Computer Architecture (CS 207 D)

Instruction Set Architecture ISA

Execution Cycle

Obtain instruction from program storage and put into IR

Determine required actions and instruction size

'control' the subsequent actions

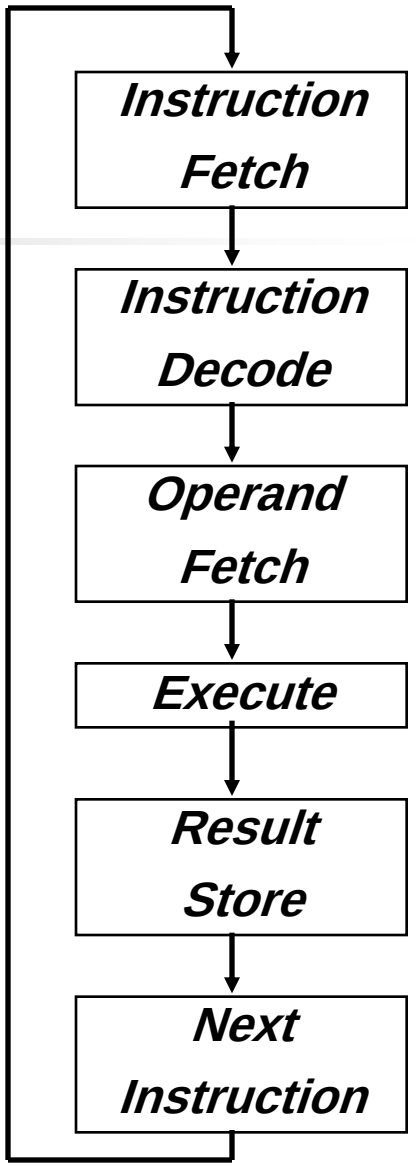
Locate and obtain operand data

Compute result value or status (Execute)

Deposit results in storage for later use

Determine successor instruction

(Fetch next instruction)



Instructions Set Architecture ISA: Overview MIPS : **M**icrocomputer without **I**nterlocked **P**ipeline **S**tages

- **Instructions** is the language of the machine
- More primitive than higher level languages, e.g., no sophisticated control flow such as *while* or *for* loops
- **Different computers** have different instruction sets
- Very restrictive e.g., **MIPS arithmetic instructions**
- Large share of embedded core market, **MIPS** ISA typical of many modern: ISAs (www.mips.com)
  - inspired most architectures developed since the 80's
  - used by NEC, Nintendo, Silicon Graphics, Sony
  - the name is not related to *millions of instructions per second* !
  - **MIPS** ISA stands for **M**icrocomputer without **I**nterlocked **P**ipeline **S**tages
  - **Design goals:** *maximize performance* ; *minimize cost* and *reduce design time*

MIPS Arithmetic

- **Design Principle 1**: *simplicity favors regularity*.
  Translation: *Regular instructions make for simple hardware*!

- *Simpler hardware reduces design time and manufacturing cost.*

- Of course this complicates some things...

Allowing variable number of operands would simplify the assembly code but complicate the hardware.

ompiler's job to associate variables with registers

C code:

```
A = B + C + D;
E = F - A;
```

MIPS code (arithmetic):

