

# Instructions: Language of the Computer

## Computer Organization

Monday, 16 September 2024

Many slides adapted from:  
Computer Organization and Design,  
Patterson & Hennessy  
5th Edition, © 2014, MK  
and from Prof. Mary Jane Irwin, PSU



**TÉCNICO** LISBOA

# Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

# Instruction Set Architecture

Instruction Set Architecture (ISA): the abstract interface between the hardware and the lowest level software, that encompasses all the information necessary to write a machine language program, including instructions, registers, memory access, I/O...

- Concept introduced by IBM in the late 1960's (IBM 370 architecture);
- Architecture that is seen from the code generators point of view;
- Interface between the processor architecture and its hardware implementation.



# Evaluating ISAs

- Design-time metrics:
  - Can it be implemented? With what performance, at what costs (design, fabrication, test, packaging), with what power, with what reliability?
  - Can it be programmed? Ease of compilation?
- Static Metrics:
  - How many bytes does the program occupy in memory?
- Dynamic Metrics:
  - How many instructions are executed? How many bytes does the processor fetch to execute the program?
  - How many clock cycles are required per instruction?
  - How fast can it be clocked?

Metric of interest: **Time to execute the program!**

# CISC vs RISC

**CISC:** Complex Instruction-Set Computer  
Versus

**RISC:** Reduced Instruction-Set Computer

Differentiating Factors:

- Number of instructions;
- Complexity of the operations that are implemented by a single instruction;
- Number of operands;
- Addressing modes;
- Memory access.

Most current processors are RISC!

# The MIPS Instruction Set

We will be using the MIPS processor as the example processor in the class.

- Stanford MIPS commercialized by MIPS Technologies ([www.mips.com](http://www.mips.com))
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern ISAs

# MIPS (RISC) Design Principles

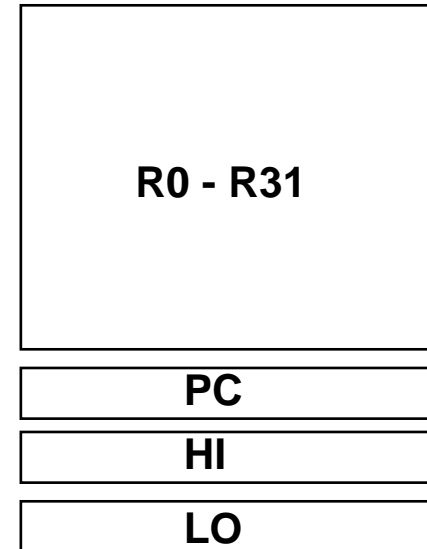
- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  -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

# MIPS-32 ISA

## Instruction Categories

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

## Registers



## 3 Instruction Formats: all 32 bits wide

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



# MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
\$zero	0	constant 0 ( <b>hardware</b> )	n.a.
\$at	1	<b>reserved</b> for assembler	n.a.
\$v0 - \$v1	2-3	returned values	no
\$a0 - \$a3	4-7	arguments	<b>yes</b>
\$t0 - \$t7	8-15	temporaries	no
\$s0 - \$s7	16-23	saved values	<b>yes</b>
\$t8 - \$t9	24-25	temporaries	no
\$k0 - \$k1	26-27	<b>reserved</b> , exception handling	no
\$gp	28	global pointer	<b>yes</b>
\$sp	29	stack pointer	<b>yes</b>
\$fp	30	frame pointer	<b>yes</b>
\$ra	31	return addr ( <b>hardware</b> )	<b>yes</b>

# Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination
  - All register operands

```
add a, b, c           # a gets b + c
```

- All arithmetic operations have this form

C code:

```
f = (g + h) - (i + j);
```

Compiled MIPS code (f, ..., j in \$s0, ..., \$s4):

