CSE3421
Computer Architecture

Chapter 2: Instructions:
Language of the Computer

Xiaodong Zhang

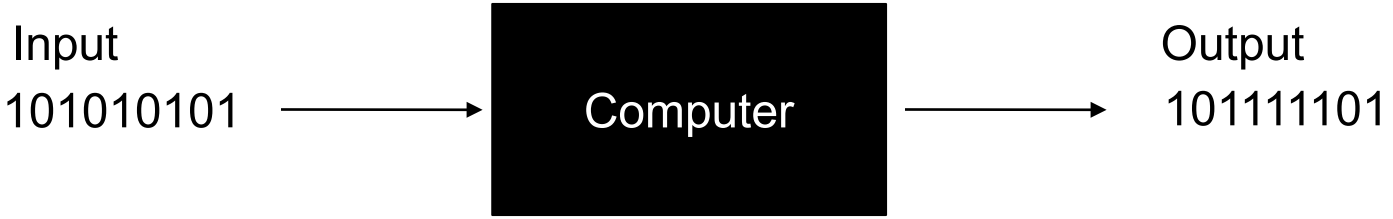
Multilevel Abstractions in Computer Systems

- Each level of abstraction builds on a more detailed lower level
  - Each level hides the detailed operations of the next
  - Multilevel abstractions create a top-down stack providing more and more detailed operations
- On the top, a processing request is made by a human expressing
  - SQL (Structured Query Language)
  - High level language programming. …
  - Converted to executable code by tools
- The request from the top will reach to the bottom level
  - To be executed by detailed operations in hardware
  - Instruction Set Architecture (ISA) serves as the interface between top logical levels and the hardware

Two Facts in Computer Harware

1. Instructions are represented as sequences of 0/1 and, as such, are indistinguishable from data

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



- Each instruction corresponds to a sequence of 0/1 signals
  - Human knows instruction, and machine understands 0/1 signals
  - Can we regulate a set of instructions and the matched 0/1 signals?
  - We borrow the idea of “vocabulary and grammar” for any language
- Instruction set architecture is a set of basic instructions and their 0/1 codes provide signals to drive a particular processor
  - With ISA of the machine, one can write assembly code for computing

IBM Compatibility Problems in Early 1960s

- By early 1960s, IBM had 4 different lines of computers
  - IBM 701 series
  - IBM 650 series
  - IBM 702 series
  - IBM 1401 series

- Each system had its own
  - Instruction set architecture (ISA)
  - I/O system and tape/disk storage systems
  - Assemblers, compilers, libraries, …
  - Providing service to business, scientific applications, real-time…

- **IBM System/360 started one ISA for all the systems!**
  - **April 7, 1964** (Fred Brooks and his team)
  - **This is a milestone in computer architecture**

Unified ISA and standardized design of CPU

- ❑ Separation between Datapath and control unit in CPU
  - **Datapath:** stored data in registers, arithmetic and logic ops (ALU)
  - **Control unit:** a sequence operations based on instructions
  - **Instruction cache:** on-chip store for frequently used instructions
  - **Data cache:** on-chip store for frequently used data

ISA and CPU design have Intellectual Properties

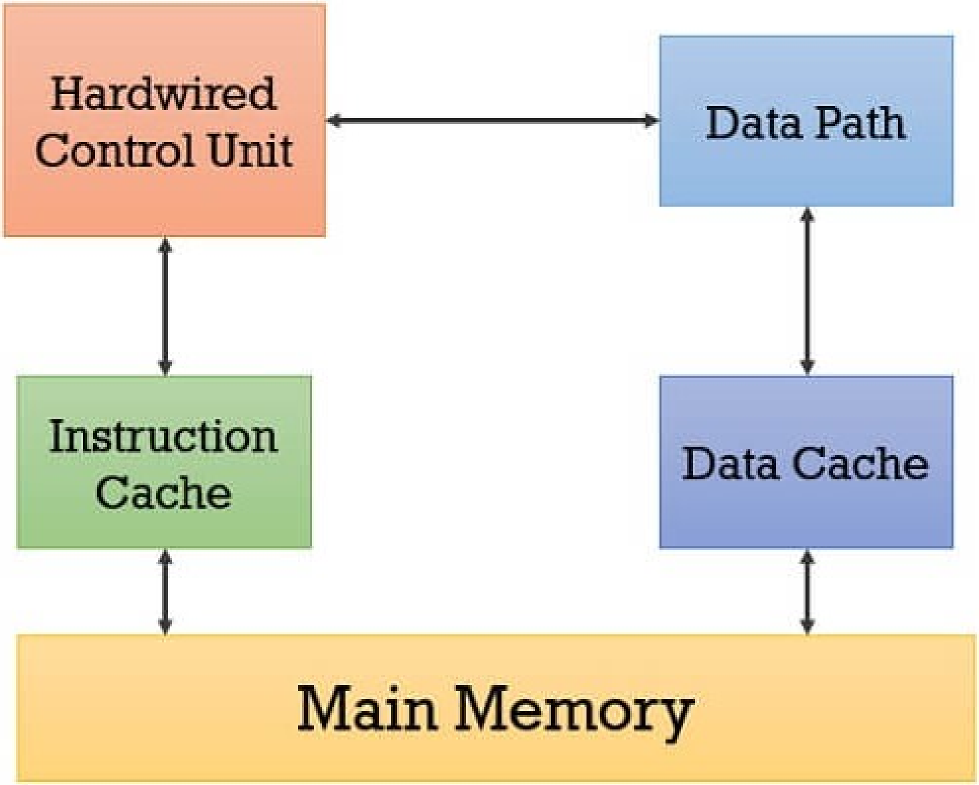
- ❑ In an automobile

Datapath: engine and gas injector

Control unit: gas and break pedals

Instruction cache: driver

Data cache: gas tank



RISC versus CISC

- CISC – Complex Instruction Set Computers
- RISC – Reduced Instruction Set Computers
  - Only supports instructions that can complete within a cycle
    - e.g., Load memory to a register, compute on two registers, store memory to a register
- Consider multiplication

Memory contents

Ans: 9	1	3
2	3	10

  - A CISC would include an assembly instruction like: MULT 2:2, 1:3, 1:1 （content in (2,2) multiply content (1,3), => (1,1)
  - A RISC would involve multiple (4) instructions: Load A, 2:2,

Load B, 1:3, Prod A, B, STORE, 1:1, A

What Instructions should hardware support?

- Approach 1: Hardware should support instructions that programmers want
  - Optimize common instructions
  - Programs have few lines of code
  - Complicated logic is built in

What Instructions should hardware support?

- Approach 1: Hardware should support instructions that programmers want
  - Optimize common instructions
  - Programs have few lines of code
  - Complicated logic is built in

- Approach 2: Hardware should support only the instructions that can complete quickly (e.g., 1 cycle)
  - Easily optimize the order and sequence of instructions
  - Hardware has simple logic
  - Depend on compilers to convert high-level code (few lines) into efficient assembly code

RISC versus CISC Today

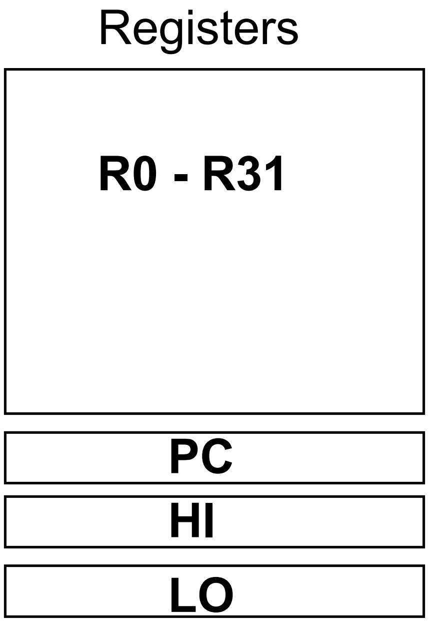
- ❑ So which methodology do we use in practice today?

RISC versus CISC Today

- RISC architectures-
  - Alpha (DEC), ARC, **ARM**, AVR (Norway), **MIPS**, PA-RISC (HP), PIC, Power Architecture (including PowerPC, IBM), SuperH (Hitachi), and SPARC (Sun), **AMD**
  - Microprocessor w/o Interlocked Pipeline Stages (MIPS)
    - PlayStation, PlayStation 2, Nintendo 64 (discontinued), PlayStation Portable game consoles, and residential gateways like Linksys WRT54G
- CISC architectures
  - System/360 (IBM), PDP-11 (DEC), VAX (DEC), 68000 (Motorola), and **x86 (Intel), AMD**
- History
  - RISC was invented by John Cocke (IBM) in 1974, 1987 Turing Award
  - RISC is promoted in research and education by P&H, 2018 Turing Award
  - CISC has been powerful with support of hardware and compiler for high-end comping, RISC is efficient for low-end apps, such as cell phones.
  - June 2020, ARM-based Fugaku became fastest supercomputer!