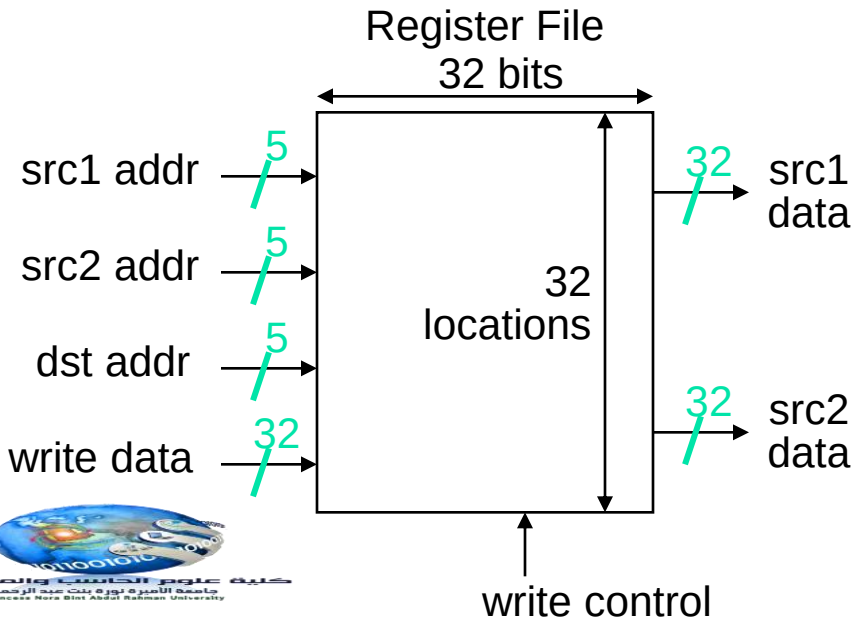
```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

- Performance penalty: high-level code translates to denser machine code.

MIPS Arithmetic

- Operands must be in registers – only 32 registers provided (which require 5 bits to select one register). Reason for small number of registers:

- **Design Principle 2**: smaller is faster. Why?
  - Electronic signals have to travel further on a physically larger chip increasing clock cycle time.
  - Smaller is also cheaper!



Example: C code

F=(g+h) - (i+j)

Compiled MIPS code

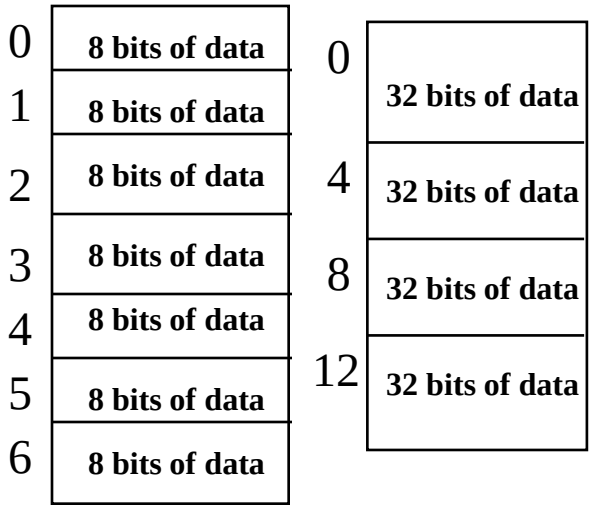
```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Registers vs. Memory

- Arithmetic instructions operands must be in registers
  - MIPS has 32 X 32 bit register file used for frequently accessed data
- Compiler associates variables with registers numbered 0 to 31
- Register names $t0, $t1, ….. $t9 for temporary values
- Register names $S0, $S1, ….. $S7 for temporary values
- What about programs with lots of variables (arrays, etc.)? Use memory, load/store operations to transfer data from memory to register – if not enough registers spill registers to memory
- **MIPS is a load/store architecture**

Memory Organization

- Viewed as a large single-dimension array with access by address
  - A memory address is an index of the memory array
  - **Byte addressing** means that the index points to a **byte** of data.



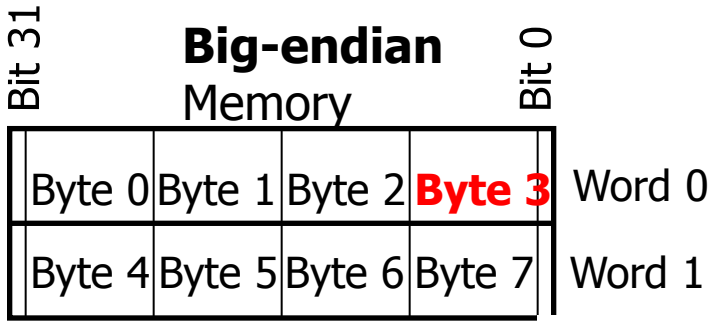
. Bytes are load/store units, but most data use larger *words* For MIPS, a word is 4 bytes.