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

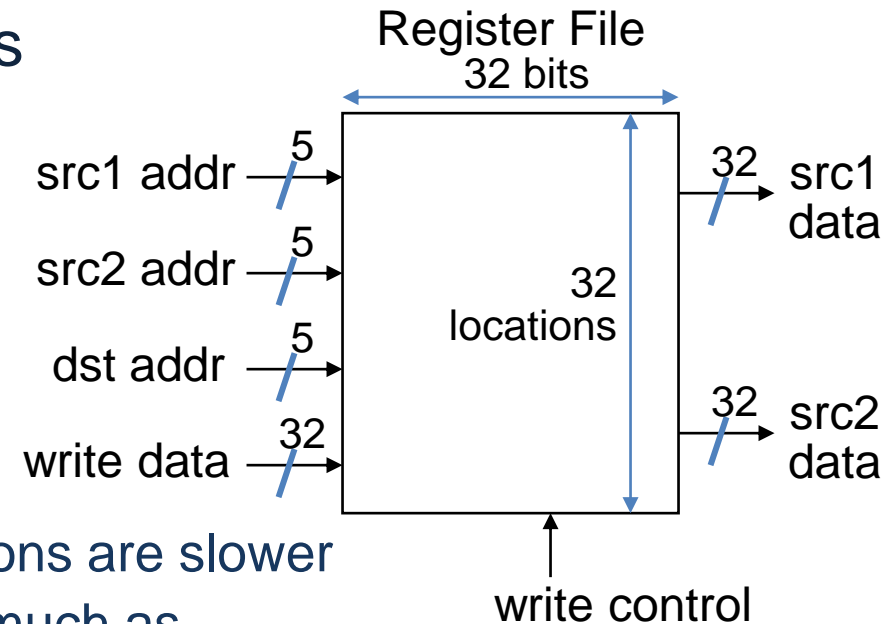
# MIPS Register File

- Holds thirty-two 32-bit registers

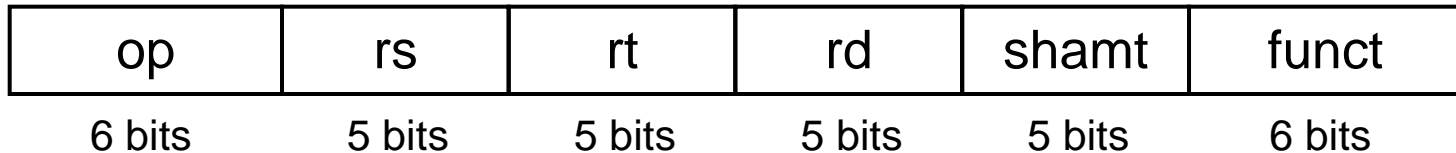
- Two read ports and
- One write port
- Each stores a “word”

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
- Code density improves
  - registers are named with fewer bits than a memory location
  - operating on memory data requires loads and stores



# MIPS R-format Instructions



## 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

ALU	\$s1	\$s2	\$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

# Immediate Operands

Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant

```
addi $s2, $s1, -1
```

- The constant is kept inside the instruction itself
  - Immediate format has 16 bits for the constant, range limited to  $+2^{15}-1$  to  $-2^{15}$

# I-format Example

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

`addi $t0, $zero, 5`

op	\$zero	\$t0	5
8	0	8	5
001000	00000	01000	00000000000000101

$$001000000000100000000000000000101_2 = 20080005_{16}$$

# Loading Larger Constants

- It is possible to load a 32 bit constant into a register, however it requires two instructions
  - "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

16	0	8	1010101010101010 <sub>2</sub>
----	---	---	-------------------------------

- then load the lower order bits (on the right), using

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

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

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

---

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



# Check@home: Unsigned Binary Integers

Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

– Range: 0 to  $+2^n - 1$

Example

$$\begin{aligned} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2 \\ &= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10} \end{aligned}$$

– Using 32 bits

0 to +4,294,967,295

# Check@home: Binary Representation

## Converting from binary to decimal representation

$$X = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0 = \sum_{i=0}^{n-1} x_i 2^i$$

$$\begin{aligned} X &= \dots b_6 \times 2^6 + b_5 \times 2^5 + b_4 \times 2^4 + b_3 \times 2^3 + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0 \\ &= \dots b_6 \times 64 + b_5 \times 32 + b_4 \times 16 + b_3 \times 8 + b_2 \times 4 + b_1 \times 2 + b_0 \times 1 \end{aligned}$$

