CS3350B
Computer Architecture
Winter 2015

Lecture 4.2: MIPS ISA -- Instruction Representation

Marc Moreno Maza
www.csd.uwo.ca/Courses/CS3350b

[Adapted from lectures on
Computer Organization and Design,
Patterson & Hennessy, 5th edition, 2013]

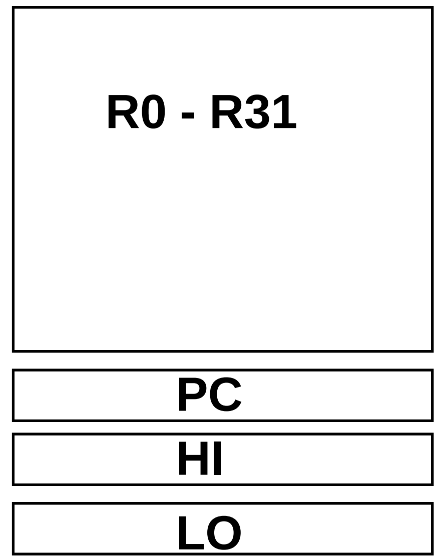
Representing Instructions

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

Overview: MIPS R3000 ISA

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

Registers



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

OP	**rs**	**rt**	**rd**	**sha**	**funct**	R-format

OP	**rs**	**rt**	**immediate**	I-format

OP	**jump target**	J-format

MIPS ISA Selected Instruction Set

Category	Instr		OP/funct	Example	Meaning
Arithmetic	add	R	0/32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	R	0/34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	I	8	addi $s1, $s2, 6	$s1 = $s2 + 6
	or immediate	I	13	ori $s1, $s2, 6	$s1 = $s2 v 6
Data Transfer	load word	I	35	lw $s1, 24($s2)	$s1 = Memory($s2+24)
	store word	I	43	sw $s1, 24($s2)	Memory($s2+24) = $s1
	load byte	I	32	lb $s1, 25($s2)	$s1 = Memory($s2+25)
	store byte	I	40	sb $s1, 25($s2)	Memory($s2+25) = $s1
	load upper imm	I	15	lui $s1, 6	$s1 = 6 * 2^{16}
Cond. Branch	br on equal	I	4	beq $s1, $s2, L	if ($s1==$s2) go to L
	br on not equal	I	5	bne $s1, $s2, L	if ($s1 != $s2) go to L
	set on less than	R	0/42	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set on less than immediate	I	10	slti $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0
Uncond. Jump	jump	J	2	j 250	go to 1000
	jump register	R	0/8	jr $t1	go to $t1
	jump and link	J	3	jal 250	go to 1000; $ra=PC+4

MIPS R-format Instructions

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

- Instruction fields
  - **op**: operation code (opcode)
  - **rs**: first source register number
  - **rt**: second source register number
  - **rd**: destination register number
  - **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

