

# 101 *Assembly*

ENGR 3410 - Computer Architecture

Mark L. Chang

Fall 2009

## What is **assembly**?

---

Very low level

1:1 correlation to CPU instructions

## Why are we learning assembly **now**?

---

Design?

Optimizing → higher level lang.

# Assembly Language

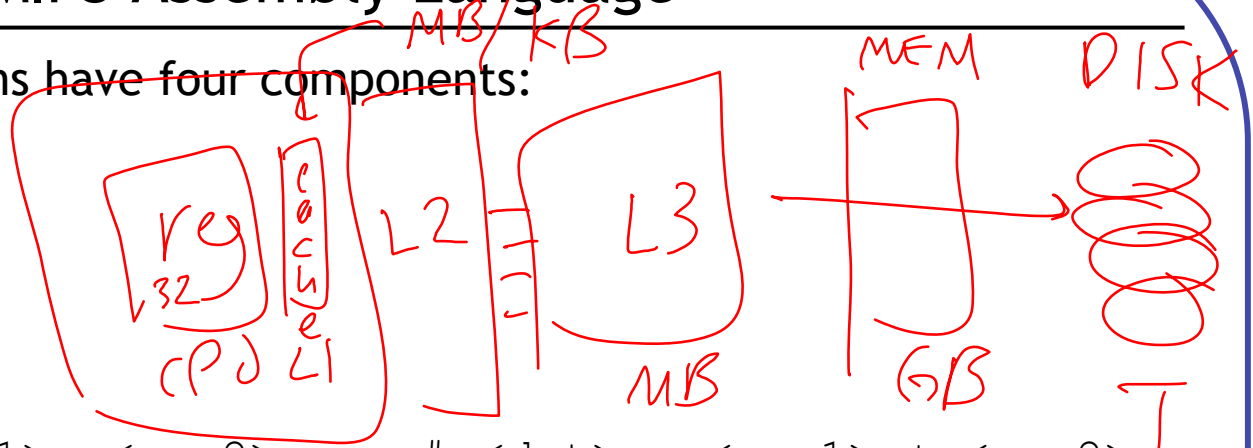
---

- Readings: Chapter 2 (2.1-2.6, 2.8, 2.9, 2.13, 2.15), Appendix A.10
- Assembly language
  - Simple, regular instructions - building blocks of C & other languages
  - Typically one-to-one mapping to machine language
- Our goal
  - Understand the basics of assembly language
  - Help figure out what the processor needs to be able to do
- Not our goal to teach complete assembly/machine language programming
  - Floating point
  - Procedure calls
  - Stacks & local variables

# MIPS Assembly Language

- The basic instructions have four components:

- Operator name
- Destination
- 1<sup>st</sup> operand
- 2<sup>nd</sup> operand



add <dst>, <src1>, <src2>

# <dst> = <src1> + <src2>

sub <dst>, <src1>, <src2>

# <dst> = <src1> - <src2>

- Simple format: easy to implement in hardware

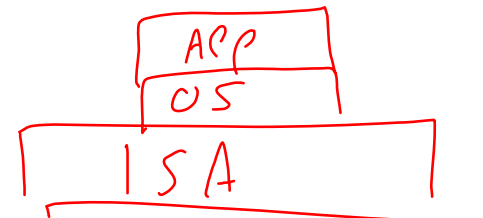
- More complex:  $A = B + C + D - E$

$$t_0 = B + C$$

$$t_1 = D - E$$

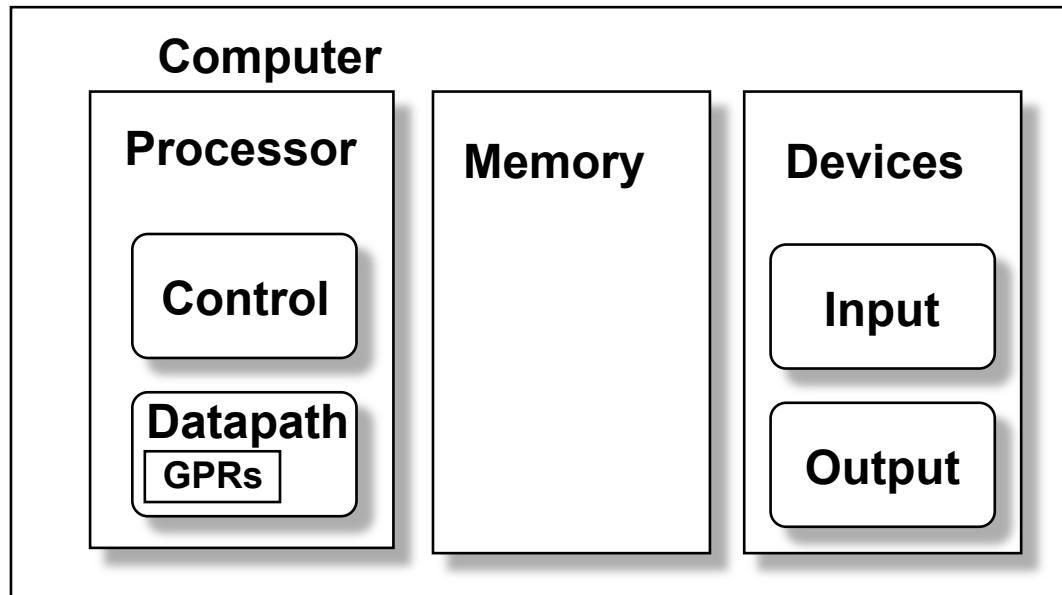
$$t_3 = t_0 + t_1$$

$$A = t_3$$



## Operands & Storage

- For speed, CPU has 32 general-purpose registers for storing most operands
- For capacity, computer has large memory (64MB+)