Example - what is the decimal value of:

$$\begin{aligned} 11001100_2 &= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= 1 \times 128 + 1 \times 64 + 0 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 0 \times 1 \\ &= 128 + 64 + 8 + 4 \\ &= 204_{10} \end{aligned}$$

# Check@home: Binary Representation

More examples:

128;64;32;16; 8;4;2;1

$$1010_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 8 + 2 = 10_{10}$$

$$1000_2 = 1 \times 2^3 = 8 = 8_{10}$$

$$10001000_2 = 1 \times 2^7 + 0 \times 2^3 = 128 + 8 = 136_{10}$$

$$1111_2 = 8 + 4 + 2 + 1 = 15_{10}$$

# Check@home: Decimal to Binary Conversion

Repeated division method:

$$x/2 = x_{n-1}2^{n-2} + x_{n-2}2^{n-3} + \dots + x_1 + x_0/2 = \sum_{i=0}^{n-1} x_i 2^i / 2$$

For Number  $\neq 0$  repeat

Number<sub>10</sub> = Number<sub>10</sub> % 2  $\Rightarrow$  Remainder is new binary digit

Example:

$$10_{10} \rightarrow 10\%2 = 5 \rightarrow 5\%2 = 2 \rightarrow 2\%2 = 1 \rightarrow 1\%2 = 0$$

$$r_0 = 0$$

$$r_1 = 1$$

$$r_2 = 0$$

$$r_3 = 1$$



Binary result =  $r_3 r_2 r_1 r_0 = 1010_2$

# Check@home: Decimal to Binary Conversion

More examples:

$$15_{10} \rightarrow 15\%2 \rightarrow 7\%2 \rightarrow 3\%2 \rightarrow 1\%2 \rightarrow 0$$
$$r_0 = 1 \quad r_1 = 1 \quad r_2 = 1 \quad r_3 = 1 \rightarrow = 1111_2$$

$$27_{10} \rightarrow 27\%2 \rightarrow 13\%2 \rightarrow 6\%2 \rightarrow 3\%2 \rightarrow 1\%2 \rightarrow 0$$
$$r_0 = 1 \quad r_1 = 1 \quad r_2 = 0 \quad r_3 = 1 \quad r_4 = 1 \rightarrow = 11011_2$$

A simpler approach :

$$33_{10} = 32_{10} + 1_{10} = 2^5 + 2^0 = 100001_2$$

$$11_{10} = 8_{10} + 2_{10} + 1_{10} = 2^3 + 2^1 + 2^0 = 1011_2$$

# 2s-Complement Signed Integers

Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

– Range:  $-2^{n-1}$  to  $+2^{n-1} - 1$

Example

$$\begin{aligned} & 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1100_2 \\ &= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= -2,147,483,648 + 2,147,483,644 = -4_{10} \end{aligned}$$

– Using 32 bits

$$-2,147,483,648 \text{ to } +2,147,483,647$$

# 2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- Non-negative numbers have the same unsigned and 2s-complement representation

- Some specific numbers

0: 0000 0000 ... 0000

-1: 1111 1111 ... 1111

Most-negative: 1000 0000 ... 0000

Most-positive: 0111 1111 ... 1111

- Signed negation: complement and add 1  
( $-(-2^{n-1})$ ) can't be represented)

$$x + \bar{x} = 1111 \dots 111_2 = -1$$

$$\bar{\bar{x}} + 1 = -x$$

# 2s-Complement Signed Integers

Examples:

Negation (8-bit)

$$24_{10} = 00011000$$

$$\begin{aligned} -24_{10} &= \overline{24_{10}} + 1_{10} = \overline{00011000}_2 + 1_2 \\ &= 11100111_2 + 1_2 = 11101000_2 \end{aligned}$$

