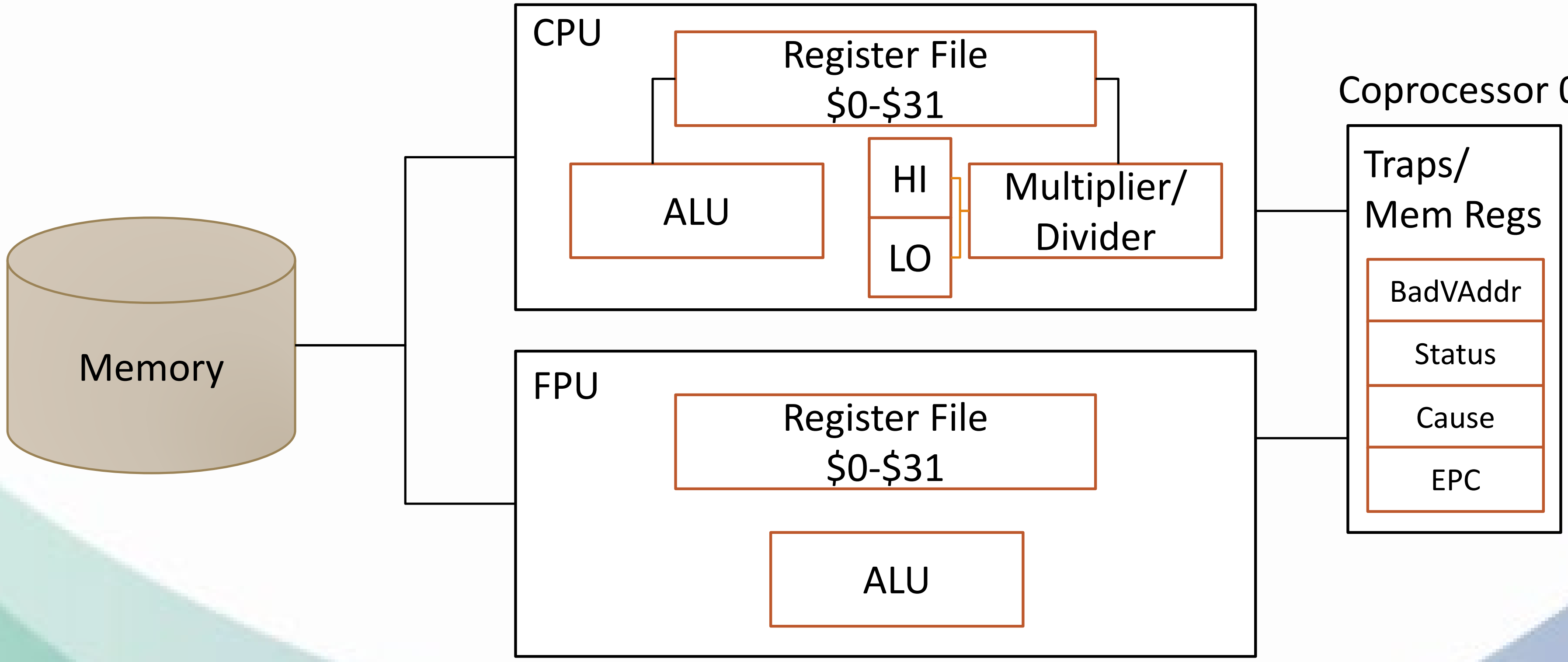
Computer Architecture and Organization

INSTRUCTOR: YAN-TSUNG PENG

DEPT. OF COMPUTER SCIENCE, NCCU

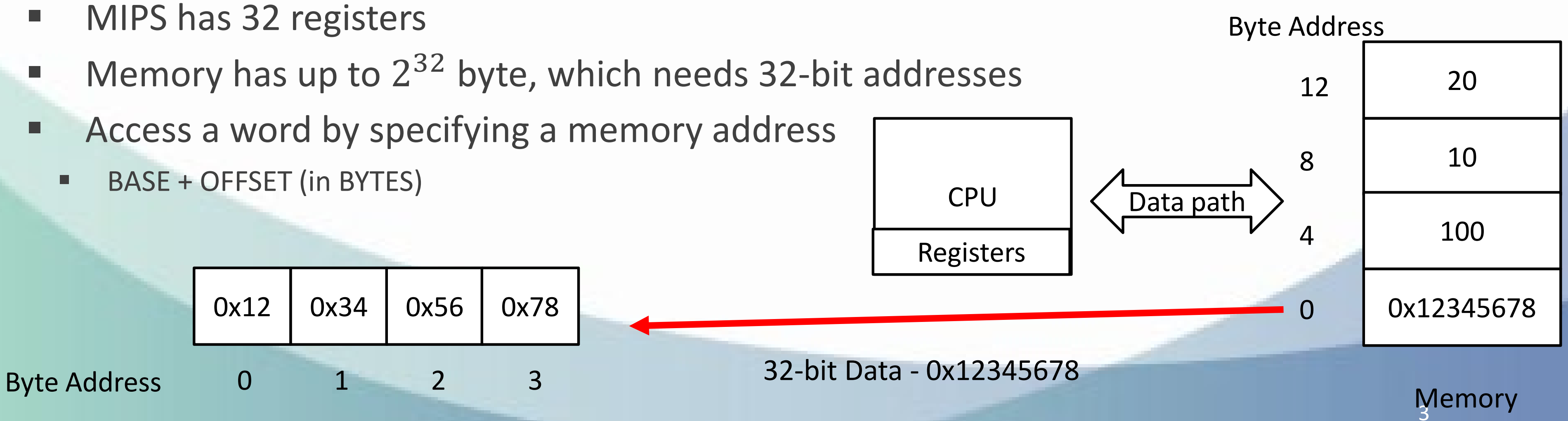
MIPS Processor Organization



MIPS R2000 CPU and FPU

Memory Operands

- MIPS must support - Transferring data between memory and registers
  - Data transfer instruction
  - Memory is byte addressed - Each address identifies an 8-bit byte
  - Addresses are multiples of 4 – Word (4 bytes) must be aligned in memory
- MIPS is Big Endian
  - Most-significant byte at least address of a word
- Memory to Register – Data Transfer
  - MIPS has 32 registers
  - Memory has up to 2^{32} byte, which needs 32-bit addresses
  - Access a word by specifying a memory address
    - BASE + OFFSET (in BYTES)



