



# **EECS 388: Embedded Systems Lecture 6 Instructions: Language of the Computer**

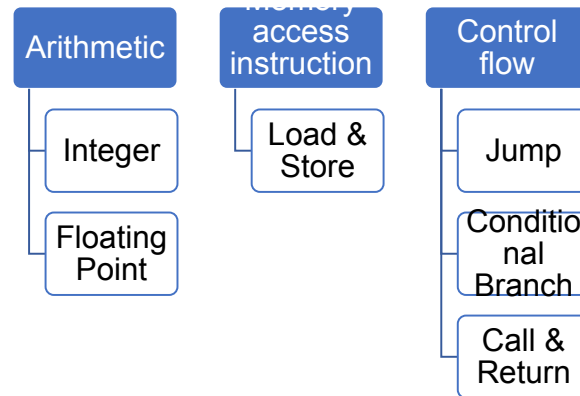
---

**Instructor: Tamzidul Hoque, Assistant Professor,  
Dept. of EECS, University of Kansas  
([hoque@ku.edu](mailto:hoque@ku.edu)), office: Eaton 2038**

**Credits: Some slides are adopted from Henk  
Corporaal's Computer Architecture and Organization  
course**



# So far: Classes of Instructions in MIPS



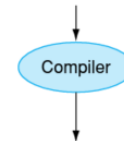
## BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct
	31	26 25	21 20	16 15	11 10	6 5
I	opcode	rs	rt	immediate		
	31	26 25	21 20	16 15		
J	opcode	address				
	31	26 25				

High-level  
language  
program  
(in C)

```

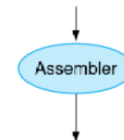
swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
    
```



Assembly  
language  
program  
(for MIPS)

```

swap:
    muli $2, $5, 4
    add $2, $4, $2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
    
```



Binary machine  
language  
program  
(for MIPS)

```

0000000001010000100000000000011000
000000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000001111100000000000000001000
    
```

# Machine Language: R-type instr

- Instructions, like registers and words of data, are also 32 bits long

- Example: add \$t0, \$s1, \$s2
- Registers have numbers: \$t0=9, \$s1=17, \$s2=18

- Instruction Format:

<i>op</i>	<i>rs</i>	<i>rt</i>	<i>rd</i>	<i>shamt</i>	<i>funct</i>
000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

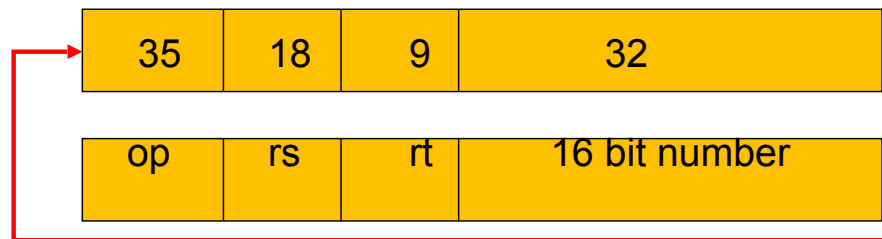
- *op*: Basic operation of the instruction, traditionally called the **opcode**.
- *rs*: The first register source operand.
- *rt*: The second register source operand.
- *rd*: The register destination operand. It gets the result of the operation.
- *shamt*: Shift amount. (Section 2.5 explains shift instructions and this term; it will not be used until then, and hence the field contains zero.)
- *funct*: Function. This field selects the specific variant of the operation in the *op* field and is sometimes called the *function code*.

NAME NUMBER

\$zero	0
\$at	1
\$v0-\$v1	2-3
\$a0-\$a3	4-7
\$t0-\$t7	8-15
\$s0-\$s7	16-23
\$t8-\$t9	24-25
\$k0-\$k1	26-27
\$gp	28
\$sp	29
\$fp	30
\$ra	31

# Machine Language: I-type instr

- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`



Go to this link to find the encoding for opcodes and operands [https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS\\_Green\\_Sheet.pdf](https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf)

Jump Register	jr	R	PC=R[rs]	0 / 08 <sub>hex</sub>
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2) 24 <sub>hex</sub>
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2) 25 <sub>hex</sub>
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7) 30 <sub>hex</sub>
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}	f <sub>hex</sub>
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2) 23 <sub>hex</sub>

35 in dec

# Translating Instruction

We can now take an example all the way from what the programmer writes to what the computer executes. If `$t1` has the base of the array `A` and `$s2` corresponds to `h`, the assignment statement

```
A[300] = h + A[300];
```

is compiled into

```
lw    $t0,1200($t1) # Temporary reg $t0 gets A[300]
```

```
add   $t0,$s2,$t0    # Temporary reg $t0 gets h + A[300]
```

```
sw    $t0,1200($t1) # Stores h + A[300] back into A[300]
```

What is the MIPS machine language code for these three instructions?

