

MIPS R-format Instructions

Arithmetic (integer) Instructions:

ADD and ADDU

SUB and SUBU

MUL, DIV (will discuss after Exam 1)

Shift Instructions:

SLL and SRL

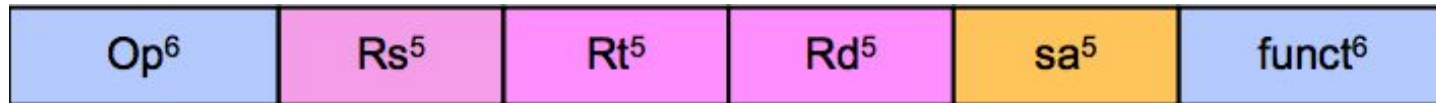
Logic Bitwise Instructions:

AND

OR

XOR

NOR



Integer add and subtract

- ADD and SUB cause an exception upon overflow
- ADDU and SUBU (U for unsigned) will ignore overflow
- An overflow is a condition that can happen when a calculation produces a result that is greater in magnitude than the storage location can hold
- Two kinds of overflow:
 - A carry out of the storage unit
 - A carry into the MSB so that the result does not have the sign we expect
- Overflow is a common cause of program bugs

Carry and Overflow

The term carry refers to unsigned operations. There has been a carry out of the storage unit.

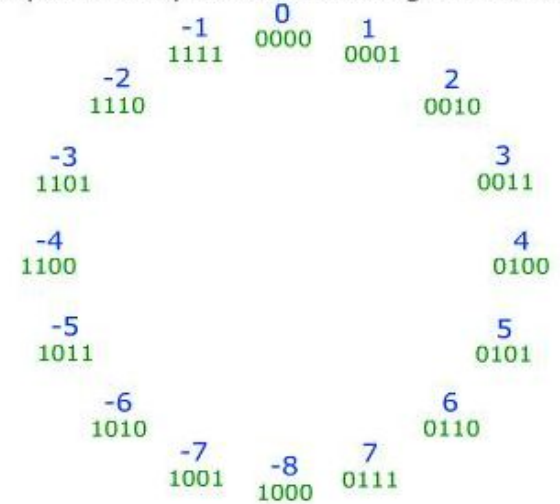
The term overflow refers to signed operations. There has been a carry into the sign bit. The result does not have the sign we expect.

Note:

- when adding numbers of opposite sign, you cannot have an overflow
- when subtracting numbers of the same sign, you cannot have an overflow

4-bit world

Two's Complement representation using 4 bit binary strings



Let's imagine 4-bit operands with bit 3 indicating the sign.

- Add $0111 + 0111$.
 - If the operands are unsigned, do we have carry, overflow, or neither?
 - If the operands are signed?
- Add $1111 + 0001$
 - If the operands are unsigned, do we have carry, overflow, or neither?
 - If the operands are signed?

Overflow Example

- Example: 1996 Ariane 5 Rocket (unmanned)
- link: https://www.youtube.com/watch?v=gp_D8r-2hwk

- Explanation by SE Professor Ian Sommerville
- link: <https://www.youtube.com/watch?v=W3YJeoYgozw>

Dealing with Overflow

MIPS provides signed and unsigned versions of ADD and SUB

ADDU and SUBU will ignore overflow

Some languages (ex: C) ignore overflow, so a MIPS compiler will use ADDU, SUBU, etc.

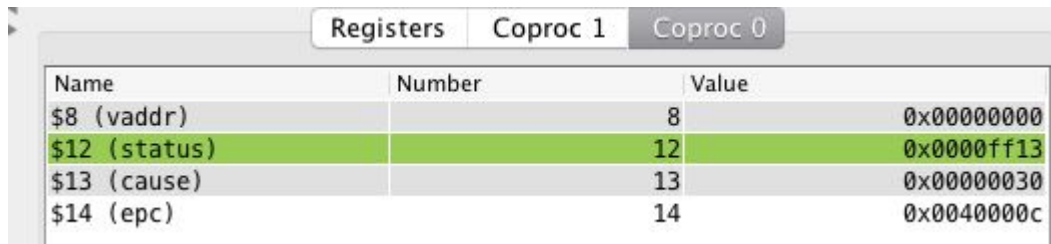
Other languages require raising an exception, so for them the MIPS compiler will use ADD, SUB, etc.