Example

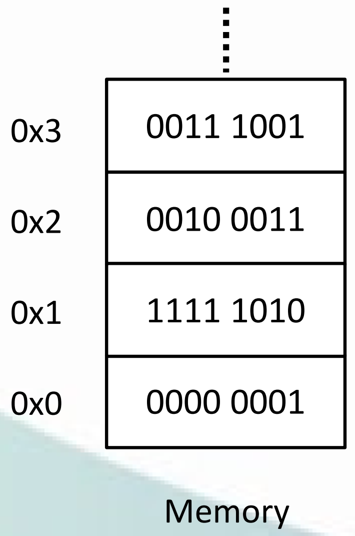
Little Endian

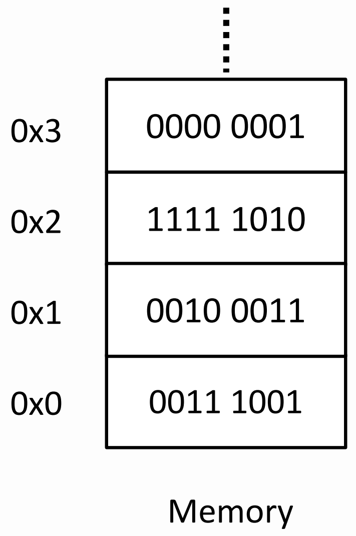
LSB at least address of a word

Big Endian

MSB at least address of a word

Ex: 0011 1001 0010 0011 1111 1010 0000 0001

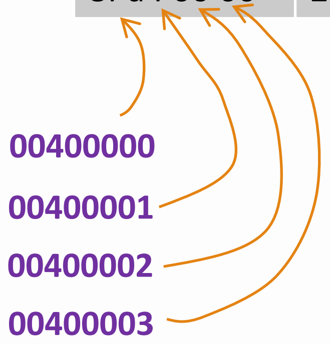




Memory Operands

Memory is byte addressed

00400000	00400004	00400008	0040000C
8f a4 00 00	27 a5 00 04	24 a6 00 04	00 04 10 80
	word	word	word



Words are aligned in memory

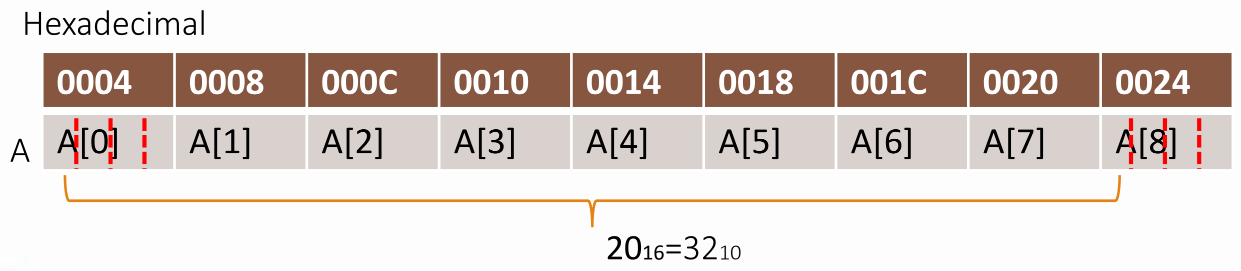
Memory Alignment

- MIPS requires that all words are aligned, meaning a word is stored at an address multiples of 4 bytes

0004	0008	000C	0010	0014	0018	001C	0020	0024
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

Access Specific Words from Memory

- Example:
  - Load A[8] from memory
  - First, you have to know what base address is (&A[0] or A)
  - Second, according to the index, calculating the byte offset for A[8]
    - 1 word = 4 bytes
    - &A[8] = &A[0] + 8 words = &A[0] + 8*4 bytes
    - 32(&A[0])



Load a Word in Memory: lw

- C code:

  g = h + A[8];

- g in $s1, h in $s2, base address of A in $s3
- MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

Base addr

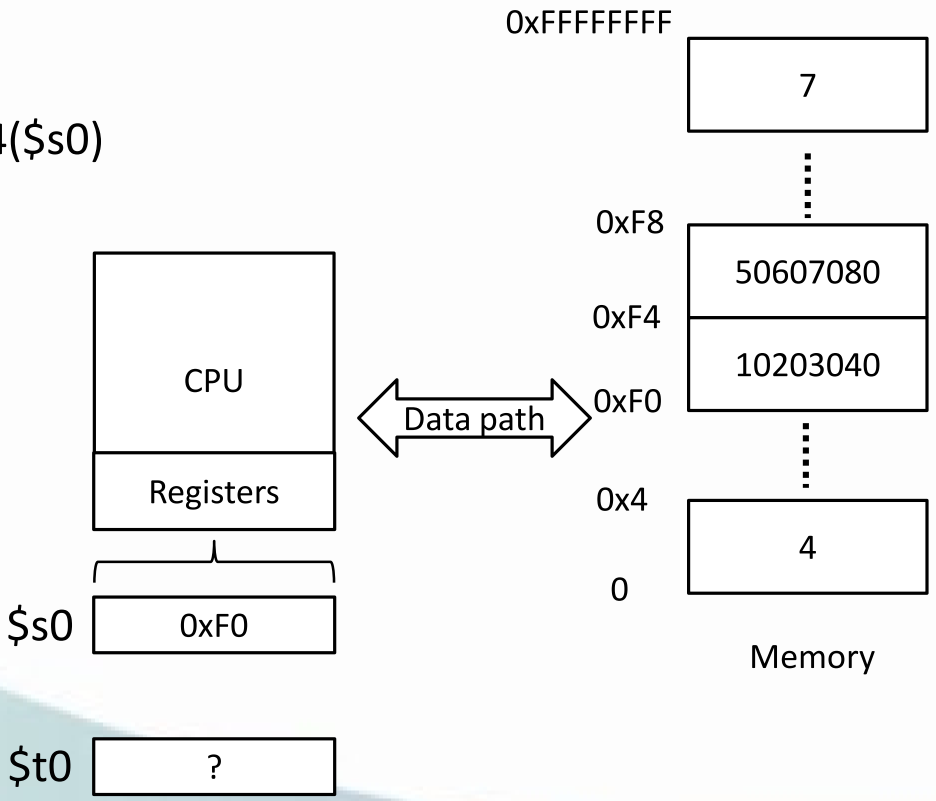
```
lw  $t0, 32($s3)    # load word - $t0 gets A[8]
```

offset

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

Example

lw $t0, 4($s0)