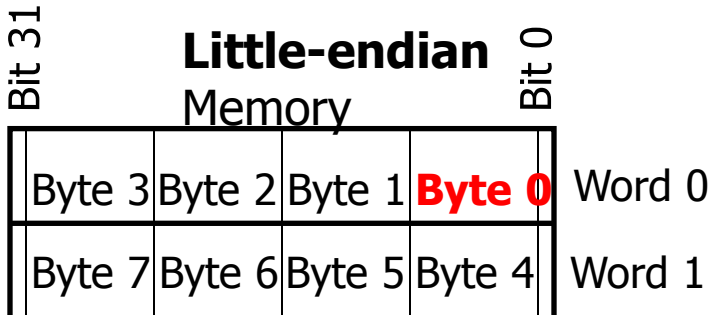
- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$
- i.e., words are *aligned*
- *what are the least 2 significant bits of a word address?*

...

Memory Organization: **Big/Little** Endian Byte Order

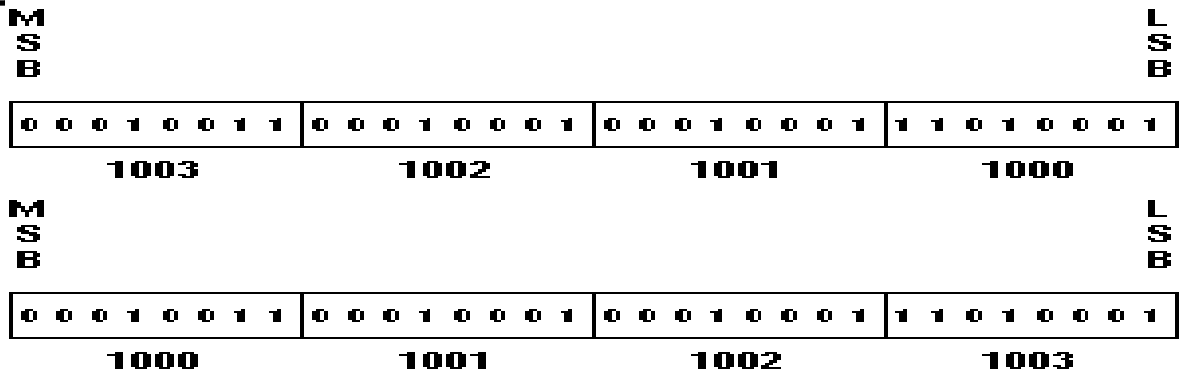
- Bytes in a word can be numbered in two ways:
  - byte 0 at the leftmost (most significant) to byte 3 at the rightmost (least significant), called *big-endian*
  - byte 3 at the leftmost (most significant) to byte 0 at the rightmost (least significant), called *little-endian*





MIPS is Big Ending

Intel 80x86 processors?



Load/Store Instructions

- *Load* and *store* instructions

Example:

C code: `A[8] = h + A[8];`

value offset address

MIPS code (load): `lw $t0, 32($s3)`

(arithmetic): `add $t0, $s2, $t0`

(store): `sw $t0, 32($s3)`

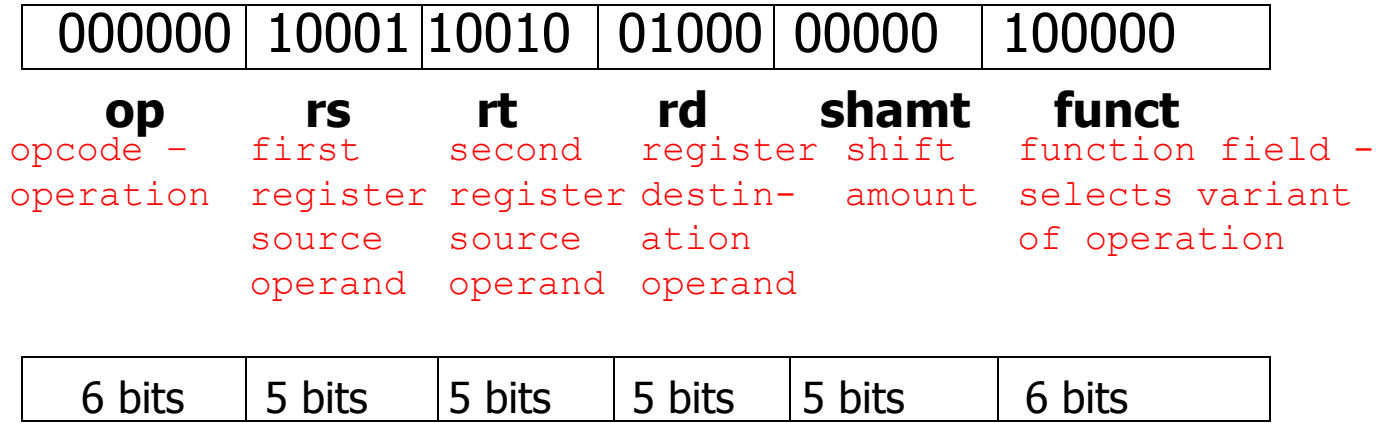
- Load instruction has destination first, **store has destination last**
- Remember MIPS arithmetic operands are registers, not memory locations
- loading words **but** addressing bytes

Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100]= $s1`

MIPS ISA format (Types)
1) R-type Instruction

- Instructions, like registers and words of data, are also 32 bits long
  - *Example*: `add $t0, $s1, $s2`
  - registers are numbered, e.g., `$t0 is 8, $s1 is 17, $s2 is 18`

- Instruction Format **R-type** ("**R**" for a**R**ithmetic):



Constants
Immediate Operands

- Small constants are used quite frequently (50% of operands)
  e.g., A = A + 5; B = B + 1; C = C - 18;

- Solutions? Will these work?
  - create hard-wired registers (like $zero) for constants like 1
  - put program constants in memory and load them as required
  - Make operand part of instruction itself!

- **Design Principle 4**: ***Make the common case fast***

001000	11101	11101	0000000000000100
6 bits	5 bits	5 bits	16 bits
op	rs	rt	16 bit number

- MIPS Example:

```
addi $sp, $sp, 4
```

2) I- Type Instruction

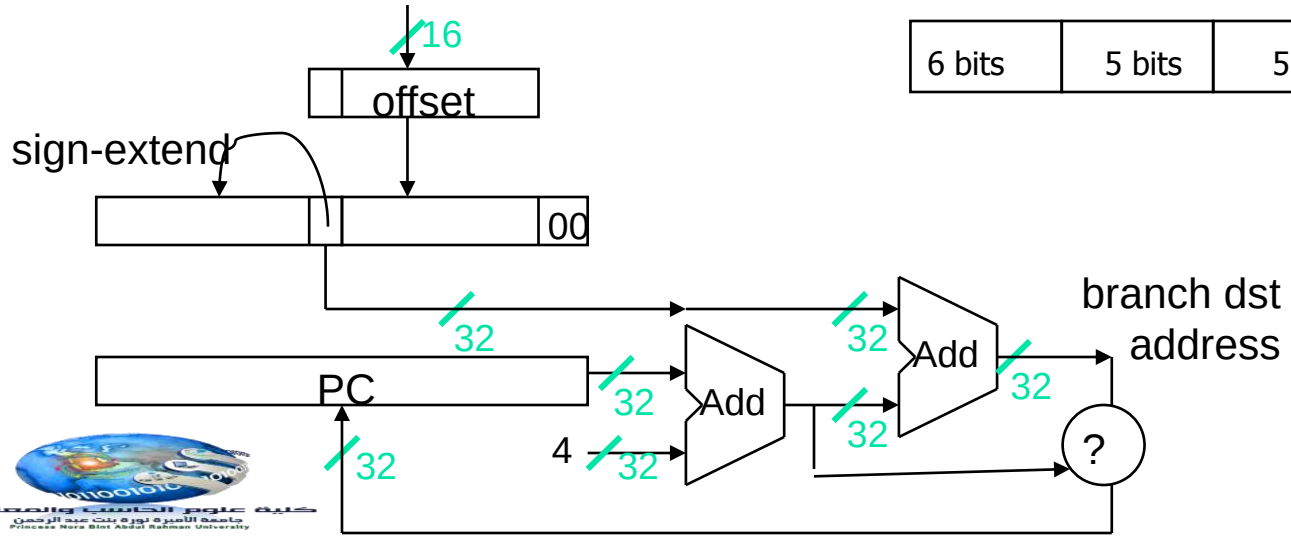
- **Design Principle 3**: *Good design demands a compromise*
- Introduce a new type of instruction format
  - **I-type** ("**I**" for **I**mmediate) for data transfer instructions
  - *Example*: `bne $t0,$s2, Lab`

