all the bits in a word left or right

sll \$t2, \$s0, 8 # \$t2 = \$s0 << 8 bits

srl \$t2, \$s0, 8 # \$t2 = \$s0 >> 8 bits

- ❑ Instruction Format (**R** format)



- ❑ Such shifts are called **logical** because they fill with **zeros**
 - Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or **31 bit positions**

MIPS Logical Operations

- There are a number of **bit-wise** logical operations in the MIPS ISA

`and $t0, $t1, $t2 #$t0 = $t1 & $t2`

`or $t0, $t1, $t2 #$t0 = $t1 | $t2`

`nor $t0, $t1, $t2 #$t0 = not($t1 | $t2)`

- Instruction Format (**R** format)

0	9	10	8	0	0x24
---	---	----	---	---	------

alu **\$t1** **\$t2** **\$t0** **unused** **and**

`andi $t0, $t1, 0xFF00 #$t0 = $t1 & ff00`

`ori $t0, $t1, 0xFF00 #$t0 = $t1 | ff00`

- Instruction Format (**I** format)

0x0d	9	8	0xff00
------	---	---	--------

MIPS Control Flow Instructions

❑ MIPS conditional branch instructions:

```
bne $s0, $s1, Lbl1 #go to Lbl1 if $s0≠$s1
beq $s0, $s1, Lbl1 #go to Lbl1 if $s0=$s1
```

● Ex: if (i==j) h = i + j;

```
          bne $s0, $s1, Lbl1
          add $s3, $s0, $s1
Lbl1:     ...
```



❑ Instruction Format (I format):

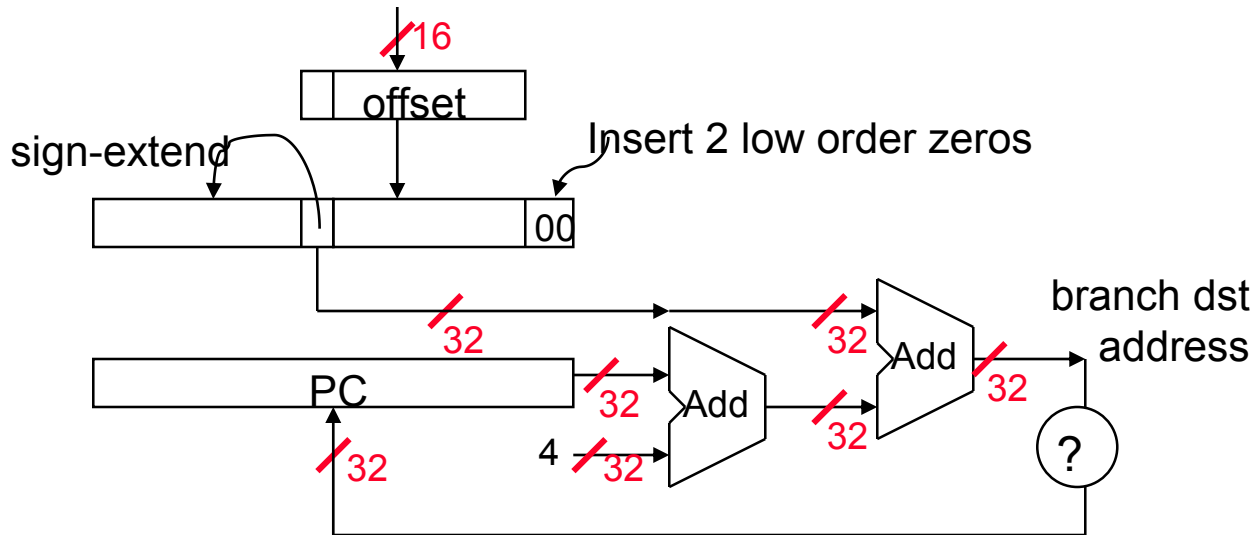
0x05	16	17	16 bit offset
bne	\$s0	\$s1	16-bit value

❑ How is the branch destination address specified?

