Computer Architecture

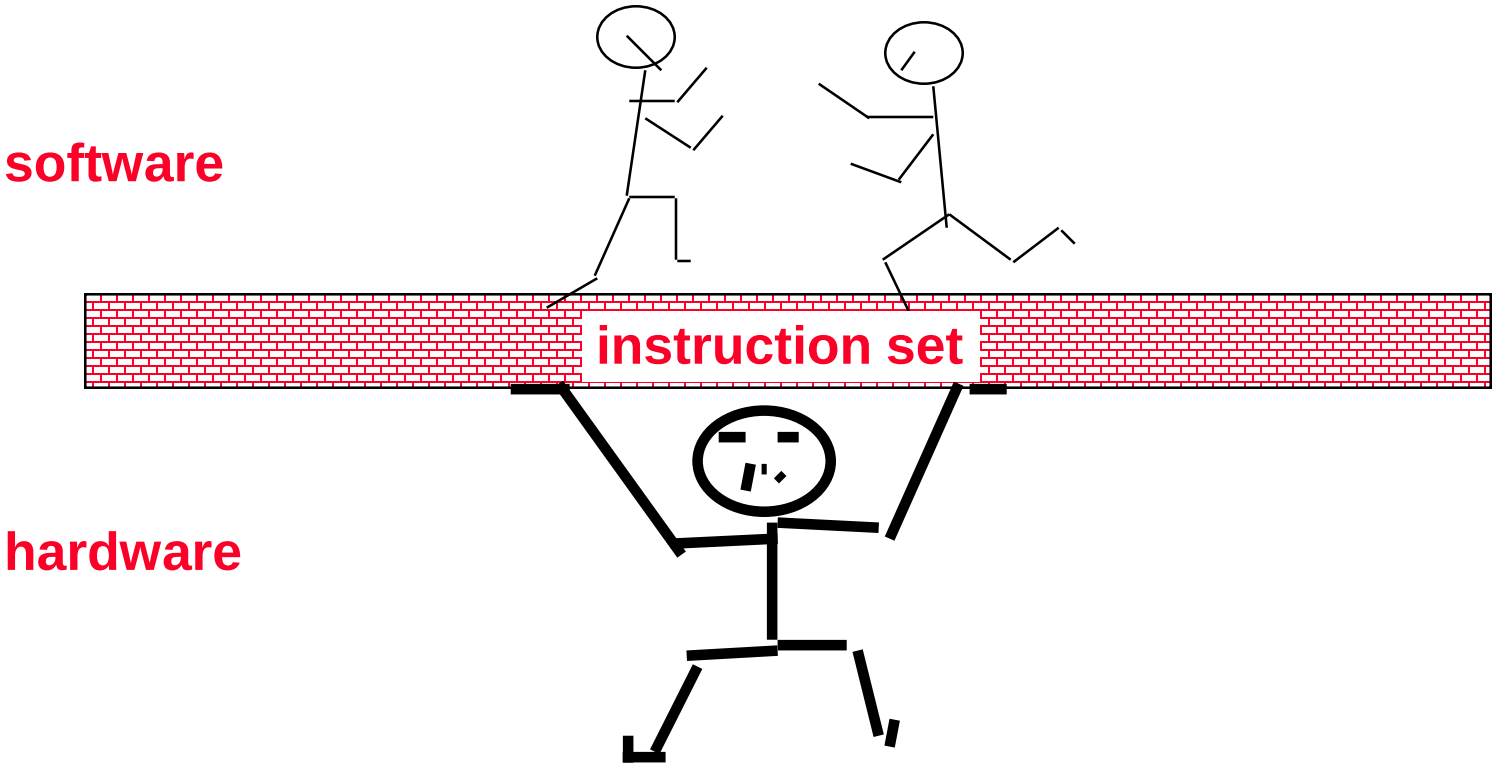
Chapter 2
Instructions: Language of the Computer

APPENDICES

ONLINE CONTENT

Instructions: Language of the Computer

The Instruction Set : a Critical Interface



Instruction : words

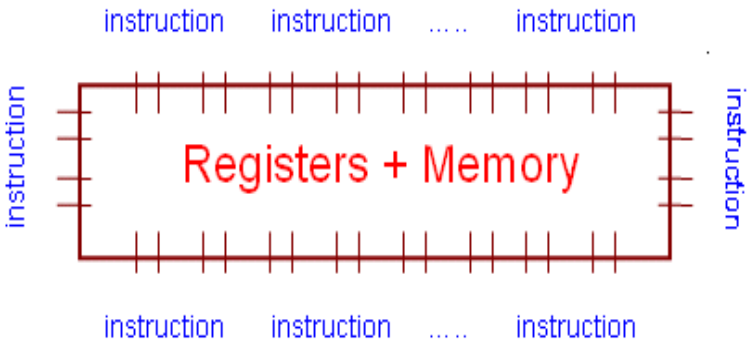
Instructions Set : vocabulary

Functions of Instruction Set

- Any Processor Must Be Able to Do at Least the Following Basic Functions
  - Arithmetic and Logic Operations
  - Data transfer to and from the memory
  - Conditional branches
    - Need a way to determine a condition
    - Need a target memory address to branch to if condition is met (or not met), go to next instruction otherwise
  - Jump and subroutine linkage (procedure call)
    - Need a large range of target memory address to branch
    - Procedure call needs a return address
- Additional functions: Examples
  - Move data between registers, to I/O, or to co-processor
  - Exception and Interrupt Instructions

Instruction Sets Classification

- An Abstract Data Type
  - Objects ≡ Registers & Memory
  - Operations ≡ Instructions
    - C code: `a = b + c`
    - MIPS 'code': `add a, b, c`
  - *Operand vs. register*



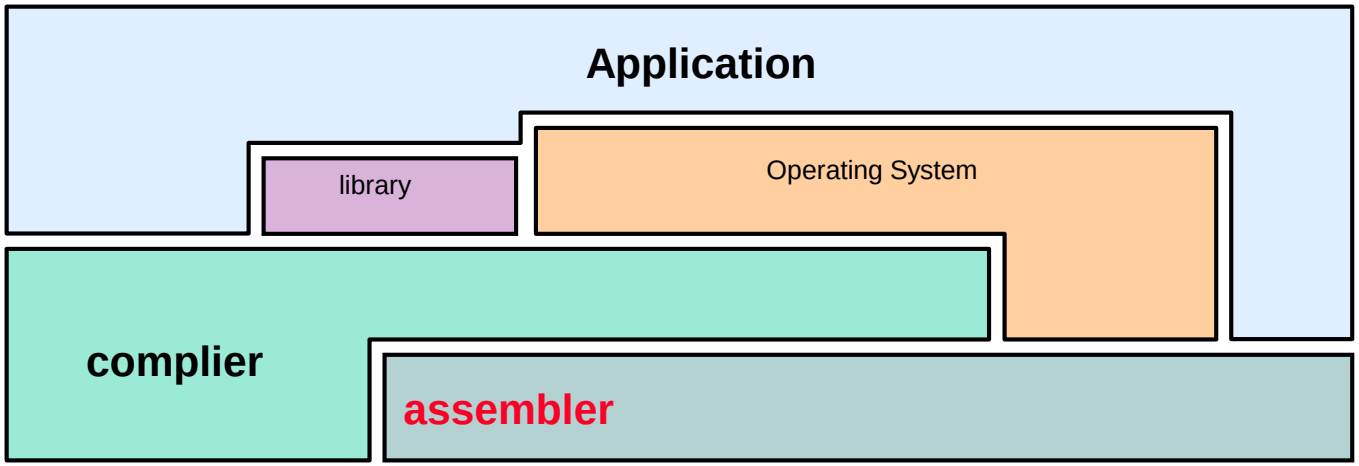
- Classifying Instruction Sets
  - Machine : number/kind of registers

    0 (stack machine) 1 (accumulator machine)

    small (2-6) general registers (e.g. 16, 32 or more)
  - # of addresses per instruction
    - 0 (stack machine)
    - 1 (accumulator machine)
    - 2 (general registers)
    - 3 (general registers)

Instruction Set Architecture (ISA)