Load a Byte in Memory: lb

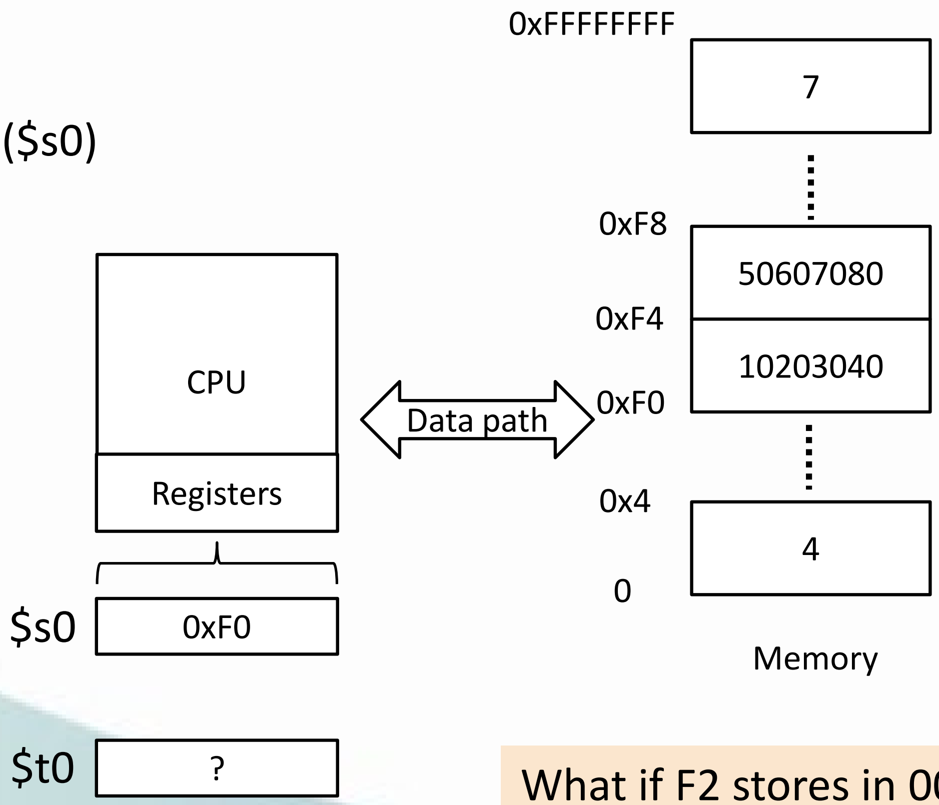
- Load Instruction Syntax:

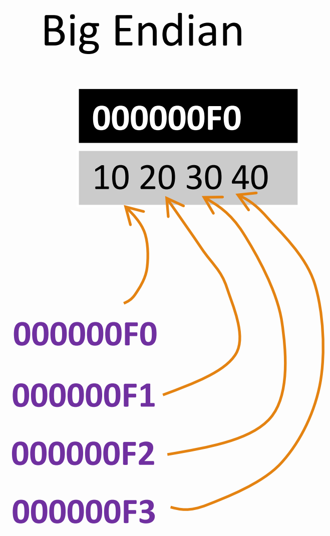
  ```
  lb $t0,1($s0)
  ```

  - lb (Load Byte, load a byte in $t0)
  - $s0 stores the base address, 1 byte is added to which, and then load the value from the memory address ($s0+1) into `$t0`
  - Sign-extended to 32 bits
    - lbu $t0,1($s0)   #Zeros filled to 32 bits
- Notes:
  - base register: `$s0`
  - offset : *1*
  - Like an array, where the offset works as the index, and the base register stores the address pointing to the beginning of the array

Example

lb $t0, 1($s0)





What if F2 stores in 000000F1??

Sign-extended

Store a Word in Memory: sw

- C code:

  A[12] = h + A[8];

- h in $s2, base address of A in $s3
- MIPS code:

```
lw  $t0, 32($s3)    # load word  -  $t0 gets A[8]
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word  - $t0 is stored in A[12]
```

Store a Byte in Memory: sb

- C code:

```
    k = (A[0]&0x000000ff);
A[1] = (A[1] &0xffffff00) | k;
```

- k in $t0, base address of A in $s3
- MIPS code:

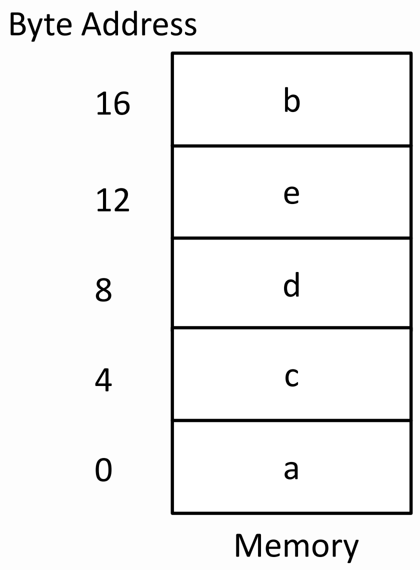
```
lb  $t0, 3($s3)    # load byte  -  $t0 gets the lsb of A[0]
sb  $t0, 7($s3)    # store byte  - $t0 is stored in lsb of A[1]
```