Demo: overflow exception

The add instruction triggered an exception,
which is handled by coprocessor 0

Changing the add to addu will not trigger an
exception, instead the result in \$t3 will be
0x80000000

```
li    $t0, 0x7fffffff # max positive integer
li    $t1, 1
add   $t3, $t0, $t1
```



The image shows a screenshot of a debugger's 'Registers' window. The 'Coproc 0' tab is selected. The window displays a table of registers with their names, numbers, and values. The registers \$12 (status), \$13 (cause), and \$14 (epc) are highlighted in green.

Name	Number	Value
\$8 (vaddr)	8	0x00000000
\$12 (status)	12	0x0000ff13
\$13 (cause)	13	0x00000030
\$14 (epc)	14	0x0040000c

Trigger or ignore overflow

Implement the following C expression:

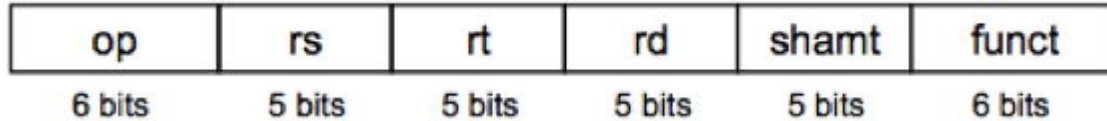
- (a) Ignoring overflow (use `subu`, `addu`)
- (b) Triggering overflow (shown)

$$g = (a - b) + (c - d)$$

```
1 # triggering or ignoring overflow
2 #
3
4 .data
5 a:    .word 5
6 b:    .word 7
7 c:    .word 12
8 d:    .word 3
9 g:    .word 0
10
11 .text
12     lw    $s1, a # load data from memory
13     lw    $s2, b
14     lw    $s3, c
15     lw    $s4, d
16     # compute g = (a - b) + (c - d)
17     # change to subu and addu to trigger overflow exception
18     sub   $t1, $s1, $s2
19     sub   $t2, $s3, $s4
20     add   $s5, $t1, $t2
21     # store result
22     sw    $s5, g
23     # exit program
24     li    $v0, 10
25     syscall
26
```

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)
0x10010000	0x00000005	0x00000007	0x0000000c	0x00000003	0x00000007	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Shift Instructions



shamt - holds the number of bits to shift

SLL - shift left logical

SRL - shift right logical

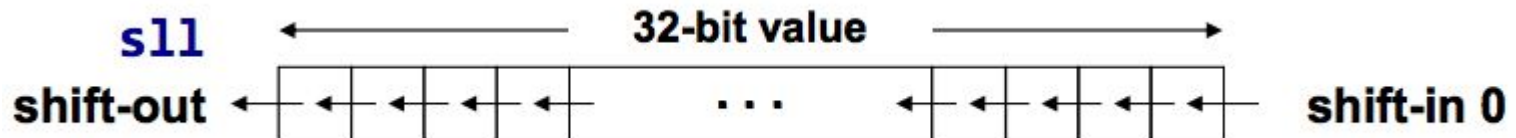
Move all bits right (or left) and fill empty spot with 0

SLL shift left logical

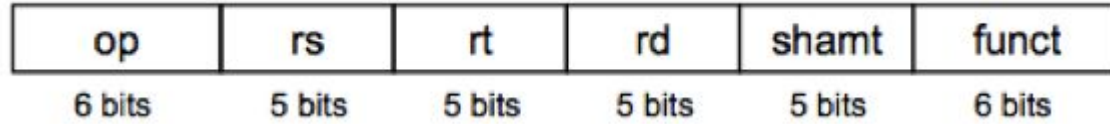
Each shift left is the same as multiplying by 2

```
li    $t2, 2  
sll   $t3, $t2, 1
```