8-bit to 16-bit representation

$$+2: 0000\ 0010 \Rightarrow 0000\ 0000\ 0000\ 0010$$

$$-2: 1111\ 1110 \Rightarrow 1111\ 1111\ 1111\ 1110$$



# Unsigned Version of Instructions

<code>add</code>	<code>addu</code>
<code>addi</code>	<code>addiu</code>
<code>sub</code>	<code>subu</code>

- Arithmetic immediate values **are** sign-extended
- Then, they are handled as signed or unsigned 32 bit numbers, depending upon the instruction
- Signed instructions can generate an overflow exception whereas unsigned instructions cannot

# 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 NOR	~( )	~( )	nor

- Useful for extracting and inserting groups of bits in a word
  - Through the use of “masks”

# 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

$a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

`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

# MIPS Memory Access Instructions

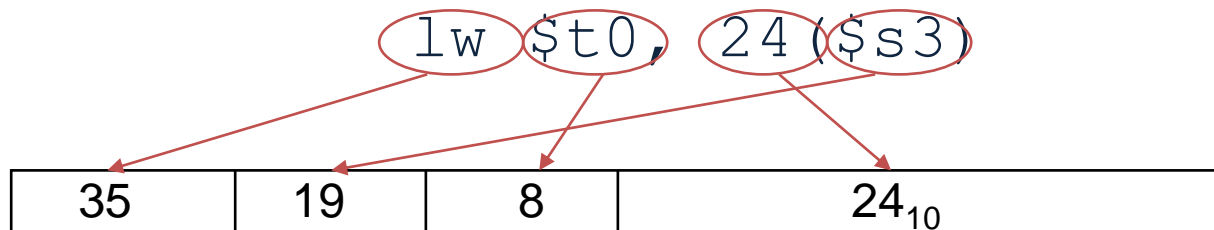
MIPS has two basic data transfer instructions for accessing memory

```
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 value to state which register to use
- The memory address – a 32 bit address