Specifying Branch Destinations

- ❑ Use a register (like in lw and sw) added to the 16-bit offset
 - which register? Instruction Address Register (the PC)
 - Its use is automatically **implied** by instruction
 - PC gets updated (PC+4) during the **fetch** cycle so that it is holding the address of the next instruction when the branch executes

from the low order 16 bits of the branch instruction



- limits the branch distance to -2^{15} to $+2^{15}-1$ (word) instructions from the (instruction after the) branch instruction, but most branches are local anyway

In Support of Branch Instructions

- ❑ We have `beq`, `bne`, but what about other kinds of branches (e.g., branch-if-less-than)? For this, we need yet another instruction, `slt`

- ❑ Set on less than instruction:

```
slt $t0, $s0, $s1    # if $s0 < $s1    then
                      # $t0 = 1         else
                      # $t0 = 0
```

- ❑ Instruction format (**R** format):

0	16	17	8		0x2a
---	----	----	---	--	------

- ❑ Alternate versions of `slt`

```
slti $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
sltu $t0, $s0, $s1    # if $s0 < $s1 then $t0=1 ...
sltiu $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
```

Aside: More Branch Instructions

- ❑ Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to **create** other conditions

- less than `blt $s1, $s2, Label`

`slt $at, $s1, $s2` `#$at set to 1 if`

`bne $at, $zero, Label` `#$s1 < $s2`

- less than or equal to `ble $s1, $s2, Label`
- greater than `bgt $s1, $s2, Label`
- great than or equal to `bge $s1, $s2, Label`

- ❑ Such branches are included in the instruction set as pseudo instructions -- recognized (and expanded) by the assembler

- Its why the assembler needs a reserved register (`$at`)

Aside: Branching Far Away

- ❑ What if the branch destination is further away than can be captured in 16 bits?
- ❑ The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition

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

becomes

```
bne    $s0, $s1, L2
```

```
j      L1
```

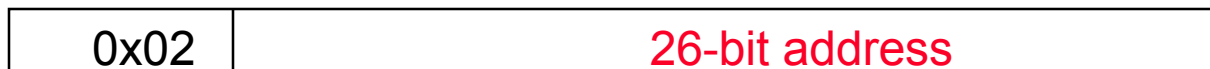
```
L2 :
```

Other Control Flow Instructions

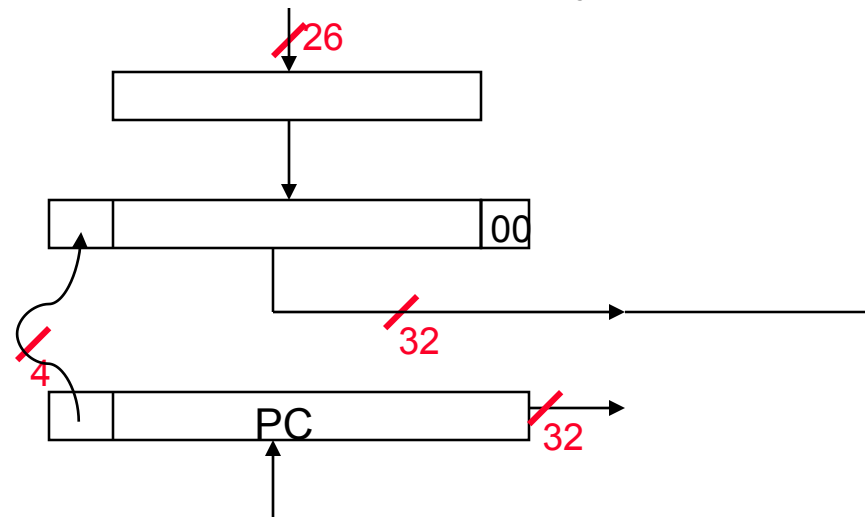
- ❑ MIPS also has an unconditional branch instruction or **jump** instruction:

j label #go to label

- ❑ Instruction Format (**J** Format):



from the low order 26 bits of the jump instruction



Instructions for Accessing Procedures

- ❑ MIPS **procedure call** instruction:

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

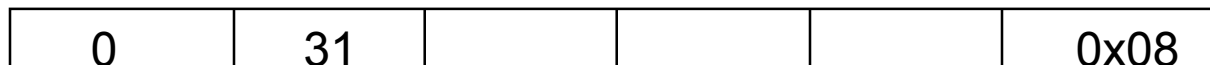
- ❑ Saves PC+4 in register \$ra to have a link to the next instruction for the procedure return
- ❑ Machine format (**J** format):



- ❑ Then can do procedure **return** with a jump register instr

```
jr    $ra                #return
```

- ❑ Instruction format (**R** format):



For Later: Atomic Exchange Instructions

❑ Hardware support for synchronization mechanisms

- Avoid **data races** where the results of the program can change depending on the relative ordering of events
- Two memory accesses from different threads/cores to the same memory (cache) location, and at least one is a write – which goes first?

❑ **Atomic exchange** (atomic swap, atomic read/write)



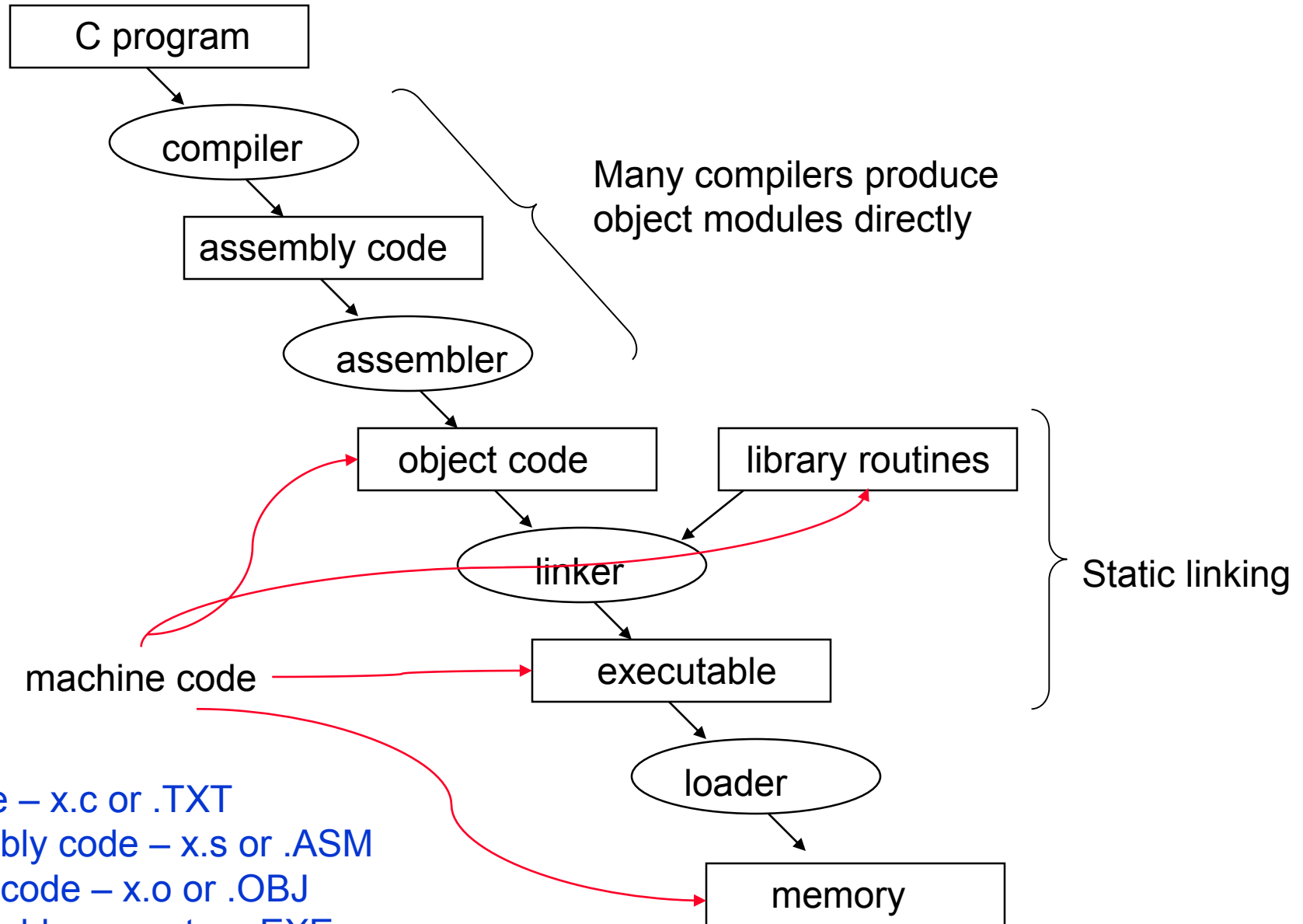
- Interchange a value in a register with a value in memory **atomically**, i.e., as one indivisible operation
- Logically requires both a memory read and a memory write in a single, uninterruptable instruction. An alternative is to have a pair of specially configured instructions where no other access to the location is allowed between the read and the write.