\$t2	10	0x00000002
\$t3	11	0x00000004



Shift instruction format



Instruction: sll \$t3, \$t2, 1 # hex machine code 0x000a5840

000000 00000 01010 01011 00001 000000

opcode=function=000000

Rs is unused; rt is source; rd is destination; shamt = 00001

SLL as NOP

Some ISAs have a no-op instruction, an instruction that does nothing

Why? Useful for various situations such as creating time delays

MIPS uses SLL for a NOP: `sll $0, $0, 0`

This instruction does nothing; no side effects. Shifting \$zero by 0 does nothing and \$zero cannot be a destination register anyway.

What do you think the machine code for this instruction is?

SRL shift right logical

Each srl divides by 2 with truncation

For positive integers only



Shift and rotate instructions

MIPS also has:

SRA - shift right arithmetic to preserve sign

Many ISAs have rotate instructions that bring the “dropped” bit around to fill the vacant spot. MIPS implements rotate instructions with pseudo-instructions.

How are these instructions used?

Encryption and compression algorithms; fast mul/div

Register \$zero aka \$0

Read-only

Other use-cases:

As a move:

```
add $t2, $s1, $zero # $t2 = $s1
```

Pseudo-instructions

There is a MOV pseudo-instruction

Pseudo-instructions get translated to real instructions by the assembler.

These instructions have the same result:

```
add $t2, $s1, $zero # $t2 = $s1
```

```
move $t2, $s1 # $t2 = $s1
```

Address	Code	Basic	Source
0x00400000	0x02205020	add \$t0,\$17,\$0	2: add \$t2, \$s1, \$zero
0x00400004	0x00115021	addu \$t0,\$0,\$17	3: move \$t2, \$s1

More pseudo-instructions

li - load immediate

la - load address

These two pseudo instructions let us use 32-bit operands in a 16-bit space by translating the pseudo instruction into 2 real instructions.

Pseudo-instructions are included to make coding a little easier.

Load immediate and load upper immediate

li is translated into lui (load upper immediate) and ori if the operand is larger than 16 bits; otherwise it is translated into addiu \$0

```
li $t0, 0x12345678 # 32-bit operand
```

Becomes:

```
lui $1, 0x00001234
```

```
ori $8, $1, 0x00005678 # $1 is the at assembler temporary register (reserved)
```

Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x3c011234	lui \$1,0x00001234	2: li \$t0, 0x12345678
<input type="checkbox"/>	0x00400004	0x34285678	ori \$8,\$1,0x00005678	

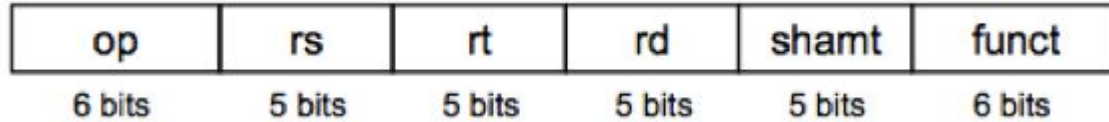
Load address

Addresses are 32-bits

Instruction la is also translated into lui and ori

Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1,0x00001001	4: la \$t0, a
<input type="checkbox"/>	0x00400004	0x34280000	ori \$8,\$1,0x00000000	

Logical instructions



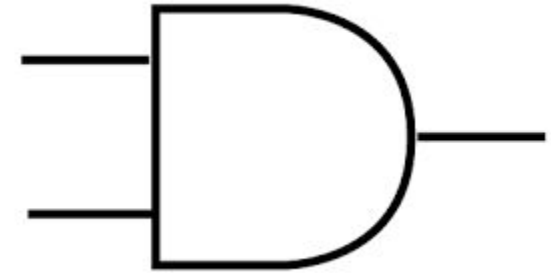
Same format as arithmetic instructions

The logical operation is performed bit-by-bit.

