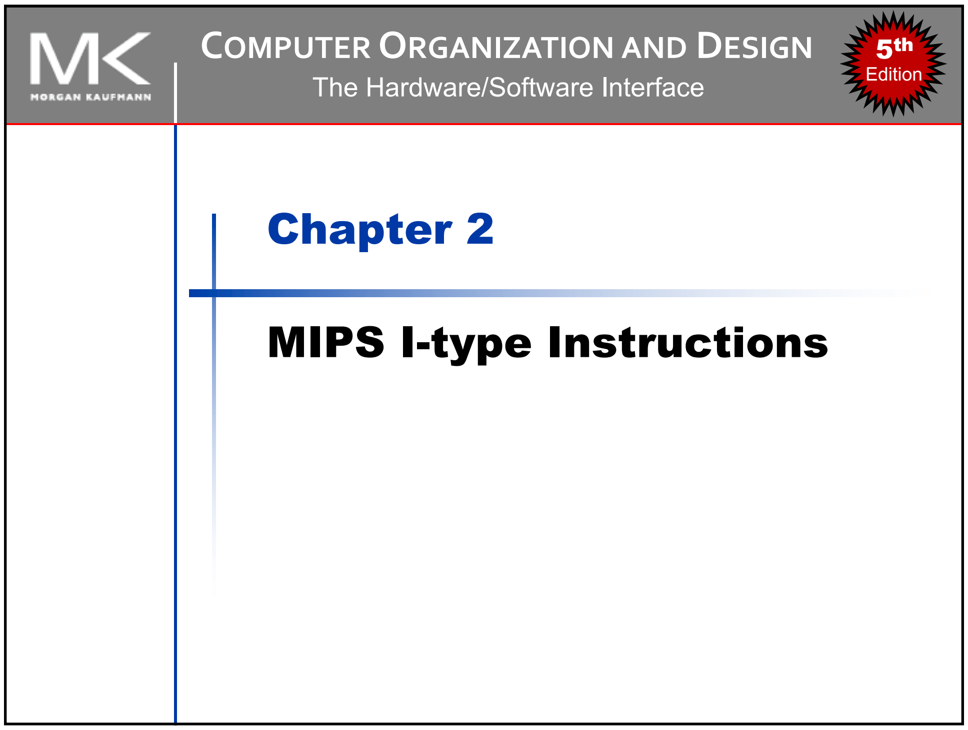


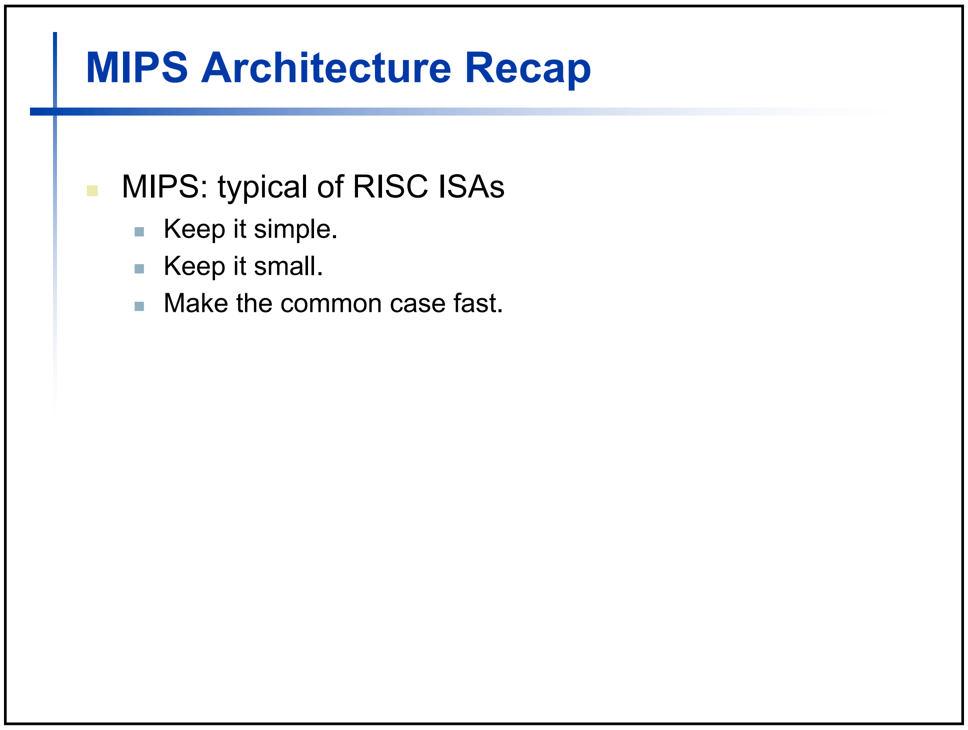
COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