# So far: Instruction formats and encoding

## MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; data in registers
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; data in registers
Data transfer	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Data from memory to register
	store word	sw \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Data from register to memory

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

# Constants

- Small constants are used quite frequently in codes (50% of operands)  
e.g.,  $i = i + 2;$   
 $j = j - 1;$   
 $k = k + 20;$
- Regular arithmetic instructions requires the constant to be present in Reg.
  - We can eliminate this load operation to register
- Possible Solution:
  - Include a new instruction type->immediate instructions
  - Immediate instructions reflect one of the key design principles: *Make the common case fast.*

# Immediate Instructions in MIPS

- MIPS Instructions:

```
addi $s3, $s3, 22
andi $s3, $s3, 9
ori $s3, $s3, 11
```

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



# Logical Operations

- Useful to operate on fields of bits within a word or on individual bits.
  - Set, clear, toggle bits
  - Faster multiplication and division
  - Data encryption, encoding etc.

Logical operations	C operators	MIPS
Shift left	<<	sll
Shift right	>>	srl
Bit-by-bit AND	&	and, andi
Bit-by-bit OR		or, ori
Bit-by-bit NOT	~	nor

and	and	\$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3
or	or	\$s1,\$s2,\$s3	\$s1 = \$s2   \$s3
nor	nor	\$s1,\$s2,\$s3	\$s1 = ~ (\$s2   \$s3)
and immediate	andi	\$s1,\$s2,100	\$s1 = \$s2 & 100
or immediate	ori	\$s1,\$s2,100	\$s1 = \$s2   100
shift left logical	sll	\$s1,\$s2,10	\$s1 = \$s2 << 10
shift right logical	srl	\$s1,\$s2,10	\$s1 = \$s2 >> 10

# Bitwise AND/OR

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

Operand 1	Operand 2	Bitwise AND (A & B)
1000	0111	0000
1100	1100	1100
0000	0000	0000

# Bitwise Shifts

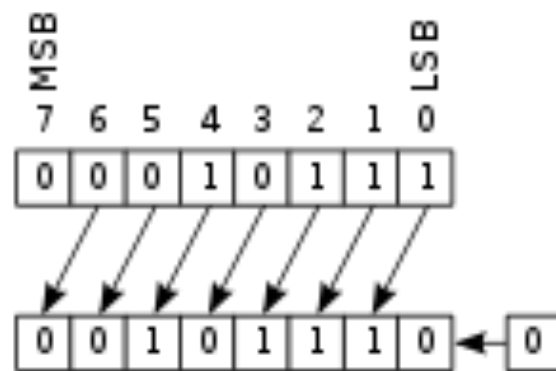
- sll = left shift (<<).

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

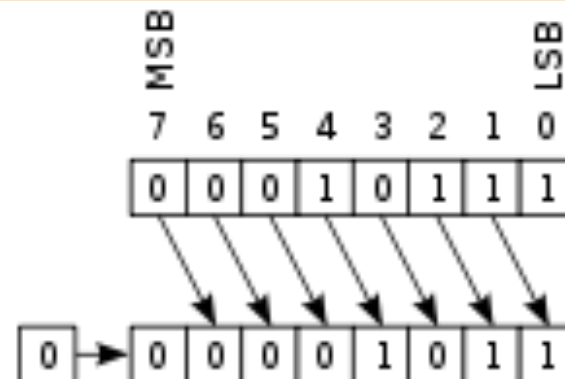
- srl = right shift (>>).

- Where do encode put the shift amount?
- What is the maximum shift amount ?

Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R-format	R	op	rs	rt	rd	shamt	funct
I-format	I	op	rs	rt	address		



Logical left shift



Logical right shift

# Instructions for Making Decisions

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed
- MIPS conditional branch instructions:

branch on equal	beq \$s1,\$s2,L	if (\$s1 == \$s2) go to L
branch on not equal	bne \$s1,\$s2,L	if (\$s1 != \$s2) go to L

- Example: if (i==j) h = i + j;  
          bne \$s0, \$s1, Label  
          add \$s3, \$s0, \$s1  
Label:     ....