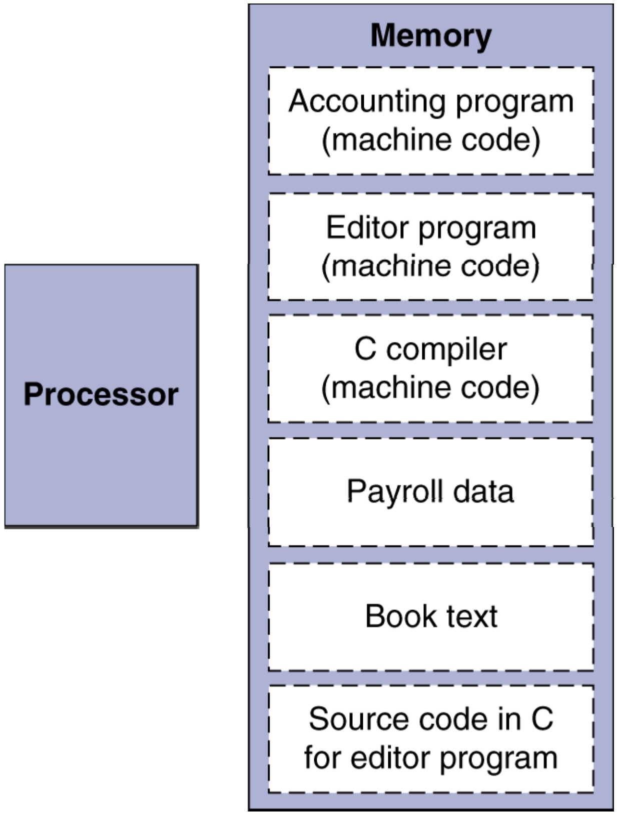
$00000010001100100100000000100000_2$

MIPS I-format Instructions

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

- **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

Stored Program Computers



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

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

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

- **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

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and $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 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
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

NOT Operations

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

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

Register 0: always read as zero

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

Conditional Operations

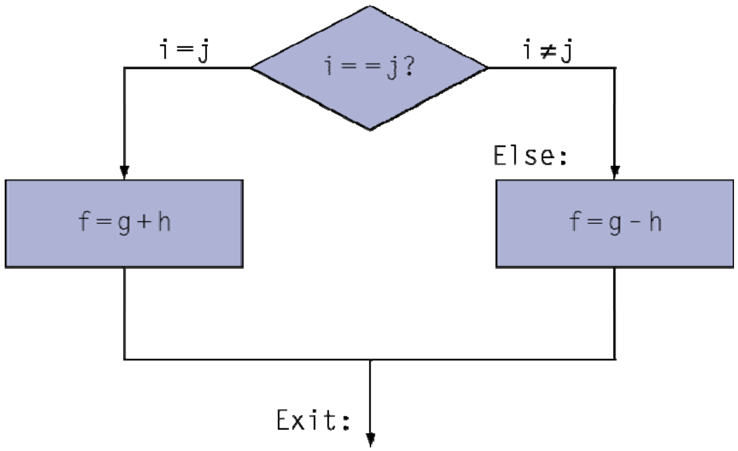
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

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:

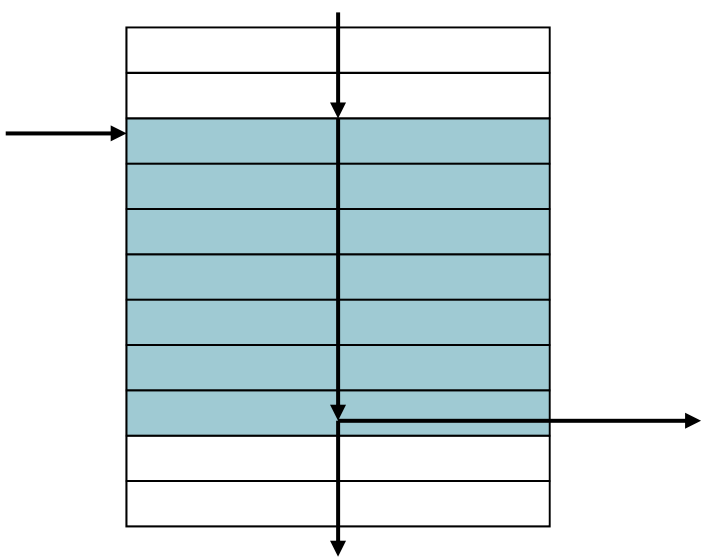
```
while (save[i] == k) i += 1;
```

  - 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: …
```

Basic Blocks

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



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

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`, `bge`, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1   # signed`
    - $-1 < +1 \Rightarrow$ $t0 = 1
  - `sltu $t0, $s0, $s1   # unsigned`
    - $+4{,}294{,}967{,}295 > +1 \Rightarrow$ $t0 = 0

Byte/Halfword Operations

- Could use bitwise operations
- MIPS **byte/halfword load/store**
  - String processing is a common case

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

  - Sign extend to 32 bits in rt

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

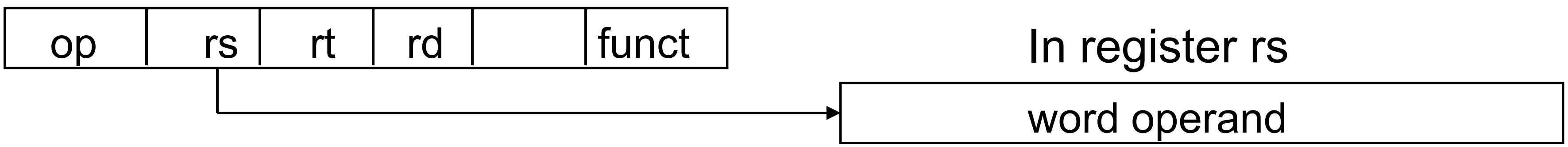
  - Zero extend to 32 bits in rt

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

  - Store just rightmost byte/halfword

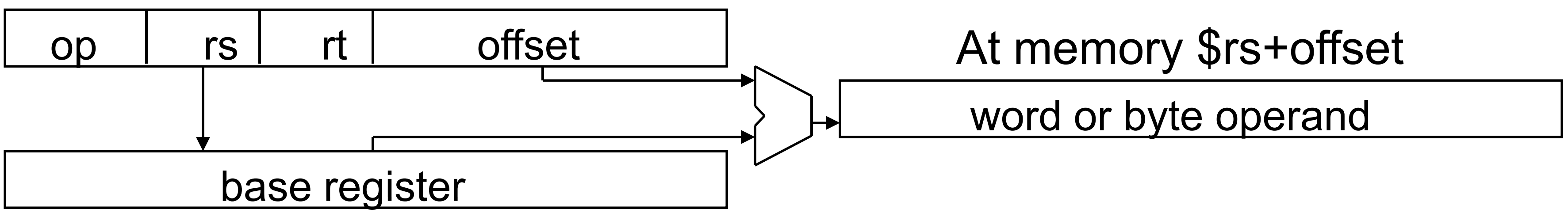
Operand Addressing Modes

(1) Register addressing – operand is in a register



Example: add $rd, $rs, $rt # $rd = $rs + $rt