Example

- Swap

C Code

```
void swap( int & A[0], int & A[4] )
{
  register int tmp;
  tmp = A[0];
  A[0] = A[4];
  A[4] = tmp;
}
```



base address of A in $s0

```
lw  $t1 0($s0)      #load A[0] in $t1
lw  $t2 16($s0)     #load A[4] in $t2
sw  $t1 16($s0)     #store $t1 in A[4]
sw  $t2 0($s0)      #store $t2 in A[0]
```

Registers vs. Memory

- MIPS has 32 registers
  - Design Principle 2: Smaller is faster
    - Having too many registers may increase the clock cycle time
- Accessing data in registers is faster than that in memory
- Accessing data in memory requires extra instructions (lw and sw)
- Using registers as much as possible
  - only store less frequently used data in memory

Immediate Operands

- Use constant data

```
addi $s3, $s3, 4    #$s3=$s3+4
```

- No subtract immediate instruction

```
addi $s2, $s1, -1
```

- <u>Design Principle 3</u>: Make the common case fast
  - Small constants are common
  - Using immediate operands avoids a load instruction

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

    ```
    add $t2, $s1, $zero
    ```

Instructions: Signed and Unsigned Numbers

- $1 \times 2^1 = 2$
  - 0000000$\underline{1}$*2^1=000000$\underline{1}$0
- $1 \times 2^2 = 4$
  - 0000000$\underline{1}$*2^2=00000$\underline{1}$00
- $1 \times 2^3 = 8$
  - 0000000$\underline{1}$*2^3=0000$\underline{1}$000
- $1 \times 2^4 = 16$
  - 0000000$\underline{1}$*2^4=000$\underline{1}$0000
- $1 \times 2^5 = 32$
  - 0000000$\underline{1}$*2^5=?

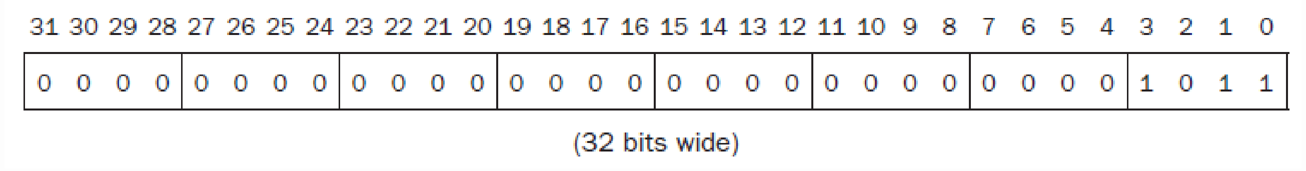
$$11011_{ten} = 1\times10^4 + 1\times10^3 + 0\times10^2 + 1\times10^1 + 1\times10^0$$

$$11011_{two} = 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$$

The value of *ith* digit *d* = $d \times \text{Base}^i$

Example

most significant bit — least significant bit



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)

$$0 \times 2^{31} + \ldots. + 1 \times 2^{3} + 0 \times 2^{2} + 1 \times 2^{1} + 1 \times 2^{0}$$

$$= 2^{3} + 2^{1} + 2^{0} = 11_{ten}$$

The maximal value of an unsigned 8-bit binary number? $\sum_{i=0}^{7} 2^{i} = 2^{8} - 1$

Signed Binary Number

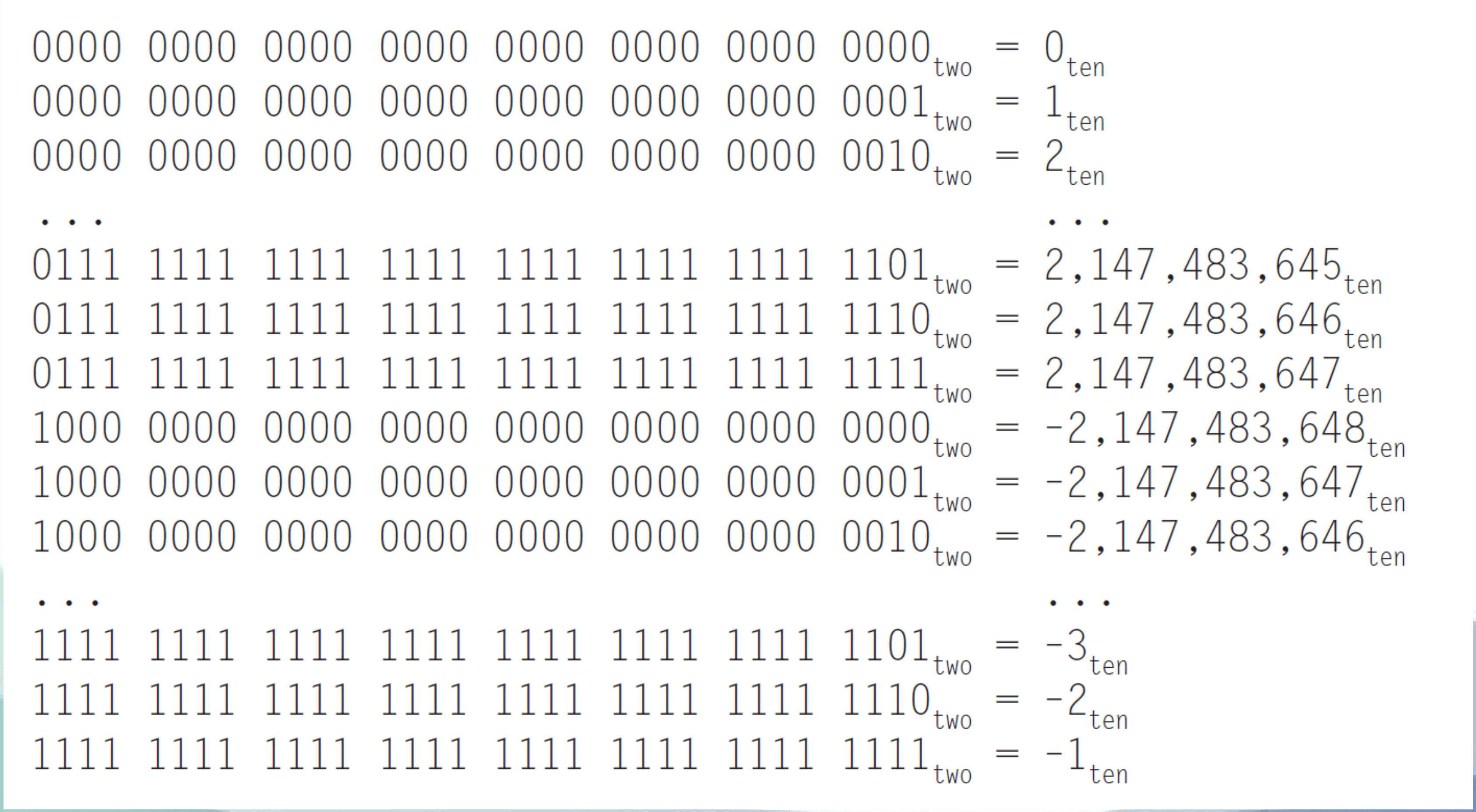
- Using the first bit to represent the sign of the number