MIPS-32 ISA

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management (PC)
  - Special: HI and LO to store long word, high bits in HI, low bits in LO



3 Instruction Formats: all 32 bits wide

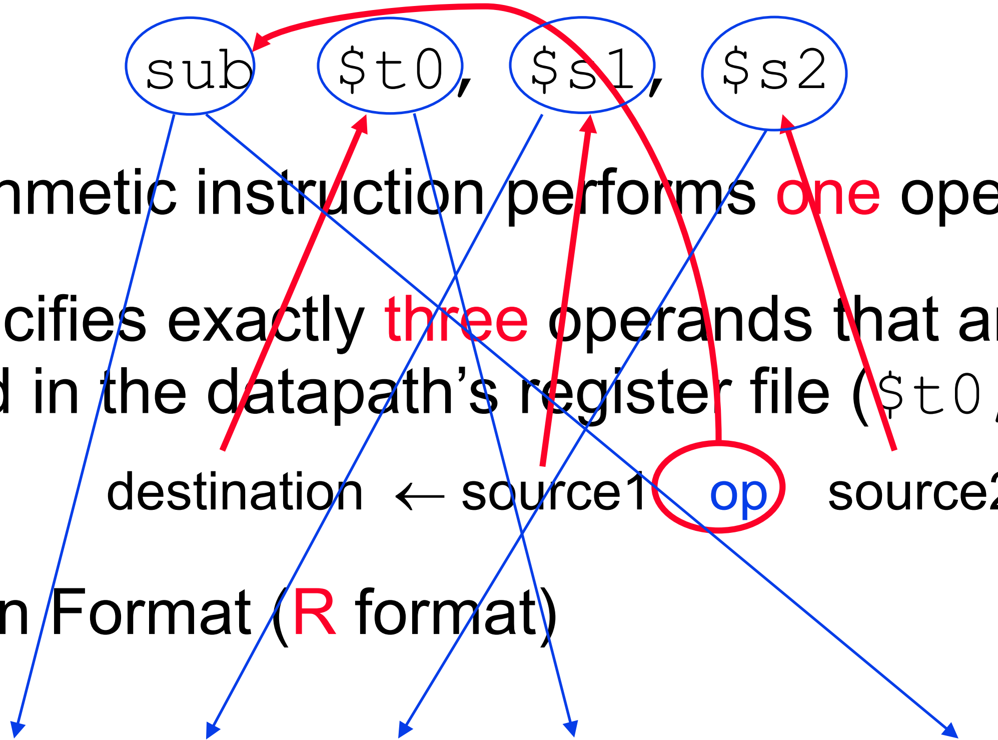
op	**rs**	**rt**	**rd**	**sa**	**funct**	R format
op	**rs**	**rt**	**immediate**			I format
op	**jump target**					J format

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)

0	17	18	8	0	0x22

Note: the binary code is represented by decimal in each entry

Prefix 0x means hexadecimal (base 16) 0x22 = 34 in decimal

MIPS Instruction Fields

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

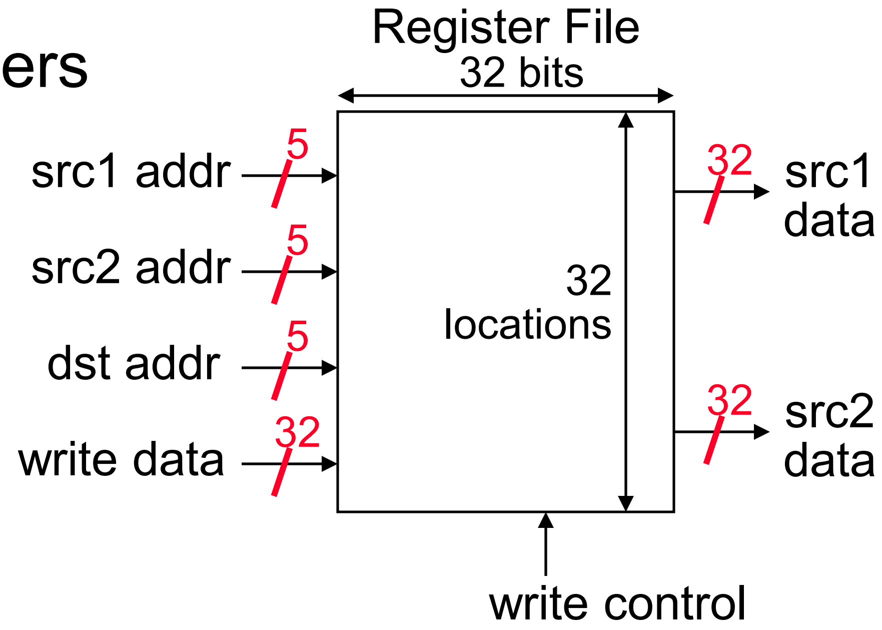
op	rs	rt	rd	shamt	funct

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 Register File

- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port
  - Each 5-bit address maps to

    a unique register location



- 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 by increasing 1 bit of the register name address)
    - Increasing # of Read/write ports would improve throughput due to concurrency
  - Can hold a small set of variables
    - 32 variables, constant, and pointers
    - Memory address is represented by at a whole word (e.g., 32 bits)

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 (fun calls)	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$k1-$k2	26-27	reserved for kernel	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

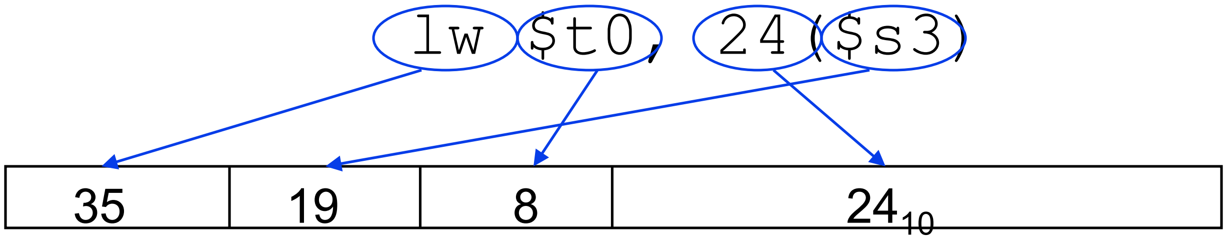
```
 lw   $t0, 4($s3)  #load word from memory

  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
- 4($s3) = 32-bit memory address stored in $3 + 4 (offset)
- 8($s3) = 32-bit memory address stored in $3 + 8 (offset)
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value (16 bits)
    - 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):



```
lw $t0, 24($s3)
```

35	19	8	24_{10}

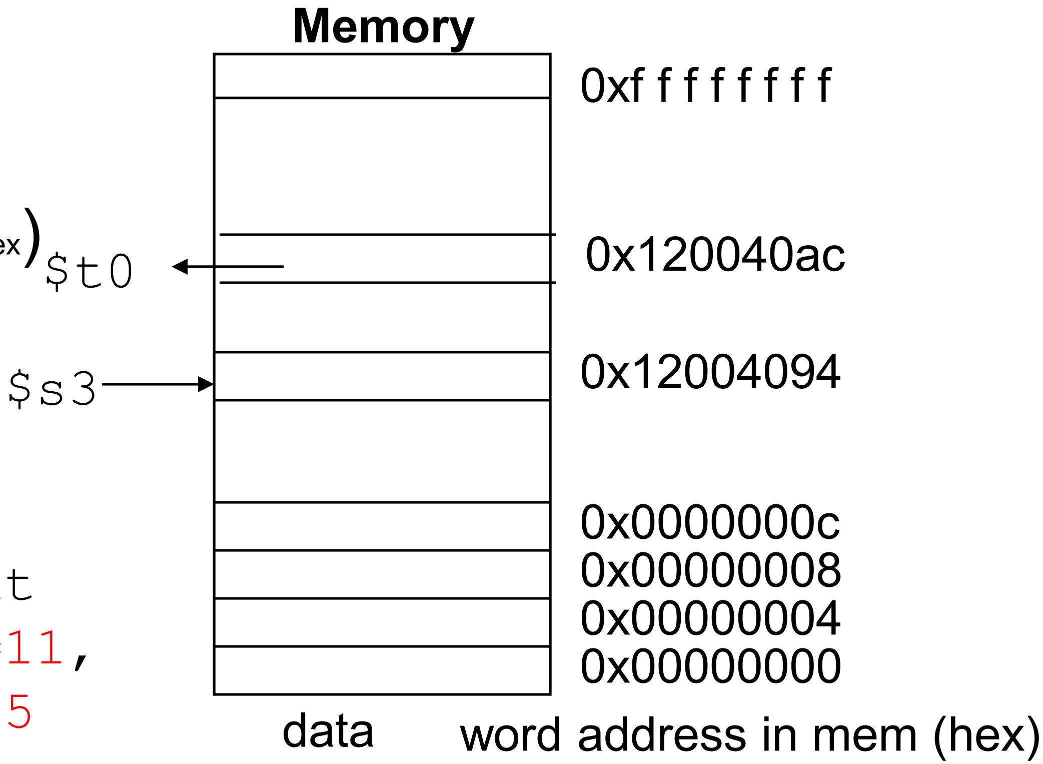
24_{10} + $s3 =

. . . 0001 1000 (24_{10}, 18_{hex})

+ . . . 1001 0100

. . . 1010 1100 =

0x120040ac



Every 4 bits represent one hex bit: a=10, b=11, c=12, d=13, e=14, f=15

Byte-addressable Memory

- Most architectures are **byte addressable:** each unique memory address points to a Byte (8 bits) in memory
  - Memory space can be viewed as an array of Bytes
  - If memory address is 8-bit, there will be 256 locations from 00000000 to 11111111, or 256-Byte memory space
- One reason for Byte is its representation of a character
  - 8-bit ASCII code is used for each character
  - Binary values 0-127 are for regular characters, 128-255 are for special characters, such as European languages
- Registers are not in the scope of memory address
  - The length of a register is 32 bits (4 Bytes)
  - In what order does a 4-Byte (word) register communicate with a Byte addressable memory?

Byte–addressable Memory (continued)

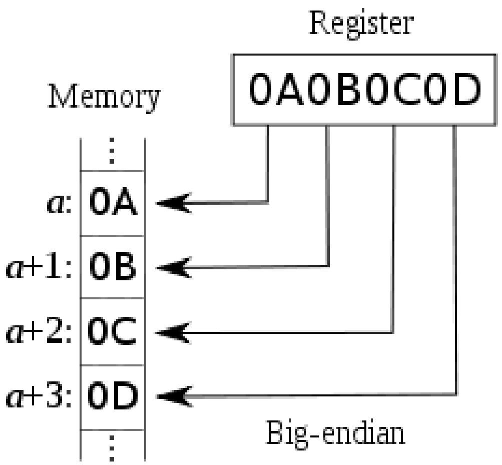
- Alignment restriction - the memory address of a word must be on word boundaries (a multiple of 4 in MIPS)
    - same concept for House lot.

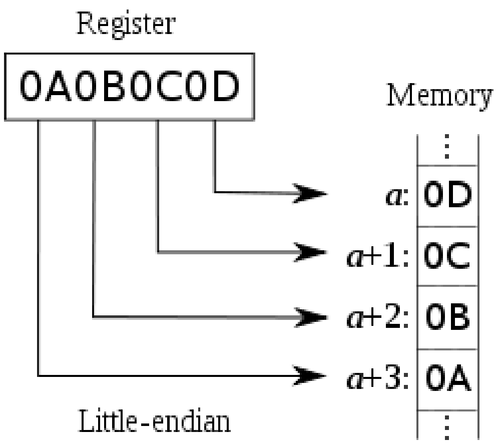
- Big Endian: most significant byte first

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

- Little Endian: least significant byte first

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





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
```