- Load/store operation moves information between registers and main memory
- All other operations work on registers

# Registers

- 32 registers for operands

Register	Name	Function	Comment
\$0	\$zero	Always 0	No-op on write
\$1	\$at	Reserved for assembler	Don't use it!
\$2-3	\$v0-v1	Function return	
\$4-7	\$a0-a3	Function call parameters	
\$8-15	\$t0-t7	Volatile temporaries	Not saved on call
\$16-23	\$s0-s7	Temporaries (saved across calls)	Saved on call
\$24-25	\$t8-t9	Volatile temporaries	Not saved on call
\$26-27	\$k0-k1	Reserved kernel/OS	Don't use them
\$28	\$gp	Pointer to global data area	
\$29	\$sp	Stack pointer	
\$30	\$fp	Frame pointer	
\$31	\$ra	Function return address	

## Basic Operations

(Note: just subset of all instructions)

Mathematic: add, sub, mult, div

Unsigned (changes overflow condition)

Immediate (one input a constant)

```
add $t0, $t1, $t2 # t0 = t1+t2
```

```
addu $t0, $t1, $t2 # t0 = t1+t2
```

```
addi $t0, $t1, 100 # t0 = t1+100
```

Logical: and, or, nor, xor

Immediate

```
and $t0, $t1, $t2 # t0 = t1&t2
```

```
andi $t0, $t1, 7 # t0 = t1&b0111
```

Shift: left & right logical, arithmetic

Immediate

```
sllv $t0, $t1, $t2 # t0 = t1<<t2
```

```
sll $t0, $t1, 6 # t0 = t1<<6
```

Example: Take bits 6-4 of \$t0 and make them bits 2-0 of \$t1, zeros otherwise:

*srli \$t1, \$t0, 4 → 32 bits*

*andi \$t1, \$t1, 7 → 32 bits*



# Memory Organization

---

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data

...

## Memory Organization (cont.)

---

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

Our registers hold 32 bits of data

- $2^{32}$  bytes with byte addresses from 0 to  $2^{32}-1$
- $2^{30}$  words with byte addresses 0, 4, 8, ...  $2^{32}-4$
- Words are aligned  
i.e., what are the least 2 significant bits of a word address?

# Endianness

---

- How do we write numbers?

1 5 2 8

Must establish a convention for the order of digits to represent numbers

# Endianness

- In memory, what is the order of a 32-bit word?

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data

0	DE
1	AD
2	BE
3	EF

Store the 32-bit word: 0xDEADBEEF

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data

0	EF
1	BE
2	AD
3	DE

# Big and Little Endian

- Big Endian - “Big End” in (first)

- Motorola 68000
- Sun SPARC
- PowerPC G5
- *Networks*

0	DE
1	AD
2	BE
3	EF

- Little Endian - “Little End” in (first)

- Intel x86
- MOS Tech 6502
  - Atari 2600, Apple ][, Commodore 64, NES

0	EF
1	BE
2	AD
3	DE

- Bi-Endian - switchable endianness

- ARM, IBM PowerPC (most)