5th Edition

Chapter 2

MIPS I-type Instructions

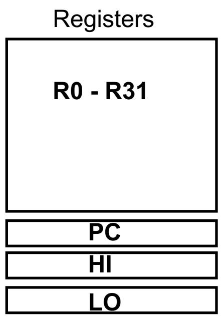


MIPS Architecture Recap

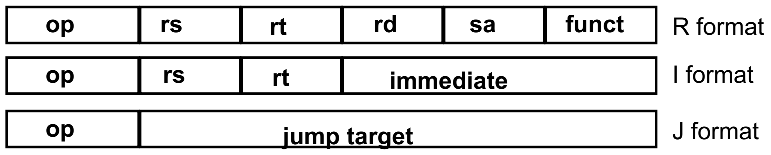
- MIPS: typical of RISC ISAs
  - Keep it simple.
  - Keep it small.
  - Make the common case fast.

MIPS-32 ISA

- Instruction Categories
  - Computational (R-type)
  - Load/Store
  - Jump and Branch
  - Floating Point



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
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

R-type Instruction Format

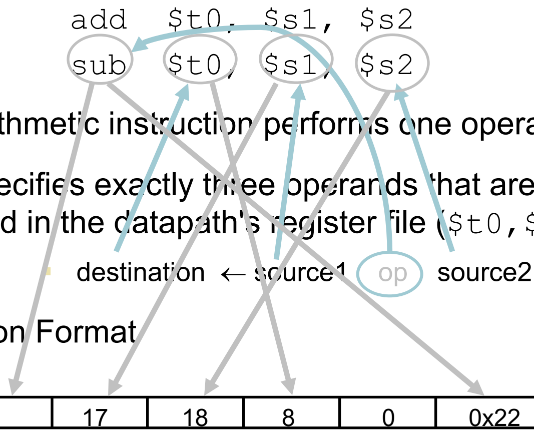
op	rs	rt	rd	shamt	funct

op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result’s destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