0x28	19	8	16 bit offset

- What 8 bits get loaded and stored?
  - load operation places the byte from memory indicated by the location in **the rightmost 8 bits** of the destination register
    - what happens to the other bits in the register? Setting to 0
  - store operation takes the byte from **the rightmost 8 bits** of a register and writes it to the location of the byte address in memory
    - what happens to the other bits in the memory word? No change

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

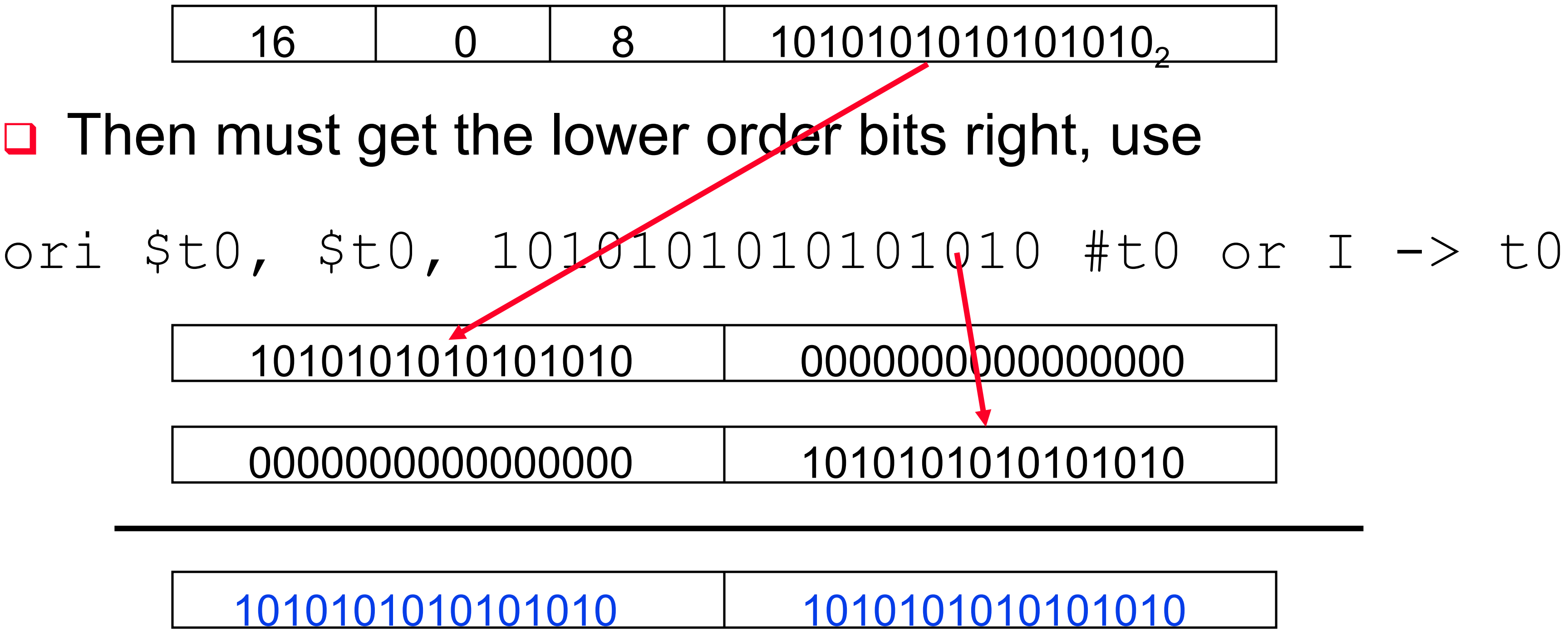
- The constant is kept inside the instruction itself!
  - Immediate format limits values to the range $+2^{15}–1$ to -2^{15} (16 bits)

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 #-> t0 upper
```



16	0	8	1010101010101010$_2$

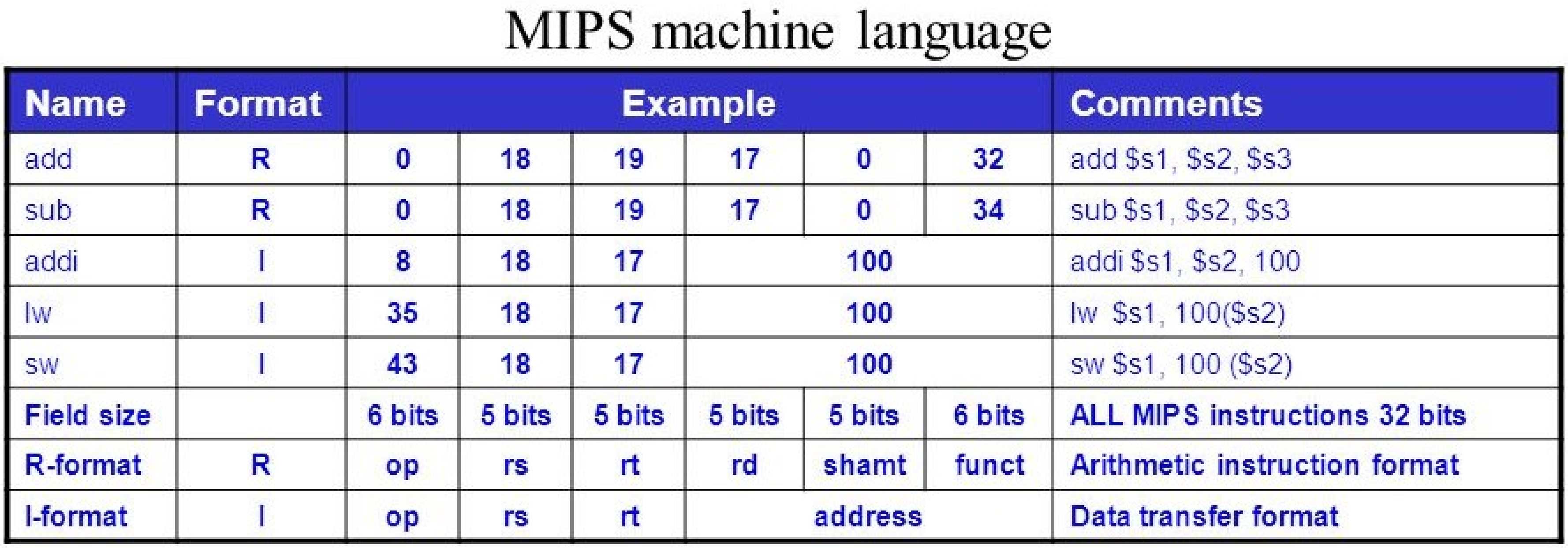
- Then must get the lower order bits right, use