- Middle-Endian

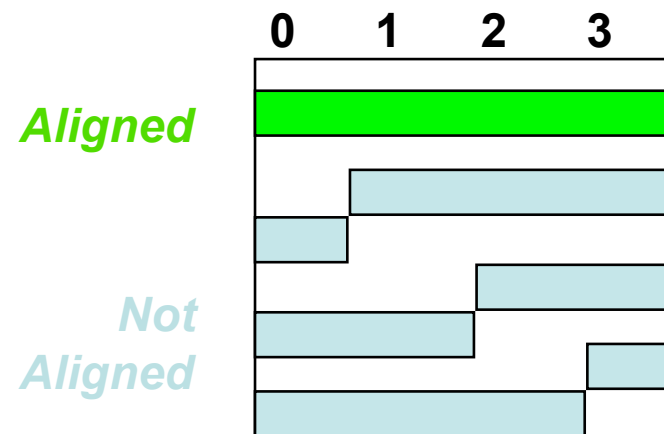
- PDP-11

0	
1	
2	
3	

# Word Alignment

---

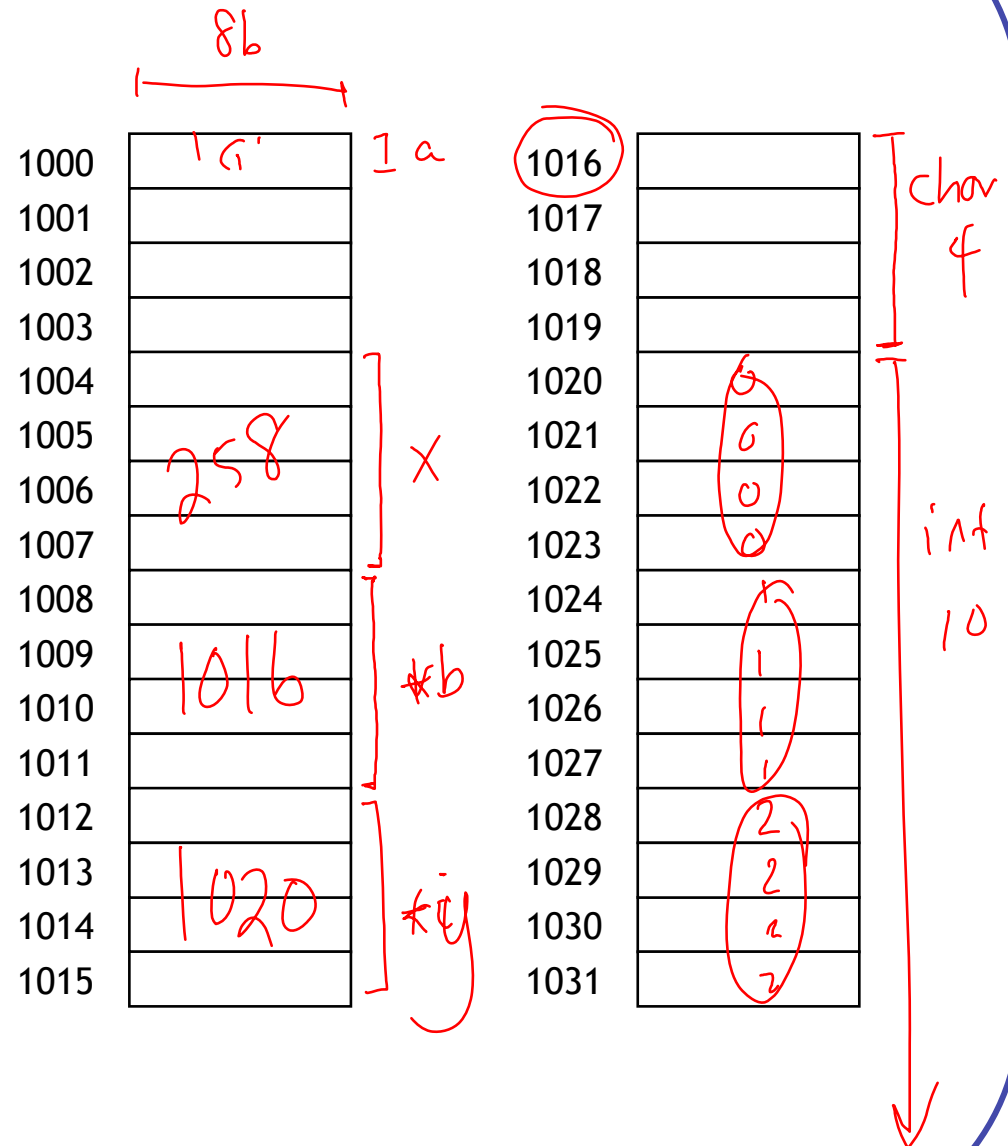
- Require that objects fall on an address that is a multiple of their size



# Data Storage

- Characters: 8 bits (byte)
- Integers: 32 bits (word)
- Array: Sequence of locations
- Pointer: Address = 32b

```
char a = 'G';
int x = 258;
char *b;
int *y;
b = new char[4];
y = new int[10];
```



# Loads & Stores

- Loads & Stores move data between memory and registers
  - All operations on registers, but too small to hold all data

`lw $t0, 16($t1)`      # `$t0 = Memory[$t1+16]`

`sw $t2, 8($t3)`      # `Memory[$t3+8] = $t2`

*if NO offset*

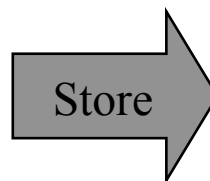
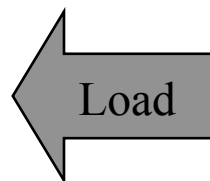
*REG*

<i>t0</i>	<i>124</i>
<i>t1</i>	<i>16</i>
<i>t2</i>	<i>140</i>
<i>t3</i>	<i>66</i>

*add \$t2, \$t1, \$t0*

*lw \$t3, \$t2*

General Purpose Registers	
t0:	66
t1:	124
t2:	723
t3:	4



Memory	
12:	723
140:	66

- Note: `lbu` & `sb` load & store bytes



# Array Example

*v[0] in 964*

```
/* Swap the kth and (k+1)th element of an array */
swap(int v[], int k) int m
```

```
{
    6 int temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
# Assume v in $a0,
   k in $a1
```