MIPS R-type Instructions

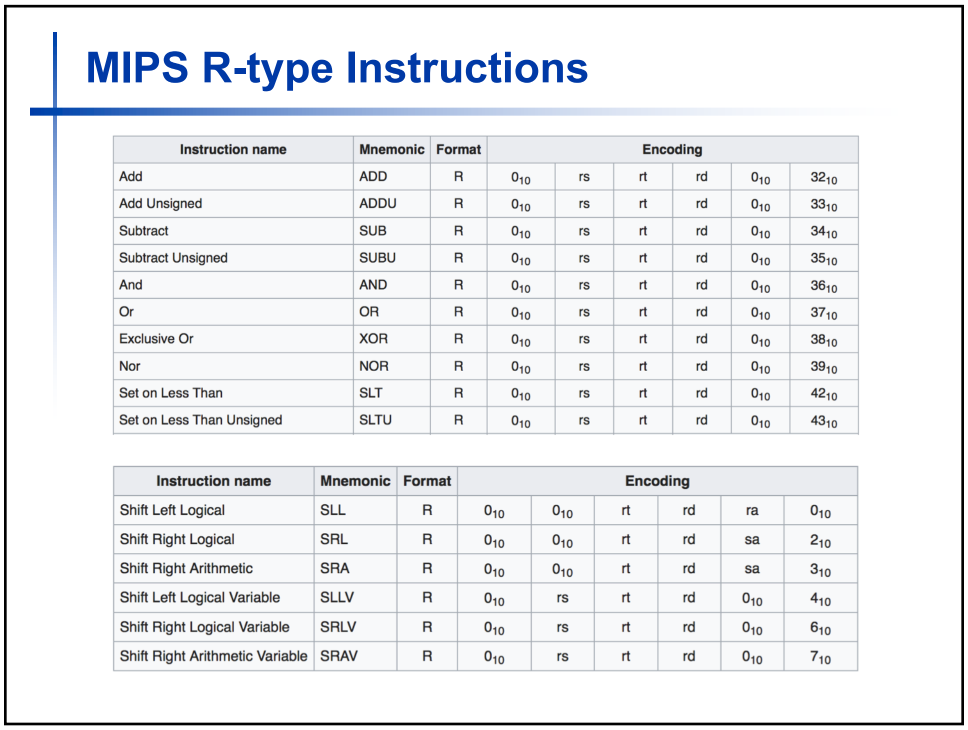
- MIPS assembly language arithmetic instructions:

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



- Each arithmetic instruction performs one operation.
- Each specifies exactly three operands that are all contained in the datapath's register file (`$t0,$s1,$s2`)
  - destination ← source1 op source2
- Instruction Format