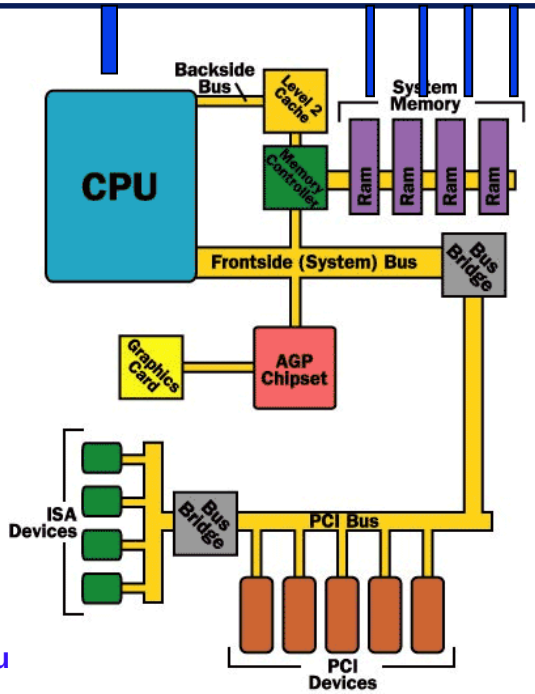
S/W



H/W

ISA
defines the permissible instructions

MIPS :
load/store,
arithmetic,
control flow,,,

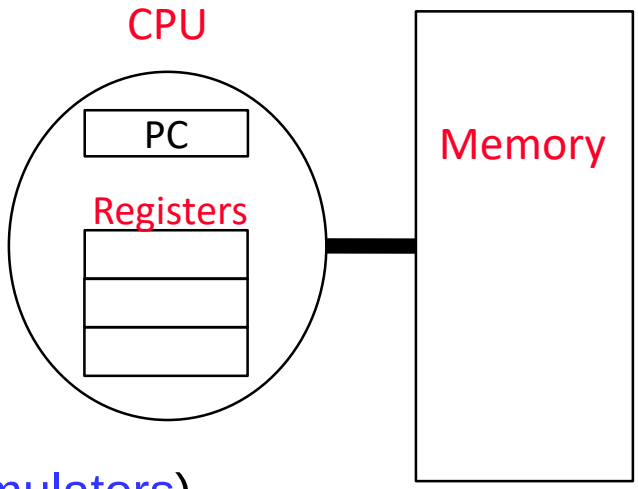


Goal of ISA Design

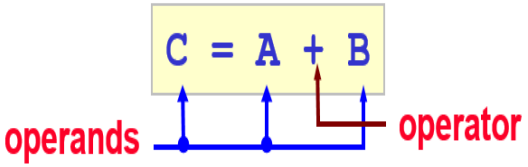
To allow high-performance and low-cost implementations while satisfying constraints imposed by applications including operating system and complier

Brief Historical Perspective on ISAs

- The ISA defines:
  - The system's state (*e.g.* registers, memory, program counter)
  - The instructions the CPU can execute
  - The effect that each of these instructions will have on the system state

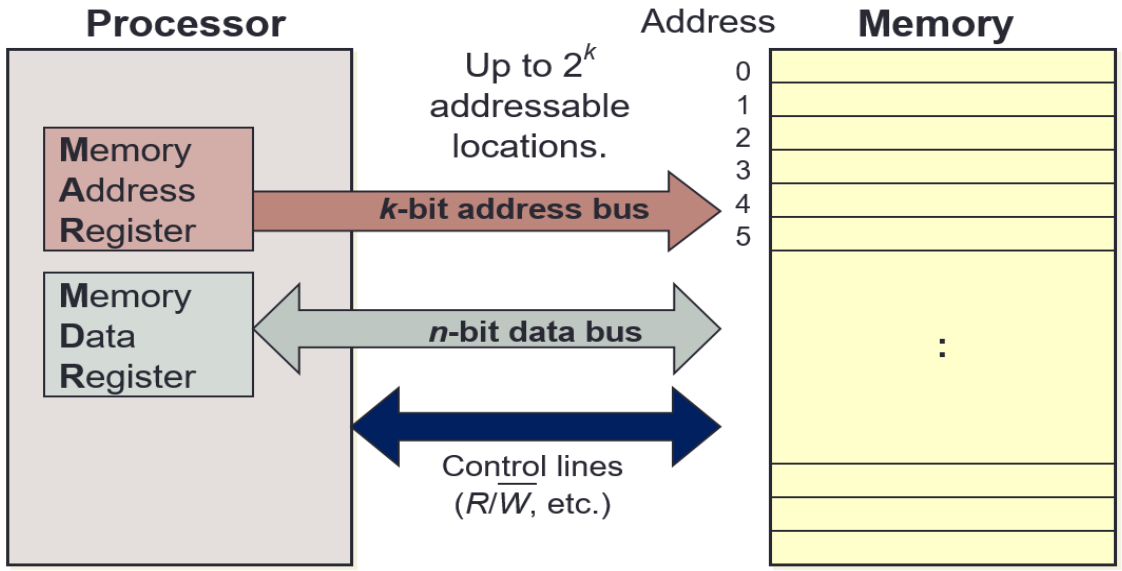


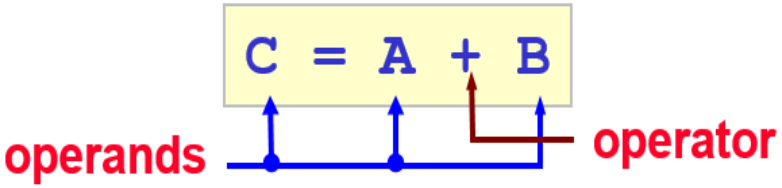
- General-purpose registers
  - Registers can be used for any purpose
  - E.g. MIPS, ARM, x86
- *Register-memory* architectures
  - One operand may be in memory (e.g. accumulators)
  - E.g. x86 processors, Motorola 68000
- *Register-register* architectures (aka load-store)
  - All operands ***must*** be in registers
  - E.g. MIPS, ARM



Memory Addressing Mode

- Memory Address and Content
  - Given *k*-bit address, the address space is of size 2^k
  - Each memory transfer consists of one word of ***n*** bits





- Addressing Mode:
  - Ways to specify an operand in an assembly language

Operations in Instructions Set

- Standard Operations

Data Movement	load (from memory) store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP add, subtract, multiply, divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control flow	Jump (unconditional), Branch (conditional)
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic r-m-w)
String	search, move, compare
Graphics	pixel and vertex operations, compression/decompression

Instruction Usage

- Designed versus actually used operations

Typical Instructions Provided by CISC

Data Movement	Load (from Memory) Store (to Memory) Memory-to-Memory Move Register-to-Register Move Input (from I/O Device) Output (to I/O Device) Push, Pop (to/from Stack)
Arithmetic	Integer (binary + decimal) or FP Add, Subtract, Multiply, Divide
Logical	not, and, or, set, clear
Shift	Shift Left/Right, Rotate Left/Right
Control (Jump/Branch)	Unconditional, Conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic read-modify-write)
String	search, translate

Top 10 80X86 Instructions

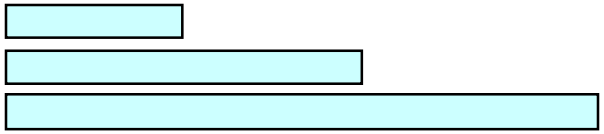
Rank	Instruction	Average % total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

Simple instructions dominate instruction frequency and most of the instructions in CISC are not used.

Make these instructions fast!
Amdahl's law – make the common cases fast!

Instruction Formats : Length, Operation

- Variable-length instructions
  - 80x86: 1 ~ 17 bytes, Digital VAX: 1 ~ 54 bytes
  - require multi-step fetch and decode
  - more flexible (but complex) and compact IS



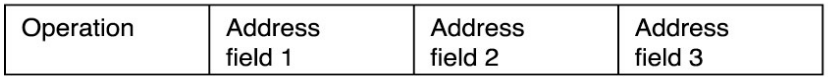
- Fixed-length instructions
  - Used in most RISC (Reduced Instruction Set Computers)
  - MIPS, PowerPC: Instructions are 4 bytes long.
  - Allow for easy fetch and decode.
  - Simplify pipelining and parallelism.
- Hybrid instructions : a mix of variable- and fixed-length instructions

- Opcode
  - unique code to specify the desired operation
- The type and size of the operands
  - Character (8 bits), half-word (eg: 16 bits), word (eg: 32 bits), single-precision floating point (eg: 1 word), double-precision floating point (eg: 2 words).