## General Purpose Registers

a0:	964
a1:	10
t0:	<i>40 → 1004</i>
t1:	<i>1111111</i>
t2:	<i>0</i>

Load

Store

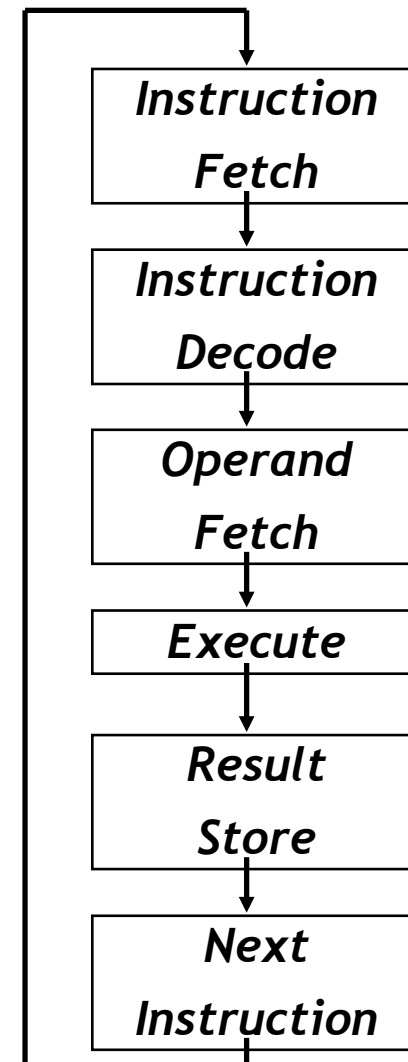
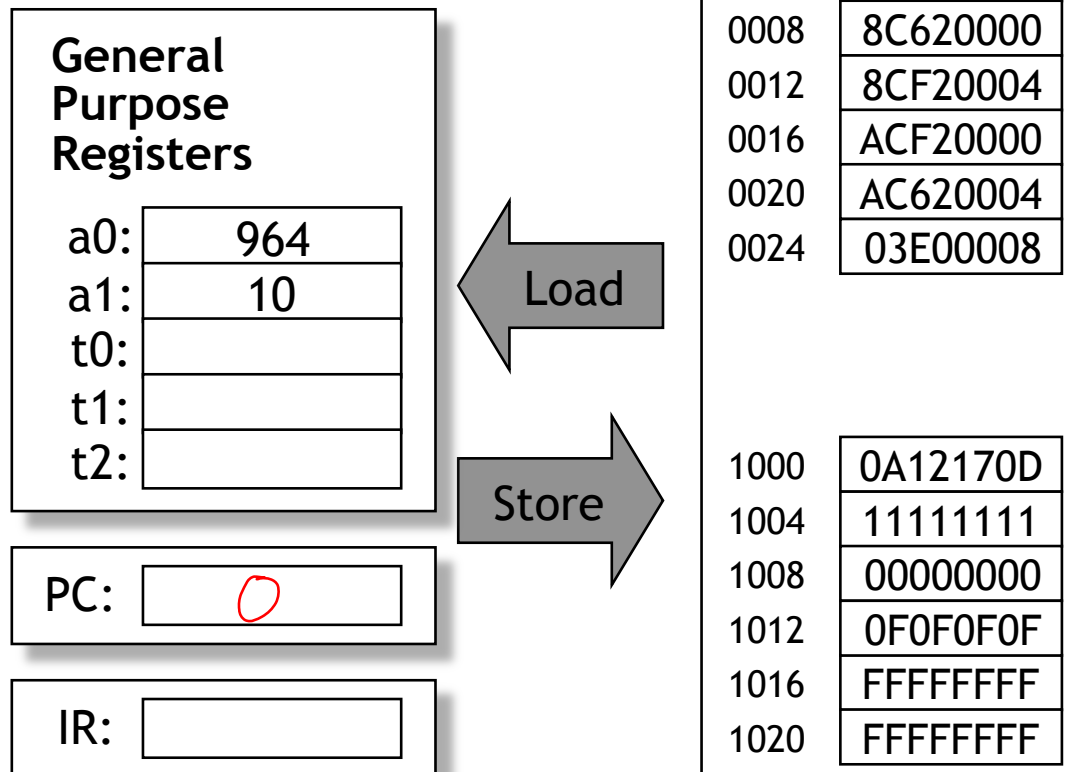
## Memory

1000	0A12170D
1004	11111111
1008	00000000
1012	0F0F0F0F
1016	FFFFFFFF
1020	FFFFFFFF

```
sll $t0, $a1, 2
add $t0, $a0, $t0
lw $t1, 0($t0)
lw $t2, 4($t0)
sw $t2, 0($t0)
sw $t1, 4($t0)
```

# Execution Cycle Example

- PC: Program Counter
- IR: Instruction Register



# Control Flow

- Jumps - GOTO different next instruction

```
j 25          # go to 100: PC = 25*4 (instructions are 32-bit)
jr $ra        # go to address in $ra: PC = value of $ra
```

- Branches - GOTO different next instruction if condition is true  
2 register: beq (==), bne (!=)

```
beq $t0, $t1, FOO # if $t0 == $t1 GOTO FOO: PC = FOO
```

- 1 register: bgez (>=0), bgtz (>0), blez (<=0), bltz (<0)

```
bgez $t0, FOO    # if $t0 >= 0 GOTO FOO: PC = FOO
```

if (a == b)	# \$a0 = a, \$a1 = b, \$a2 = c	
a = a + 3;	bne    \$a0, \$a1, ELSEIF	# branch if a!=b
else	addi   \$a0, \$a0, 3;	# a = a + 3
b = b + 7;	j DONE;	# avoid else
c = a + b;	ELSEIF:	
	addi   \$a1, \$a1, 7;	# b = b + 7
	DONE:	
	add    \$a2, \$a0, \$a1;	# c = a + b

## Loop Example

- Compute the sum of the values 1...N-1

```
int sum = 0;
for (int I = 0; I != N; I++) {
    sum += I;
}
```

# \$t0 = N, \$t1 = sum, \$t2 = I

	add \$t1, \$zero, \$zero	0
	add \$t2, \$zero, \$zero	4
Top:	beq \$t0, \$t2, END	8
	add \$t1, \$t1, \$t2	12
	addi \$t2, \$t2, 1	16
	j Top	20
END:	<u>~~~~~</u>	24

## Comparison Operators

- For logic, want to set a register TRUE (1) / FALSE(0) based on condition

```
slt $t0, $t1, $t2      # if ($t1 < $t2) $t0 = 1 else $t0 = 0;
```

```
if (a >= b)
    c = a + b;
a = a + c;
```

```
# $t0 = a, $t1 = b, $t2 = c
```

```
slt $t3, $t0, $t1      # a < b
```

```
bne $t3, $zero, DONE   # if !(a >= b)
```

```
add $t2, $t0, $t1
```

```
DONE: add $t0, $t0, $t2
```