***Two's complement* representation**

Sign bit

	Sign bit		
Positive	0	$111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$	=Maximum
	0	$000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$	=0
Negative	1	$111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$	=-1
	1	$000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$	=Minimum

Two's Complement Representation



$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$

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

$\ldots \qquad \ldots$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$

$\ldots \qquad \ldots$

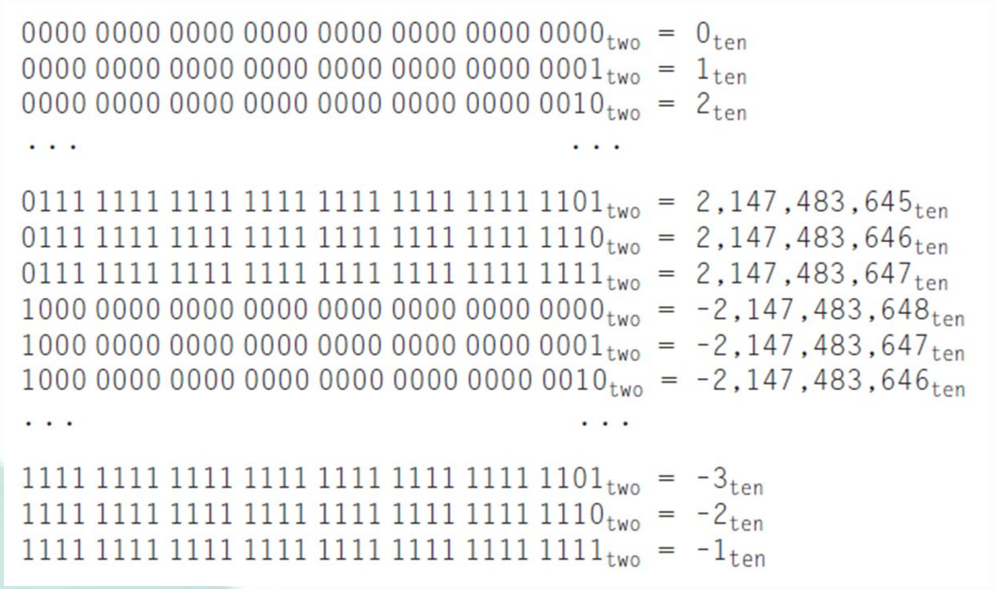
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$

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

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

Signed Negation

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



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

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

if x = 2, meaning x=0010

0010 + 1101 = 1111 <- -1

⬇

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

Take 4-bit binary number as an example

-1 -> $\overline{1}$ + 1-> 1110 + 1 = 1111

More examples

- 4_{10} = 0100
- What is -4 ?

Sign Extension

- Replicate the sign bit to the left
  - unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0 000 0010 => 0000 0000 0000 0010
  - –2: 1 111 1110 => 1111 1111 1111 1110

Hexadecimal (Base_{16})

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

Base_{10}	Base_{16}	Base_{2}
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

MIPS Instruction Format

- Stored-program Computer Concept
  - **Instructions and data** are both stored in the memory
  - Each instruction takes 32 bits
- For a 32-bit instruction
  - Divide 32 bits into different “fields.”
  - CPU decode each field to know how to execute the instruction
- MIPS defines three types of instruction formats
  - R-type (R-format) – R for registers
  - I-type (I-format) – I for immediate
  - J-type (J-format) – J for jump

R-type Instruction

- R-type Instruction
  - MIPS Fields

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 (for shift instructions)
  - funct: function code
    - combined with `opcode` to define different functions
    - Why do we need it? **Why not using a 12-bit opcode**?

https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf

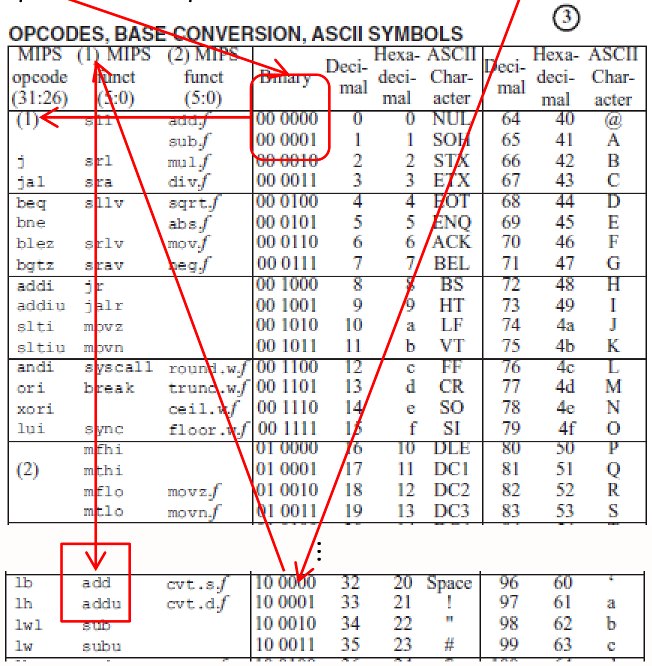
R-type Instruction

- R-type Instruction
  - MIPS Fields

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

  - Since MIPS only has 32 registers, each register takes 5 bits
  - For a shift instruction
    - `shamt`: contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits
    - `shamt` only works for shift instruction. For all the other instructions, it is set to 0.