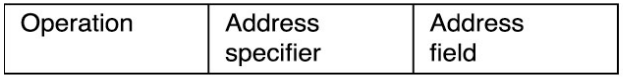
Encoding the Instruction Set

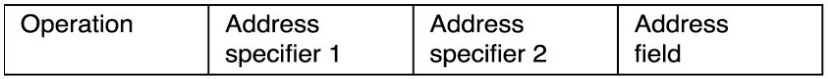
- How are instructions represented in binary format for execution by the processor?
- Things to be decided :
  - Number of registers
  - Number of addressing modes
  - Number of operands in an instruction
- Choices

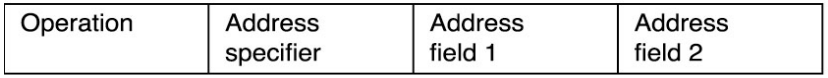
(a) Variable (e.g., VAX, Intel 80x86)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)







(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

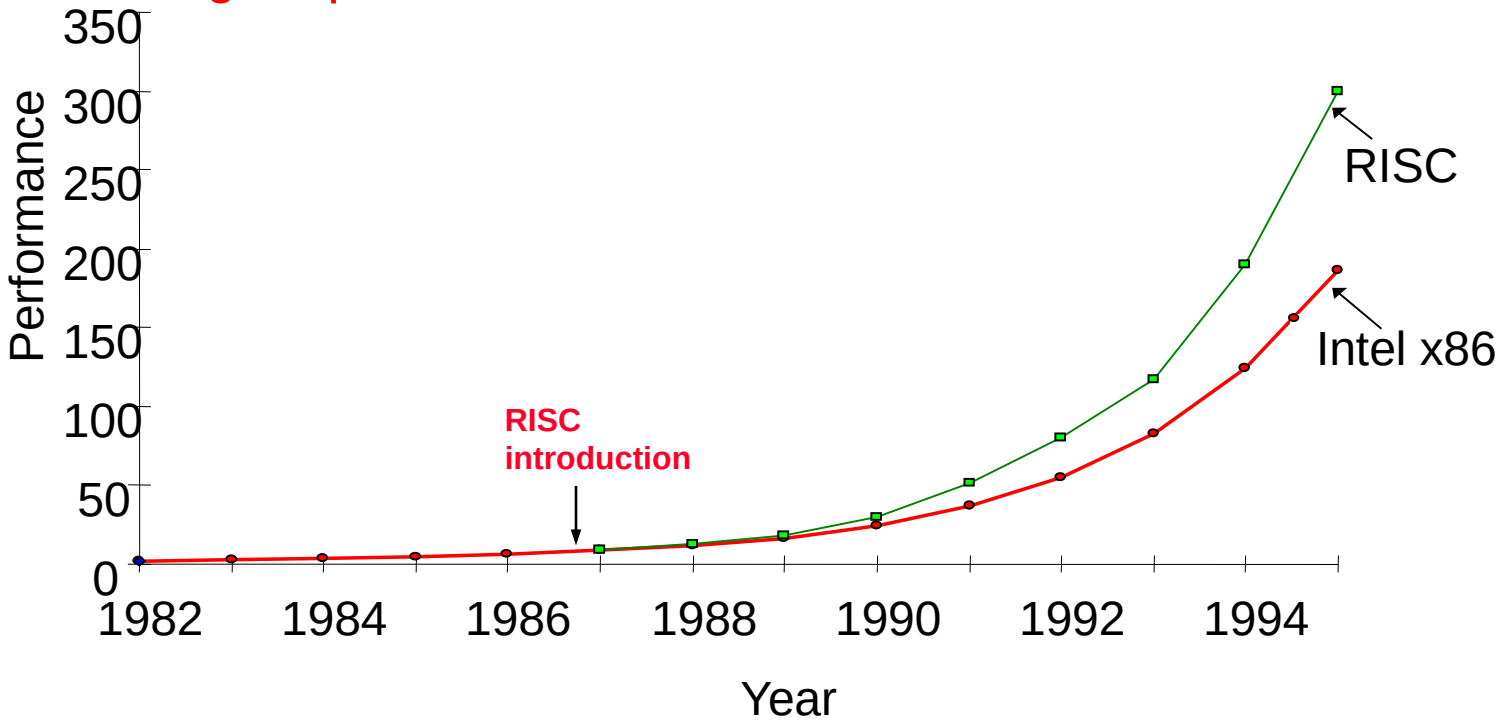
RISC vs. CISC

- Complex Instruction Set Computer (CISC) : x86-32(IA32)
  - > 1000 instructions, 1 to 15 bytes each
  - Single instruction performs complex operation
  - 10s of addressing modes
    - e.g. Mem[segment + reg + reg*scale + offset]
- Desktops/Servers

- Reduced Instruction Set Computer (RISC) : MIPS, ARM
  - Keep the instruction set small and simple, makes it easier to build/optimize hardware
    - ≈ 200 instructions, 32 bits each, 3 formats
    - all operands in registers
- Energy efficiency, Embedded Systems, Phones/Tablets

<참고> What is RISC and Why?

- RISC is an architecture design concept based on the principle that simpler hardware runs faster (e.g. MIPS). It uses smaller and regular instruction set to achieve performance, while relying on compiler technology to achieve functions used to done by complex instructions.
- Opposite to RISC is Complex Instruction Set Computer (CISC) (e.g. Intel x86). CISC believes complex instructions implemented in hardware can achieve higher performance.

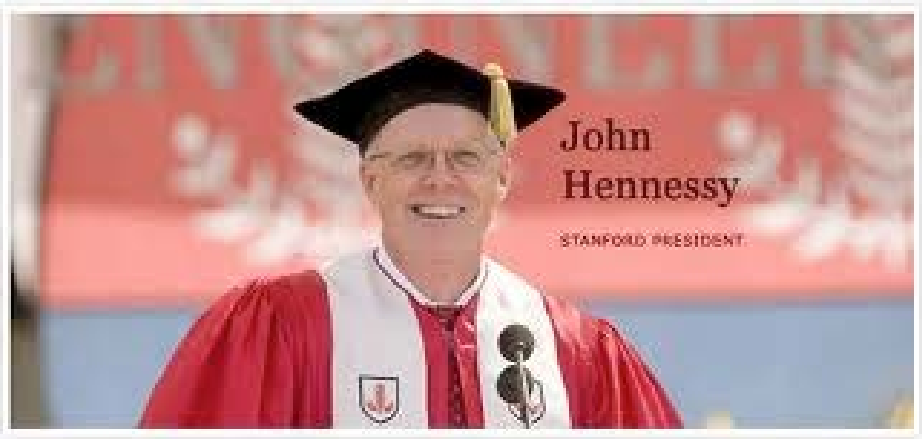


Courtesy D. Patterson

<참고> Reduced Instruction Set Computer (RISC)

- Dave Patterson
  - RISC Project, 1982
  - UC Berkeley
  - RISC-I: ½ transistors & 3x faster
  - Influences: Sun SPARC, namesake of industry

- John L. Hennessy
  - MIPS, 1981
  - Stanford
  - Simple pipelining, keep full
  - Influences: MIPS computer system, PlayStation, Nintendo



Mainstream ISAs

x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

Macbooks & PCs, Servers
(Core i3, i5, i7, M)
x86-64 Instruction Set

ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
Endianness	Bi (little as default)

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
ARM Instruction Set

MIPS

Designer	MIPS Technologies, Inc.
Bits	64-bit (32→64)
Introduced	1981; 35 years ago
Design	RISC
Type	Register-Register
Encoding	Fixed
Endianness	Bi

Digital home & networking equipment
(Blu-ray, PlayStation 2)
MIPS Instruction Set

* RISC-V: The *Free and Open RISC* Instruction Set Architecture https://riscv.org/

Summary

- Many possible implementations of the same ISA
  - x86 implementations: *8086 (c. 1978), 80186, 286, 386, 486, Pentium, Pentium Pro, Pentium-4, Core i7, AMD Athlon, AMD Opteron, Transmeta Crusoe, SoftPC*
  - MIPS implementations: *R2000, R4000, R10000, ...*
  - JVM: *HotSpot, PicoJava, ARM Jazelle, ...*
  - *RISC-V: RV32I, RV32E, RV64I, RV128I, …*
    - *Open-Source*

- ISA classes : Stack, Accumulator, and General purpose register

- Most current systems use general-purpose register(GPR) based ISA

Operations and Operands

- Examples pp. 65 ~ 69

a = b + c;

add a, b, c

Example Instructions(1)

- Basic arithmetic operation

```
add a, b, c        # a = b + c
sub a, b, c        # a = b – c
```

Regularity makes implementation simpler

- Arithmetic operation

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

It is the compiler's job to associate program variables with registers.