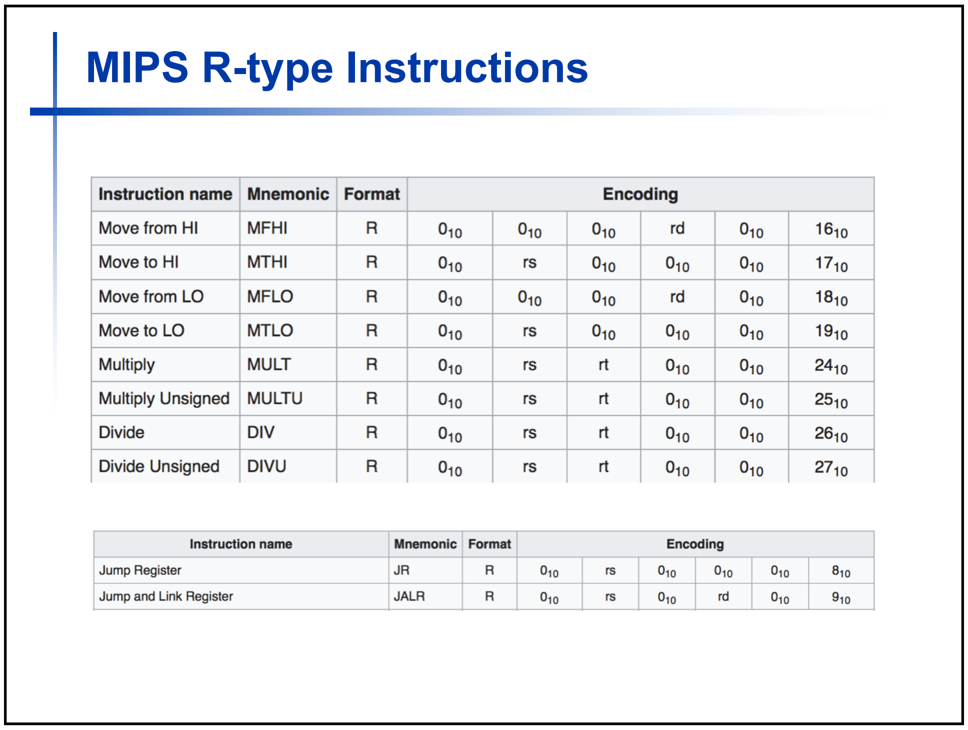
0	17	18	8	0	0x22



MIPS R-type Instructions

Instruction name	Mnemonic	Format	Encoding					
Add	ADD	R	0_{10}	rs	rt	rd	0_{10}	32_{10}
Add Unsigned	ADDU	R	0_{10}	rs	rt	rd	0_{10}	33_{10}
Subtract	SUB	R	0_{10}	rs	rt	rd	0_{10}	34_{10}
Subtract Unsigned	SUBU	R	0_{10}	rs	rt	rd	0_{10}	35_{10}
And	AND	R	0_{10}	rs	rt	rd	0_{10}	36_{10}
Or	OR	R	0_{10}	rs	rt	rd	0_{10}	37_{10}
Exclusive Or	XOR	R	0_{10}	rs	rt	rd	0_{10}	38_{10}
Nor	NOR	R	0_{10}	rs	rt	rd	0_{10}	39_{10}
Set on Less Than	SLT	R	0_{10}	rs	rt	rd	0_{10}	42_{10}
Set on Less Than Unsigned	SLTU	R	0_{10}	rs	rt	rd	0_{10}	43_{10}

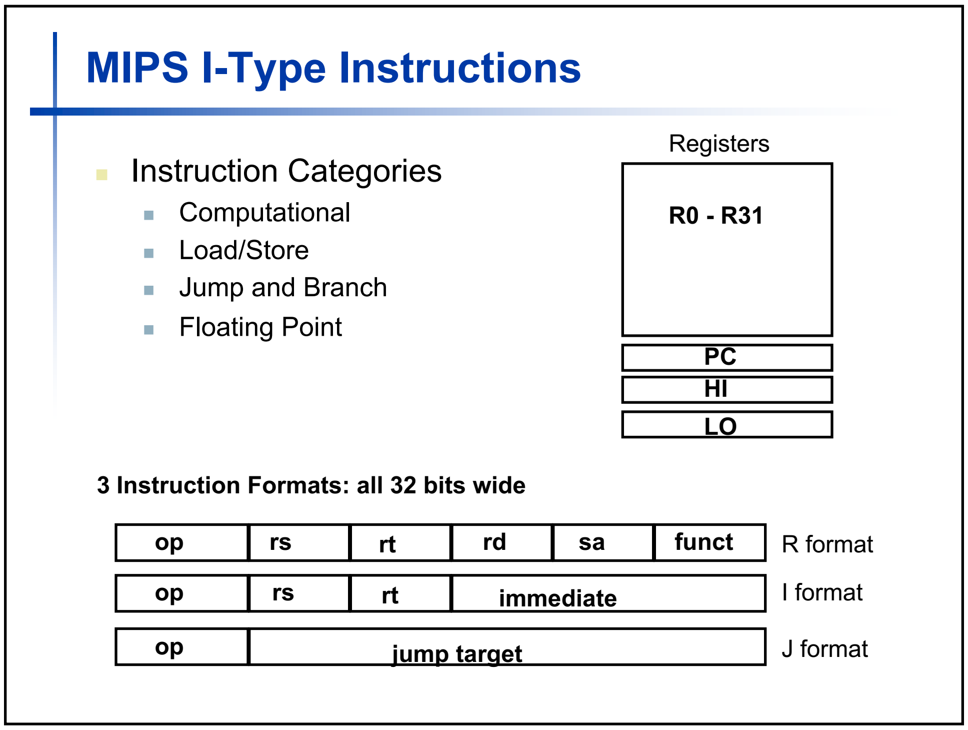
Instruction name	Mnemonic	Format	Encoding					
Shift Left Logical	SLL	R	0_{10}	0_{10}	rt	rd	ra	0_{10}
Shift Right Logical	SRL	R	0_{10}	0_{10}	rt	rd	sa	2_{10}
Shift Right Arithmetic	SRA	R	0_{10}	0_{10}	rt	rd	sa	3_{10}
Shift Left Logical Variable	SLLV	R	0_{10}	rs	rt	rd	0_{10}	4_{10}
Shift Right Logical Variable	SRLV	R	0_{10}	rs	rt	rd	0_{10}	6_{10}
Shift Right Arithmetic Variable	SRAV	R	0_{10}	rs	rt	rd	0_{10}	7_{10}



