Instructions: Language of the Computer

Computer Organization

Monday, 28 September 2020

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

TÉCNICO LISBOA

Summary

- Previous Class
  - Fundamentals of computer architecture
  - Performance metrics

- Today:
  - Instruction Set Architecture
  - MIPS ISA
    - Registers
    - Computational instructions
    - Signed vs unsigned operands
    - Instruction encoding

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)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs

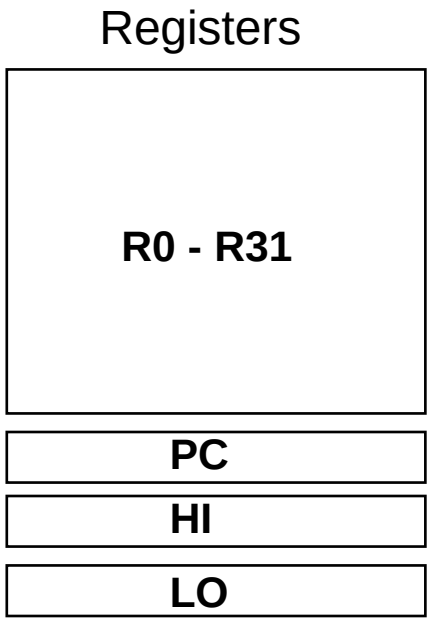
MIPS (RISC) Design Principles

- 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



3 Instruction Formats: all 32 bits wide

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 - $a3`	4-7	arguments	yes
`$t0 - $t7`	8-15	temporaries	no
`$s0 - $s7`	16-23	saved values	yes
`$t8 - $t9`	24-25	temporaries	no
`$k0 - $k1`	26-27	reserved, exception handling	no
`$gp`	28	global pointer	yes
`$sp`	29	stack pointer	yes
`$fp`	30	frame pointer	yes
`$ra`	31	return addr (hardware)	yes

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:

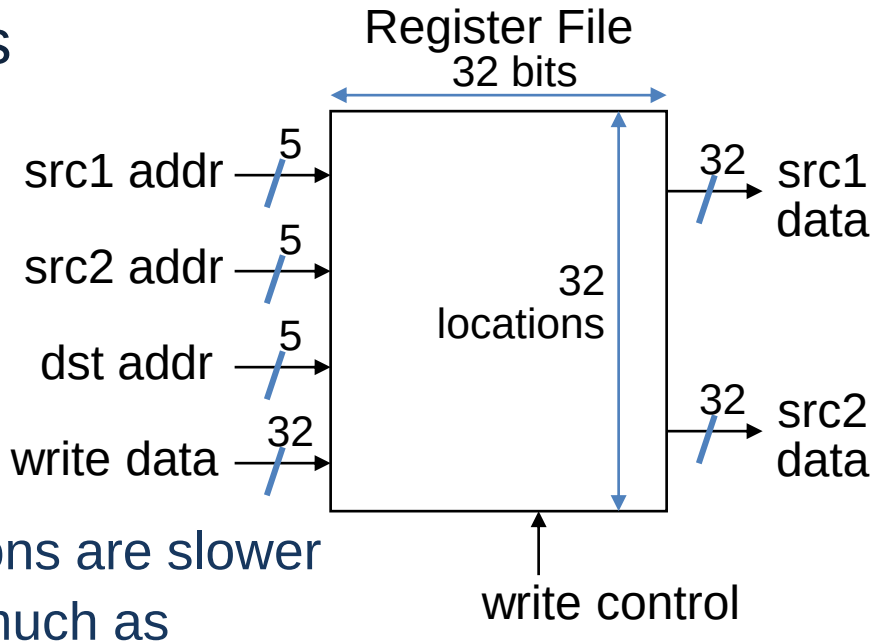
```
f = (g + h) - (i + j);
```

Compiled MIPS code (`f`, …, `j` in `$s0`, …, `$s4`):

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

MIPS Register File

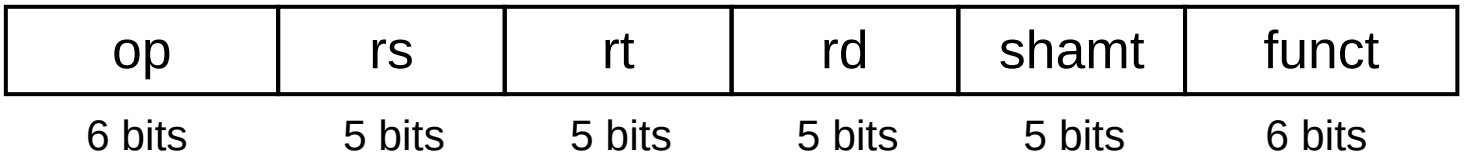
- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port
  - Each stores a “word”