Example Instructions(2)

- Memory Operand Example

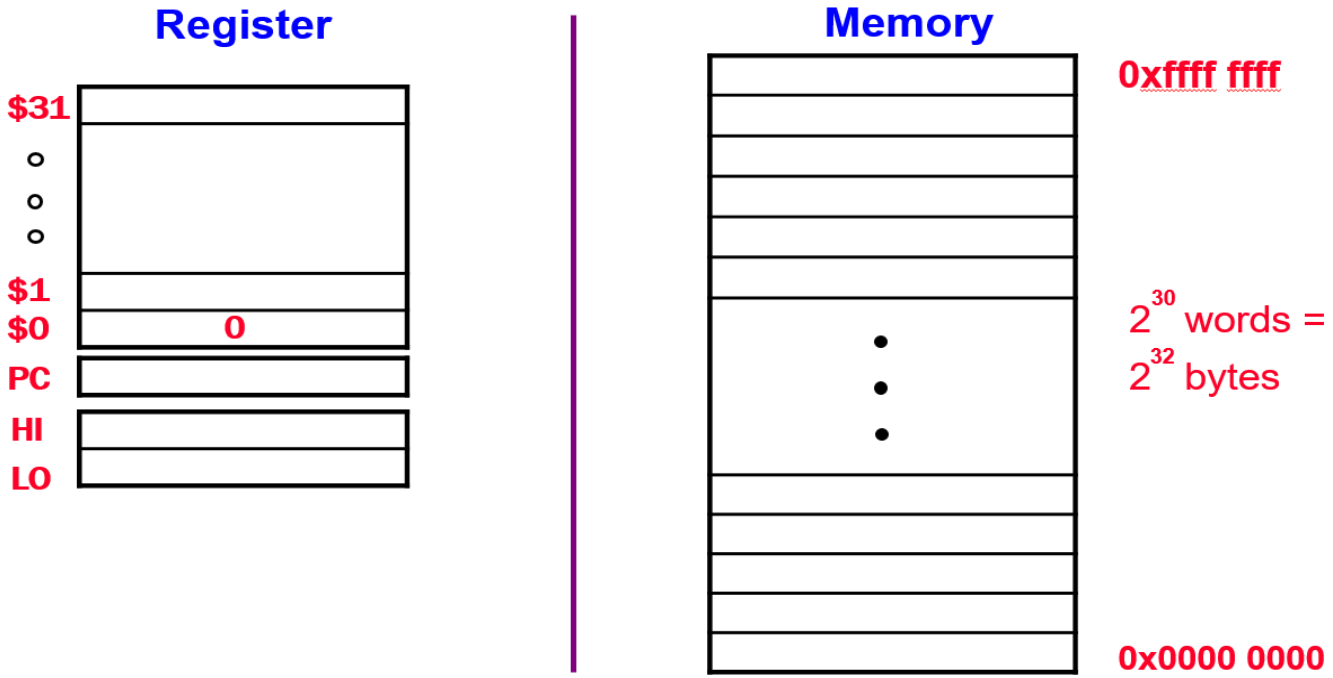
```
g = h + A[8];          // Assume g in $s1, h in $s2
                       // Base address of A in $s3
```

Compiled MIPS (Index 8 requires offset of 32)

```
lw $t0, 32($s3)        #32 is offset, $s3 base register
add $s1, $s2, $t0      #g = h + A[8]
```

MIPS Operands

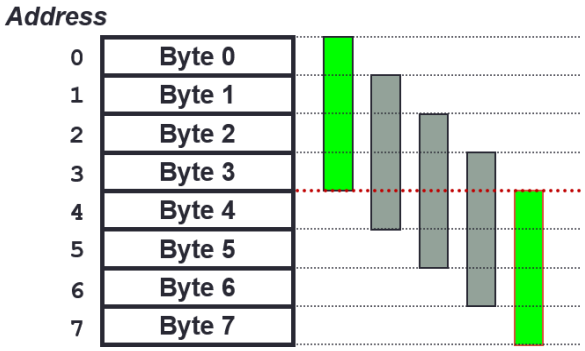
- MIPS is a load-store register architecture



- 32 registers, each 32-bit (4 bytes) long
- Memory addresses are 32-bit long, "byte / word addressable"
- Each word contains 32 bits (4 bytes)
  - The common unit of transfer between processor and memory
  - Also commonly coincide with the register size, the integer size and instruction size in most architectures

Memory Content : Big Endian vs. Little Endian

- Since 8-bit bytes are so useful, 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)

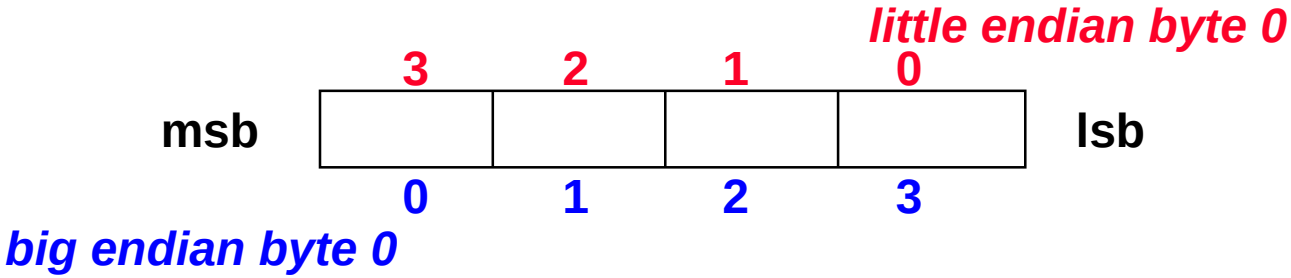


- Big Endian: leftmost byte is word address

  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- Little Endian: rightmost byte is word address

  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



https://www.youtube.com/watch?v=jhErugDB-34

Example Instructions(3)

❑ Immediate Operands

What about constant data?

e.g. a = b + 5

Constant data specified in an instruction:

addi $s3, $s4, 5

Make the common case fast!

Small constants are common (e.g. i++, i+=2)