```
ori $t0, $t0, 1010101010101010 #t0 or I -> t0
```

1010101010101010	0000000000000000
0000000000000000	1010101010101010

1010101010101010	1010101010101010

A Summary of MIPS Instruction Set (R and I Types)



MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1, $s2, $s3
sub	R	0	18	19	17	0	34	sub $s1, $s2, $s3
addi	I	8	18	17	100			addi $s1, $s2, 100
lw	I	35	18	17	100			lw $s1, 100($s2)
sw	I	43	18	17	100			sw $s1, 100 ($s2)
Field size		**6 bits**	**5 bits**	**5 bits**	**5 bits**	**5 bits**	**6 bits**	**ALL MIPS instructions 32 bits**
R-format	**R**	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**	**Arithmetic instruction format**
I-format	**I**	**op**	**rs**	**rt**	**address**			**Data transfer format**

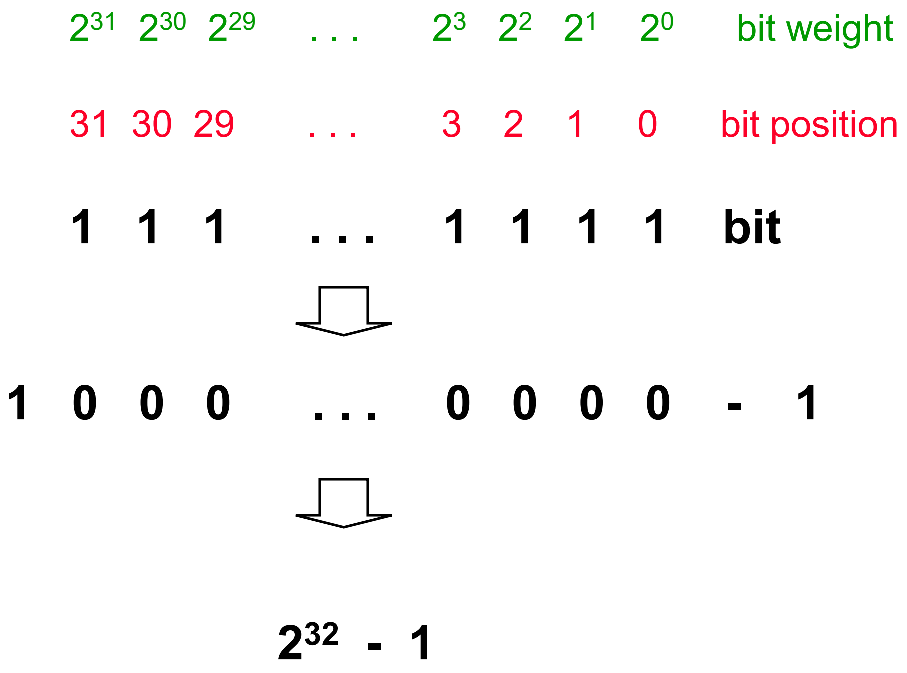
- ❑ Each instruction corresponds to a sequence of 0/1 signals
  - Human knows instruction, and machine understand 0/1
  - Can we regulate a set of instructions and the matched 0/1 signals?
  - We borrow the idea of “vocabulary and grammar” for any language
- Instruction set architecture is a set of basic instructions and 0/1
  - With ISA of the machine, one can write assembly code for computing

Number systems used in computer architecture

- Decimal number system based on Arabic number
  - Independently contributed by Chinese, Egyptians, Greeks, Indian, Iraqis, and Iranians about 2,000 years ago.
  - The number system is based on 10 fingers of human being and is probably the only commonly accepted system to communicate
    - Languages, Weight, Distance, Temperature, others are different
- Binary system
  - Used by Egyptian for ½, ¼, 1/8, …. 1200 BC
  - Used by Chinese for Yin and Yang, more than 1,000 years ago
  - Used by Indians in proportions 200 BC
  - Leibniz (German) studied binary numbers in mathematics in 1676
  - The real applications of binary system were as early as logic circuit
    - The basic signal of a gate is on and off by diode
- Hexadecimal: 16-based (0-15)
  - Expressed by 4-bit: a 4-byte word is represented by 8 numbers

Review: Unsigned Binary Representation

Hex	Binary	Decimal
0x00000000	0…0000	0
0x00000001	0…0001	1
0x00000002	0…0010	2
0x00000003	0…0011	3
0x00000004	0…0100	4
0x00000005	0…0101	5
0x00000006	0…0110	6
0x00000007	0…0111	7
0x00000008	0…1000	8
0x00000009	0…1001	9
	…	
0xFFFFFFFC	1…1100	$2^{32} - 4$
0xFFFFFFFD	1…1101	$2^{32} - 3$
0xFFFFFFFE	1…1110	$2^{32} - 2$
0xFFFFFFFF	1…1111	$2^{32} - 1$



2^{31} 2^{30} 2^{29} . . . 2^{3} 2^{2} 2^{1} 2^{0} bit weight

31 30 29 . . . 3 2 1 0 bit position

1 1 1 . . . 1 1 1 1 bit

1 0 0 0 . . . 0 0 0 0 - 1

2^{32} - 1

Prefix 0x = Hexadecimal
0x0000000A = 10
0x0000000B = 11
…
0x0000000F = 15

How to represent positive and negative numbers?

- Most significant digit is used as the sign (32 bits in total)
  - 0 is positive and 1 is negative
  - The maximal positive number is **2^{31} - 1 => (0 111….111)**
  - The maximal negative number is **-2^{31} => (1 000 … 000)**
- Converting between positive and negative values
  - 2's compliment: Not each bit, and then add 1 to the whole word
- Signed binary to decimal conversion

$x31*(-2^{31}) + x30*(2^{30}) + x29*(2^{29}) + \ldots + x1*(2^{1}) + x0*(2^{0})$,

where xi, i=0,… 31, represents each bit value.

Review: Signed Binary Representation

-2^3 =

$-(2^3 - 1)$ =

$2^3 - 1$ =

2’sc binary	decimal
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

complement all the bits

0101

and add a 1

0110

1011

and add a 1

1010

complement all the bits

MIPS Shift Operations

- To pack and unpack 8-bit characters into 32-bit words
- Shifts move all the bits in a word left or right

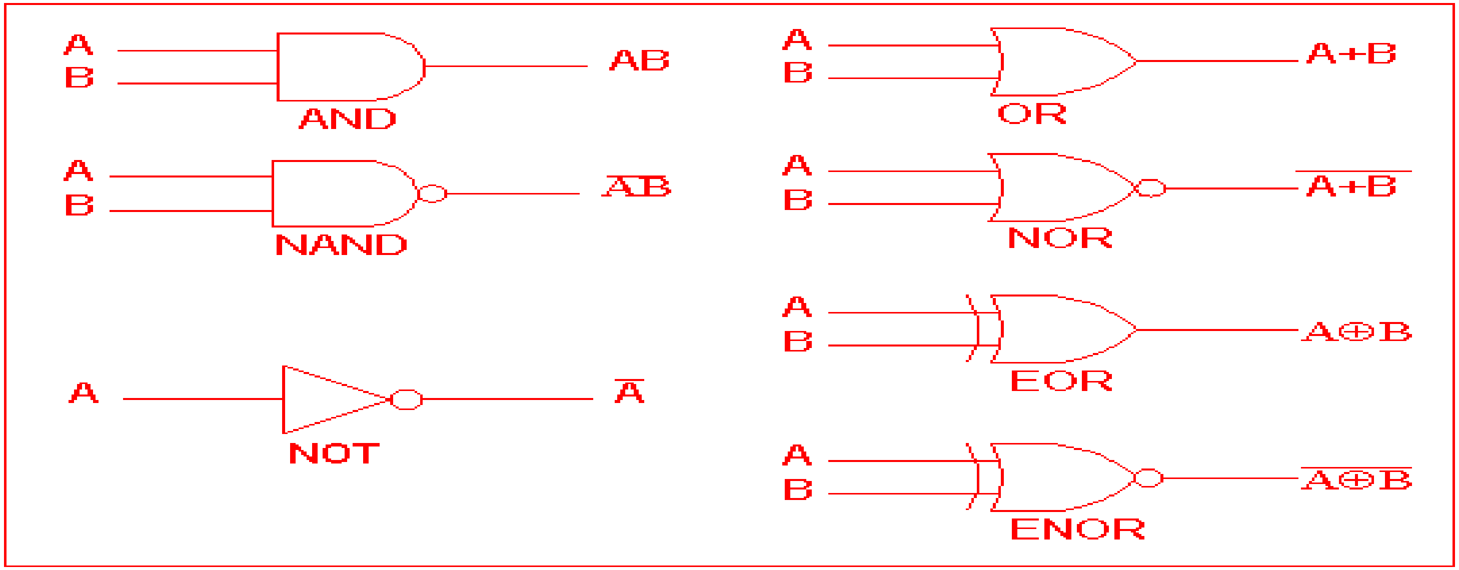
```
sll $t2, $s0, 8    #$t2 = $s0 << 8 bits
srl $t2, $s0, 8    #$t2 = $s0 >> 8 bits
```

- Shift left (right) logic => sl(r)l
- Instruction Format (R format)

0		16	10	8	0x00

- Such shifts are called logical because they fill with zeros (the “seats” become empty after shifting)
  - Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 – 1$ or 31 bit positions

Basic Logic gates and operations



INPUTS		OUTPUTS					
A	B	AND	NAND	OR	NOR	EXOR	EXNOR
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

EOR = XOR: “exclusive or”, output is 1 only one or other input is 1.

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

```
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, 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

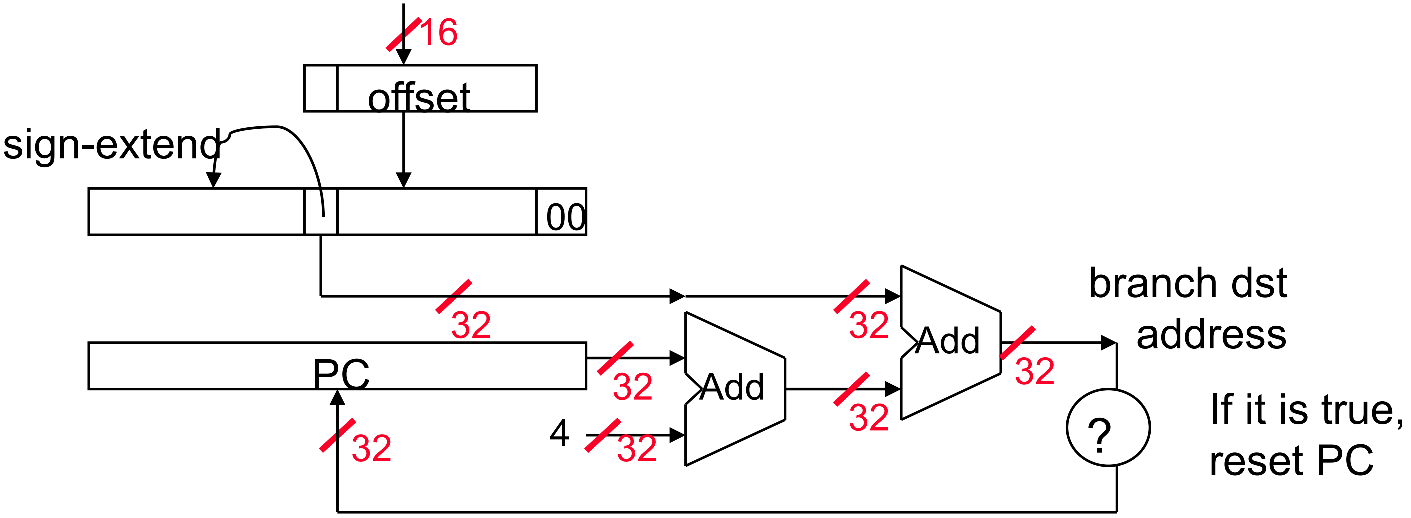
- How is the branch destination address specified?

