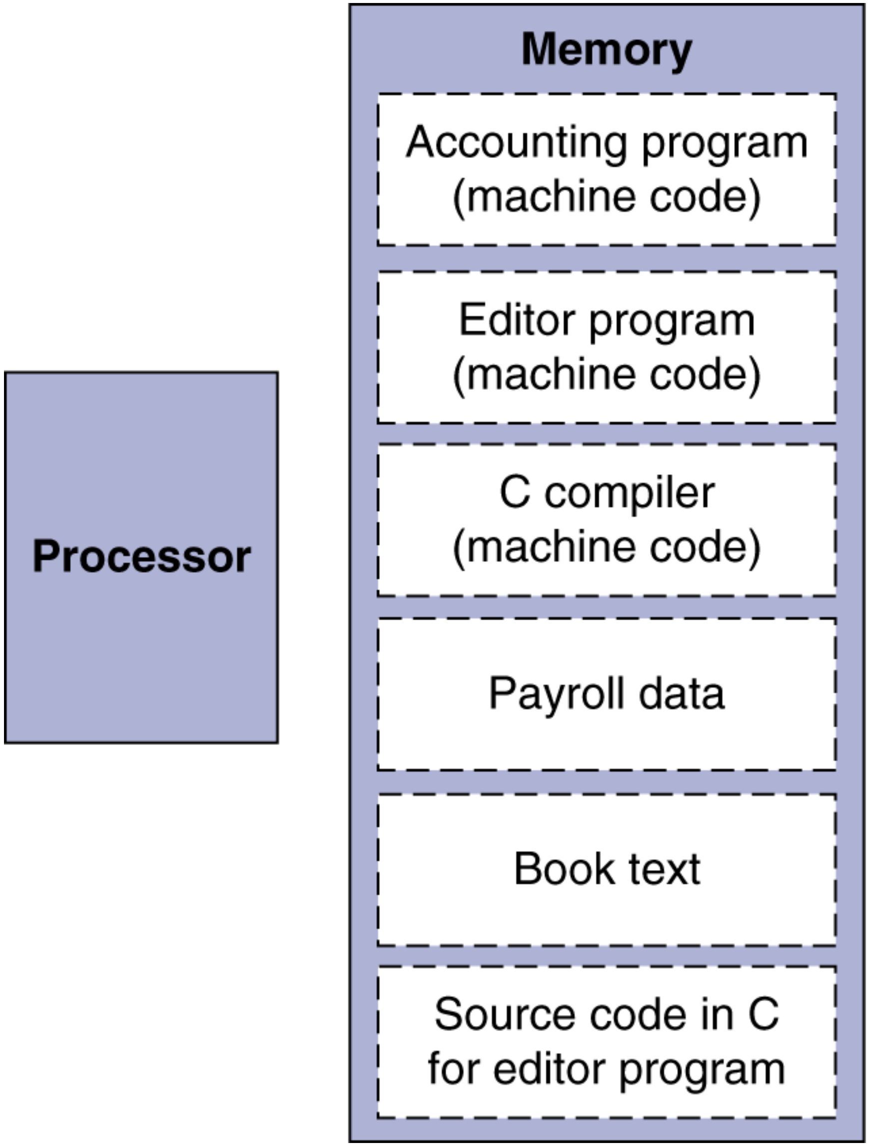
CHAPTER 2-2

Instruction Language of the Computer

By Pattama Longani
Collage of arts, media and Technology


Processor
Memory
Accounting program
(machine code)
Editor program
(machine code)
C compiler
(machine code)
Payroll data
Book text
Source code in C
for editor program

NUMBER SYSTEM

- Binary Number System
  - 0, 1 (binary digits/bits)
- Octal Number System
  - 0,1,2,3,4,5,6,7
- Decimal Number System
  - 0,1,2,3,4,5,6,7,8,9
- Hexadecimal Number System
  - 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

MIPS = 32 bits long

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 bits wide)

most significant bit

least significant bit

- MIPS can represent 2^{32} different patterns.
- It is natural to let these combinations represent the numbers from 0 to $2^{32} - 1$ ($4{,}294{,}967{,}295_{ten}$)

$$
\begin{array}{l}
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten} \\
\ldots \qquad\qquad\qquad\qquad \ldots \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}
\end{array}
$$

- Hardware can be designed to add, subtract, multiply, and divide these binary bit patterns.
- ***Overflow*** occurs when an operation result cannot display every bits of the result properly

- Computer programs calculate both positive and negative number
- We use **<u>two's complement</u>** *to* representation the positive & negative number system:
  - leading **<u>0s</u>** mean **<u>positive</u>**
  - leading **<u>1s</u>** mean **<u>negative</u>**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
$\ldots \qquad \ldots$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$
$\ldots \qquad \ldots$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

Sign bit

- So there are 31 bits to represent numbers
  - The positive half of the numbers, from 0 to $2{,}147{,}483{,}647_{ten}$ ($2^{31} - 1$ or $0111 \ldots 1111_{two}$).
  - The negative half of the numbers, from -1 ($1111 \ldots 1111_{two}$) to number $-2{,}147{,}483{,}648_{ten}$ (-2^{31} or $1000 \ldots 0001_{two}$).
- Two's complement have one negative number, $-2{,}147{,}483{,}648_{ten}$, that has no corresponding positive number.

