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
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](https://www.youtube.com/watch?v=gp_D8r-2hwk)

- Explanation by SE Professor Ian Sommerville
  link: [https://www.youtube.com/watch?v=W3YJeoYgozw](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.
Trigger or ignore overflow

Implement the following C expression:

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

g = (a - b) + (c - d)
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
```
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: \texttt{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
```
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)
Load address

Addresses are 32-bits

Instruction la is also translated into lui and ori
Logical instructions

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

Same format as arithmetic instructions

The logical operation is performed bit-by-bit.
AND

An AND yields a 1 in the result only if both bits of the operands are 1.
A destination bit will be 1 if at least 1 of the source bits is 1.
XOR

A destination bit will be 1 if one of the source bits is 1, but not both.
NOR

A destination bit will be 1 if both source operand bits are 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.
Immediate arithmetic/logic instructions

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

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

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
```

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Binary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x21080001</td>
<td>001000</td>
<td>addi $8, $8, 0x00000001</td>
</tr>
</tbody>
</table>

10: `addi $t0, $t0, 1`

- Opcode = 001000 = 8
- Rs and Rt (destination) = 01000 $t0
- Constant = 0000000000000001
Immediate operands

Let's say our constant is 5.

16 bits: 0000000000000101

2 bytes: 00000000 00000101

4 hex digits: 0005
ANDI example

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

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

```
li $t0, 0x55555555
andi $t2, $t0, 0xffffffffa
```
**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
```
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
XOR swap algorithm

- use XOR bitwise operation to swap the contents of 2 registers
- \( x = x \text{ xor } y \)
- \( y = x \text{ xor } y \)
- \( x = x \text{ xor } y \)

- proof:
  
  [Proof link](https://en.wikipedia.org/wiki/XOR_swap_algorithm)
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

<table>
<thead>
<tr>
<th>'a'</th>
<th>0110 0001</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0111 0111</td>
</tr>
<tr>
<td>xor</td>
<td>0001 0110</td>
</tr>
</tbody>
</table>

0001 0110  encrypted char

| 7   | 0111 0111 |
| xor | 0110 0001 | decrypted char |