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

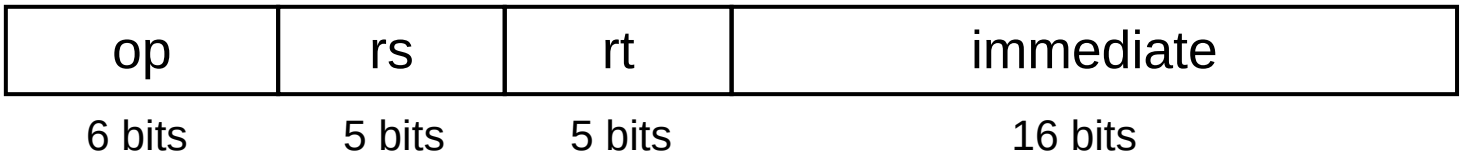
```
addi $s3, $s3, 4
```

- 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




```
addi $t0, $zero, 5
```

op	$zero	$t0	5
8	0	8	5
001000	00000	01000	0000000000000101

$$00100000000010000000000000000101_2 = 20080005_{16}$$

Loading Larger Constants

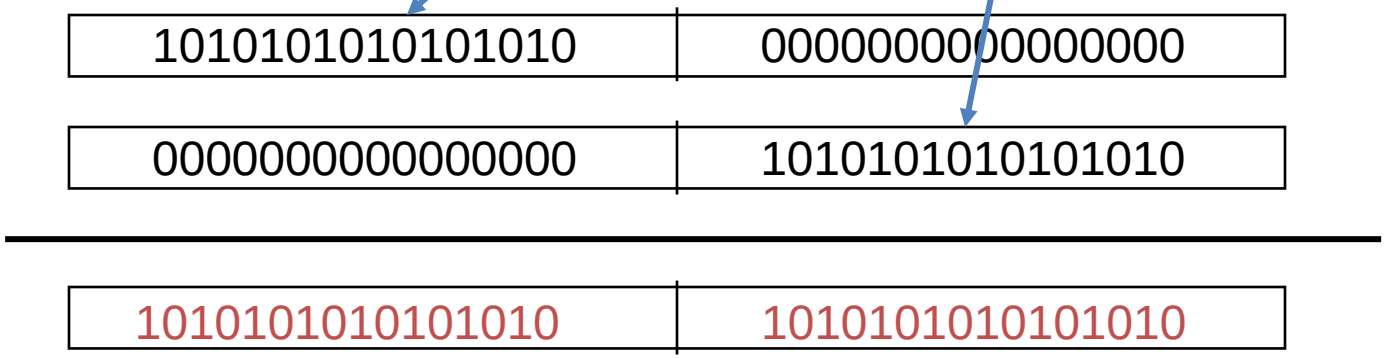
- It is possible to load a 32 bit constant into a register, however it requires two instructions
  - "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

16	0	8	1010101010101010_2

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

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



Unsigned Binary Integers

Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

– Range: 0 to $+2^n - 1$

Example

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
$= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
$= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

– Using 32 bits

0 to +4,294,967,295

Binary Representation

Converting from binary to decimal representation

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

$$X = \ldots b_6 \times 2^6 + b_5 \times 2^5 + b_4 \times 2^4 + b_3 \times 2^3 + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$
$$= \ldots b_6 \times 64 + b_5 \times 32 + b_4 \times 16 + b_3 \times 8 + b_2 \times 4 + b_1 \times 2 + b_0 \times 1$$

Example - what is the decimal value of:

$$11001100_2 = 1\times2^7 + 1\times2^6 + 0\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0$$
$$= 1\times128 + 1\times64 + 0\times32 + 0\times16 + 1\times8 + 1\times4 + 0\times2 + 0\times1$$
$$= 128 + 64 + 8 + 4$$
$$= 204_{10}$$

Binary Representation

More examples: 128;64;32;16; 8;4;2;1

$1010_2 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 = 8 + 2 = 10_{10}$

$1000_2 = 1\times2^3 = 8 = 8_{10}$

$10001000_2 = 1\times2^7 + 0\times2^3 = 128 + 8 = 136_{10}$

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

Decimal to Binary Conversion

Repeated division method:

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

For Number ≠ 0 repeat

$Number_{10} = Number_{10} \% 2$ => 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 = $r_3\, r_2\, r_1\, r_0 = 1010_2$

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

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

– Range: -2^{n-1} to $+2^{n-1} - 1$

Example

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
$= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
$= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

– Using 32 bits

–2,147,483,648 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)

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

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

2s-Complement Signed Integers

Examples:

Negation (8-bit)