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
000000	10001	10010	01000	00000	100000
	R17	R18	R8		
	$s1	$s2	$t0		



OPCODES, BASE CONVERSION, ASCII SYMBOLS ③

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII Char-acter	Deci-mal	Hexa-deci-mal	ASCII Char-acter
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.*f*	00 0100	4	4	EOT	68	44	D
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.*f*	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.*f*	00 0111	7	7	BEL	71	47	G
addi	jr		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.*f*	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.*f*	00 1101	13	d	CR	77	4d	M
xori		ceil.w.*f*	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.*f*	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movz.*f*	01 0010	18	12	DC2	82	52	R
	mtlo	movn.*f*	01 0011	19	13	DC3	83	53	S
⋮									
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	`
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c

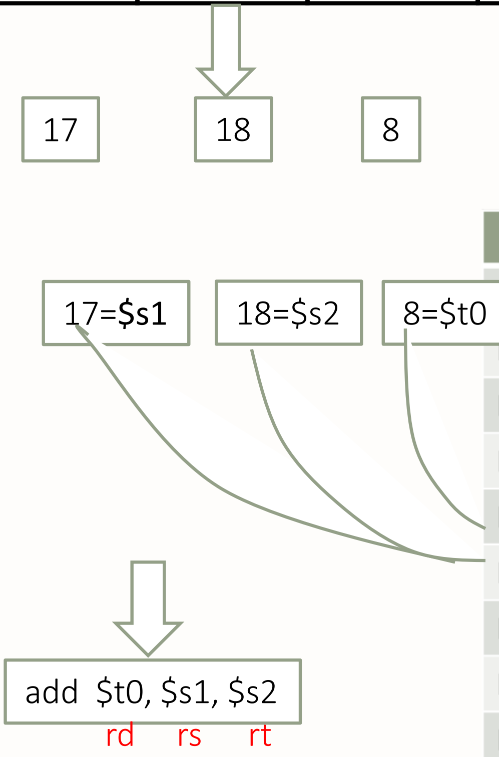
Interpreting the machine code to know this is the "add" instruction

add $t0, $s1, $s2   # $t0 = $s1 + $s2

Register	MIPS register name
R0	$zero
R1	$at
R2-R3	$v0-$v1
R4-R7	$a0-$a3
R8-R15	$t0-$t7
R16-R23	$s0-$s7
R24-R25	$t8-$t9
R26-R27	$k0-$k1
R28	$gp
R29	$sp
R30	$fp
R31	$ra

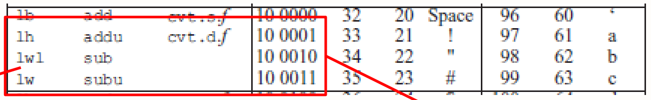
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
000000	10001	10010	01000	00000	100000

add



Register	MIPS register name
R0	$zero
R1	$at
R2-R3	$v0-$v1
R4-R7	$a0-$a3
R8-R15	$t0-$t7
R16-R23	$s0-$s7
R24-R25	$t8-$t9
R26-R27	$k0-$k1
R28	$gp
R29	$sp
R30	$fp
R31	$ra

$$\overset{}{\texttt{sub}}\ \overset{rd}{\texttt{\$t1}},\ \overset{rs}{\texttt{\$s3}},\ \overset{rt}{\texttt{\$s4}}$$



lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	`
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c

Place funct (sub)

000000	rs	rt	rd	shamt	100010
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Reset shamt

000000	rs	rt	rd	00000	100010
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

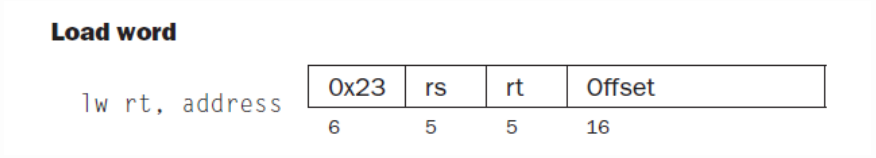
Find register code and make it binary

$t1=R9, $s3=R19, $s4=R20

R9=01001, R19=10011, R20=10100

000000	10011	10100	01001	00000	100010
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-format Instructions



Load word

```
lw rt, address
```

0x23	rs	rt	Offset
6	5	5	16

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

#base_reg

lw: #reg to hold the loaded value
sw: write #reg's value to memory

- For "lw" and "sw"
  - rt: lw: load data in rt
  - sw: write data from rt
- 16-bit constant: signed offset added to <u>base address stored in rs</u>

ex: lw $t0, 32($s3) #$t0 ← A[8], $s3:A[0] base address (constant: 32)
rt rs

I-format (I-type) Instructions

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

- ❑ When an instruction needs longer fields than R-format instructions
  - ❑ The 5-bit field is too small to be useful for representing constants
  - ❑ 16-bit signed constant or address: -2^{15} to $+2^{15} - 1$ (32, 768 bytes of the address in the base register)
  - ❑ ex: addi \$s1, \$s2, 120 #\$s1 $\leftarrow$ \$s2 +120

    rt: s1, rs:s2, constant: 120

Design Principle 3: Good design demands good compromises.