- PC register (current instruction address)+4 (next word) + offset*4
- Branch address is a multiple of 4 from the offset (two-bit shift to left)

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 holds the address of the next 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

Process the low order 16 bits of the branch instruction by **2-bit shifting (*4),** and **sign-extend** => 32-bit destination



Homework 2: Question 5

For the following MIPS program, determine a sequence of 32 bits for each instruction:

```
Loop: lw R6, -1(R31)
        addi R18, R3, -513
        sw   R28, -3 (R5)
        bne  R7, R5, Loop
```

Let's focus on the last instruction:

```
   bne  R7, R5, Loop
```

How do we or machine fill the 16 digits for label "Loop"?

bne (5) R7 (7) R5 (5) **-4** => 5 7 5 (2's complement of -4) =>
000101 00111 00101 1111111111111100

Why -4 not -3?

Before this instruction is executed, PC=PC+4 (pointing to the next instruction)

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		0x24

- Alternate versions of slt

```
slti $t0, $s0, 25 # if $s0 < 25 then $t0=1
sltu $t0, $s0, $s1 # if $s0 < $s1(unsigned)then $t0=1
sltiu $t0, $s0, 25 # if $s0(unsigned) < 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 #if s1<s2 goto L`

```
        slt   $at, $s1, $s2        #$at=1 if $s1<$s2
        bne   $at, $zero, Label    #if $at=/0, goto L
```

  - If less than or equal to, then go to L: `ble $s1, $s2, L`
  - If greater than, then go to L `bgt $s1, $s2, L`
  - If great than or equal to, then go to L `bge $s1, $s2, L`

Other Control Flow Instructions

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

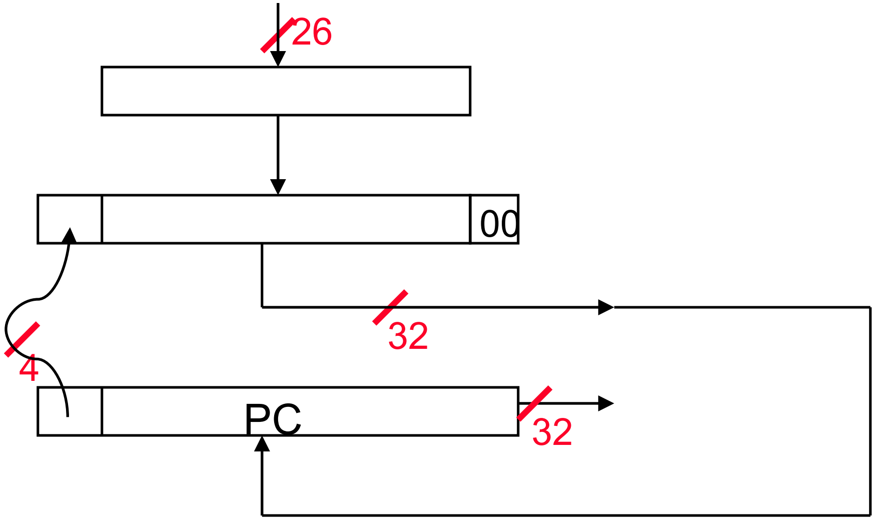
```
    j  label        #go to label
```

- Instruction Format (J Format):

0x02	26-bit address

- Jumping address is formed at runtime by two operations:
  - An address is always a multiple of 4: shifting two bits with 00
  - The high order 4-bits comes from the 4-bits in PC (relative to PC)
  - As an instruction is fetched, PC increments, PC+4

from the low order 26 bits of the jump instruction



Explanation of MIPS Jump instruction

1. The target address is the number of words, we need to multiple by 4 => shifting the 26 bits two positions left

2. We have 28 bits now. Why do we use the high order 4 bits of PC to fill in the same positions for target address?

3. In 32-MIPS, the memory is divided into 16 blocks of 2^{28} bytes, which is controlled by the high order 4 bits in PC

4. The actual addresses within a block are 32 bits starting from the 4-bit for the block and the lower order of 28 bits

5. The MIPS compiler is forced to put a program within one block only so that

    - all instructions within a program are in the 32-bit addresses starting with the 4-bit block address.

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 – one branch plus another jump, thus

```
beq  $s0, $s1, L1 #if $s0=$s1,L1(16 bits)
```

becomes

```
bne  $s0, $s1, L2 #if $s0!=$s1, L2
j    L1           #long bits L1 (26 bits)
L2:
```

Instructions for Accessing Procedures

- MIPS procedure call instruction:

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

- Saves PC+4 in register $ra (register 31) to have a link to the next instruction for the procedure return

- Machine format (J format):

0x03	26 bit address

- The procedure return with a

```
jr    $ra          #$ra=returning-address
```

- Instruction format (R format):

0	31				0x08

Branch instructions and Jump instructions J, JR, Jal

- The destination address in branch instruction is a 16-bit immediate value in the instruction
  - The real branch address is always a multiple of 4 (2-bit shifting)
  - e.g. beq $s1, $s2, 25 #if ($s1==$s2), go to PC+4 + 25*4 (100)

- The destination address of **J** and **Jal** jump instructions is in a 26-bit immediate value
  - The destination address is always a multiple of 4 (2-bit shifting)
  - e.g. J 2500 # go to 2500*4 (10000) (instruction starting address)
  - e.g. Jal 2500 #$ra=PC+4, go to 10000 (procedure starting address)

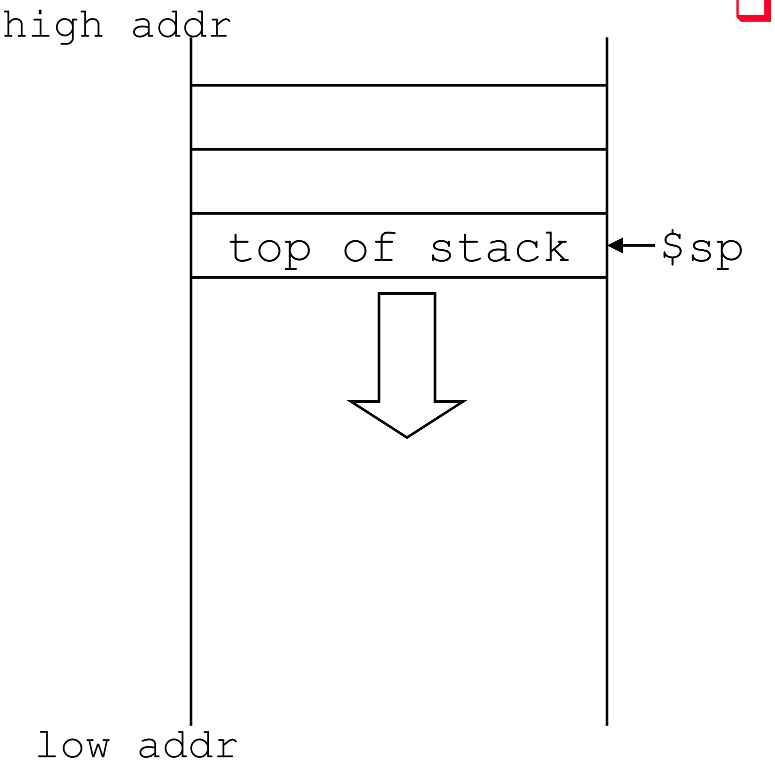
- The destination address of **Jr** instruction is stored in register $ra
  - e,.g. Jr $ra # go to the address in $ra (returning address)

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 (after saving returning address in $ra)
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
   - `$v0` - `$v1`: two value registers for result values
6. Callee returns control to the caller
   - `$ra`: by return address register to return to the point of origin

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 bottom (which "grows" from high address to low address)
  - add data onto the stack – push

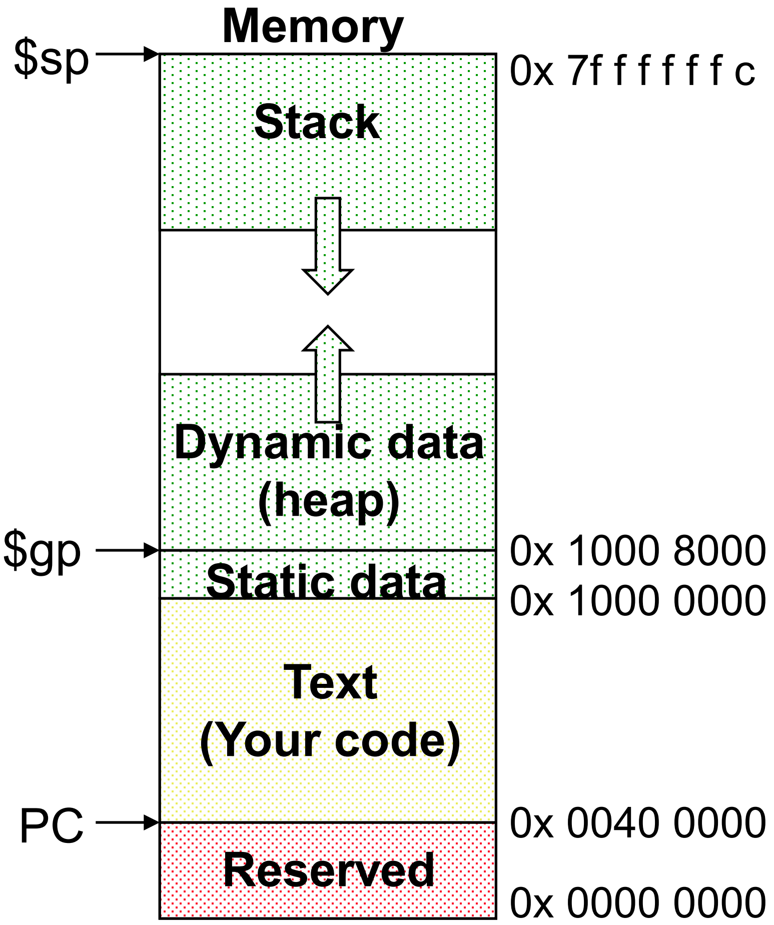
    `$sp` = `$sp` – 4
    data on stack at new $sp

  - remove data from the stack – pop

    data from stack at $sp
    `$sp` = `$sp` + 4 (increment top)

Aside: Allocating Space on the Heap

- Static data segment for constants and other static variables (e.g., arrays), with 8000 (hex) words
- A heap is a tree-based structure, e.g. a B-tree, stack pointer is $gp ($28)
- Dynamic data segment (aka heap) for structures that grow and shrink (e.g., linked lists)
  - Allocate space on the heap with `malloc()` and free it with `free()` in C
  - If malloc (size) succeeds, returns a pointer of the first byte



Atomic Exchange Support

- Need hardware support for synchronization mechanisms to avoid data races where the results of the program can change depending on how events happen to occur
  - Two or more threads in a single process access to the same memory location, and at least one thread is a write
- Atomic exchange (atomic swap) – interchanges a value in a register for a value in memory atomically, i.e., as one operation (instruction) while other threads are waiting
  - Implementing an atomic exchange would require both a memory read and a memory write in a single, uninterruptable instruction. An alternative is to have **a pair** of specially configured instructions