Immediate operand avoids a load instruction

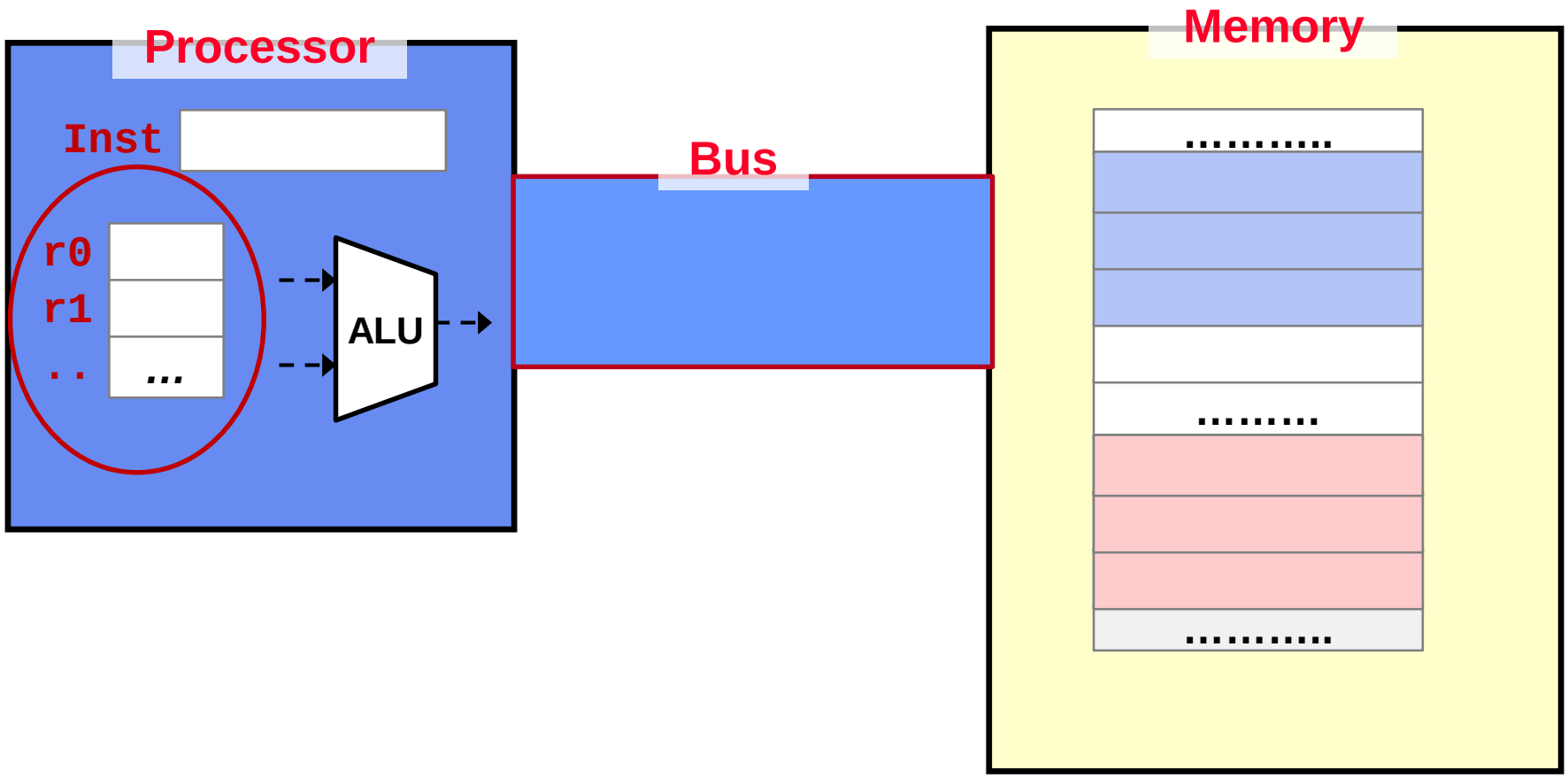
❑ Constant Zero

In MIPS, register 0 is a constant zero

Can be addressed as $zero

Useful for common operations *e.g. Copy registers*

add $t2, $s1, $zero



MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by 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 word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Signed and Unsigned Numbers (*Chap. 2.4*)

- Binary digit or bits
  - lsb vs. msb

- Three ways in which signed binary numbers may be expressed :
  - Signed Magnitude
  - 1's Complement
  - 2's Complement
  - For binary 1010 ?

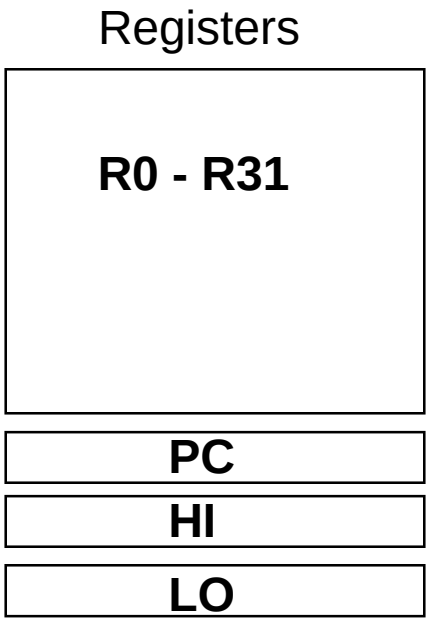
- Sign Extension

MIPS (RISC) Design Principles

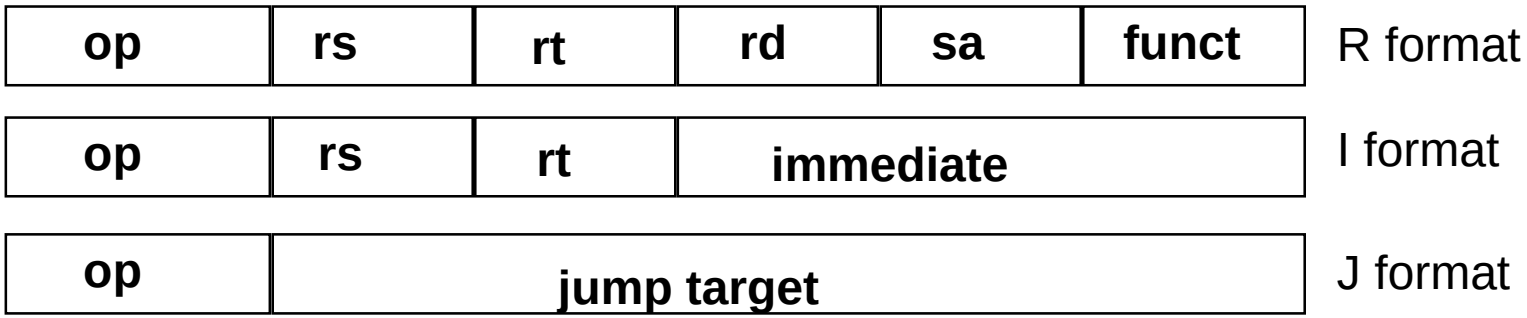
- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  -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

MIPS-32 ISA

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



- 3 Instruction Formats: all 32 bits wide



MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

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

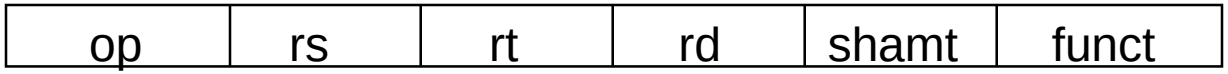
- Each arithmetic instruction performs one operation
- Each specifies exactly three operands that are all contained in the datapath's register file (`$t0,$s1,$s2`)

destination ← source1 op source2

- Instruction Format (R format)

MIPS Instruction Fields

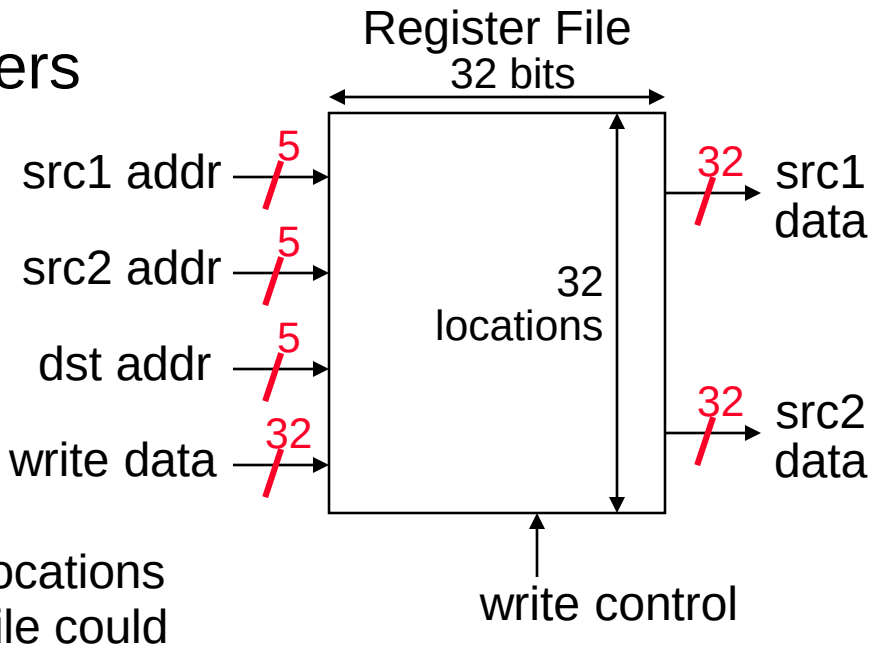
- MIPS fields are given names to make them easier to refer to



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 Register File

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



- Registers are
  - Faster than main memory
    - But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
    - Read/write port increase impacts speed quadratically
  - 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)

Aside: MIPS Register Convention

Name	Register Number	Usage
$zero	0	constant 0 (hardware)
$at	1	reserved for assembler
$v0 - $v1	2-3	returned values
$a0 - $a3	4-7	arguments
$t0 - $t7	8-15	temporaries
$s0 - $s7	16-23	saved values
$t8 - $t9	24-25	temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return addr (hardware)

$k0-$k1 (registers 26-27) are reserved for the operation system.