## String toUpper

---

- Convert a string to all upper case

```
char *index = string;
while (*index != 0) { /* C strings end in 0 */
    if (*index >= 'a' && *index <= 'z')
        *index = *index + ('A' - 'a');
    index++;
}
```

```
# $t0=index, $t2='a', $t3='z', $t4='A'-'a', Memory[100]=string
```

# String toUpper

- Convert a string to all upper case

```
char *index = string;
while (*index != 0) { /* C strings end in 0 */
    if (*index >= 'a' && *index <= 'z')
        *index = *index + ('A' - 'a');
    index++;
}
```

```
# $t0=index, $t2='a', $t3='z', $t4='A'-'a', Memory[100]=string
```

```
    lw    $t0, 100($zero)      # index = string
```

```
LOOP:
```

```
    lbu   $t5, 0($t0)          # t5 = *index
    beq   $t5, $zero, END      # exit if *index == 0
    slt   $t6, $t5, $t2        # is *index < 'a'?
    bne   $t6, $zero, NEXT     # don't change if < 'a'
    slt   $t6, $t3, $t5        # is 'z' < *index?
    bne   $t6, $zero, NEXT     # don't change if 'z' < *index
    add   $t5, $t5, $t4        # t5 = *index + ('A' - 'a')
    sb    $t5, 0($t0)          # *index = new value;
```

```
NEXT:
```

```
    addi  $t0, $t0, 1          # index++;
    j     LOOP                 # continue the loop
```

```
END:
```

# Machine Language vs. Assembly Language

- Assembly Language
  - mnemonics for easy reading
  - labels instead of fixed addresses
  - easier for programmers
  - almost 1-to-1 with machine language
- Machine language
  - Completely numeric representation
  - format CPU actually uses

SWAP:

```
sll    $2, $5, 2
add    $2, $4, $2    // Compute address of v[k]
lw     $15, 0($2)    // get v[k]
lw     $16, 4($2)    // get v[k+1]
sw     $16, 0($2)    // save new value to v[k]
sw     $15, 4($2)    // save new value to v[k+1]
jr     $31           // return from subroutine
```

```
000000 00000 00101 00010 00010 000000
000000 00100 00010 00010 00000 100000
100011 00010 01111 00000 00000 000000
100011 00010 10000 00000 00000 000100
101011 00010 10000 00000 00000 000000
101011 00010 01111 00000 00000 000100
000000 11111 00000 00000 00000 001000
```



# Labels

- Labels specify the address of the corresponding instruction
  - Programmer doesn't have to count line numbers
  - Insertion of instructions doesn't require changing entire code

```
# $t0 = N, $t1 = sum, $t2 = I
```

```
0 add $t1, $zero, $zero # sum = 0
```

```
4 add $t2, $zero, $zero # I = 0
```

```
TOP:
```

```
8 bne $t0, $t2, END # I!=N
```

```
12 add $t1, $t1, $t2 # sum += I
```

```
16 addi $t2, $t2, 1 # I++
```

```
20 j TOP # next iteration
```

```
24 END:
```

- Notes:

- Jumps are pseudo-absolute:
  - $PC = \{ PC[31:26], 26\text{-bit unsigned-Address}, "00" \}$
- Branches are PC-relative:
  - $PC = PC + 4 + 4 \times (16\text{-bit signed Address})$

inst(.) 000010 j target

# Instruction Types

---

- Can group instructions by # of operands

3-register

2-register

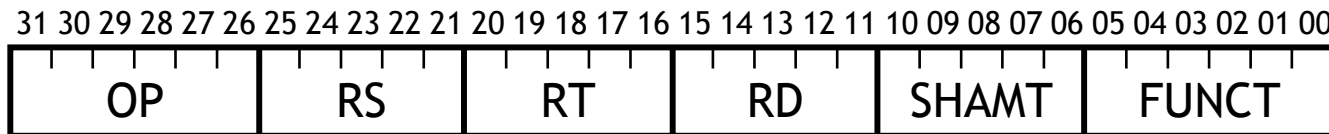
1-register

0-register

add	\$t0, \$t1, \$t2	# t0 = t1+t2
addi	\$t0, \$t1, 100	# t0 = t1+100
and	\$t0, \$t1, \$t2	# t0 = t1&t2
andi	\$t0, \$t1, 7	# t0 = t1&b0111
sllv	\$t0, \$t1, \$t2	# t0 = t1<<t2
sll	\$t0, \$t1, 6	# t0 = t1<<6
lw	\$t0, 12(\$t1)	# \$t0 = Memory[\$t1+10]
sw	\$t2, 8(\$t3)	# Memory[\$t3+10] = \$t2
j	25	# go to 100 - PC = 25*4 (instr are 32-bit)
jr	\$ra	# go to address in \$ra - PC = value of \$ra
beq	\$t0, \$t1, FOO	# if \$t0 == \$t1 GOTO FOO - PC = FOO
bgez	\$t0, FOO	# if \$t0 >= 0 GOTO FOO - PC = FOO
slt	\$t0, \$t1, \$t2	# if (\$t1 < \$t2) \$t0 = 1 else \$t0 = 0;