```
ll $t1,0($s1)#$t1=Mem[$s1+0],load linked word

sc $t0,0($s1)#store $t0 to Mem[$s1+0] if Mem[]
is changed, $t0=0, otherwise $t0=1
```

More on atomic exchange

- Exchanging the value between variables A and B:
  - C=A;
  - A=B;
  - B=C;
- We want to ensure if the read value is changed before we store
- In MIPS, this is accomplished by the pair of ll and sc

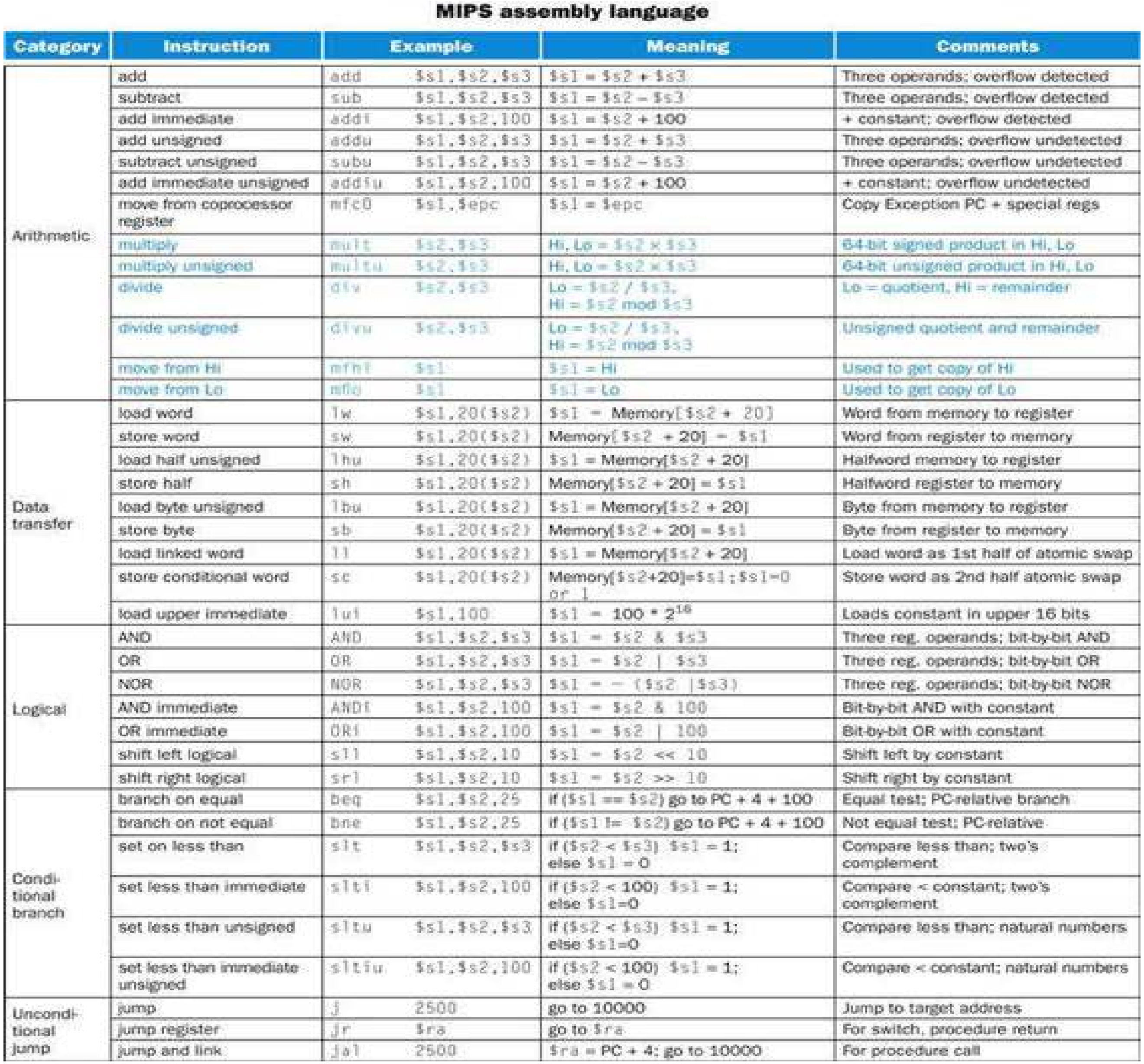
Atomic Exchange between $s4 and 0($s1)

- If the contents of the memory location specified by the `ll` are changed before the `sc` to the same address occurs, the `sc` fails (returns a zero in $t0)