000100 10010 01000 0000001111101010

Op	rs	rt	16 bit offset

from the low order 16 bits of the branch instruction

6 bits	5 bits	5 bits	16 bits



Addresses in Branch

- Further extend reach of branch by observing all MIPS instructions are a word (= 4 bytes), therefore ***word-relative addressing:***
- MIPS branch destination address = (PC + 4) + (4 * offset)

Because hardware typically increments PC early in execute cycle to point to next instruction

  - so offset = (branch destination address – PC – 4)/4
  - *but SPIM does* offset = (branch destination address – PC)/4

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

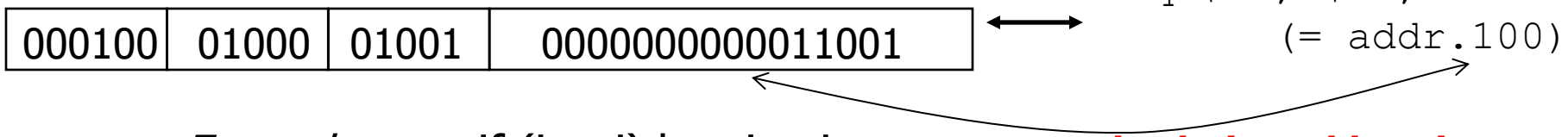
80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Control: Conditional Branch

- Decision making instructions
  - alter the control flow,
    - i.e., change the next instruction to be executed
- MIPS conditional branch instructions:

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

} I-type instructions



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

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

word-relative addressing:
25 words = 100 bytes;
also *PC-relative* (more…)

Addresses in Branch

- Instructions:

```
bne $t4,$t5,Label     Next instruction is at Label if $t4 != $t5
beq $t4,$t5,Label     Next instruction is at Label if $t4 = $t5
```

- Format:

I	op	rs	rt	16 bit offset

- 16 bits is too small a reach in a 2^{32} address space
- Solution: specify a register (as for `lw` and `sw`) and add it to offset
  - use PC (= program counter), called *PC-relative* addressing, based on
  - *principle of locality*: most branches are to instructions near current instruction (e.g., loops and *if* statements)

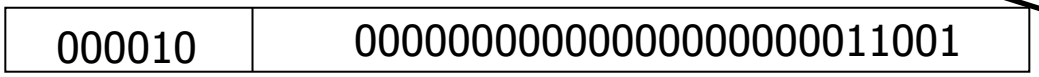
3) J-Type Instruction

- Control: Unconditional Branch (Jump)
  - MIPS unconditional branch instructions: `j Label`

- *Example*:

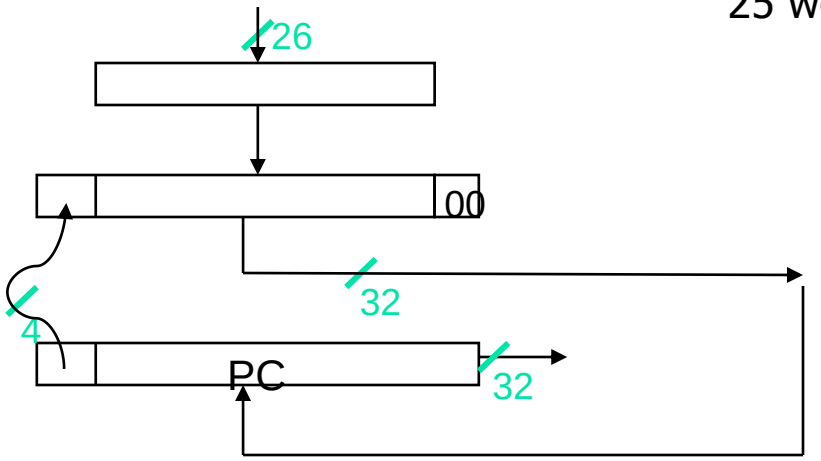
```
if (i!=j)                beq $s4, $s5, Lab1
 h=i+j;                  add $s3, $s4, $s5
else                     j Lab2
h=i-j;                   Lab1: sub $s3, $s4, $s5
                         Lab2:
```