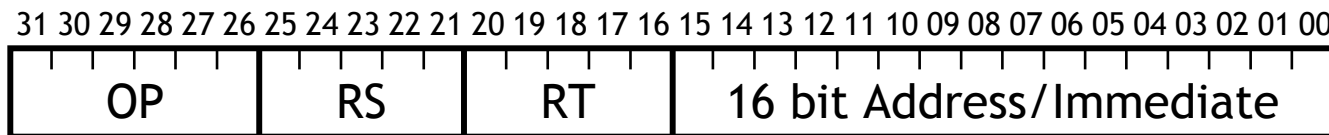
# Instruction Formats

- All instructions encoded in 32 bits (operation + operands/immediates)
- Register (R-type) instructions



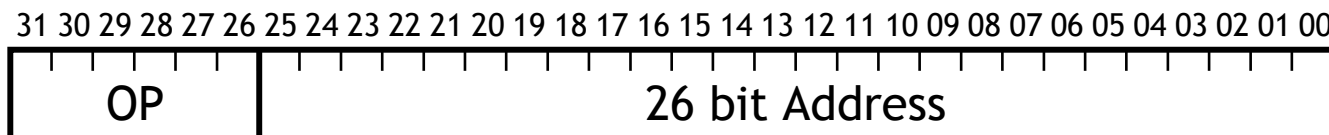
(OP = 0,16-20)

- Immediate (I-type) instructions



(OP = any but 0,2,3,16-20)

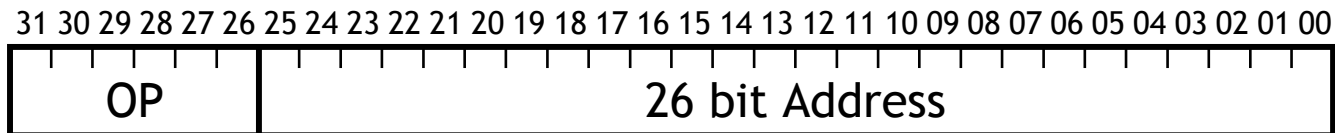
- Jump (J-type) instructions



(OP = 2,3)

## J-Type

- Used for unconditional jumps

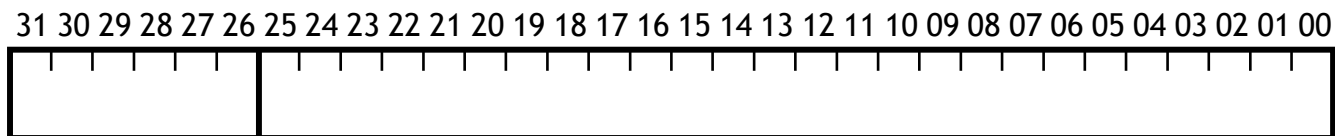


2: j (jump)

3: jal (jump and link)

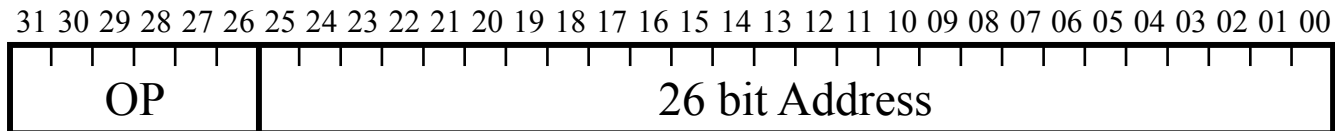
- Note: top 6 bits of jumped-to address come from current PC
- Example:

j 25 # go to 100, PC = 25\*4 (instr are 32-bit)



## J-Type

- Used for unconditional jumps

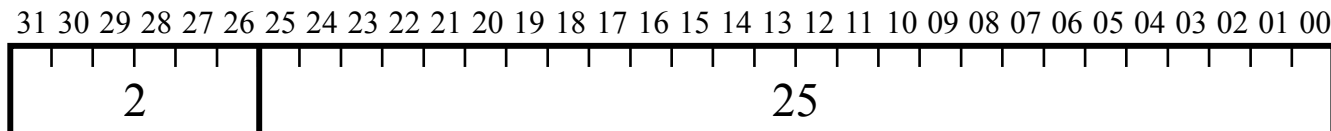


2: j (jump)

3: jal (jump and link)

- Note: top 4 bits of jumped-to address come from current PC
- Example:

j 25 # go to 100 - PC = 25\*4 (instr are 32-bit)



# I-Type

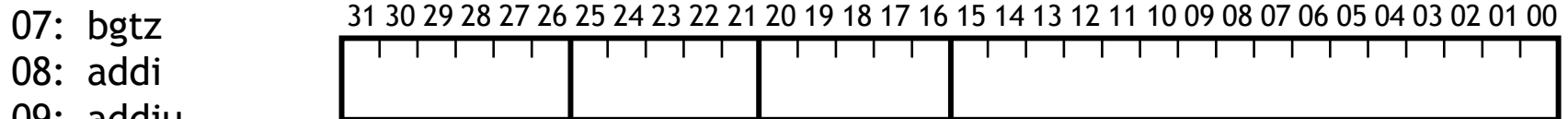
- Used for operations with immediate (constant) operand

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00



04: beq      Op1,      Op2, Dest,  
05: bne      L/S addr      L/S targ

06: blez      addi      \$8, \$9, 100      # \$8 = \$9+100



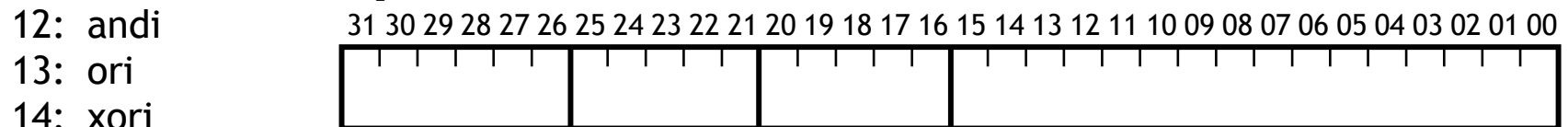
08: addi

09: addiu

10: slti

11: sltiu

beq      \$8, \$9, -11      # if \$8 == \$9 GOTO (PC+4+FOO\*4)