- Uniformity of 32-bit instructions
  - First 16 bits in R-type Instructions are the same as those in I-type ones
- I-type instructions include lw/sw and immediate instructions (addi)
- Keep formats as similar as possible

Example

- Translating MIPS code into machine language

$t1 : base address of A

$s2 : h

A[300] = h + A[300]

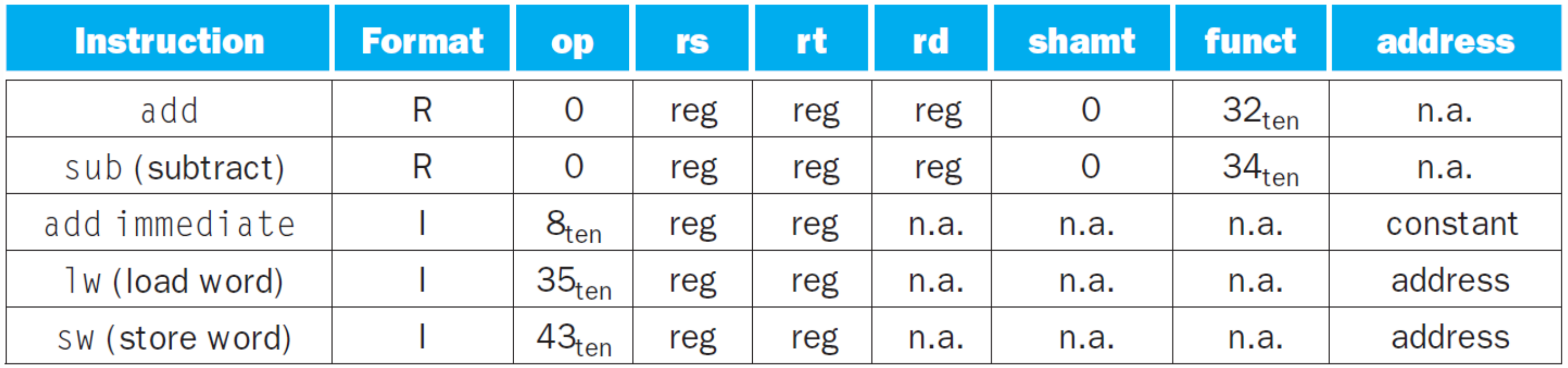
```
lw   $t0, 1200($t1)
add  $t0, $s2, $t0
sw   $t0, 1200($t1)
```

Register	MIPS register name
R0	$zero
R1	$at
R2-R3	$v0-$v1
R4-R7	$a0-$a3
R8-R15	$t0-$t7
R16-R23	$s0-$s7
R24-R25	$t8-$t9
R26-R27	$k0-$k1
R28	$gp
R29	$sp
R30	$fp
R31	$ra

Register	MIPS Reg	Code
R8	$t0	01000
R9	$t1	01001
R18	$s2	10010

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

R-format and I-format Instructions



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

Table is from "Computer Organization and Design, 5th version, p. 84

Logical Operations

- Instructions for bitwise manipulation

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

(singed shift right: sra)

Shift Operations (R-type Instruction)

Why shamt takes only 5 bits?

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

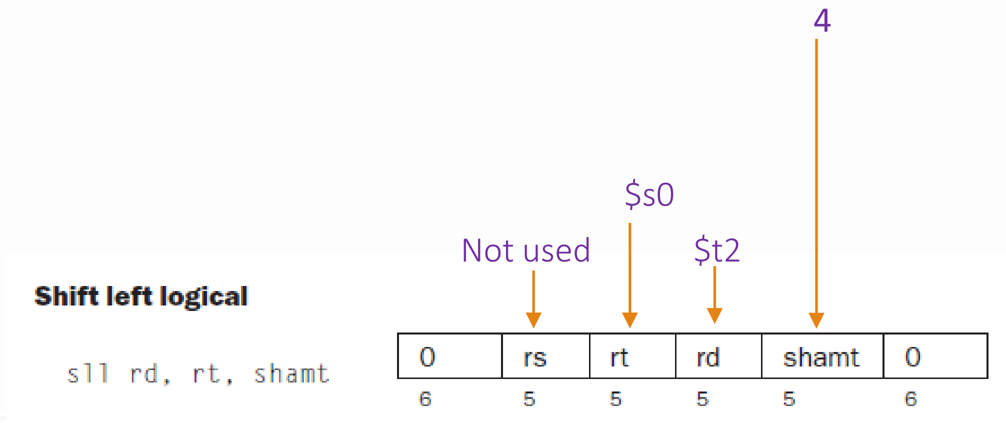
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits == multiplies by 2^i (useful when calculating byte offset for words)

- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits == divides by 2^i (unsigned only)

Shift Operations

- shift instructions:
  - `sll` (shift left logical): shifts left, filler: 0s
  - `srl` (shift right logical): shifts right, filler: 0s
  - `sra` (shift right arithmetic): shifts right, empties is filled with sign extension

sll $t2, $t0, 4 # $t2 = $s0 << 4



Shift Operations (logical)

`sll` shift left by 4 bits

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

$$9 \times 2^4 = 144$$

shift by 4 bits

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

Fill with 0s

`srl` shift right by 2 bits

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

shift by 2 bits

$$\boxed{0}000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$$

Fill with 0s

Shift Operations (Arithmetic)

`sra: shift 1 bit right (a`rithmetic)
Keep the sign bit the same

$$0100 \rightarrow 0010$$

$$4 \rightarrow 2$$

$$1100 \rightarrow 1110$$

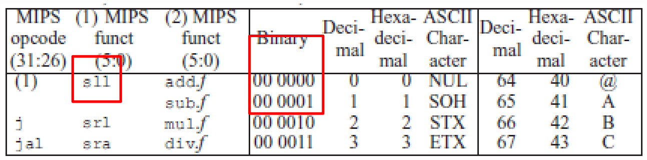
$$-4 \rightarrow -2$$

Shift Left Logical

Register	MIPS register name
R0	$zero
R1	$at
R2-R3	$v0-$v1
R4-R7	$a0-$a3
R8-R15	$t0-$t7
R16-R23	$s0-$s7
R24-R25	$t8-$t9
R26-R27	$k0-$k1
R28	$gp
R29	$sp
R30	$fp
R31	$ra