MIPS R-type Instructions

Instruction name	Mnemonic	Format	Encoding					
Move from HI	MFHI	R	0_{10}	0_{10}	0_{10}	rd	0_{10}	16_{10}
Move to HI	MTHI	R	0_{10}	rs	0_{10}	0_{10}	0_{10}	17_{10}
Move from LO	MFLO	R	0_{10}	0_{10}	0_{10}	rd	0_{10}	18_{10}
Move to LO	MTLO	R	0_{10}	rs	0_{10}	0_{10}	0_{10}	19_{10}
Multiply	MULT	R	0_{10}	rs	rt	0_{10}	0_{10}	24_{10}
Multiply Unsigned	MULTU	R	0_{10}	rs	rt	0_{10}	0_{10}	25_{10}
Divide	DIV	R	0_{10}	rs	rt	0_{10}	0_{10}	26_{10}
Divide Unsigned	DIVU	R	0_{10}	rs	rt	0_{10}	0_{10}	27_{10}

Instruction name	Mnemonic	Format	Encoding					
Jump Register	JR	R	0_{10}	rs	0_{10}	0_{10}	0_{10}	8_{10}
Jump and Link Register	JALR	R	0_{10}	rs	0_{10}	rd	0_{10}	9_{10}



MIPS I-Type Instructions

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point

Registers

R0 - R31

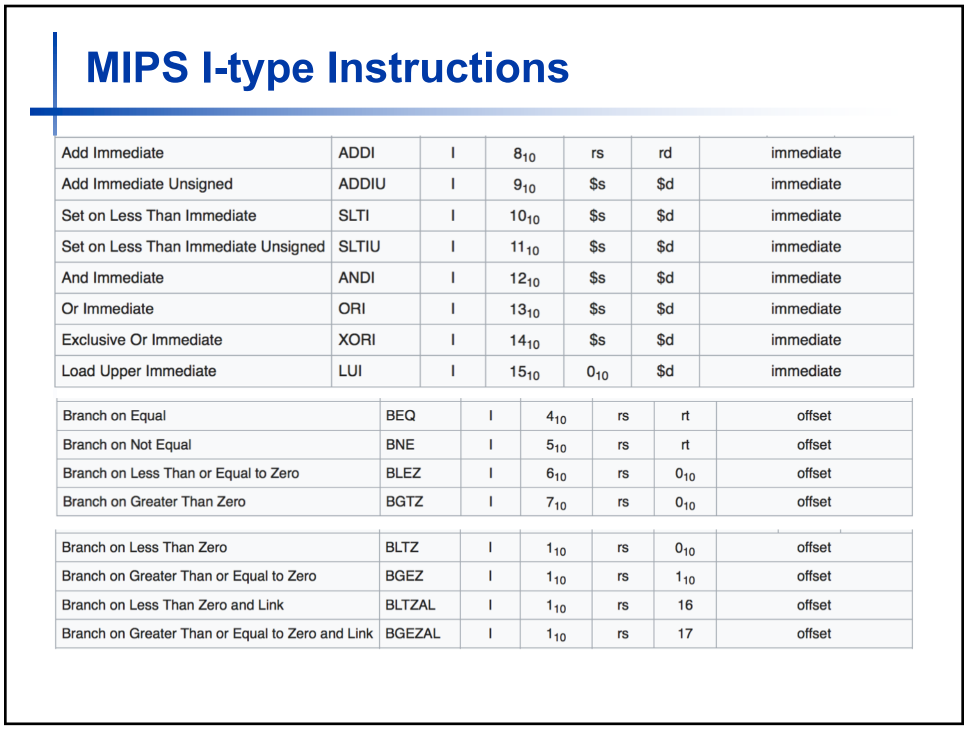
PC

HI

LO

3 Instruction Formats: all 32 bits wide

op	**rs**	**rt**	**rd**	**sa**	**funct**	R format
op	**rs**	**rt**	**immediate**			I format
op	**jump target**					J format