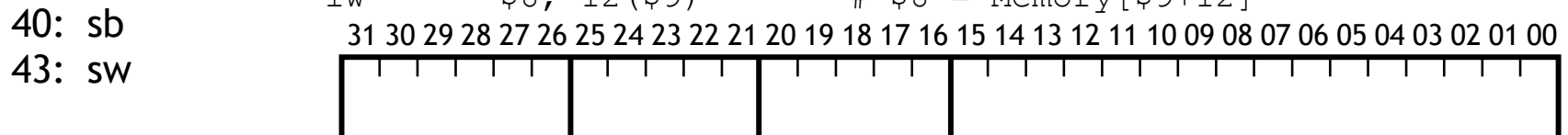
13: ori

14: xori

32: lb

35: lw

lw      \$8, 12(\$9)      # \$8 = Memory[\$9+12]



43: sw

# I-Type

- Used for operations with immediate (constant) operand

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00



04: beq      Op1,      Op2, Dest, L/  
              L/S addr      S targ  
 05: bne  
 06: blez      addi      \$8, \$9, 100      # \$8 = \$9+100  
 07: bgtz  
 08: addi  
 09: addiu  
 10: slti  
 11: sltiu  
 12: andi  
 13: ori  
 14: xori  
 32: lb  
 35: lw  
 40: sb  
 43: sw

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00



beq      \$8, \$9, -11      # if \$8 == \$9 GOTO (PC-11\*4)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00



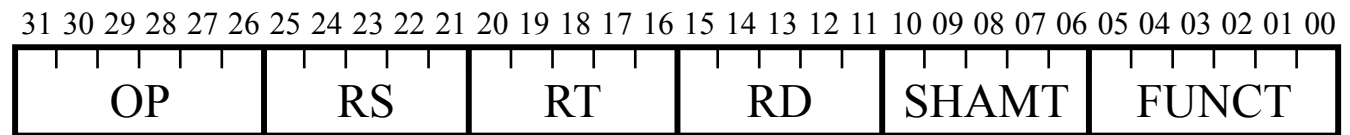
lw      \$8, 12(\$9)      # \$8 = Memory[\$9+12]

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00



# R-Type

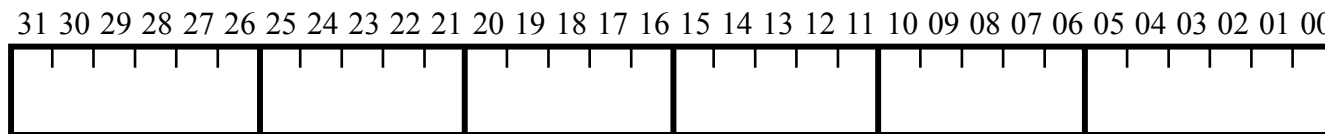
- Used for 3 register ALU operations



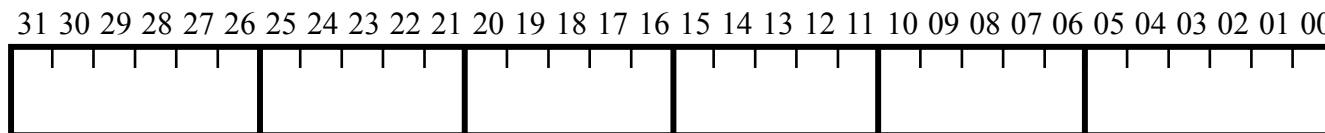
00 (16-20 for FP)	Op1	Op2	Dest	Shift amount (0 for non-shift)	
----------------------	-----	-----	------	-----------------------------------	--

00:	sll
02:	srl
03:	sra
04:	sllv
06:	srlv
07:	srav
08:	jr
24:	mult
26:	div
32:	add
33:	addu
34:	sub
35:	subu
36:	and
37:	or
38:	xor
39:	nor
42:	slt

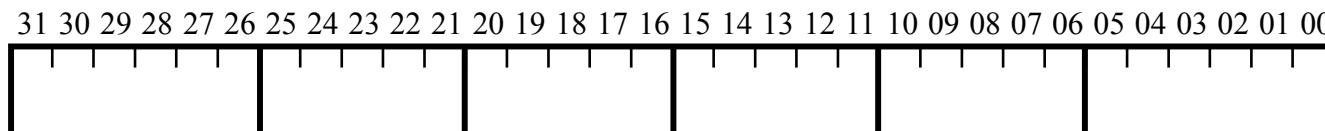
add        \$8, \$9, \$10        # \$8 = \$9+\$10



sll        \$8, \$9, 6        # \$8 = \$9<<6



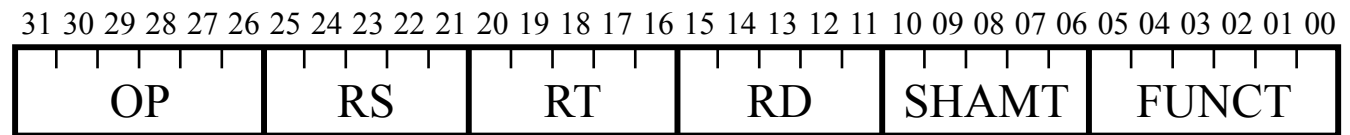
sllv        \$8, \$9, \$10        # \$8 = \$9<<\$10