AND

```
$t1 0000 0000 0000 0000 0000 1101 1100 0000
$t2 0000 0000 0000 0000 0011 1100 0000 0000
AND
-----
$t0
```

x	y	x and y
0	0	0
0	1	0
1	0	0
1	1	1

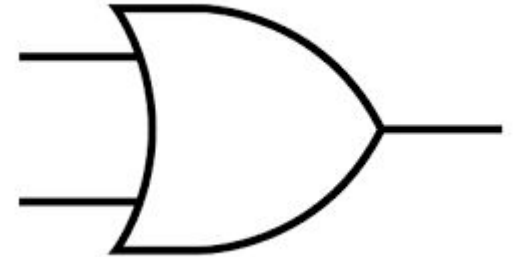
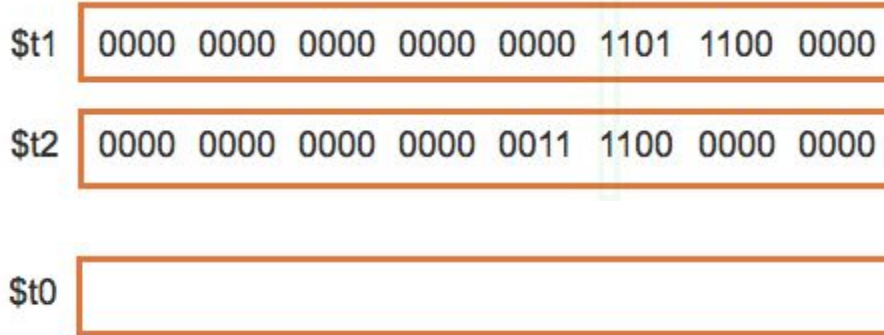


An AND yields a 1 in the result only if both bits of the operands are 1.

OR

A destination bit will be 1 if at least 1 of the source bits is 1.

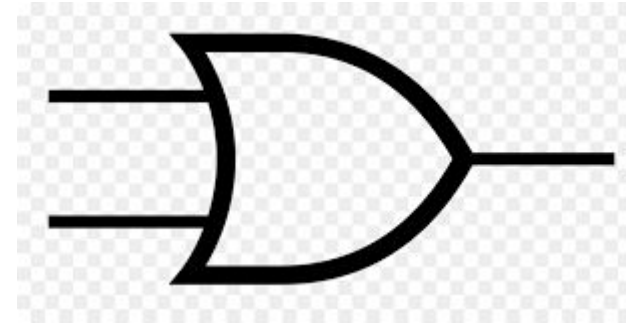
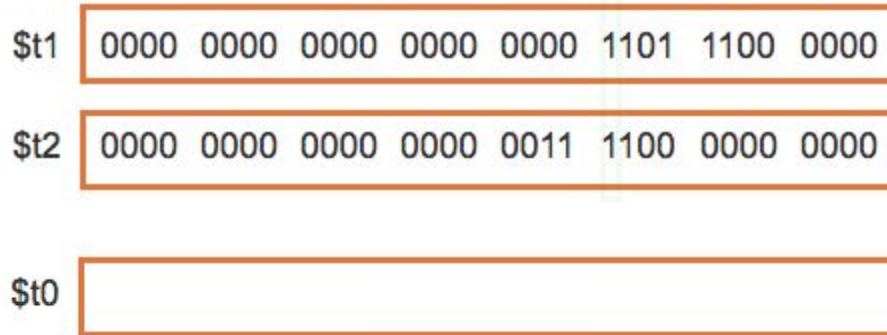
x	y	x or y
0	0	0
0	1	1
1	0	1
1	1	1



XOR

A destination bit will be 1 if one of the the source bits is 1,
but not both.

