

Assembly Language And Machine Code

C Statement:

```
int foo; foo = 15; foo = foo + 7;
```

MIPS Assembly Language:

```
ori $1,$0,15      # set foo to 15
addiu $1,$1,7      # add 7 to foo
```

(register 1 holds the value of foo)

MIPS Machine Instructions:

```
001101000000000010000000000001111
0010010000010000010000000000000111
```



Number Representation

Decimal: base 10, digits: '0', '1', ..., '9'

$$(683)_{10} = 6 \cdot 10^2 + 8 \cdot 10^1 + 3 \cdot 10^0$$

Binary: base 2, digits: '0', '1'

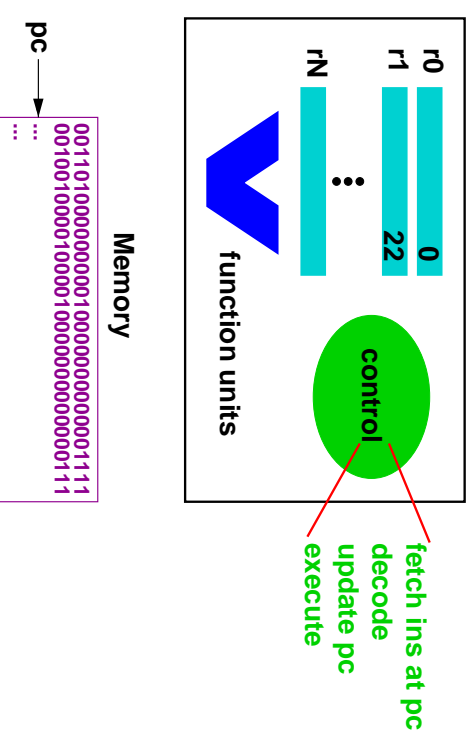
$$\begin{aligned}(1101)_2 &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 8 + 4 + 0 + 1 \\ &= (13)_{10}\end{aligned}$$

Hexadecimal: base 16, digits: '0' .. '9', 'a' .. 'f'

$$\begin{aligned}\text{'a'} = 10, \text{'b'} = 11, \text{'c'} = 12, \text{'d'} = 13, \text{'e'} = 14, \text{'f'} = 15 \\ (a6)_{16} = 10 \cdot 16^1 + 6 \cdot 16^0 = (166)_{10}\end{aligned}$$

Often write 0xa6 instead of $(a6)_{16}$.

A Simple Computer



Number Representation

A Useful Trick: Converting between hexadecimal (hex) and binary.

$$\begin{aligned}0xe3f8 &= 14 \cdot 16^3 + 3 \cdot 16^2 + 15 \cdot 16^1 + 8 \cdot 16^0 \\ &= 14 \cdot (2^4)^3 + 3 \cdot (2^4)^2 + 15 \cdot (2^4)^1 + 8 \cdot (2^4)^0 \\ &= (1110)_2 \cdot (2^4)^3 + (0011)_2 \cdot (2^4)^2 \\ &\quad + (1111)_2 \cdot (2^4)^1 + (1000)_2 \cdot (2^4)^0 \\ &= (\underbrace{1110}_{0xe} \underbrace{0011}_{0x3} \underbrace{1111}_{0xf} \underbrace{1000}_{0x8})_2\end{aligned}$$

1 hex digit = 4 bits

Negative Numbers

Various representations possible for signed binary arithmetic.

Sign-Magnitude: reserve left-most bit for the sign

- + Easy to negate a number
- Multiple zeros
- Arithmetic is more complicated

Example: 4-bit numbers

- $(+5)_{10}$ is given by 0 101
- $(-5)_{10}$ is given by 1 101



Why 2's complement?