```
try:  add $t0, $zero, $s4 #temporally store $s4 in $t0
      ll  $t1, 0($s1)     #load memory value to $t1
      sc  $t0, 0($s1)     #try to store $s4 to memory
                          #if mem value is changed, $t0
                          #will be set to 0
      beq $t0, $zero, try #try again if $t0 is 0
      add $s4, $zero, $t1 #else load mem value in $s4
```

- If the value in memory between the `ll` and the `sc` instructions changes, then `sc` sets `$t0=0` causing the code sequence to try again.

Figure 3.13 MIPS core architecture, p. 243, 4th edition



MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0 $s1,$epc	$s1 = $epc	Copy Exception PC + special regs
	multiply	mult $s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo

	divide	div $s2,$s3	Lo = $s2 / $s3,
			Hi = $s2 mod $s3
				Lo = quotient, Hi = remainder

	divide unsigned	divu $s2,$s3	Lo = $s2 / $s3,
			Hi = $s2 mod $s3
				Unsigned quotient and remainder

	move from Hi	mfhi $s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo $s1	$s1 = Lo	Used to get copy of Lo
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 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 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 conditional word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half atomic swap
	load upper immediate	lui $s1,100	$s1 = 100 * 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 $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND with constant
	OR immediate	ORI $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR 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; two's complement

	set less than immediate	slti $s1,$s2,100	if ($s2 < 100) $s1 = 1;
			else $s1=0
				Compare < constant; two's complement

	set less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1;
			else $s1=0
				Compare less than; natural numbers

	set less than immediate unsigned	sltiu $s1,$s2,100	if ($s2 < 100) $s1 = 1;
			else $s1 = 0
				Compare < constant; natural numbers

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

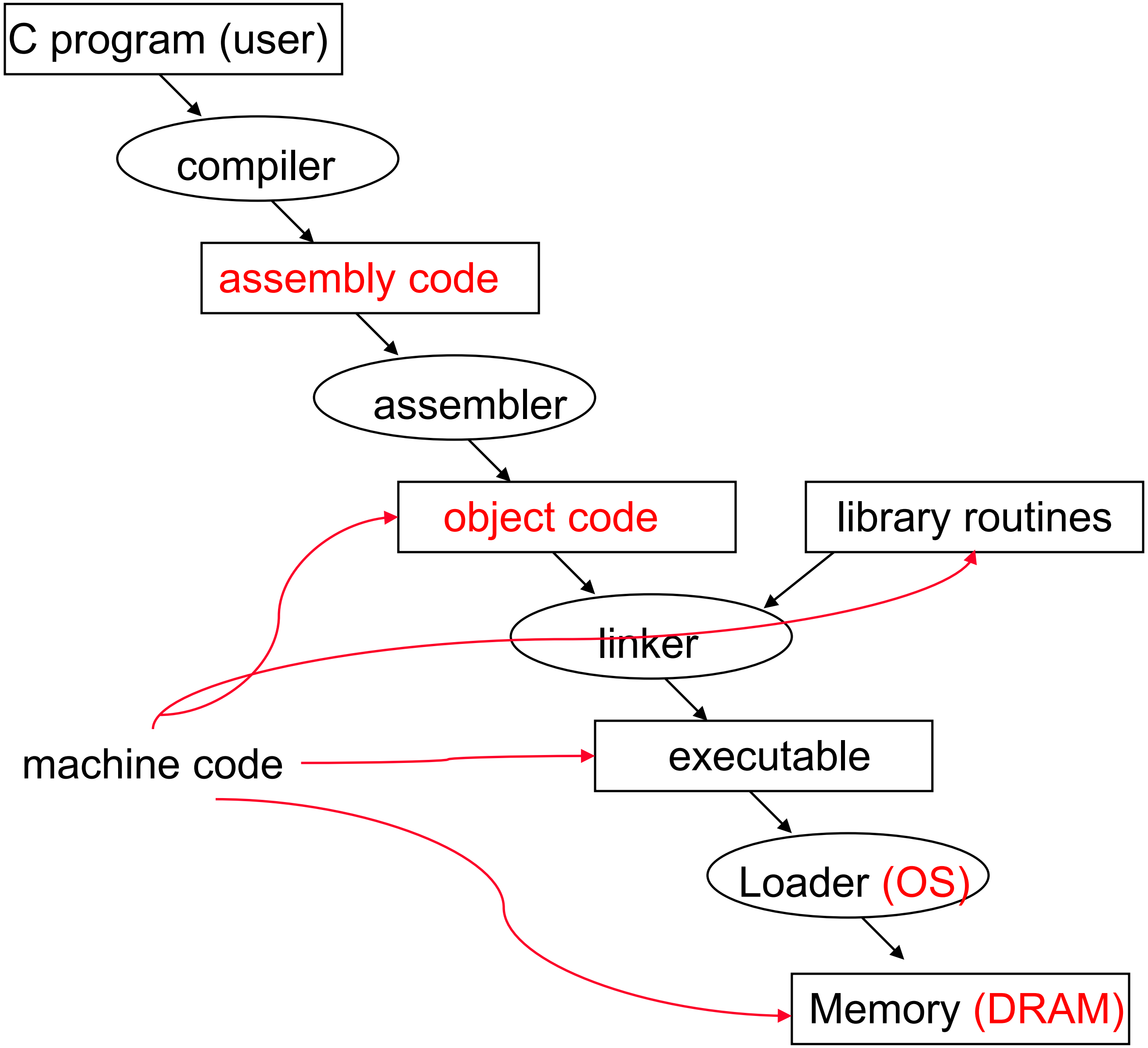
MIPS Instruction Classes Distribution

- Frequency of MIPS instruction classes for SPEC2006

Instruction Class	Frequency	
	Integer	Ft. Pt.
Arithmetic	16%	**48%**
Data transfer	**35%**	**36%**
Logical	12%	4%
Cond. Branch	**34%**	8%
Jump	2%	0%

- Highest frequency: floating point calculation, but low cost
- Data movement and integer branch are high and expensive

The C Code Translation Hierarchy



Compiler Benefits

- Comparing performance for bubble (exchange) sort
  - To sort 100,000 words by a C program 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

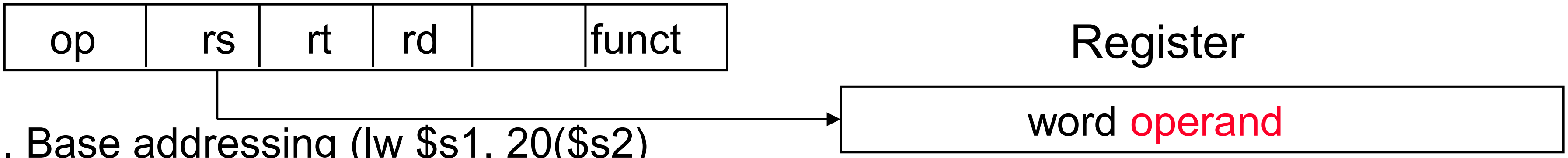
gcc opt	Relative performance	Clock cycles (M)	Instr count (M)	CPI
None	1.00	158,615	114,938	**1.38**
O1 (medium)	2.37	66,990	**37,470**	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (proc mig)	**2.41**	**65,747**	44,993	1.46

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

- Focusing on minimizing data movement

Addressing Modes Illustrated

1. Register addressing (add $s1, $s2, $s3) to each

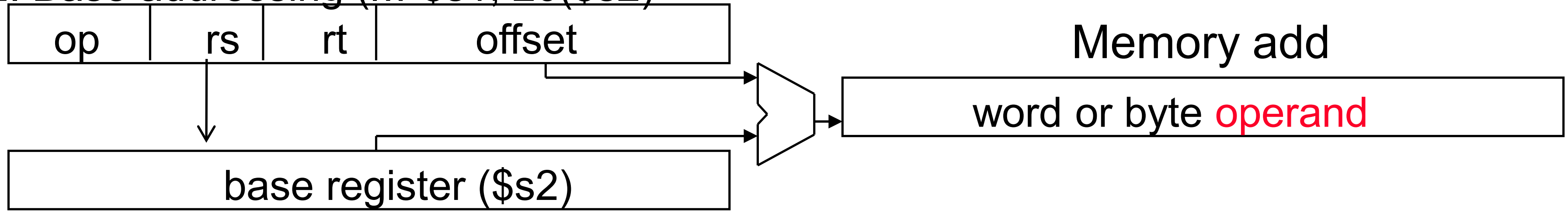


op	rs	rt	rd		funct

Register

word operand

2. Base addressing (lw $s1, 20($s2)



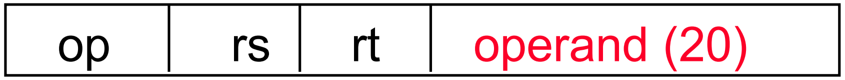
op	rs	rt	offset

Memory add

word or byte operand

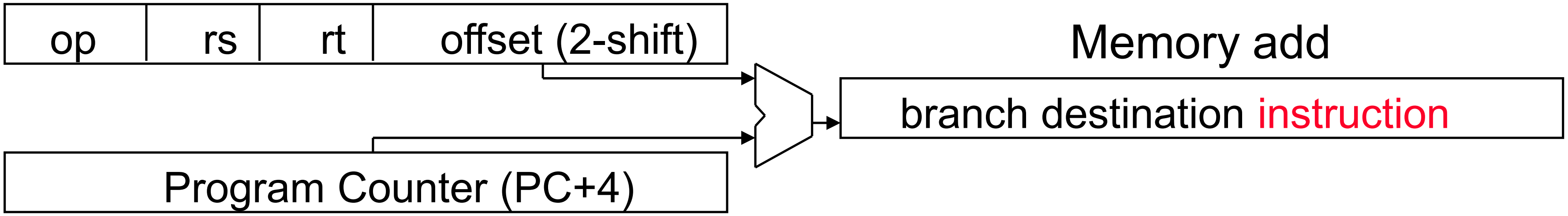
base register ($s2)

3. Immediate addressing (addi $s1, $s2, 20)



op	rs	rt	operand (20)

4. PC-relative addressing (beq $s1, $s2, offset)



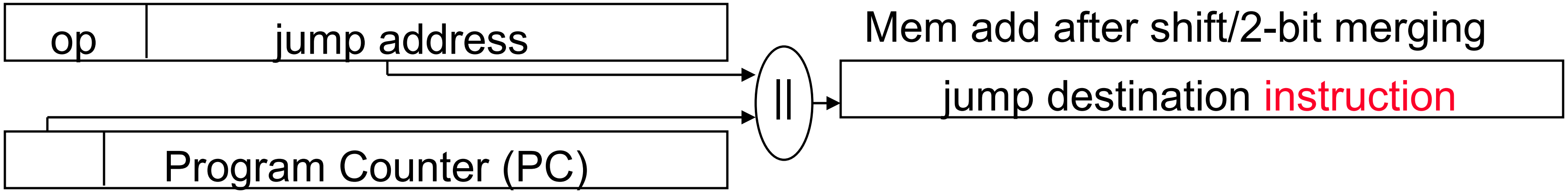
op	rs	rt	offset (2-shift)

Memory add

branch destination instruction

Program Counter (PC+4)

5. Pseudo-direct addressing (j 250)



op	jump address