MIPS Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory

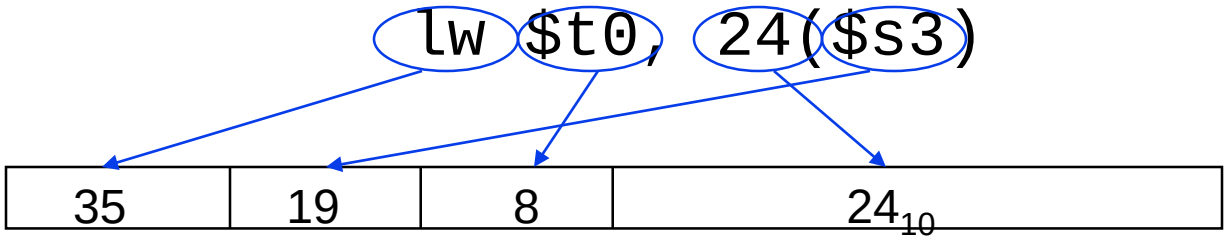
```
lw   $t0, 4($s3)  #load word from memory

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 offset value
  - A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of the address in the base register

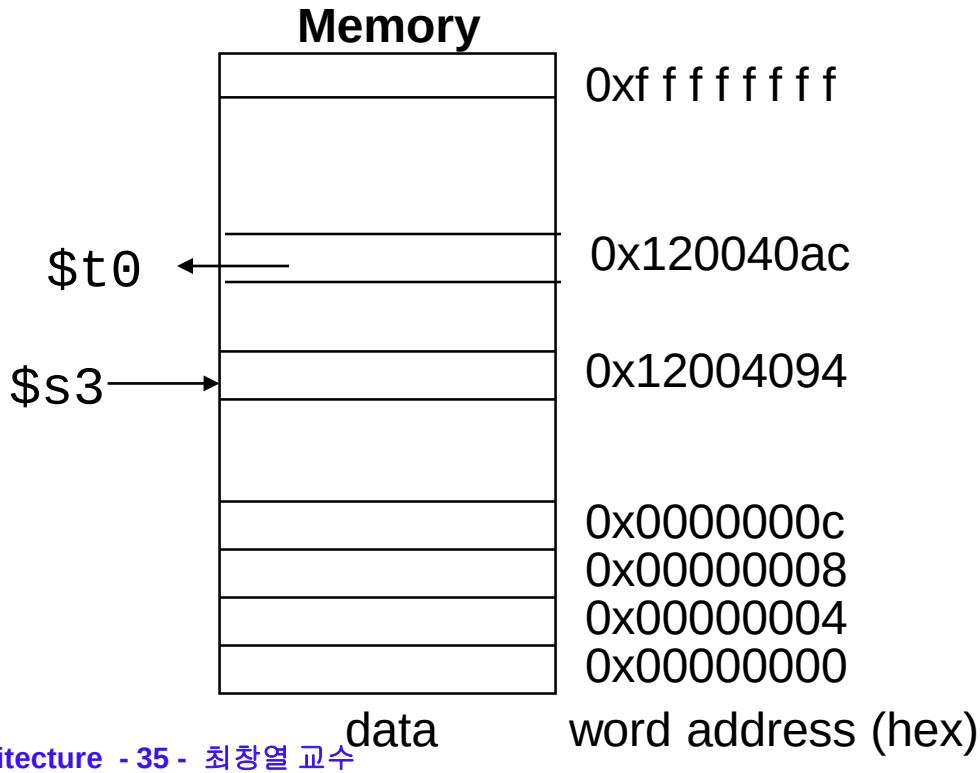
Machine Language - Load Instruction

- Load/Store Instruction Format (I format):



24_{10} + $s3 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac

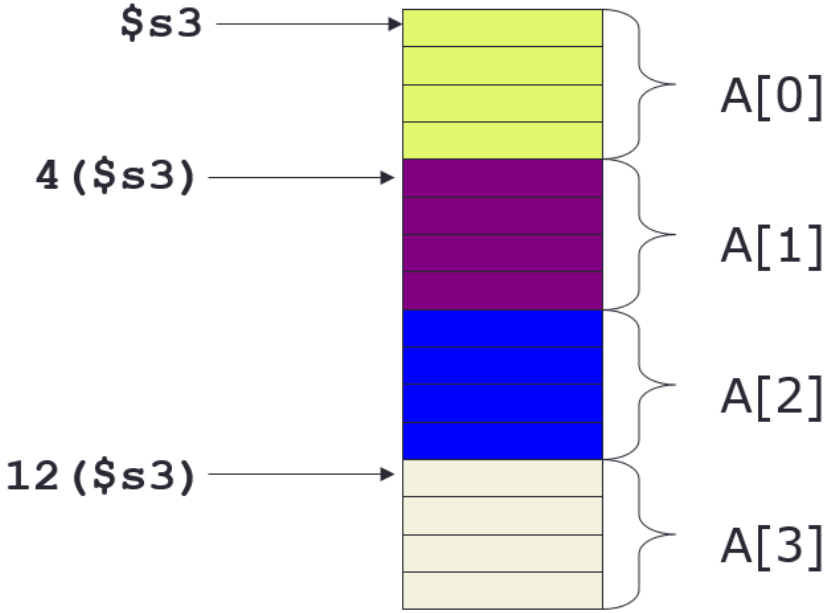


Memory Access Example : Array

- Assume 4 bytes per element

C Statement to translate	Variables Mapping
`A[3] = h + A[1];`	`h` ➔ `$s2` base of `A[]` ➔ `$s3`

```
lw   $t0, 4($s3)
add  $t0, $s2, $t0
sw   $t0, 12($s3)
```



MIPS Immediate Instructions

- Small constants are used often in typical code
- Possible approaches?
  - put “typical constants” in memory and load them
  - create hard-wired registers (like $zero) for constants like 1
  - have special instructions that contain constants !

```
addi $sp, $sp, 4     #$sp = $sp + 4

slti $t0, $s2, 15    #$t0 = 1 if $s2<15
```

- Machine format (I format):

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}

- Example pp. 84

- Figure 2.6

- Check Yourself!

Two Key Principles of Machine Design

- Instructions are represented as numbers and, as such, are indistinguishable from data

  Program = A sequence of instructions

- Programs are stored in alterable memory (that can be read or written to) just like data

- Stored-program concept Fig. 2.7
  - 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 – leads industry to align around a small number of ISAs

MIPS Logical Operations Overview

- Arithmetic instructions view the content of a register as a single quantity (signed or unsigned integer)
- View register as 32 raw bits rather than as a single 32-bit number ➔ Possible to operate on individual bits or bytes within a word

Logical operation	C operator	Java operator	MIPS instruction
Shift Left	<<	<<	sll
Shift right	>>	>>, >>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT*	~	~	nor
Bitwise XOR	^	^	xor, xori

*with some tricks

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)


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

- Instruction Format (I format)

MIPS Instructions for Making Decisions

- To perform general computing tasks, we need to:
  - Make decisions
  - Perform iterations
- Decision-making instructions
  - Alter the control flow of the program
  - Change the next instruction to be executed
- Two types of decision-making statements in MIPS
  - Conditional (branch)

```
bne $t0, $t1, label      #"branch if not equal"
                         #C code: if (a != b) goto label

beq $t0, $t1, label      #"branch if equal"
                         # C code: if (a == b) goto label
```

  - Unconditional (jump)