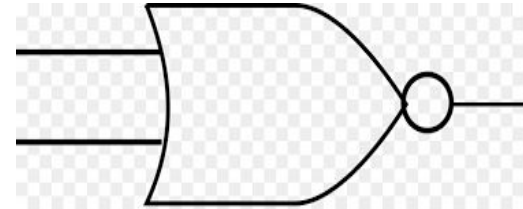
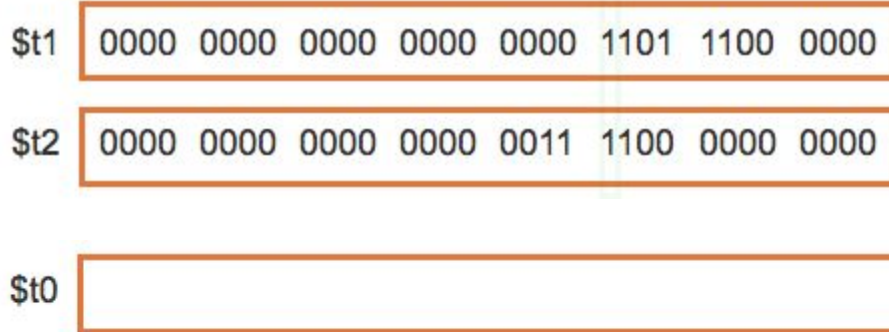
x	y	x xor y
0	0	0
0	1	1
1	0	1
1	1	0



NOR

A destination bit will be 1 if both source operand bits are 0.

x	y	x nor y
0	0	1
0	1	0
1	0	0
1	1	0



NOT?

MIPS does not implement a NOT instruction since NOR could be used:

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

First \$t1 and \$zero are ORed and then inverted.

- $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

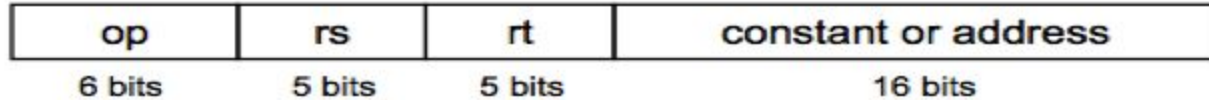
```
nor $t0, $t1, $zero ← Register 0: always read as zero
```

\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 1111 1111 1111 1111 1100 0011 1111 1111

Immediate arithmetic/logic instructions

addi, andi, ori, slli, etc., use the I-format:



rt is the destination operand; rs is the source operand

The constant can be -2^{15} to $-2^{15}-1$, that is, -32,768 to +32,767

Assemble by hand:

addi \$t0, \$t0, 1

```
0x21080001 addi $8,$8,0x00000001 10:          addi $t0, $t0, 1
```

001000 01000 01000 00000000000000000001

Opcode = 001000 = 8

Rs and Rt (destination) = 01000 \$t0

Constant = 0000000000000001

Immediate operands

Let's say our constant is 5.

16 bits 00000000000000101

2 bytes 00000000 00000101

4 hex digits 0005

ANDI example

To force bits to be 0, use an AND instruction.

ANDing by 0xffffffa forces bits 0 and 2 to be 0, leaving all other bits unchanged.

```
li      $t0, 0x55555555
andi    $t2, $t0, 0xffffffa
```

\$t0	8	0x55555555
\$t1	9	0x00000000
\$t2	10	0x55555550

ORI example

To force bits to be 1, use OR.

The following code forces bits 1 and 3 to be 1, leaving all others unchanged.

```
li    $t0, 0
ori   $t2, $t0, 0xa
```

\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x0000000a

Bitwise operations

We do have bitwise operators in higher-level languages as well

`&&` is often used for logical operations

`&` is often used for bit-wise operations

Why do we need bit-wise operations?