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

sll $t2, $s0, 4



MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII Char-acter	Deci-mal	Hexa-deci-mal	ASCII Char-acter
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C

000000	00000	10000	01010	00100	000000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Bitwise AND Operation

```
           rd      rs      rt
and $t0, $t1, $t2      #$t0 = $t1 & $t2
        result  filter
                (mask)
```

$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

Example

```
and $t0, $t1, $t2
```

R8=01000

Register	MIPS register name
R0	$zero
R1	$at
R2-R3	$v0-$v1
R4-R7	$a0-$a3
R8-R15	$t0-$t7
R16-R23	$s0-$s7
R24-R25	$t8-$t9
R26-R27	$k0-$k1
R28	$gp
R29	$sp
R30	$fp
R31	$ra

AND

100100

```
and rd, rs, rt
```

0	rs	rt	rd	0	0x24
6	5	5	5	5	6

OR Operations

```
or $t0, $t1, $t2
```

$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

NOT Operations

- Inverting bits in a word (0->1, 1->0)
- MIPS has NOR 3-operand instruction

How to implement NOT using NOR

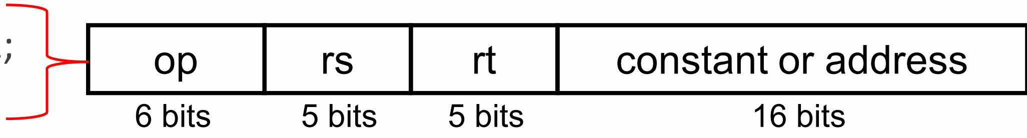
NOR		
A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

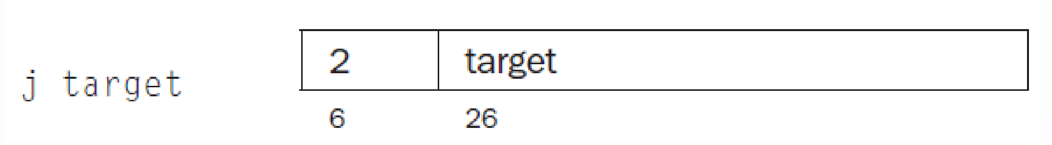
```
nor $t0, $t1, $zero
```

```
$t1   0000 0000 0000 0000 0011 1100 0000 0000
$t0   1111  1111  1111 1111  1100 0011 1111 1111
```

Conditional/Unconditional Operations

- Branch to a labeled instruction if true
  - Otherwise, continue sequentially
- **`beq rs, rt, L1`**
  - if (rs == rt) branch to instruction labeled L1;
- **`bne rs, rt, L1`**
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - jump to instruction labeled L1
  - goto L1





J-format instruction

Compiling If Statements (Equal)

- C code:

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

- Compiled MIPS code:

```
        bne $s3, $s4, Exit
        add $s0, $s1, $s2
Exit: …
```

VAR	REG
f	$s0
g	$s1
h	$s2
i	$s3
j	$s4

Compiling If/Else Statements

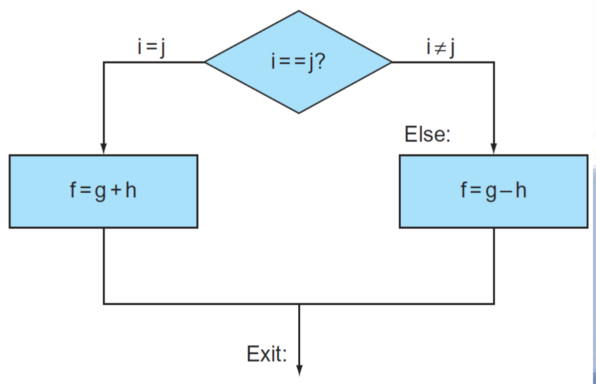
- C code:

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

VAR	REG
f	$s0
g	$s1
h	$s2
i	$s3
j	$s4

- Compiled MIPS code:

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



Practice

```
main:
   li $s3, 1
   li $s4, 10
   li $s0, 0
Loop:
   beq  $s3, $s4, Exit
   add  $s0, $s0, $s3
   addi $s3, $s3, 1
   j Loop
Exit:
   jr $ra
```

li: load immediate

li $t0, 1 ⟶ addi $t0, $zero, 1

Translate it into C code

It seems not like what we would write…

Conditional Operations: Less than

- Set result to 1 if the condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
       slt $t0, $s1, $s2     # if ($s1 < $s2) t0=1
       beq $t0, $zero, ELSE  # branch to ELSE if t0==0
       …
ELSE:
```

Observation:
Using "less than" will have one more instruction than using "equal to"

Less than and Branch Equal

- In C language, it is intuitive to write if(a<b) … else … . Why don't we have `blt` (branch less than) or `ble`(branch less than or equal to) instead?
- branch less than basically combine a comparing instruction with a branch one, involving more work per instruction and thus more clocks
  - All other instructions would be slowed down.
- ***Good design demands good compromises***

Compiling If/Else Statements (less than)

- C code:

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

VAR	REG
f	$s0
g	$s1
h	$s2
i	$s3
j	$s4

- Compiled MIPS code:

```
      slt $t0, $s3, $s4     #if($s3<$s4) $t0=1 else $t0=0
      beq $t0, $zero, Else  #$if($t0==0) goto Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

Loop Statements: while

- C code:

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

VAR	REG
i	$s3
k	$s5
save	$s6

while

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
  Exit: …
```

Loop Statements: for

- Code

  for (i=1, i <=10; i++)

  f=g+h

VAR	REG
f	$s0
g	$s1
h	$s2
i	$s3

- Compiled MIPS code:

```
       addi $s3, $zero, 1      # $s3=1
loop: slti $t0, $s3, 11       # if($s3<11) $t0=1 else $t0=0
      beq  $t0, $zero, Exit   # if($t0==0) goto Exit
      add  $s0, $s1, $s2      # $s0=$s1+$s2
      addi $s3, $s3, 1        # $s3=$s3+1
      j    loop               # goto loop
Exit: …
```