```
j label
```

MIPS Control Flow Instructions

- MIPS conditional branch instructions:

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

- How is the branch destination address specified?

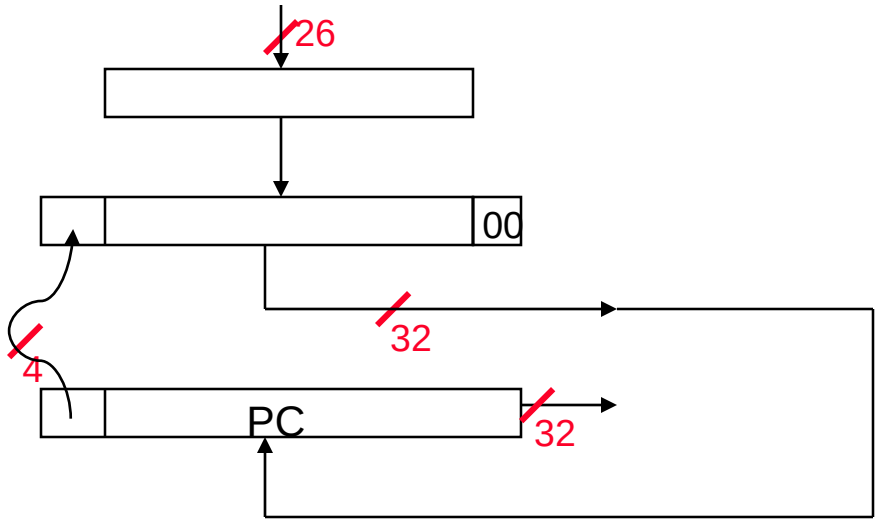
Other Control Flow Instructions

- 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

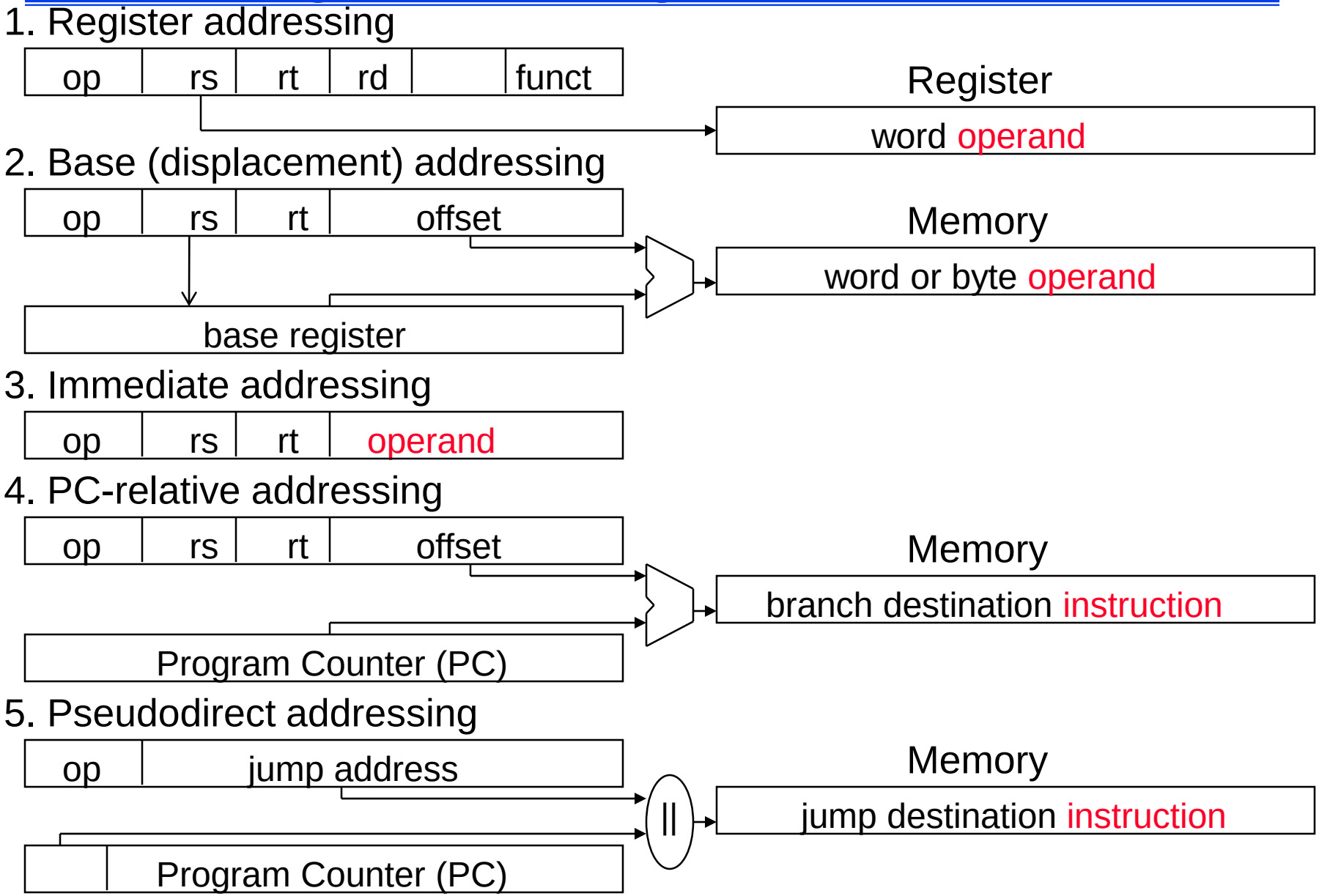
```
jr   $ra                  #return
```

- Instruction format (R format):

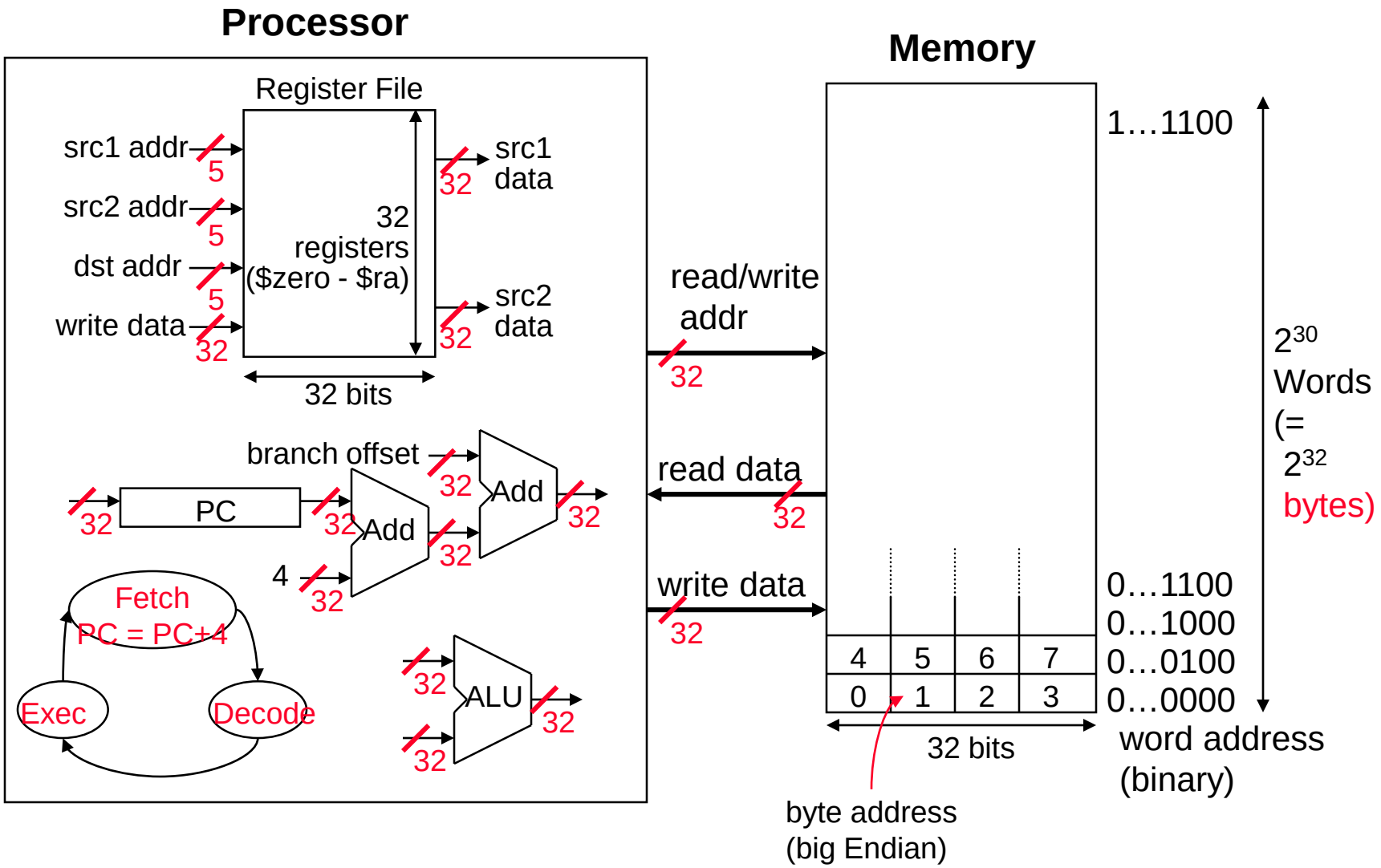
MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional 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

Addressing Modes Fig. 2.18



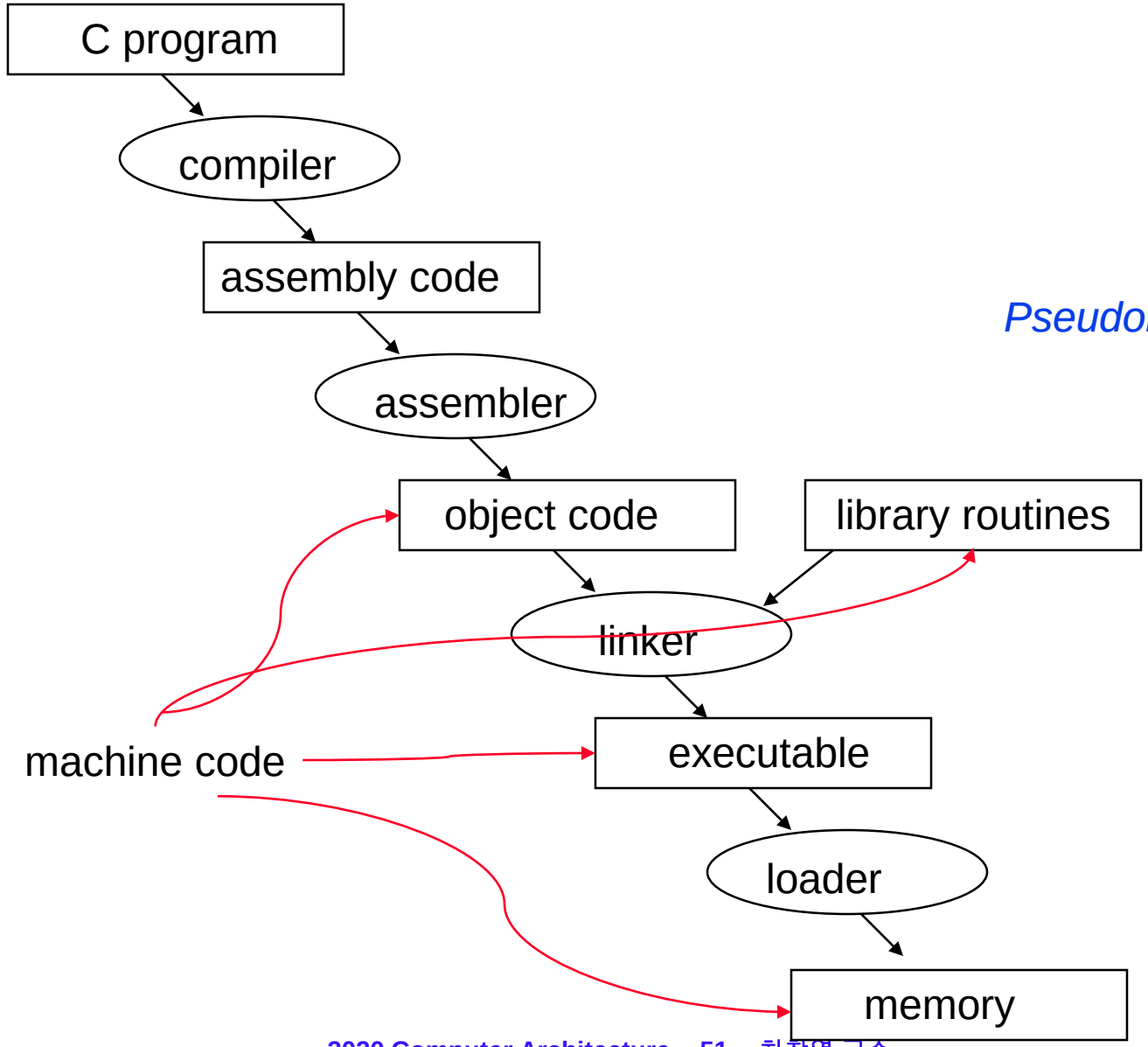
MIPS Organization So Far



- Check Yourself!

- Reading : Hardware / Software Interfaces