MIPS I-type Instructions

Add Immediate	ADDI	I	8_{10}	rs	rd	immediate
Add Immediate Unsigned	ADDIU	I	9_{10}	\$s	\$d	immediate
Set on Less Than Immediate	SLTI	I	10_{10}	\$s	\$d	immediate
Set on Less Than Immediate Unsigned	SLTIU	I	11_{10}	\$s	\$d	immediate
And Immediate	ANDI	I	12_{10}	\$s	\$d	immediate
Or Immediate	ORI	I	13_{10}	\$s	\$d	immediate
Exclusive Or Immediate	XORI	I	14_{10}	\$s	\$d	immediate
Load Upper Immediate	LUI	I	15_{10}	0_{10}	\$d	immediate

Branch on Equal	BEQ	I	4_{10}	rs	rt	offset
Branch on Not Equal	BNE	I	5_{10}	rs	rt	offset
Branch on Less Than or Equal to Zero	BLEZ	I	6_{10}	rs	0_{10}	offset
Branch on Greater Than Zero	BGTZ	I	7_{10}	rs	0_{10}	offset

Branch on Less Than Zero	BLTZ	I	1_{10}	rs	0_{10}	offset
Branch on Greater Than or Equal to Zero	BGEZ	I	1_{10}	rs	1_{10}	offset
Branch on Less Than Zero and Link	BLTZAL	I	1_{10}	rs	16	offset
Branch on Greater Than or Equal to Zero and Link	BGEZAL	I	1_{10}	rs	17	offset

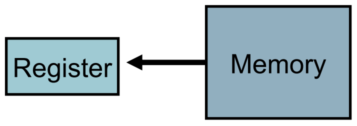
MIPS I-type Instructions

Instruction name	Mnemonic	Format	Encoding			
Load Byte	LB	I	32_{10}	rs	rt	offset
Load Halfword	LH	I	33_{10}	rs	rt	offset
Load Word Left	LWL	I	34_{10}	rs	rt	offset
Load Word	LW	I	35_{10}	rs	rt	offset
Load Byte Unsigned	LBU	I	36_{10}	rs	rt	offset
Load Halfword Unsigned	LHU	I	37_{10}	rs	rt	offset
Load Word Right	LWR	I	38_{10}	rs	rt	offset
Store Byte	SB	I	40_{10}	rs	rt	offset
Store Halfword	SH	I	41_{10}	rs	rt	offset
Store Word Left	SWL	I	42_{10}	rs	rt	offset
Store Word	SW	I	43_{10}	rs	rt	offset
Store Word Right	SWR	I	46_{10}	rs	rt	offset

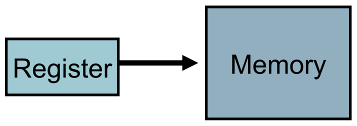
Memory Operands

- Values must be fetched from memory before instructions can operate on them.

Load Word
lw $t0, memory-address



Store Word
sw $t0, memory-address

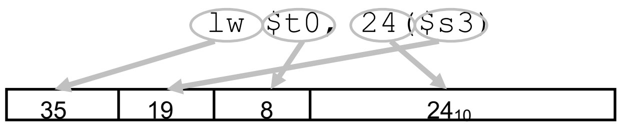


Deciphering the LW instruction

- lw Register1, Offset(Register2)
  - *Register1* – where the data from memory is placed.
  - The address of where the data resides in memory is calculated by adding the *offset* to the contents of *register2.*
  - The offset value is a 16-bit field, meaning access is limited to memory locations within a region of ± 2^{13} or 8,192 words (± 2^{15} or 32,768 bytes) of the address in the base register.
- The operation of the sw instruction is analogous.