MIPS Instruction Classes Distribution

- ❑ Frequency of MIPS instruction classes for SPEC2006

Instruction Class	Frequency	
	SPECint	SPECfp
Arithmetic	16%	48%
Data transfer	35%	36%
Logical	12%	4%
Cond. Branch	34%	8%
Jump	2%	0%

The C Code Translation Hierarchy



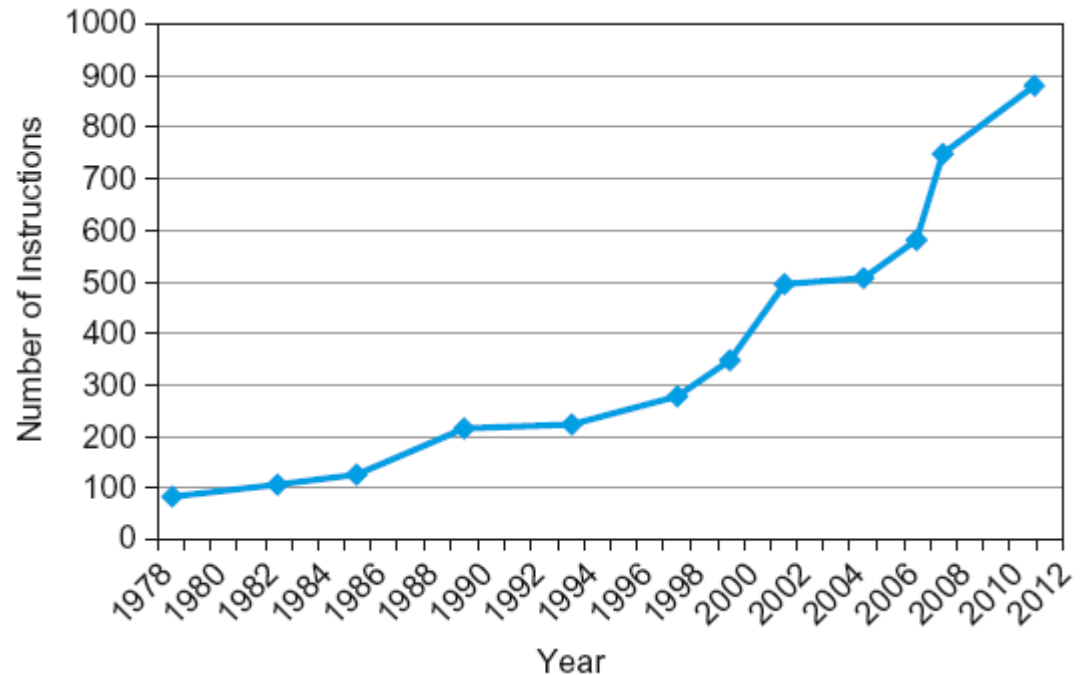
C code – x.c or .TXT
assembly code – x.s or .ASM
object code – x.o or .OBJ
executable – a.out or .EXE

Fallacies and Pitfalls

❑ Fallacies:

- More powerful instructions mean higher performance
- Write in assembly language for highest performance
- Binary compatibility means successful ISAs (e.g., x86) don't change

x86 Instruction Set Growth

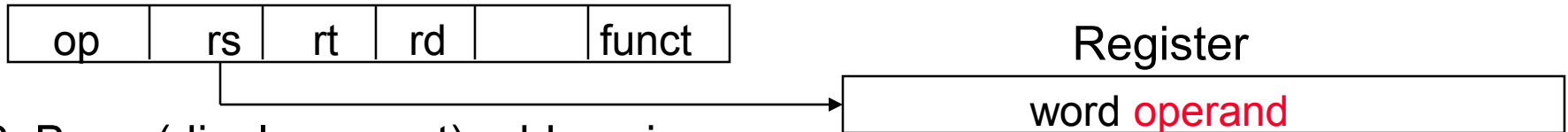


❑ Pitfalls:

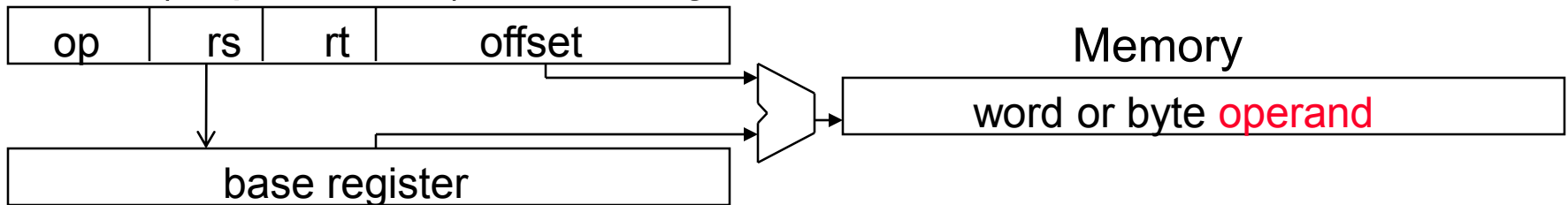
- Forgetting that sequential word addresses in machines with byte addressing don't differ by one (but by 4 !)
- Using a pointer to an automatic variable outside its defining procedure