$24_{10} = 00011000$

$-24_{10} = \overline{24}_{10}+1_{10} = \overline{00011000}_2 + 1_2$

$= 11100111_2 + 1_2 = 11101000_2$

8-bit to 16-bit representation

+2: 0000 0010 => 0000 0000 0000 0010

–2: 1111 1110 => 1111 1111 1111 1110

Unsigned Version of Instructions

```
add    addu
addi   addiu
sub    subu
```

- 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

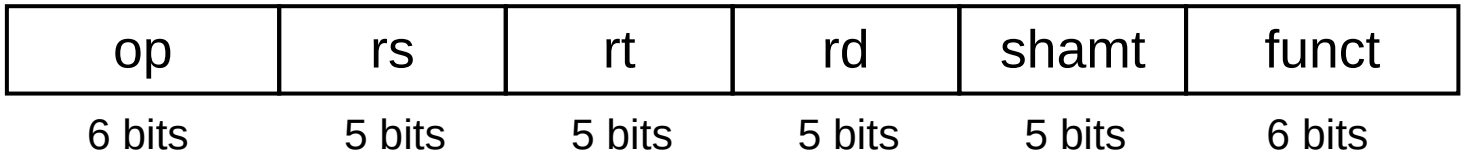
Logical Operations

- 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

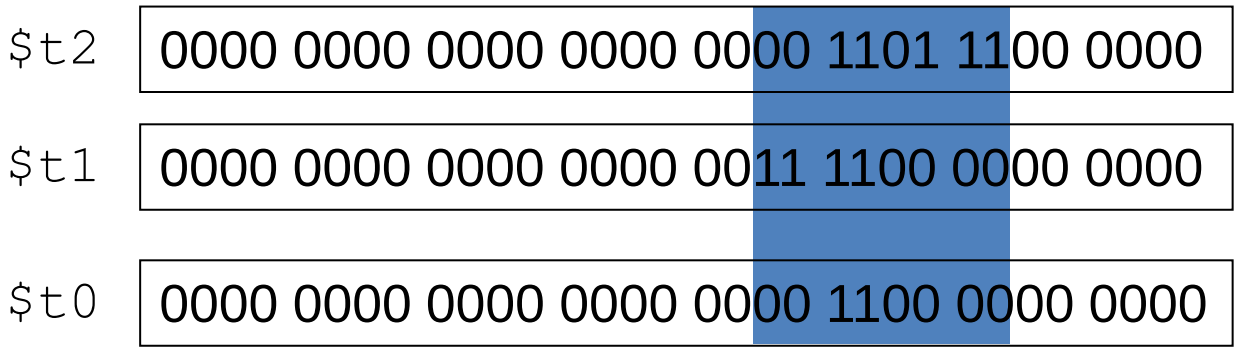


- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

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

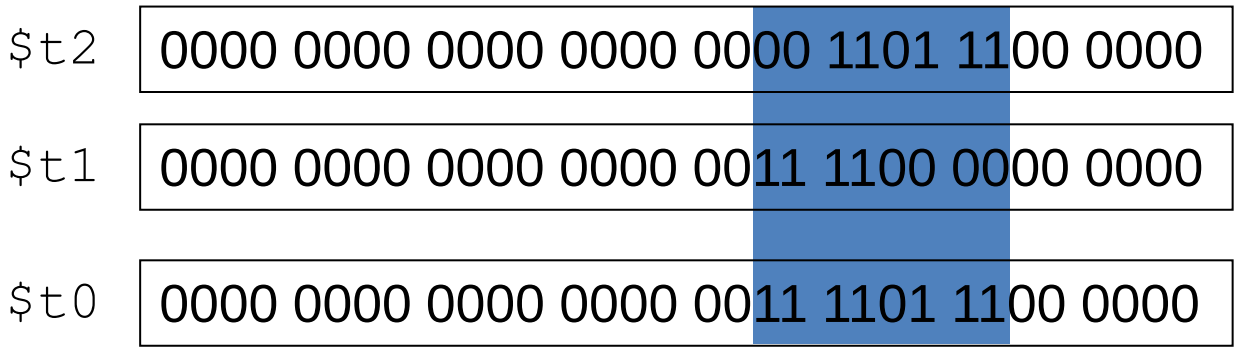
```
and $t0, $t1, $t2
```



OR Operations

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

```
or $t0, $t1, $t2
```



NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction

a NOR b == NOT (a OR b)

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

Register 0: always read as zero

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

Next Class

- MIPS ISA, conclusion
  - Memory Access Instructions
  - Jump/Branch instructions
  - Procedure calls
  - Memory organization

Instructions: Language of the Computer

Computer Organization

Monday, 28 September 2020

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

TÉCNICO LISBOA