Mem add after shift/2-bit merging

jump destination instruction

Program Counter (PC)

MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions (32 bit each, or a word long)
  - 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

Summary of Instruction Formats

- All MIPS instructions are encoded in binary.

- All MIPS instructions are 32 bits long.
  - (Note: some assembly langs do not have uniform length for all instructions)

- There are three instruction categories: R-format (most common), I-format, and J-format.

- All instructions have:
  - op (or opcode): operation code (specifies the operation) (first 6 bits)

R-format Instructions

- Have op 0. (all of them!)

- Also have:
  - rs: 1st register operand (register source) (5 bits)
  - rt: 2nd register operand (5 bits)
  - rd: register destination (5 bits)
  - shamt: shift amount (0 when N/A) (5 bits)
  - funct: function code (identifies the specific R-format instruction) (6 bits)

- Example
  - add $s0, $s1, $s2   #registers $16 = $17 + $18

I-format Instructions

- Have a constant value immediately present in the instruction.
- Also have:
  - rs: register containing base address (5 bits)
  - rt: register destination/source (5 bits)
  - immediate: value or offset (16 bits)
- Example
  - lw $t0, 32($s3) #registers 8 and 19 + immediate value of 32

J-format Instructions

- Have an address in the instruction.

- Example
  - j Destination

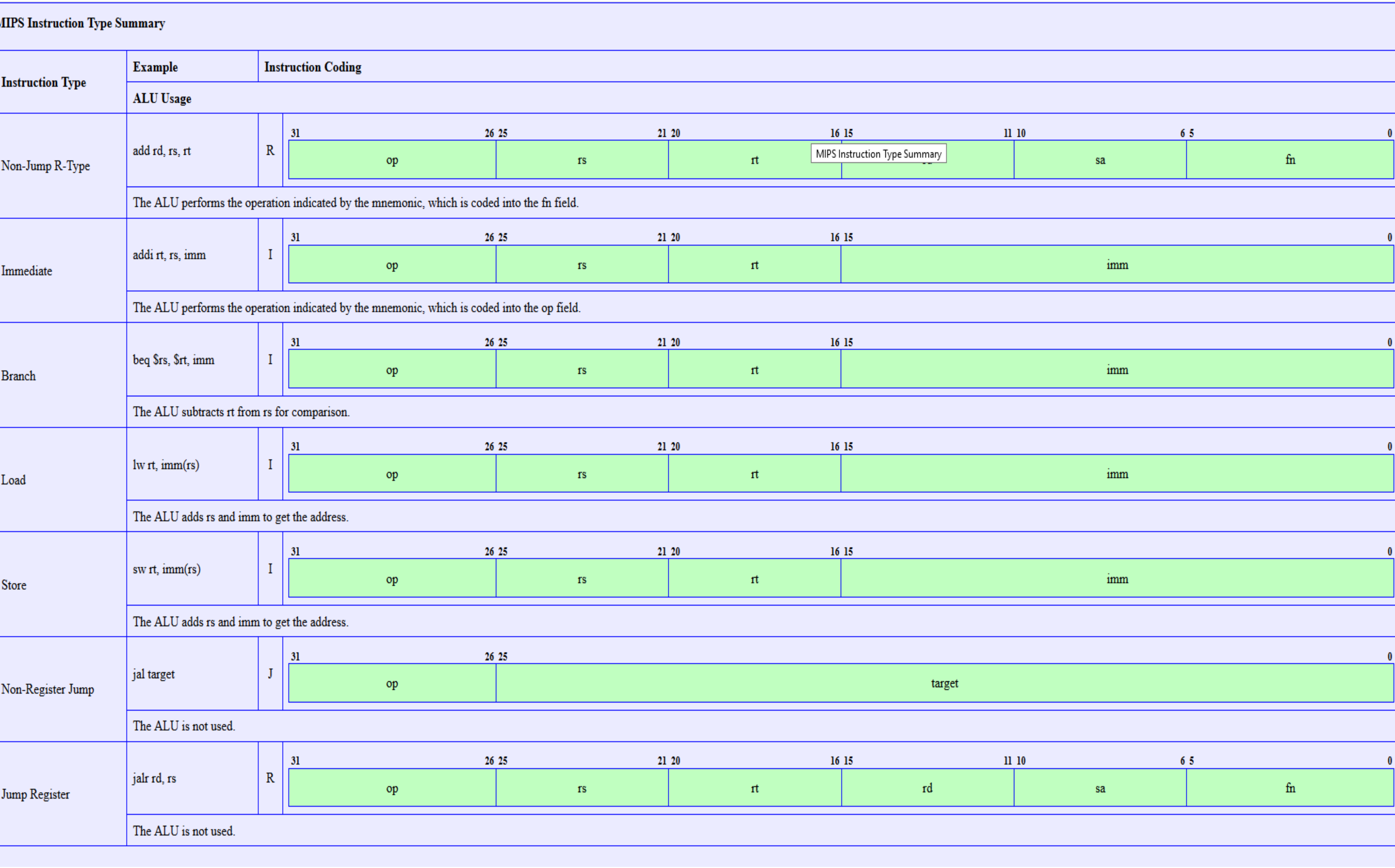
Branch instructions and Jump instructions J, JR, Jal

- The destination address in branch instruction is a 16-bit immediate value in the instruction
  - The real branch address is always a multiple of 4 (2-bit shifting)
  - e.g. beq $s1, $s2, 25 #if ($s1==$s2), go to PC+4 + 25*4 (100)

- The destination address of **J** and **Jal** jump instructions is in a 26-bit immediate value
  - The destination address is always a multiple of 4 (2-bit shifting)
  - e.g. J 2500 # go to 2500*4 (10000)
  - e.g. Jal 2500 #$ra=PC+4, go to 10000

- The destination address of **Jr** instruction is stored in register $ra
  - e,.g. Jr $ra # go to the address in $ra

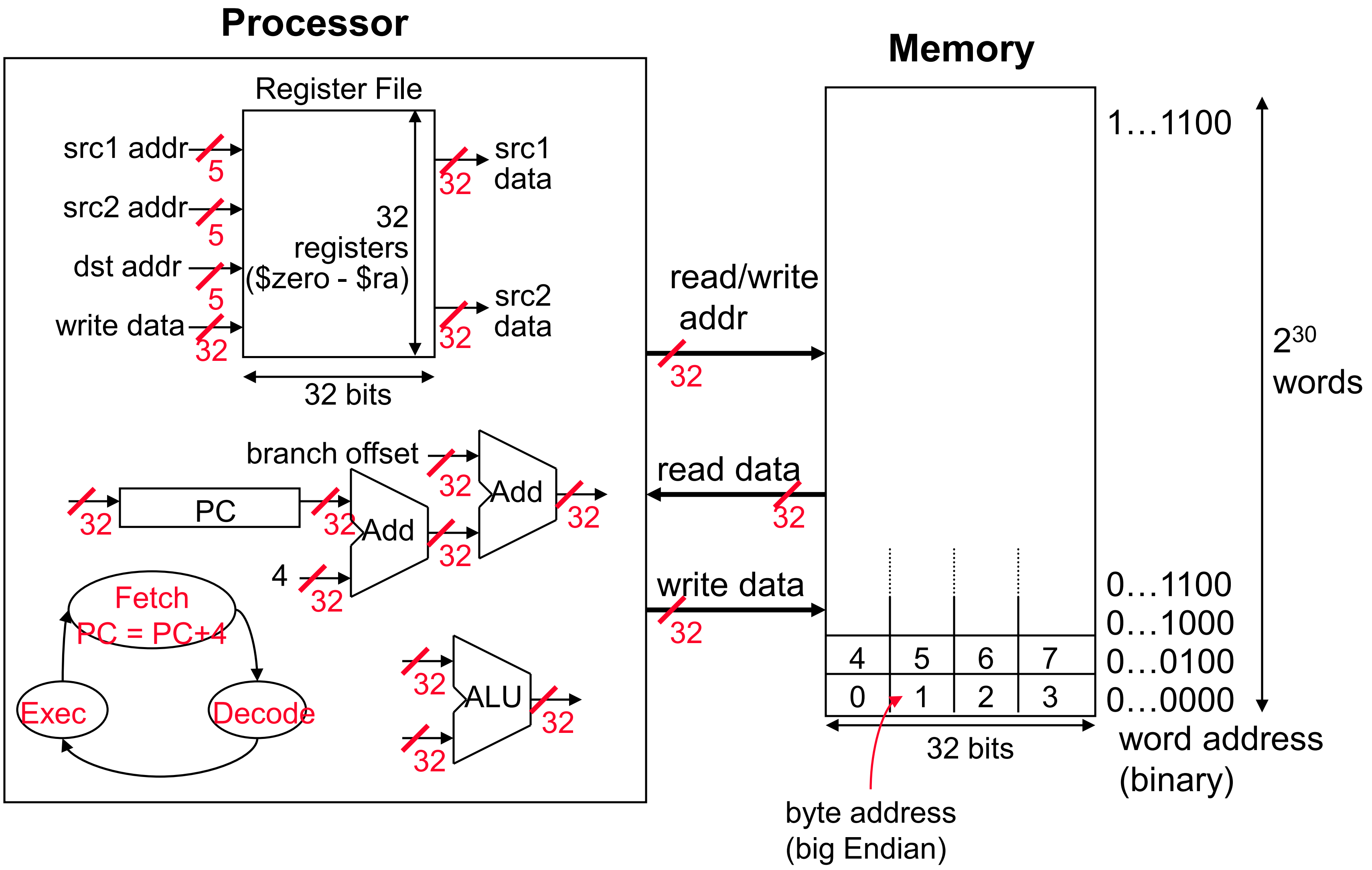
A summary of Simple MIPS Instruction Set



MIPS Instruction Type Summary

Instruction Type	Example		Instruction Coding					
	ALU Usage							
Non-Jump R-Type	add rd, rs, rt	R	31 26	25 21	20 16	15 11	10 6	5 0
			op	rs	rt	MIPS Instruction Type Summary	sa	fn
	The ALU performs the operation indicated by the mnemonic, which is coded into the fn field.							
Immediate	addi rt, rs, imm	I	31 26	25 21	20 16	15 0		
			op	rs	rt	imm		
	The ALU performs the operation indicated by the mnemonic, which is coded into the op field.							
Branch	beq $rs, $rt, imm	I	31 26	25 21	20 16	15 0		
			op	rs	rt	imm		
	The ALU subtracts rt from rs for comparison.							
Load	lw rt, imm(rs)	I	31 26	25 21	20 16	15 0		
			op	rs	rt	imm		
	The ALU adds rs and imm to get the address.							
Store	sw rt, imm(rs)	I	31 26	25 21	20 16	15 0		
			op	rs	rt	imm		
	The ALU adds rs and imm to get the address.							
Non-Register Jump	jal target	J	31 26	25 0				
			op	target				
	The ALU is not used.							
Jump Register	jalr rd, rs	R	31 26	25 21	20 16	15 11	10 6	5 0
			op	rs	rt	rd	sa	fn
	The ALU is not used.							

How Does MIPS Instructions interact with Circuits?


Processor
Register File
src1 addr
src2 addr
dst addr
write data
5
5
5
32
32
registers
($zero - $ra)
src1
data
src2
data
32
32
32 bits
branch offset
PC
Add
Add
4
32
32
32
32
32
32
Fetch
PC = PC+4
Decode
Exec
ALU
32
32
32
Memory
read/write
addr
32
read data
32
write data
32
1…1100
0…1100
0…1000
0…0100
0…0000
2^{30}
words
4
5
6
7
0
1
2
3
32 bits
word address
(binary)
byte address
(big Endian)