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
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1{\times}2^3 + 0{\times}2^2 + 1{\times}2^1 + 1{\times}2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2'S-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
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1{\times}2^{31} + 1{\times}2^{30} + \ldots + 1{\times}2^2 + 0{\times}2^1 + 0{\times}2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

NEGATION SHORTCUT

Observe that

0000_2 = 0 and 1111_2 = -1

0001_2 = 1 and 1110_2 = -2

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

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

Complement and add 1

- Complement means $1 \longrightarrow 0$, $0 \longrightarrow 1$

- Example: negate +2

$$2_{ten} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$$

$$
\begin{array}{rr}
 & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} \\
+ & 1_{two} \\
\hline
= & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} \\
= & -2_{ten}
\end{array}
$$

- Example: negate -2

$$2_{ten} = \ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$$

$$
\begin{array}{lr}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} \\
+ & 1_{two} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} \\
= & 2_{ten} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad
\end{array}
$$

SIGN EXTENSION

- convert a *n* bits binary number to more than *n* bits binary number.
  - Replicate the sign bit to the left
  - Examples: 8-bits to 16-bits number
    - +2: 0000 0010 => 0000 0000 0000 0010
    - –2: 1111 1110 => 1111 1111 1111 1110

Ex: Convert 16-bit binary versions of 2_{ten} and -2_{ten} to 32-bit

- The 16-bit binary version of the number 2 is

  $0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

- converted to a 32-bit number

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

- The 16-bit binary version of the number -2 is

  $1111\ 1111\ 1111\ 1110_{two} = 2_{ten}$

- converted to a 32-bit number

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$

REPRESENTING INSTRUCTIONS

- Instructions are kept in the computer as a series of high and low electronic signals
- there must be a convention to map register names into numbers.
  - \$t0 – \$t7 are reg's 8 – 15
  - \$t8 – \$t9 are reg's 24 – 25
  - \$s0 – \$s7 are reg's 16 – 23
- we call the numeric version of instructions **machine language**

Translating a MIPS Assembly Instruction into a Machine Instruction

field

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

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

- $t0 – $t7 are reg's 8 – 15
- $t8 – $t9 are reg's 24 – 25
- $s0 – $s7 are reg's 16 – 23

special	$s1	$s2	$t0	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

DESIGN PRINCIPLE 4 :Good design demands good compromises

- To compromise to keep all instructions the same length, so it require different kinds of instruction formats

MIPS **R**-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- 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)

MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15}-1$
  - Address: offset added to base address in rs

Instruction	Format	op	rs	rt	rd	shamt	funct	address
`add`	R	0	reg	reg	reg	0	32_{ten}	n.a.
`sub` (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
`add immediate`	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
`lw` (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
`sw` (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

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 are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

- $to – $t7 are reg's 8 – 15
- $t8 – $t9 are reg's 24 – 25
- $so – $s7 are reg's 16 – 23

EX: A[300] = h + A[300];

Translating MIPS Assembly Language into Machine Language. $t1 has the base of the array A and $s2 corresponds to h

Solution: Compile to

- lw $t0,1200($t1)
- add $t0,$s2,$t0
- sw $t0,1200($t1)

- $t0 – $t7 are reg's 8 – 15
- $t8 – $t9 are reg's 24 – 25
- $s0 – $s7 are reg's 16 – 23

- MIPS machine language code

op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

- the binary equivalent to the decimal form is:

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

- $t0 – $t7 are reg's 8 – 15
- $t8 – $t9 are reg's 24 – 25
- $s0 – $s7 are reg's 16 – 23

EX: What MIPS instruction does this represent?

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

- **Answer**

sub $t2, $t0, $t1

LOGICAL OPERATIONS

- bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

SHIFT

move all the bits in a word to the left or right filling the emptied bits with 0s.

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{two} = 9_{ten}$

- After shift left by 4

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two} = 144_{ten}$

- Two MIPS shift instructions
  - **shift left logical (sll)**
  - **shift right logical (srl)**

SHIFT

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)

```
sll  $t2,$s0,4   # reg $t2 = reg $s0 << 4 bits
```

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

- $to – $t7 are reg's 8 – 15
- $t8 – $t9 are reg's 24 – 25
- $so – $s7 are reg's 16 – 23

AND

`and $t0, $t1, $t2`

Used for selecting some bits (clear other bits to 0)

Bit 1	Bit 2	Result bits
1	1	1
1	0	0
0	1	0
0	0	0

\$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

`or $t0, $t1, $t2`

- Used for including bits in a word
  - Set some bits to 1, leave others unchanged

Bit 1	Bit 2	Result bits
1	1	1
1	0	1
0	1	1
0	0	0

$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

NOR

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

- **NOT** operation used to invert bits in a word
  - Change 0 to 1, and 1 to 0
- **NOR** (Not Or)
  - a NOR b == NOT (a OR b)

Bit 1	Bit 2	Result bits
1	1	0
1	0	0
0	1	0
0	0	1

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

XOR

- Exclusive or
- sets the bit to 1 when two corresponding bits differ, and to 0 when they are the same.

Bit 1	Bit 2	Result bits
1	1	0
1	0	1
0	1	1
0	0	0

\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$zero	0000 0000 0000 0000 0000 0000 0000 0000
\$t0	0000 0000 0000 0000 0011 1100 0000 0000