- Manipulate flag registers in embedded systems
- Any time bit-manipulation is needed such as encryption algorithms

Arithmetic/Logic/Shift Instruction

We will cover MUL, DIV and floating-point arithmetic after exam 1.

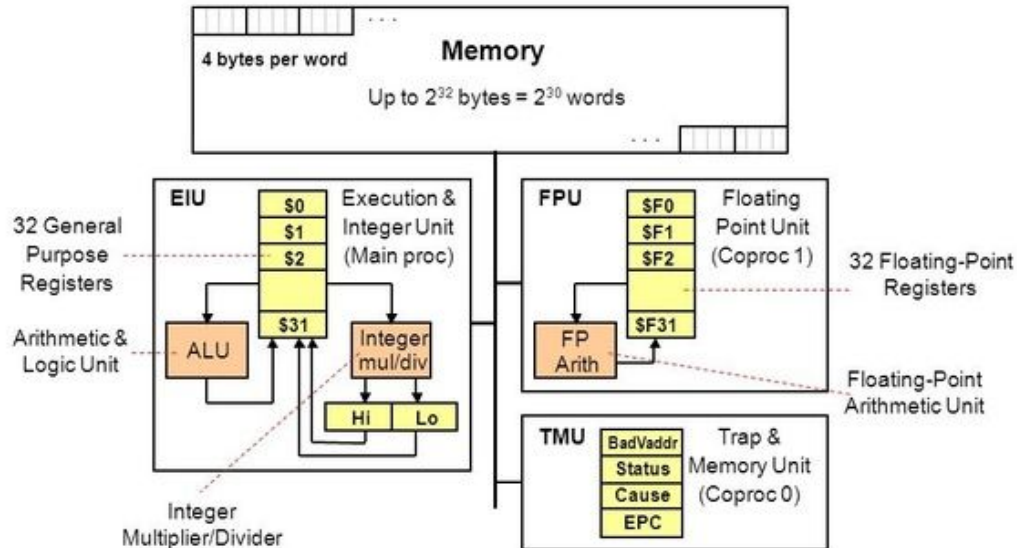
All arithmetic/logic/shift instructions use the R format.

If the opcode ends in “i” it is the immediate version which uses the I format.

Trick question: What instruction format does li use?

Integer arithmetic/logic/shift functions

Overview of the MIPS Processor



XOR swap algorithm

- use XOR bitwise operation to swap the contents of 2 registers

- `x = x xor y`

- `y = x xor y`

- `x = x xor y`

- proof:

https://en.wikipedia.org/wiki/XOR_swap_algorithm

<u>x</u>		<u>y</u>		
1010	\oplus	0011	=	1001 \rightarrow x
1001	\oplus	0011	=	1010 \rightarrow y
1001	\oplus	1010	=	0011 \rightarrow x
0011		1010		

The diagram illustrates the XOR swap algorithm. It shows three rows of calculations. The first row shows the initial values of x (1010) and y (0011) being XORed to produce 1001, which is assigned to x. The second row shows the new value of x (1001) and the original value of y (0011) being XORed to produce 1010, which is assigned to y. The third row shows the new value of x (1001) and the new value of y (1010) being XORed to produce 0011, which is assigned to x. Arrows indicate the flow of values: a green arrow points from the first row's result (1001) to the second row's first operand (1001), and a blue arrow points from the second row's result (1010) to the third row's second operand (1010). A blue arrow also points from the third row's result (0011) to the final value of x (0011), and a green arrow points from the third row's second operand (1010) to the final value of y (1010).

XOR cypher

Encrypt a string by xor-ing each character with a 'key'; read more here:

https://en.wikipedia.org/wiki/XOR_cipher

Using cypher key 7 = 0111 on char 'a' = 0x61 = 0110 0001

```
'a'    0110 0001
7      0111 0111
xor    0001 0110    encrypted char
```

```
      0001 0110    encrypted char
7      0111 0111
xor    0110 0001    decrypted char
```