(2) Base (displacement) addressing – operand is at the **memory location** whose address is the sum of a register and a 16-bit constant contained within the instruction



Example: lw $rt, offset($rs) # $rt = Memory($rs+offset)

- **Register relative (indirect)** with 0($a0) (that is, offset = 0), or **jr**
- **Pseudo-direct** with addr($zero), that is, $rs = $zero = 0

Operand Addressing Modes (ctn'd)

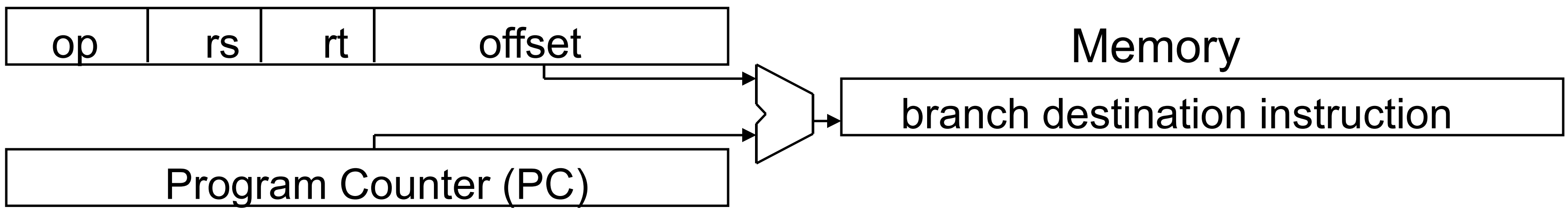
(3) Immediate addressing – operand is a 16-bit **constant** contained within the instruction

op	rs	rt	operand

Example: addi $rt, $rs, operand # $rt = $rs + operand

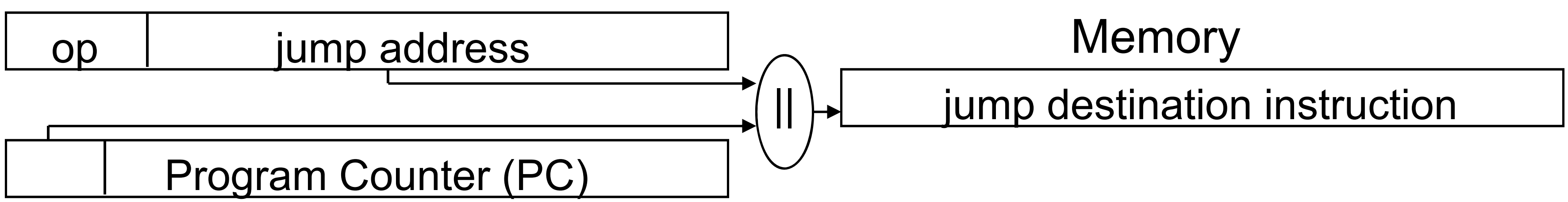
Instruction Addressing Modes

(1) PC-relative addressing –instruction address is the sum of the PC and a 16-bit constant contained within the instruction



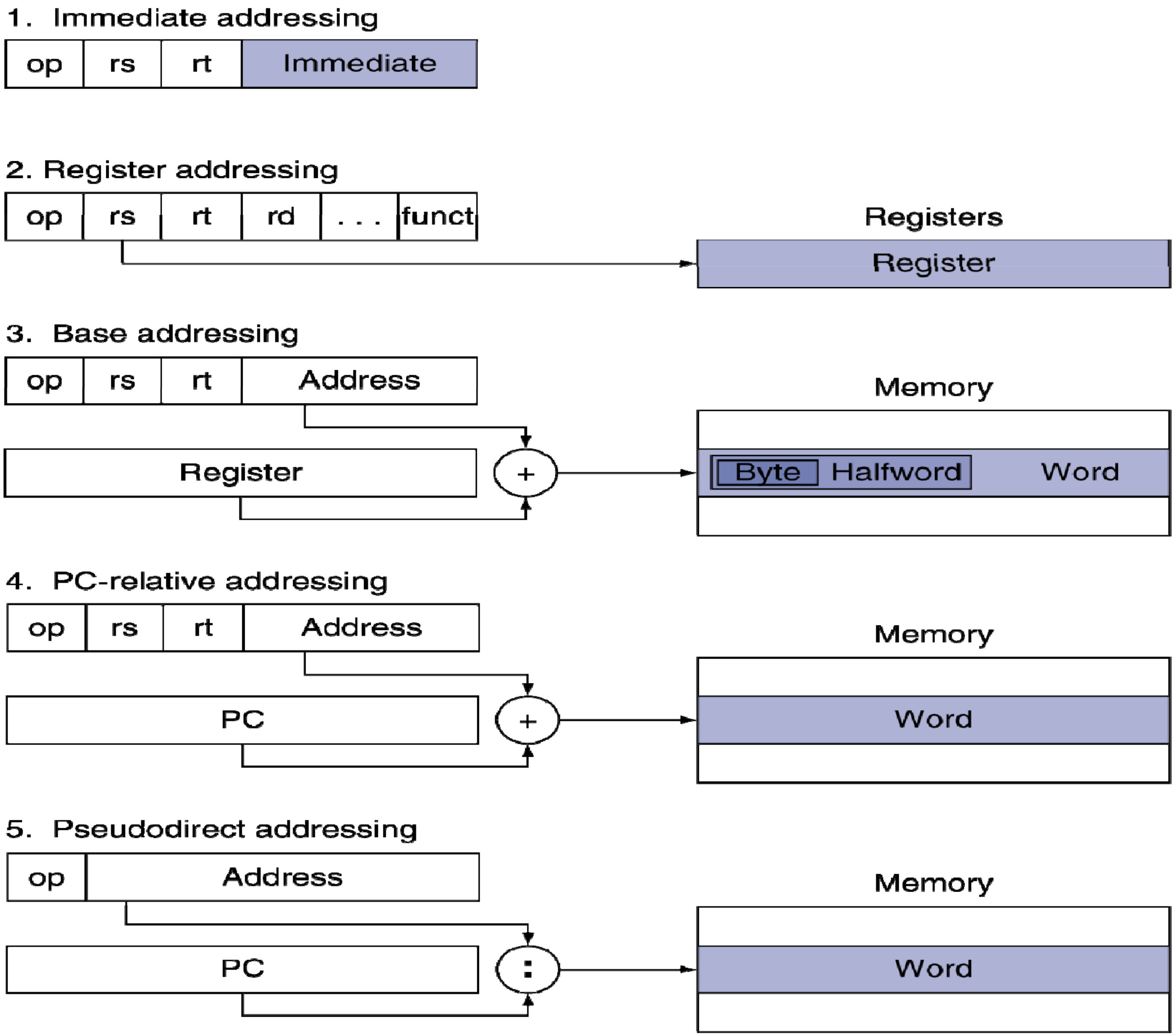
- Used for **beq** and **bne:** # if rs==rt (or rs!=rt), go to offset (PC=PC+4+4*offset)

(2) Pseudo-direct addressing – instruction address is the 26-bit constant contained within the instruction concatenated with the upper 4 bits of the PC



- Used for **j** (jump): **PC** | xxxx | jump address | 00 |

Addressing Mode Summary