Let b be the integer we're trying to negate. (N -bits)

- Flip bits \equiv subtract b from $\underbrace{111\dots1}_{N \text{ 1s}}$

$$\begin{array}{r} 1 \ 1 \ 1 \ 1 \\ - \ 0 \ 1 \ 0 \ 1 \\ \hline 1 \ 0 \ 1 \ 0 \end{array}$$

$$\underbrace{111\dots1}_{N \text{ 1s}} = 2^N - 1$$

$$\begin{array}{r} 1 \ 0 \ 0 \ 0 \ 0 \\ - \ 0 \ 0 \ 0 \ 1 \\ \hline 1 \ 1 \ 1 \ 1 \end{array}$$

- Add 1

$$result = 2^N - b$$



Negative Numbers

2's complement

- Flip all the bits and add 1
- + No wasted bits
- + Arithmetic works out
- Asymmetric range for positive and negative numbers

Example: 4-bit numbers

- $(+5)_{10}$ is given by 0101
- Flip bits: 1010
- Add 1: 1011



Why 2's complement?

For 2's complement: $-b$ is represented by $2^N - b$.

... which is $-b$ modulo 2^N .

\Rightarrow we can use the same computation structure to add positive and negative numbers if we use modulo 2^N arithmetic.



Sign Extension

How do I convert an 8-bit number into a 16-bit number?

- If the number is non-negative, left-most bit is 0
 \Rightarrow add 0s to the left
- If the number is $-b$, then it corresponds to $2^8 - b$.
 $2^{16} - b = (2^8 - b) + (2^{16} - 2^8)$
 \Rightarrow add 1s to the left

In both cases, **replicate left-most bit**
Known as “sign-extension”



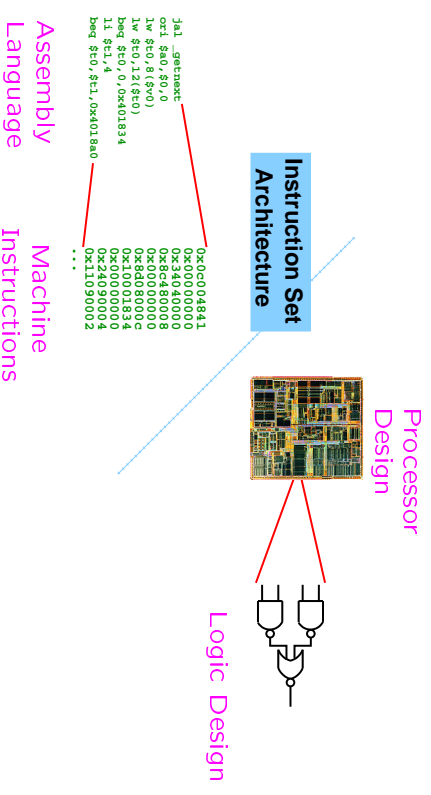
MIPS Instruction Set Architecture

Basic features:

- Load/store architecture
 - Data must be in registers to be operated on
 - Keeps hardware simple
 - Memory operations only transfer data between registers and memory
- Emphasis on efficient implementation
- Very simple: basic operations rather than support for any specific language construct



Instruction Set Architecture



ISA: operands, data types, operations, encoding



MIPS Data Representation

Integer data types:

- Byte: 8 bits
- Half-words: 16 bits
- Words: 32 bits
- Double-words: 64 bits (not in basic MIPS I)

MIPS supports operations on signed and unsigned data types.

Converting a byte to a word? Sign-extend!



MIPS Instruction Types

- **Arithmetic/Logical**
 - three operands: result + two sources
 - operands: registers, 16-bit immediates
 - signed + unsigned operations
- **Memory access**
 - load/store between registers and memory
 - half-word and byte operations
- **Control flow**
 - conditional branches, fixed offsets and pc-relative