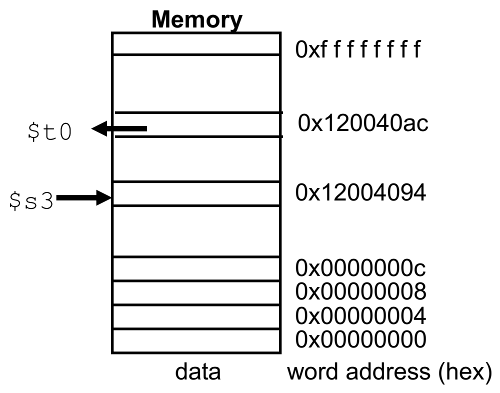
Machine Language - Load Instruction

- Load/Store Instruction Format (I format):



`24`$_{10}$ `+ $s3 =`

```
  . . . 0001 1000
+ . . . 1001 0100
  . . . 1010 1100 =
          0x120040ac
```



Example Code

C code: d[3] = d[2] + a;

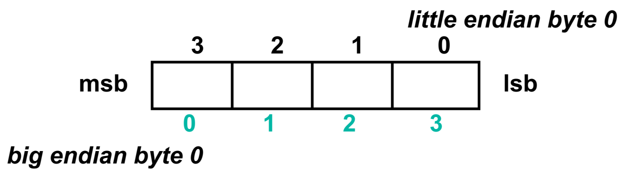
- Note that in MIPS assembly code, **$** is used to denote a register and **#** is used to denote a comment.
- Register $s4 contains the base address of the array **d**
- Variable **a** is stored in register $t1

Assembly: # implementation of C code

```
lw    $t0, 8($s4)    #  d[2] is brought into $t0
add   $t0, $t0, $t1  #  the sum is in $t0
sw    $t0, 12($s4)   #  $t0 is stored into d[3]
```

Byte Addresses

- Since 8-bit bytes are useful for many things, most architectures allow *byte-level* addressing:
  - MIPS Alignment restriction - the memory address of a word must be on natural word boundaries (a multiple of 4).
- **Big Endian:** Leftmost byte is least-significant
  - IBM 360/370, Motorola 68k, Sparc, HP PA
- **Little Endian**: Rightmost byte is least-significant
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)
- MIPS can actually be configured to work either way. The QTSpim simulator uses the byte ordering of the computer it is running on.



MIPS Immediate Instructions

- Small constants are used often in typical code.

```
addi  $sp, $sp, 4     #$sp = $sp + 4
slti $t0, $s2, 15     #$t0 = 1 if $s2<15
```

- Machine format (I format):

0x0A	18	8	0x0F

- The constant is kept inside the instruction itself
  - Immediate format limits values to the range +2^{15} – 1 to -2^{15}

What About Larger Constants?

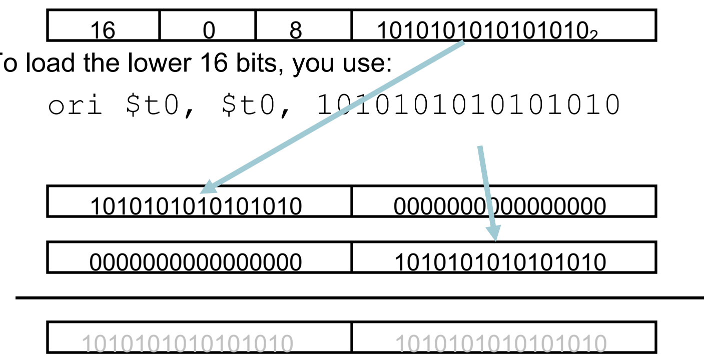
- Sometimes you need to load a 32-bit constant into a register. For this, you must use two instructions.
- The "load upper immediate" (lui) instruction loads the upper 16 bits:

```
lui $t0, 1010101010101010
```