Fig. 2.21 The C Code Translation Hierarchy



Pseudoinstruction?

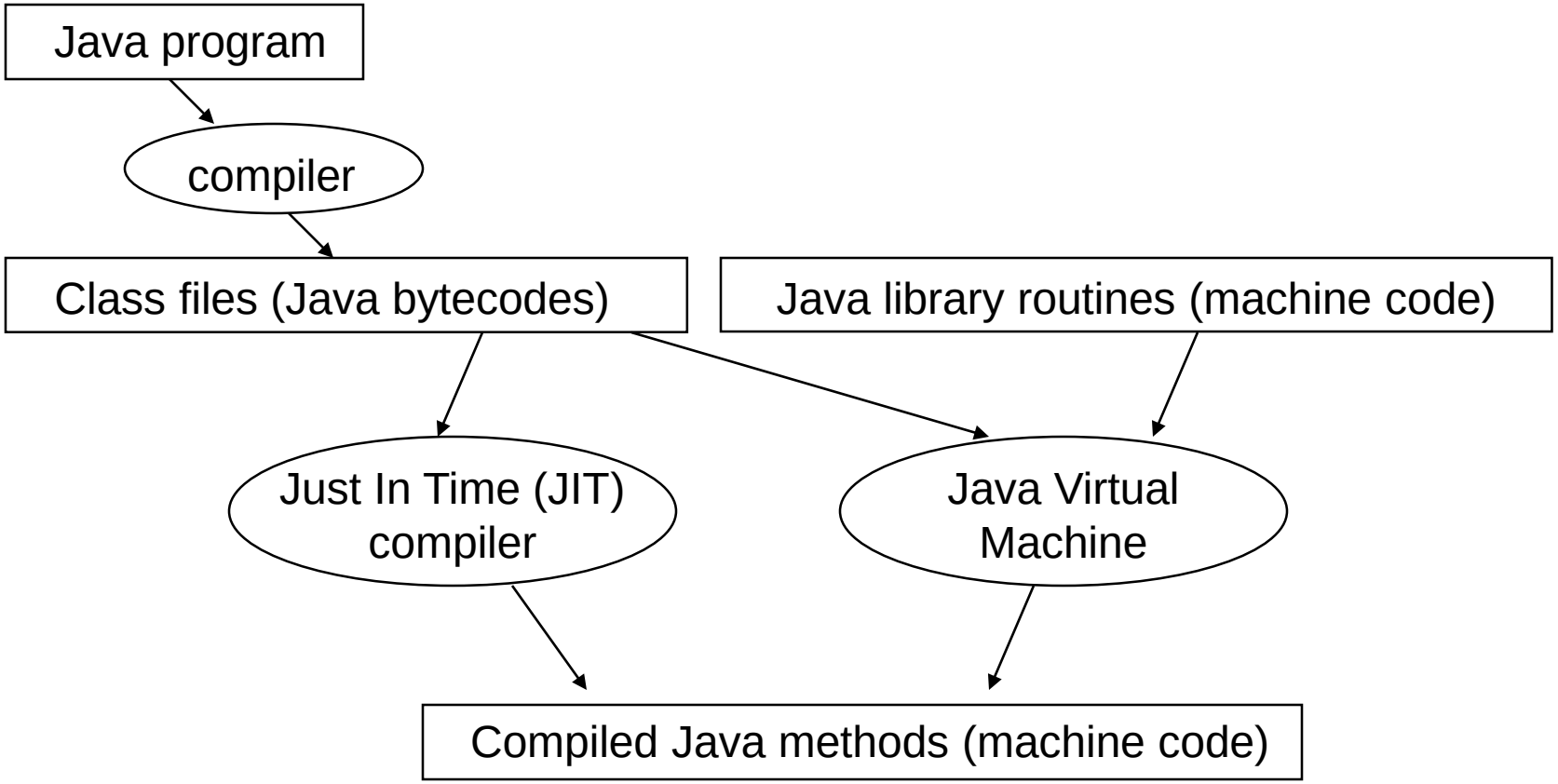
Compiler Benefits

- Comparing performance for bubble (exchange) sort
  - To sort 100,000 words with the array initialized to random values on a Pentium 4 with a 3.06 clock rate, a 533 MHz system bus, with 2 GB of DDR SDRAM, using Linux version 2.4.20

gcc opt	Relative performance	Clock cycles (M)	Instr count (M)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (proc mig)	2.41	65,747	44,993	1.46

- The unoptimized code has the best CPI, the O1 version has the lowest instruction count, but the O3 version is the fastest. Why?

The Java Code Translation Hierarchy



Summary

- Real Stuff : ARM, x86

- Concluding Remarks