- *Example*: `j Label #  Label = 100`



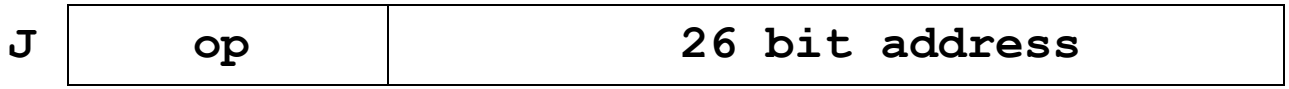
6 bits 26 bits

word-relative addressing:
25 words = 100 bytes



Addresses in Jump

- Word-relative addressing also for jump instructions



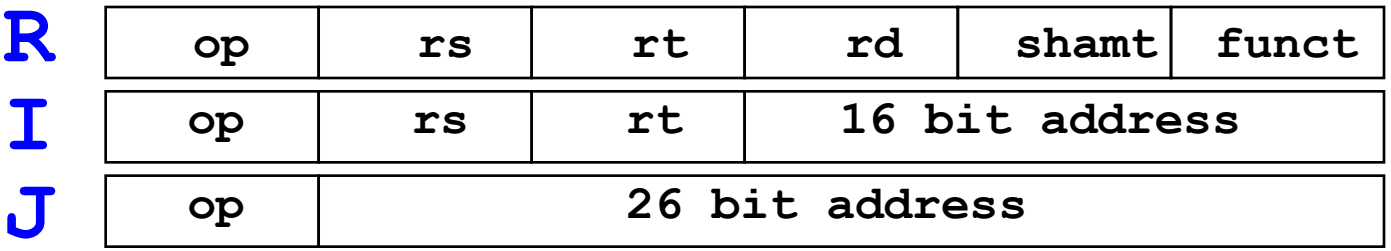
- MIPS jump `j` instruction replaces *lower* 28 bits of the PC with `A00` where `A` is the 26 bit address; it *never changes* upper 4 bits
  - *Example*: if `PC = 1011X` (where X = 28 bits), it is replaced with `1011A00`
  - there are 16(=2^4) partitions of the 2^{32} size address space, each partition of size 256 MB (=2^{28}), *such that*, in each partition the upper 4 bits of the address is same.
  - if a program crosses an address partition, then a **`j`** that reaches a different partition has to be replaced by **`jr`** with a full 32-bit address first loaded into the jump register **jr $ra**
  - therefore, *OS should always try to load a program inside a single partition*

In Summary

Instruction	Format	Meaning
`add $s1,$s2,$s3`	R	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	R	`$s1 = $s2 - $s3`
`lw $s1,100($s2)`	I	`$s1 = Memory[$s2+100]`
`sw $s1,100($s2)`	I	`Memory[$s2+100] = $s1`
`bne $s4,$s5,Lab1`	I	Next instr. is at Lab1 if `$s4 != $s5`
`beq $s4,$s5,Lab2`	I	Next instr. is at Lab2 if `$s4 = $s5`
`j Lab3`	J	Next instr. is at Lab3

Overview of MIPS

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



Control Flow

- We have: beq, bne. What about ***branch-if-less-than***?

- New instruction:

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

- Can use this instruction to **build** `blt $s1, $s2, Label` (How?)
  - *how?* We generate more than one instruction – ***pseudo-instruction***
  - can now build general control structures bgt, bge and ble

- Assembly can provide *pseudo-instructions*
  - e.g., `move $t0, $t1` exists only in assembly (How?)