16	0	8	1010101010101010_2

- To load the lower 16 bits, you use:

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



MIPS Shift Operations (R format)

- Shifts move all the bits in a word left or right

```
sll $t2, $s0, 8    #$t2 = $s0 << 8 bits
srl $t2, $s0, 8    #$t2 = $s0 >> 8 bits
```

- Instruction Format (**R** format)

0		16	10	8	0x00

- Such shifts are called logical because they fill with zeros
  - Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or 31 bit positions.

MIPS Logical Operations

- There are a number of *bit-wise* logical operations in the MIPS ISA:

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

or  $t0, $t1, $t2    #$t0 = $t1 | $t2

nor $t0, $t1, $t2    #$t0 = not($t1 | $t2)
```

- Instruction Format (**R** format)

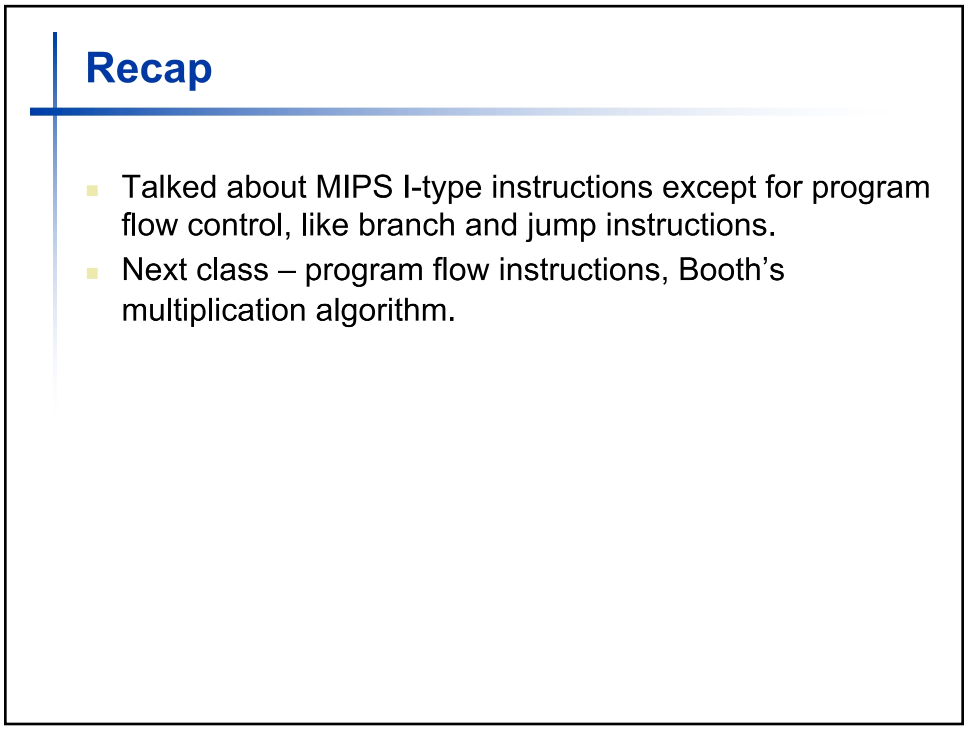
0	9	10	8	0	0x24

```
andi $t0, $t1, 0xFF00   #$t0 = $t1 & ff00

ori  $t0, $t1, 0xFF00   #$t0 = $t1 | ff00
```

- Instruction Format (**I** format)

0x0D	9	8	0xFF00



Recap

- Talked about MIPS I-type instructions except for program flow control, like branch and jump instructions.
- Next class – program flow instructions, Booth's multiplication algorithm.