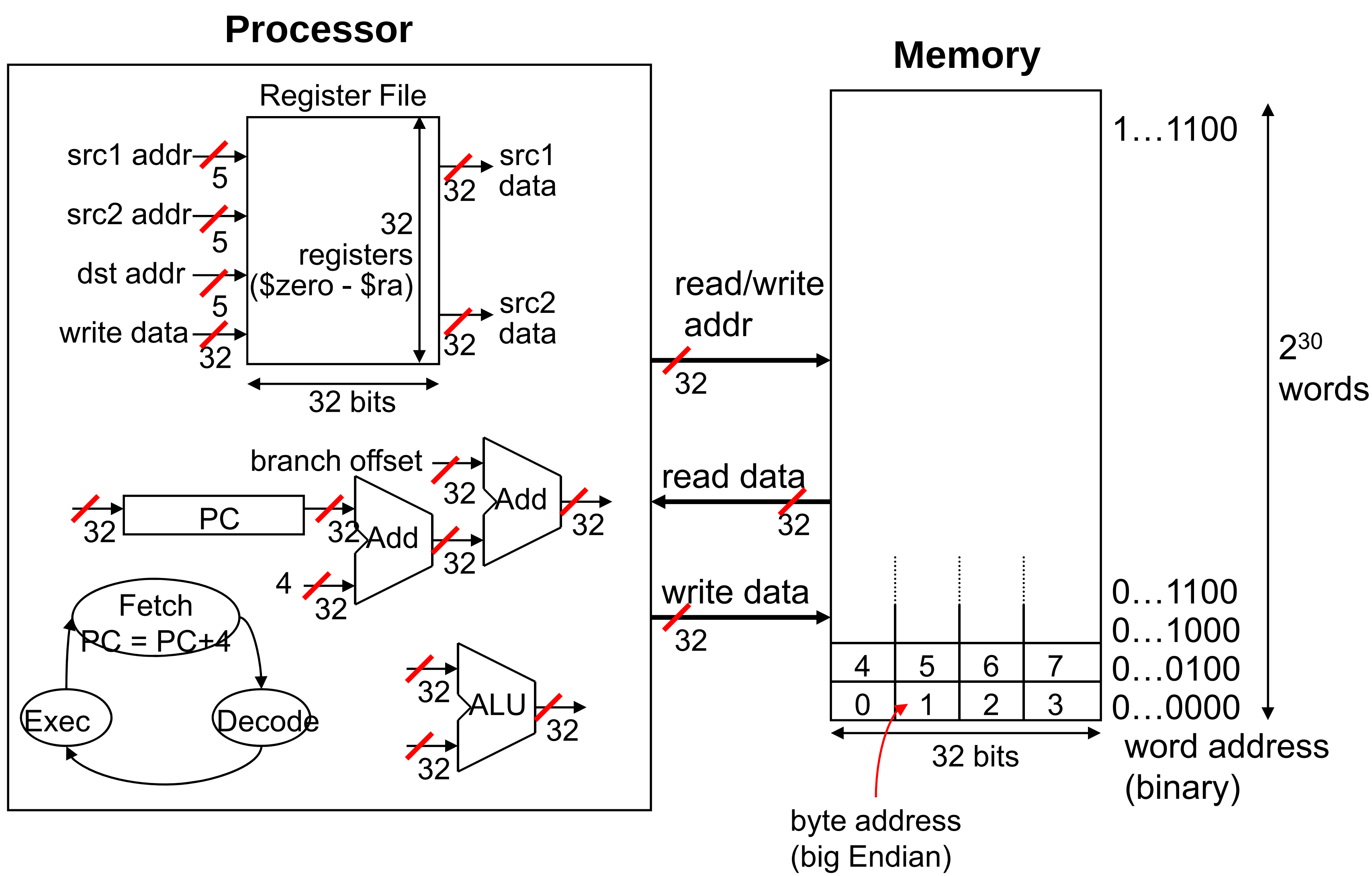
Caution: Addressing mode is not Instruction Types

- Addressing mode is how an address (memory or register) is determined.
- Instruction type is how the instruction is put together.
- Example: addi, beq, and lw are all I-Format instructions. But,
  - addi uses immediate addressing mode (and register)
  - beq uses pc-relative addressing (and register)
  - lw uses base addressing (and register)

Summary of MIPS Addressing Modes

- **Register**: a source or destination operands specified as content of one of the registers \$0-\$31.
- **Immediate**: a numeric value embedded in the instruction is the actual operand.
- **PC-relative**: a data or instruction memory location is specified as an offset relative to the incremented PC.
- **Base**: a data or instruction memory location is specified as a signed offset from a register.
- **Register-direct**: the value of the effective address is in a register.
- **Pseudo-direct**: the memory address is (mostly) embedded in the instruction.

MIPS Organization So Far



Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

Aside: Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual **bytes** in memory: **byte-addressable**
  - it means that a byte is the smallest unit with its address

- **Naturally aligned data**: **doublewords** that lie on addresses that are multiples of eight, **words** that lie on addresses that are multiples of four, **halfwords** that lie on addresses that are multiples of two, and **single bytes** that lie at any byte address. Such data is located on its natural size boundary, to maximize storage potential and to provide for fast, efficient memory access.
- **Little Endian:** rightmost byte is word address

  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)
- **Big Endian:** leftmost byte is word address

  IBM 360/370, Motorola 68k, **MIPS**, Sparc, HP PA
- **MIPS** memory is **byte-addressable;** supports 32-bit address (an address is given as a 32-bit unsigned integer)

Aside: Compiler storage of data objects by byte alignment

Type	Bytes	Alignment
char, bool	1	Located at any byte address.
short	2	Located at any address that is evenly divisible by 2.
float, int, long, pointer	4	Located at an address that is evenly divisible by 4.
long long, double, long double	8	Located at an address that is evenly divisible by 8.