Review: Addressing Modes Illustrated

1. Register addressing



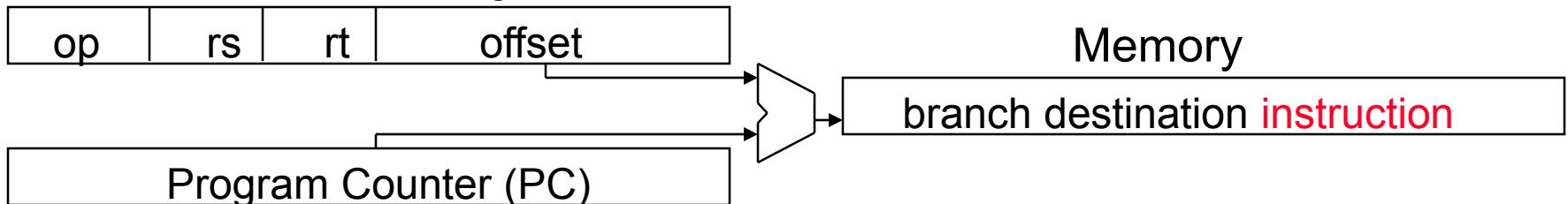
2. Base (displacement) addressing



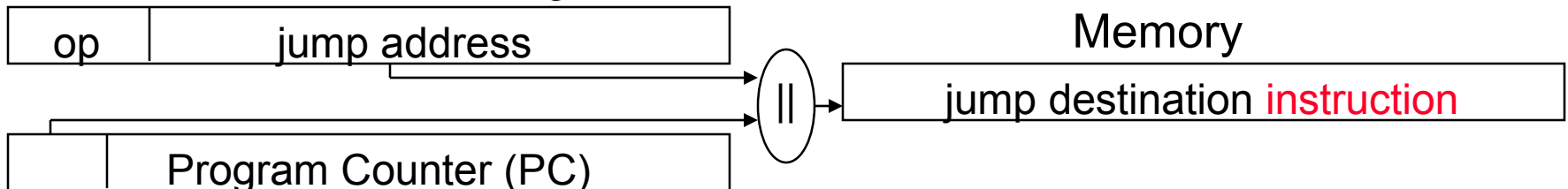
3. Immediate addressing



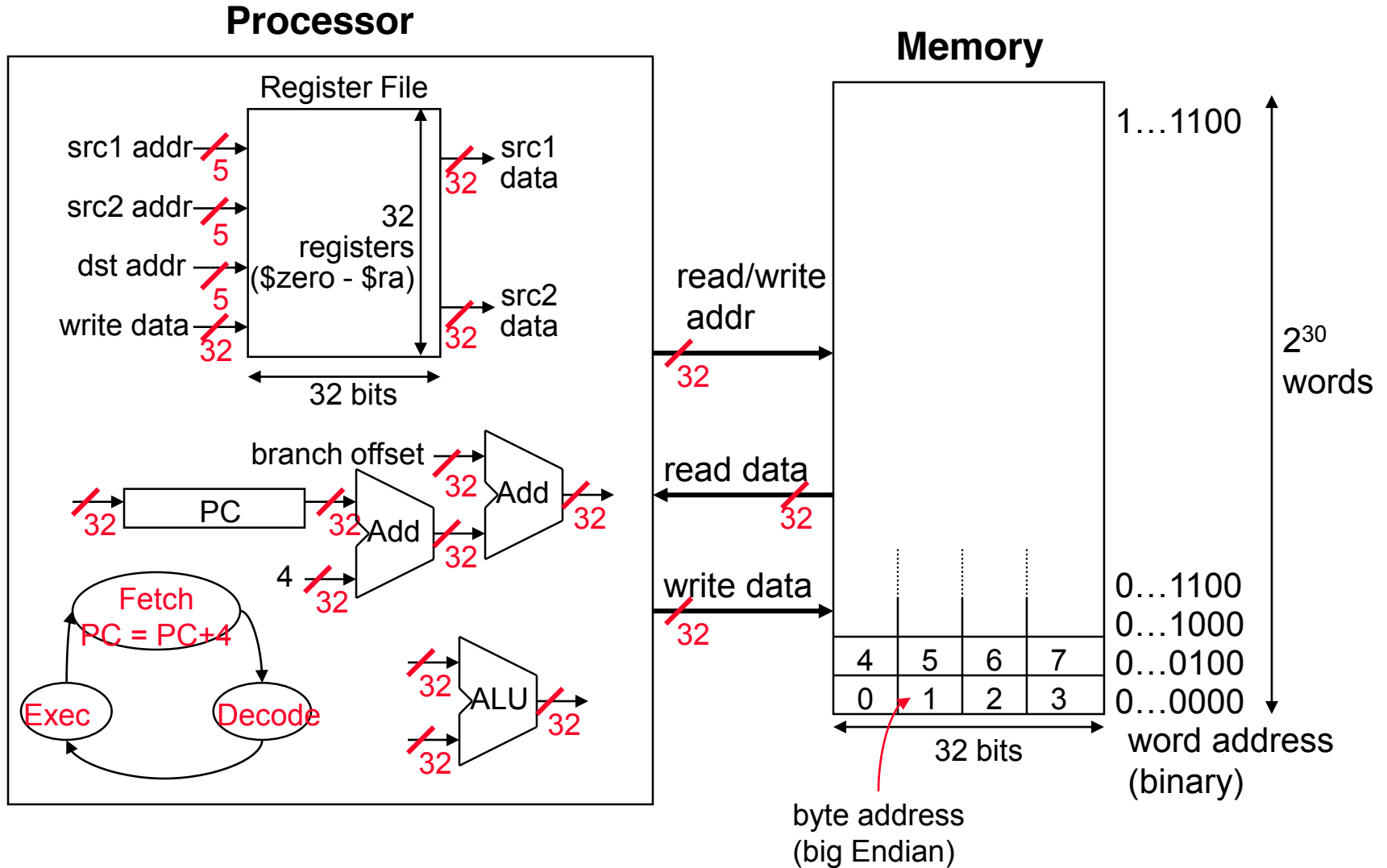
4. PC-relative addressing



5. Pseudo-direct addressing



Review: MIPS Organization So Far



Reading Assignment

- ❑ Read all of Chapter 2 in PH