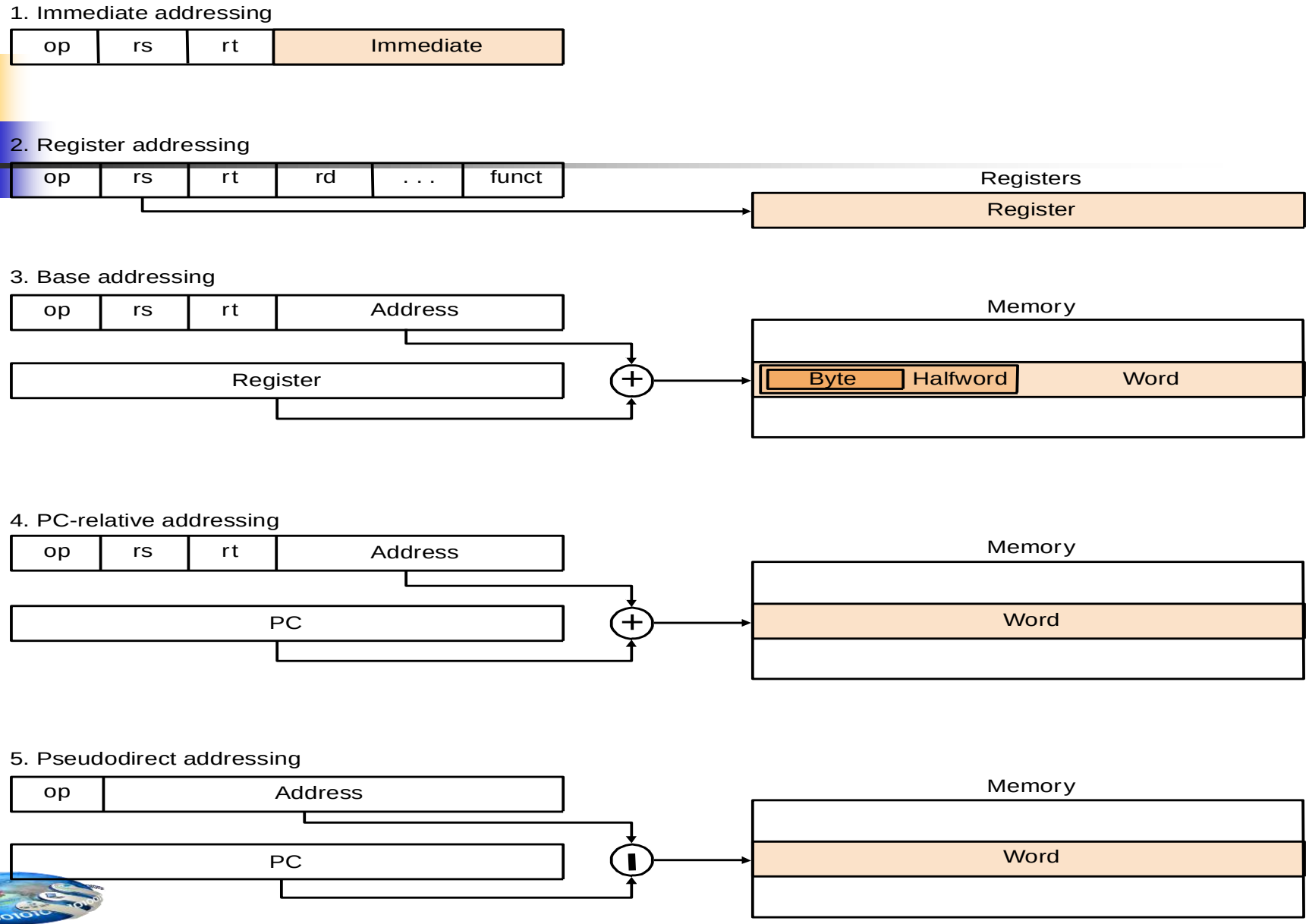
Policy-of-Use Convention for Registers

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

There is a *convention* for use of registers.

- Register 1, called **$at**, is reserved for the assembler; registers 26-27, called **$k0 and $k1** are reserved for the operating system.