  - formed by adding the contents of the base address register to the offset value (16 bit value)

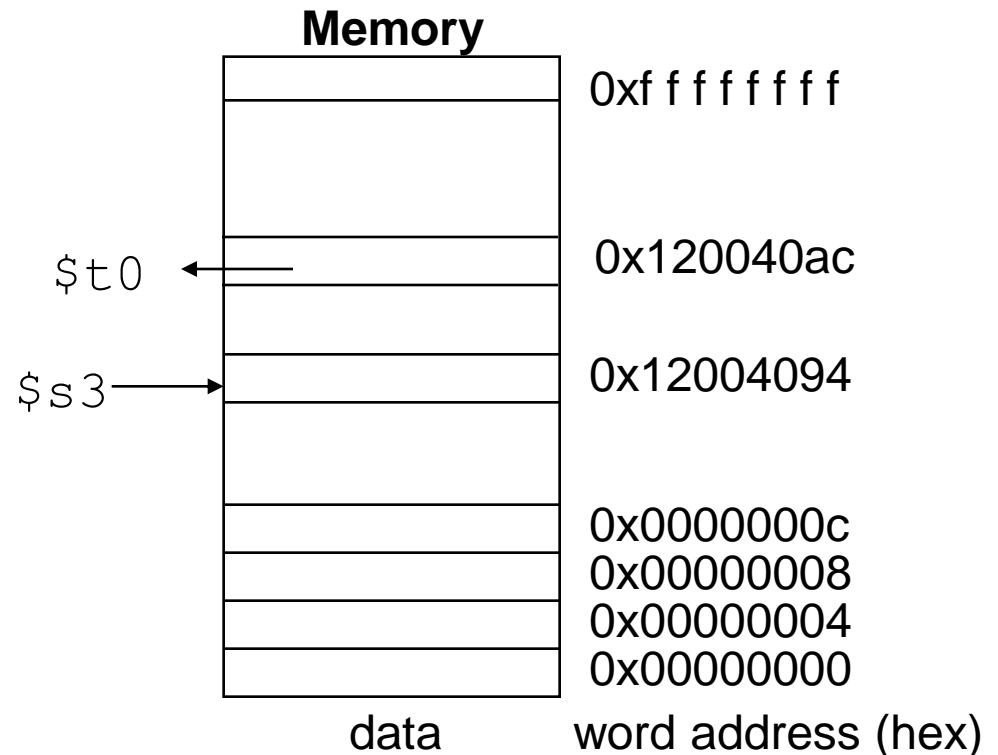
# Machine Language - Load Instruction

## Load/Store Instruction Format (I format):



$$24_{10} + \$s3 =$$

$$\begin{array}{r}
 \dots 0001\ 1000 \\
 + \dots 1001\ 0100 \\
 \hline
 \dots 1010\ 1100 = \\
 0x120040ac
 \end{array}$$





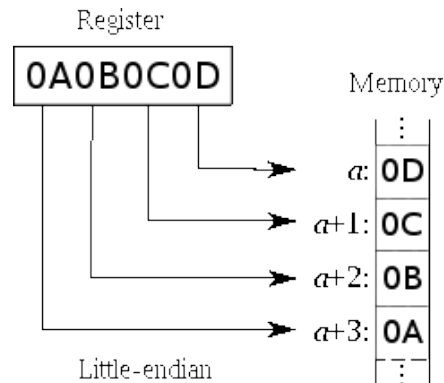
# Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is **byte addressed**
  - Each address identifies an 8-bit value
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at lowest address of a word
  - Little Endian: least-significant byte at lowest address

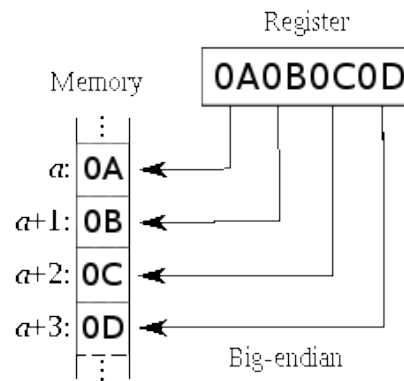
# Check@home: Endianness

- Endianness usually refers to the individual byte order within a longer data word stored in memory
  - Once defined, such order is irrelevant within a computer system
  - However, it is frequently the source of many data transfer problems between different architectures!

**Little Endian:** **least** significant byte stored in the lowest memory address;



**Big Endian:** **most** significant byte stored in the lowest memory address.



# Memory Operand Example 1

C code:

```
g = h + A[8];
```

g in \$s1, h in \$s2, base address of A in \$s3

Compiled MIPS code:

- Index 8 requires offset of 32 (4 bytes per word)

```
lw    $t0, 32($s3)    # load word
add   $s1, $s2, $t0
```

offset

base register

# Memory Operand Example 2

C code:

```
A[12] = h + A[8];
```

h in \$s2, base address of A in \$s3

Compiled MIPS code:

– Indices multiplied by 4

```
lw    $t0, 32($s3)    # load word
add   $t0, $s2, $t0
sw    $t0, 48($s3)    # store word
```

# Byte/Halfword 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**

# Aligned Memory Accesses

	031				031				0
Width	xx0	xx1	xx2	xx3	xx4	xx5	xx6	xx7	...
1 byte (byte)	Alin	Alin	Alin	Alin	Alin	Alin	Alin	Alin	
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned		
2 bytes (half word)	Unaligned		Unaligned		Unaligned		Unaligned		
4 bytes (word)	Aligned				Aligned				
4 bytes (word)	Unaligned				Unaligned				
4 bytes (word)	Unaligned				Unaligned				
4 bytes (word)	Unaligned				Unaligned				
8 bytes (dword)	Aligned								
8 bytes (dword)	Unaligned								
8 bytes (dword)	Unaligned								
8 bytes (dword)	Unaligned								
8 bytes (dword)	Unaligned								
8 bytes (dword)	Unaligned								
8 bytes (dword)	Unaligned								
8 bytes (dword)	Unaligned								
8 bytes (dword)	Unaligned								
8 bytes (dword)	Unaligned								

# Control Instructions

## Computer Organization

Monday, 16 September 2024

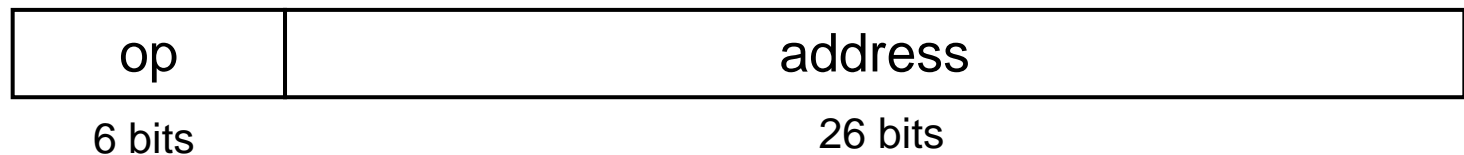
Many slides adapted from:  
Computer Organization and Design,  
Patterson & Hennessy  
5th Edition, © 2014, MK  
and from Prof. Mary Jane Irwin, PSU



**TÉCNICO** LISBOA

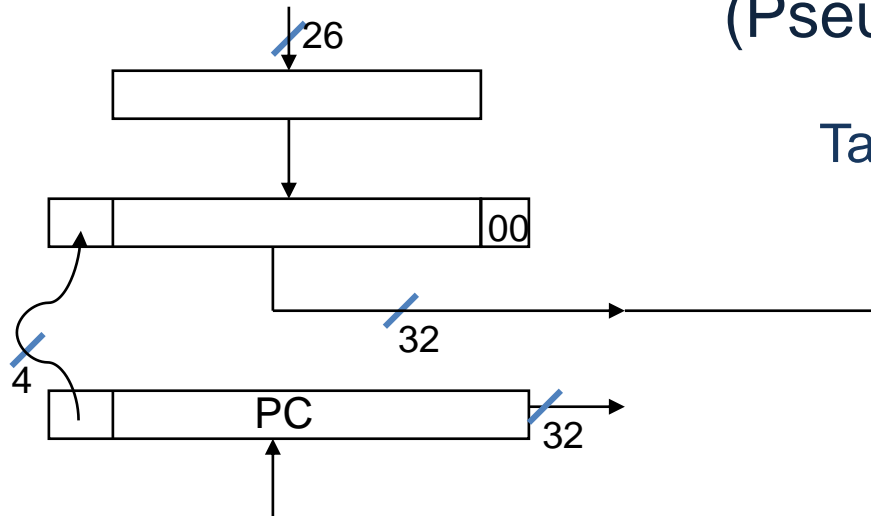
# Jump Instruction

- Jump (`j` and `jal`) targets could be anywhere in program
  - Encode “full” address in instruction



(Pseudo)Direct jump addressing

$$\text{Target address} = \text{PC}_{31..28} : (\text{address} \times 4)$$



- Jump register (`jr`)  
Copies register to PC



# Conditional Operations

- **No flags!**
- 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

- For unconditional branch to instruction labeled L1, use jump

`j L1`

# Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward



- PC-relative addressing
  - Target address =  $PC + \text{offset} \times 4$
  - (PC already incremented by 4 by this time)

# Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code

Example, L1 too far:

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

Rewritten as:

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

```
j L1
```

```
L2: ...
```

# 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
```

# 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`
  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`
- Such branches are included in the instruction set as pseudo instructions
  - Recognized (and expanded) by the assembler
  - Reason why the assembler needs a reserved register (`$at`)