# If-then-else conditional branches

- To implement the both if and else, we need an unconditional jump

```
j Exit    # go to Exit
```

- Example:

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

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j Exit
Else:    sub $s0, $s1, $s2
Exit:
```

# Loops

- while loop in C:

```
while (save[i]==k)
    i+=1;
```

- $i \rightarrow \$s3$ ,  $k \rightarrow \$s5$ , base of the array save is in  $\$s6$ .

```
Loop: sll    $t1,$s3,2    # reg $t1=4*i
      add    $t1,$t1,$s6  # $t1= address of save[i]
      lw     $t0,0($t1)   # reg t0=save[i]
      bne    $t0, $s5, Exit # go to Exit if save[i]!=k
      addi   $s3,$s3,1    # i=i+1
      j      Loop        # go to Loop

Exit:
```

# Loops

- while loop in C:

```
while (save[i]==k)
    i+=1;
```

- Assume: i and k are present in \$s3 and \$s5, base address of the array is in \$s6.
- **Step 1:** Check if save[i]==k.
  - Copy save[i] and K to register;
  - k is already in reg, use base address of save[] copy save[i]

```
sll    $t1,$s3,2    # reg $t1=4*i
add    $t1,$t1,$s6  # $t1= address of save[i]
lw     $t0,0($t1)   # reg t0=save[i]
bne    $t0, $s5, Exit # go to Exit if save[i]!=k
```

# Loops

- while loop in C:

```
while (save[i]==k)
    i+=1;
```

- Assume: i and k are present in \$s3 and \$s5, base address of the array is in \$s6.
- **Step 2:** if the condition is met, increment i, repeat condition check

```
Loop: sll    $t1,$s3,2    # reg $t1=4*i
      add    $t1,$t1,$s6  # $t1= address of save[i]
      lw     $t0,0($t1)   # reg t0=save[i]
      bne    $t0, $s5, Exit # go to Exit if save[i]!=k
      addi    $s3,$s3,1    # i=i+1
      j      Loop        # go to Loop

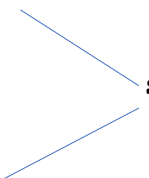
Exit:
```



# Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction: set on less than (slt)
- Compares two register data and sets (stores 1) a third register if first one is smaller

```
if $s3 < $s4 then  
    $t0 = 1  
else  
    $t0 = 0
```



**slt \$t0, \$s3, \$s4**

- Since its common to compare variables with constants, we have slti
  - `slti $t0,$s2,10`      # \$t0=1 if \$s2 < 10
- We don't have set on greater than (sgt), why?
  - Keeping the hardware simple, improves clock cycle speed

# Summary of instructions studied so far (p.77):

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; data in registers
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; data in registers
Data transfer	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Data from memory to register
	store word	sw \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Data from register to memory
Logical	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \mid \$s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \mid \$s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 \mid 100$	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
Conditional branch	branch on equal	beq \$s1,\$s2,L	if ( $\$s1 == \$s2$ ) go to L	Equal test and branch
	branch on not equal	bne \$s1,\$s2,L	if ( $\$s1 \neq \$s2$ ) go to L	Not equal test and branch
	set on less than	slt \$s1,\$s2,\$s3	if ( $\$s2 < \$s3$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compare less than; used with beq, bne
	set on less than immediate	slti \$s1,\$s2,100	if ( $\$s2 < 100$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compare less than immediate; used with beq, bne
Unconditional jump	jump	j L	go to L	Jump to target address

# Encoding

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
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
and	R	0	18	19	17	0	36	and \$s1,\$s2,\$s3
or	R	0	18	19	17	0	37	or \$s1,\$s2,\$s3
nor	R	0	18	19	17	0	39	nor \$s1,\$s2,\$s3
andi	I	12	18	17	100			andi \$s1,\$s2,100
ori	I	13	18	17	100			ori \$s1,\$s2,100
sll	R	0	0	18	17	10	0	sll \$s1,\$s2,10
srl	R	0	0	18	17	10	2	srl \$s1,\$s2,10
beq	I	4	17	18	25			beq \$s1,\$s2,100
bne	I	5	17	18	25			bne \$s1,\$s2,100
slt	R	0	18	19	17	0	42	slt \$s1,\$s2,\$s3
j	J	2	2500					j 10000 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, <a href="#">branch</a> format

# Next Class: Procedures

- A function commonly used in high level programs like C or Java
  - Makes code readable
  - Allows code reuse