MIPS Addressing Modes



Summarize MIPS:

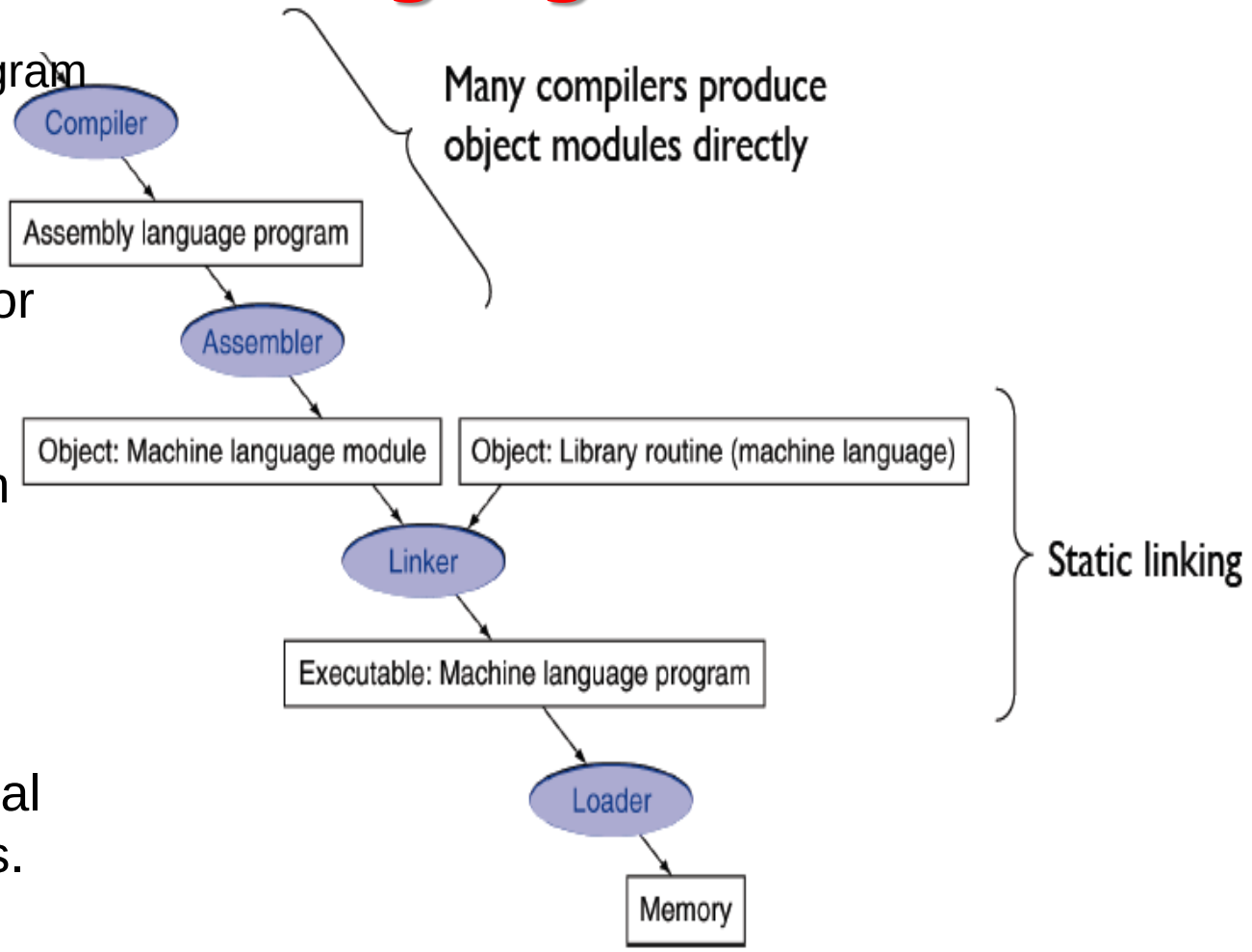
MIPS operands

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

MIPS assembly language

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

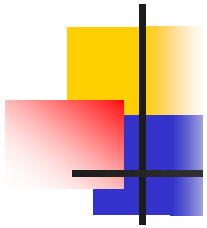
Translation from High Level Language to MIPS; Machine Language



There are three steps for the linker:

1. Place code and data modules symbolically in memory.
2. Determine the addresses of data and instruction labels.
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 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

Summary

- Instruction set architecture
  - a very important abstraction indeed!
- **Design Principles:**
  - **simplicity favors regularity**
  - **smaller is faster**
  - **make the common case fast**
  - **Good design demands good compromises**
- Layers of software/ hardware
  - Compiler; assembler; hardware

MIPS is typical of RISC ISAs