# Branch Instruction Design

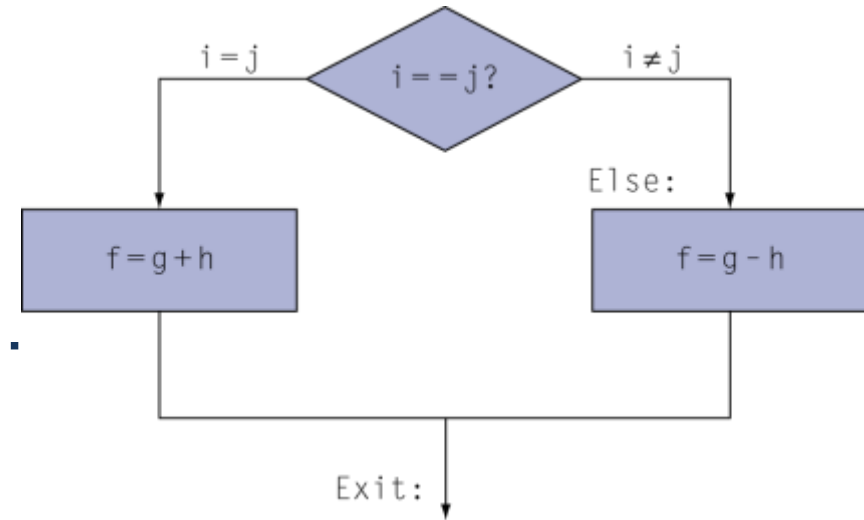
- Why not `blt`, `bge`, etc?
- Hardware for `<`, `≥`, ... slower than `=`, `≠`
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case

# 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:  ...
```



# Target Addressing

- Assume Loop at location 80000

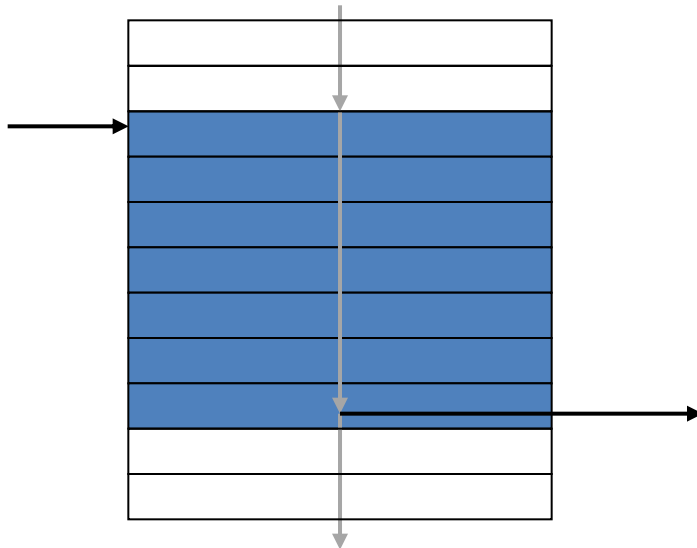
```

Loop: sll  $t1, $s3, 2      80000
      add  $t1, $t1, $s6    80004
      lw   $t0, 0($t1)      80008
      bne  $t0, $s5, Exit   80012
      addi $s3, $s3, 1      80016
      j    Loop             80020
Exit: ...                   80024
    
```

0	0	19	9	2	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

# 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

# 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):

0x03	26 bit address
------	----------------

- Procedure return with

`jr $ra #return`

- Instruction format (R format):

0	31				0x08
---	----	--	--	--	------

# Procedure Calling

## Steps required

1. Place parameters in registers

`$a0` – `$a3`: four argument registers

2. Transfer control to procedure

3. Acquire storage for procedure

- `$t0`–`$t9`: temporaries, can be overwritten by callee
- `$s0`–`$s7`: saved, must be saved/restored by callee

4. Perform procedure's operations

5. Place result in register for caller

`$v0` – `$v1`: two value registers for result values

6. Return to place of call

# Summary of MIPS Instructions

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)	$\$s1 = \text{Memory}[\$s2 + 20]$	Word from memory to register
	store word	sw \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Word from register to memory
	load half	lh \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	store half	sh \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Load word as 1st half of atomic swap
	store condition. word	sc \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1; \$s1 = 0 \text{ or } 1$	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	$\$s1 = \sim (\$s2   \$s3)$	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	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Shift right by constant

# Summary of MIPS Instructions

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

# Next Class

- Role of the compiler
- Comparison between ISAs

# Instructions: Language of the Computer

## Computer Organization

Monday, 16 September 2024

Many slides adapted from:  
Computer Organization and Design,  
Patterson & Hennessy  
5th Edition, © 2014, MK  
and from Prof. Mary Jane Irwin, PSU



**TÉCNICO** LISBOA