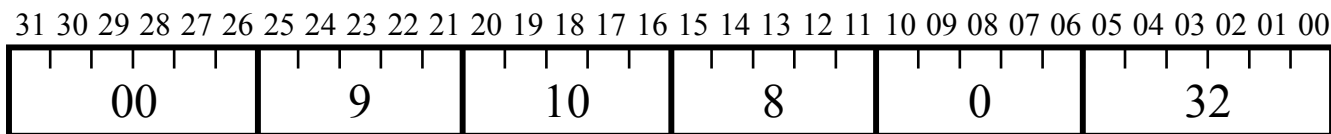
# R-Type

- Used for 3 register ALU operations

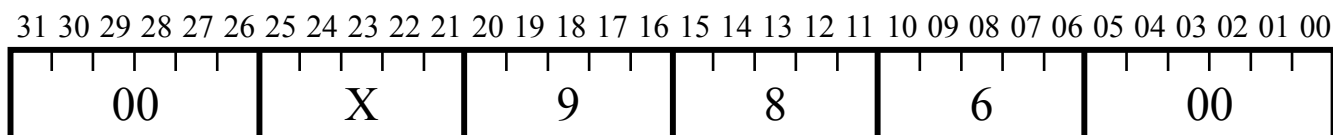


00      Op1      Op2      Dest      Shift amount      00: sll  
 (10-13 for FP)      (0 for non-shift)      02: srl

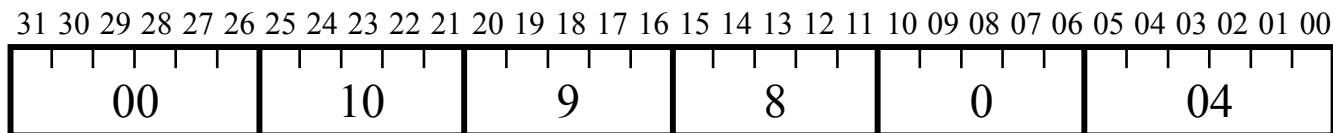
add      \$8, \$9, \$10      # \$8 = \$9+\$10



sll      \$8, \$9, 6      # \$8 = \$9<<6



sllv      \$8, \$9, \$10      # \$8 = \$9<<\$10



03: sra  
 04: sllv  
 06: srlv  
 07: srav  
 08: jr  
 24: mult  
 26: div  
 32: add  
 33: addu  
 34: sub  
 35: subu  
 36: and  
 37: or  
 38: xor  
 39: nor  
 42: slt

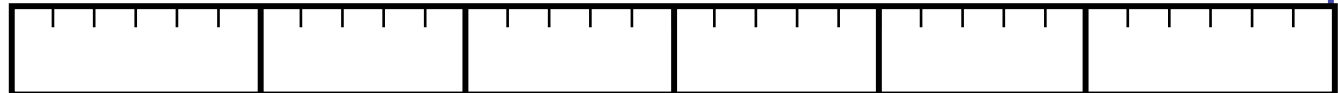
## Conversion example

- Compute the sum of the values 0...N-1

004 add \$9, \$0, \$0

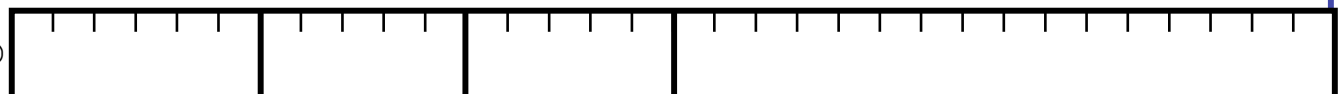


008 add \$10, \$0, \$0

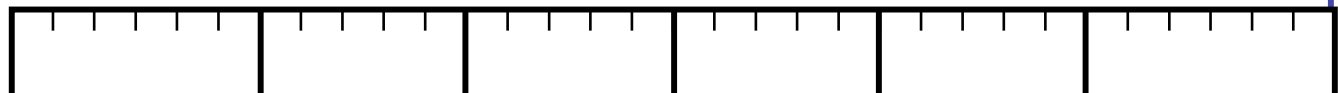


TOP:

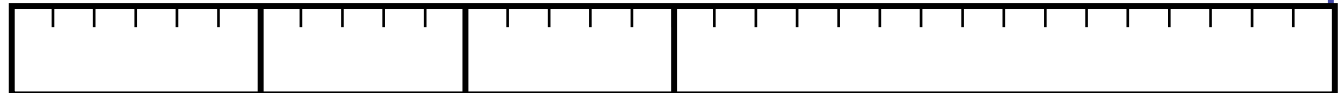
012 bne \$8, \$10, END



016 add \$9, \$9, \$10



020 addi \$10, \$10, 1



024 j TOP



END:

028

## Conversion example

- Compute the sum of the values 0...N-1

004	add	\$9, \$0, \$0	00	0	0	9	0	32
008	add	\$10, \$0, \$0	00	0	0	10	0	32
TOP:								
012	bne	\$8, \$10, END	05	8	10	+3		
016	add	\$9, \$9, \$10	00	9	10	9	0	32
020	addi	\$10, \$10, 1	08	10	10	1		
024	j	TOP	02	3				
END:								
028								

# Assembly & Machine Language

---

- Assembly
- Machine Language