Data Storage

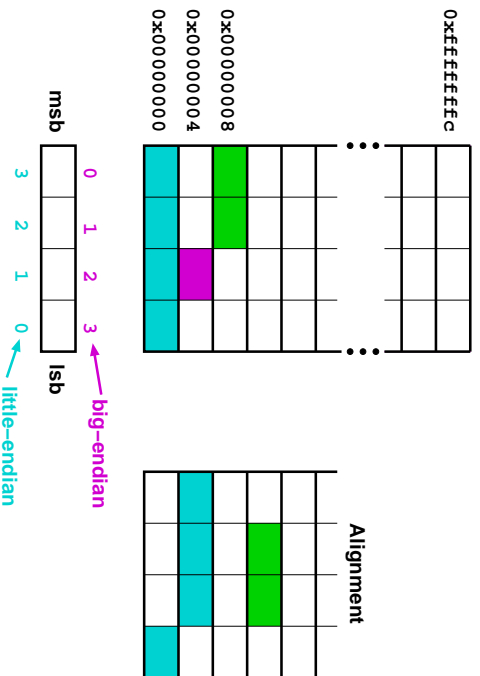
- 32 32-bit registers, register 0 is always zero.
- 2^{32} bytes of memory
- hi, lo: special 32-bit registers for multiply/divide
- pc, program counter
- 16 floating-point registers

Memory access:

- Byte addressing: can address individual bytes of memory
- How do bytes map into words?



Byte Ordering And Alignment



“On Holy Wars and a Plea for Peace”, Cohen (1980)



Data Movement

Load/store architecture

- Read data from memory: “load”
- Write data to memory: “store”

Load:

- Normally overwrites **entire** register
- Loading bytes/half-words
 - unsigned: zero-extend
 - signed: sign-extend

Store: writes bottom byte/bottom half-word/word of register to memory.



Addressing Modes For Data Movement

How do we specify an address in memory?

- Instructions compute **effective address** (EA)

MIPS: One addressing mode for loads/stores

- register indirect with immediate offset
- EA = register + signed immediate

Example:

```
lh $5, 8($29)
lw $7, -12($29)
lbu $7, 1($30)
```

Requires aligned addresses!



MIPS Load/Store Instructions

```
lb  rt, imm(rs) # load byte (signed)
lbu rt, imm(rs) # load byte (unsigned)

lh  rt, imm(rs) # load half-word (signed)
lhu rt, imm(rs) # load half-word (unsigned)

lw  rt, imm(rs) # load word
sw  rt, imm(rs) # store word

sb  rt, imm(rs) # store byte
sh  rt, imm(rs) # store half-word
```

op	rs	rt	imm
----	----	----	-----

6 bits 5 bits 5 bits 16 bits



Addressing Modes

Other architectures have more than one way to specify EA.

- EA = signed immediate
- EA = register
- EA = register + k × register (k=1,2,4,8)
- EA = register + k × register + signed immediate

MIPS favors simplicity ⇒ fast hardware



MIPS Load/Store Instructions

C Code

```
foo = x[3]; x[4] = foo + 1;
```

Assembly

```
lw    $16, 12($17) # reg 16 contains foo, reg 17
                    # contains the address of x
addiu $8, $16, 1   # add 1 to foo
sw    $8, 16($17)  # store into x[4]
```



Integer Arithmetic Operations

- **Constants**
 - register zero is always zero
 - immediates are 16-bits wide
- **Signed + unsigned operations**
- **Logical operations**
 - bitwise operations on operands
 - always unsigned



Integer Arithmetic Operations

```
add    rd, rs, rt    # rd = rs + rt
addi   rt, rs, imm   # rt = rs + s_ext(imm)
addiu  rt, rs, imm   # rt = rs + s_ext(imm)
addu   rd, rs, rt    # rd = rs + rt
slt     rd, rs, rt    # rd = (rs <_s rt)
slti   rt, rs, imm   # rt = (rs <_s s_ext(imm))
sltiu  rt, rs, imm   # rt = (rs < s_ext(imm))
sltu   rd, rs, rt    # rd = (rs < rt)
sub     rd, rs, rt    # rd = rs - rt
subu   rd, rs, rt    # rd = rs - rt
```

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

