# Lecture 6
# Decision + Shift + I/O

# Instructions so far

| MIPS | C Program |
|---|---|
| add, sub, addi, multi, div | a=b+c, a=b-c, a=b+10, a=b*c, a=b/c |
| lw $t0,12($s0)<br>sw $t0, 12($s0) | t= A[3]<br>A[3]=t |
| beq $s0, $s1, L1<br>bne $s0, $s1, L1 | If (a==b) go to L1<br>If (a!=b) go to L1 |
| j L1<br>(unconditional branch) | goto L1 |
| slt reg1,reg2,reg3 | if (reg2 < reg3)<br>reg1 = 1;<br>else reg1 = 0; |

# Optional Individual Submission for Today's Quiz

- Turnin  under quiz2
- In the program, include your name/perm number
- By 11:59PM Tonight

# Shift Instructions

## TODAY'S FOCUS

# Bitwise Operations

- Up until now, we've done arithmetic (add, sub,addi ), memory access (lw and sw), and branches and jumps.
- All of these instructions view contents of register as a **single** quantity (such as a signed or unsigned integer)
- New Perspective: View contents of register as 32 **individual bits** rather than as a single 32-bit number

# Bitwise Operations

- Since registers are composed of 32 bits, we may want to access individual bits (or groups of bits) rather than the whole.
- Introduce two new classes of instructions:
  - Logical Operators
  - Shift Instructions

# Logical Operators

- Two basic logical operators:
  - AND: outputs 1 only if both inputs are 1
  - OR: outputs 1 if at least one input is 1
- In general, can define them to accept >2 inputs, but in the case of MIPS assembly, both of these accept exactly 2 inputs and produce 1 output
  - Again, rigid syntax, simpler hardware

# Logical Operators

- Truth Table: standard table listing all possible combinations of inputs and resultant output for each
- Truth Table for AND and OR

| A | B | A AND B | A OR B |
|---|---|---|---|
| 0 | 0 | **0** | **0** |
| 0 | 1 | **0** | **1** |
| 1 | 0 | **0** | **1** |
| 1 | 1 | **1** | **1** |

# Logical Operators

- Logical Instruction Syntax:

  1 2,3,4

  - where

    1) operation name

    2) register that will receive value

    3) first operand (register)

    4) second operand (register) or immediate (numerical constant)

# Logical Operators

- Instruction Names:
  - **and**, **or**: Both of these expect the third argument to be a register
  - **andi**, **ori**: Both of these expect the third argument to be an immediate
- MIPS Logical Operators are all bitwise, meaning that bit n of the output is produced by the respective bit n's of the inputs, bit 1 by the bit 1's, etc.

# Uses for Logical Operators

- Note that anding a bit with 0 produces a 0 at the output while anding a bit with 1 produces the original bit.
- This can be used to create a mask.
  - Example:

**mask last 12 bits**

1011 0110 1010 0100 0011 1101 1001 1010

0000 0000 0000 0000 0000 1111 1111 1111

  - The result of anding these:

0000 0000 0000 0000 0000 1101 1001 1010

# Uses for Logical Operators

- The second bitstring in the example is called a mask. It is used to isolate the rightmost 12 bits of the first bitstring by masking out the rest of the string (e.g. setting it to all 0s).
- The `and` operator can also be used to set certain portions of a bitstring to 0s, while leaving the rest alone.
  - In particular, if the first bitstring in the above example were in $t0, then the following instruction would mask the last 12 bits:

```
andi    $t0,$t0,0x0FFF
```

# Uses for Logical Operators

- Similarly, note that `or`ing a bit with 1 produces a 1 at the output while `or`ing a bit with 0 produces the original bit.
- This can be used to force certain bits of a string to 1s.
  - For example, if $t0 contains 0x12345678, then after this instruction:

    ```
    ori   $t0, $t0, 0xFFFF
    ```

  - … $t0 contains 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).

# Exercises

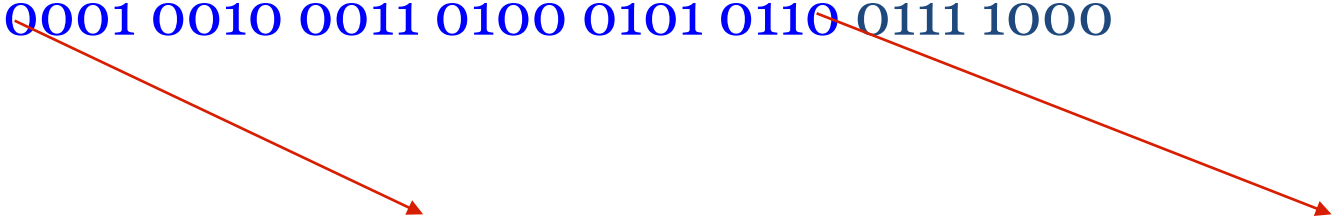
- $t0 holds the value of 0x1100FF
- What is the result of $t0 in each step:

```
and $t0, $t0, $t0
andi $t0, $t0, 0xFF
and $t0, $t0, $zero
ori $t0, $t0, 0xFFFF
or  $t0, $t0, $zero
ori $t0, $t0, 0x1001
```

# Shift Instructions

- Move (shift) all the bits in a word to the left or right by a number of bits.
  - Example: shift right by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000



0000 0000 0001 0010 0011 0100 0101 0110

- Example: shift left by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000

# Shift Instructions

- Shift Instruction Syntax:

  1 2,3,4

  - where

    1) operation name

    2) register that will receive value

    3) first operand (register)

    4) shift amount (constant <= 32)

# Shift Instructions

- MIPS shift instructions:
  1. `sll` (shift left logical): shifts left and fills emptied bits with 0s
  2. `srl` (shift right logical): shifts right and fills emptied bits with 0s
  3. `sra` (shift right arithmetic): shifts right and fills emptied bits by sign extending

# Shift Instructions

- Example: shift right arithmetic by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- Example: shift right arithmetic by 8 bits

1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110

# Uses for Shift Instructions

- Suppose we want to isolate byte 0 (rightmost 8 bits) of a word in $t0. Simply use:

```
andi    $t0,$t0,0x00FF
```

- Suppose we want to isolate byte 1 (bit 15 to bit 8) of a word in $t0. We can use:

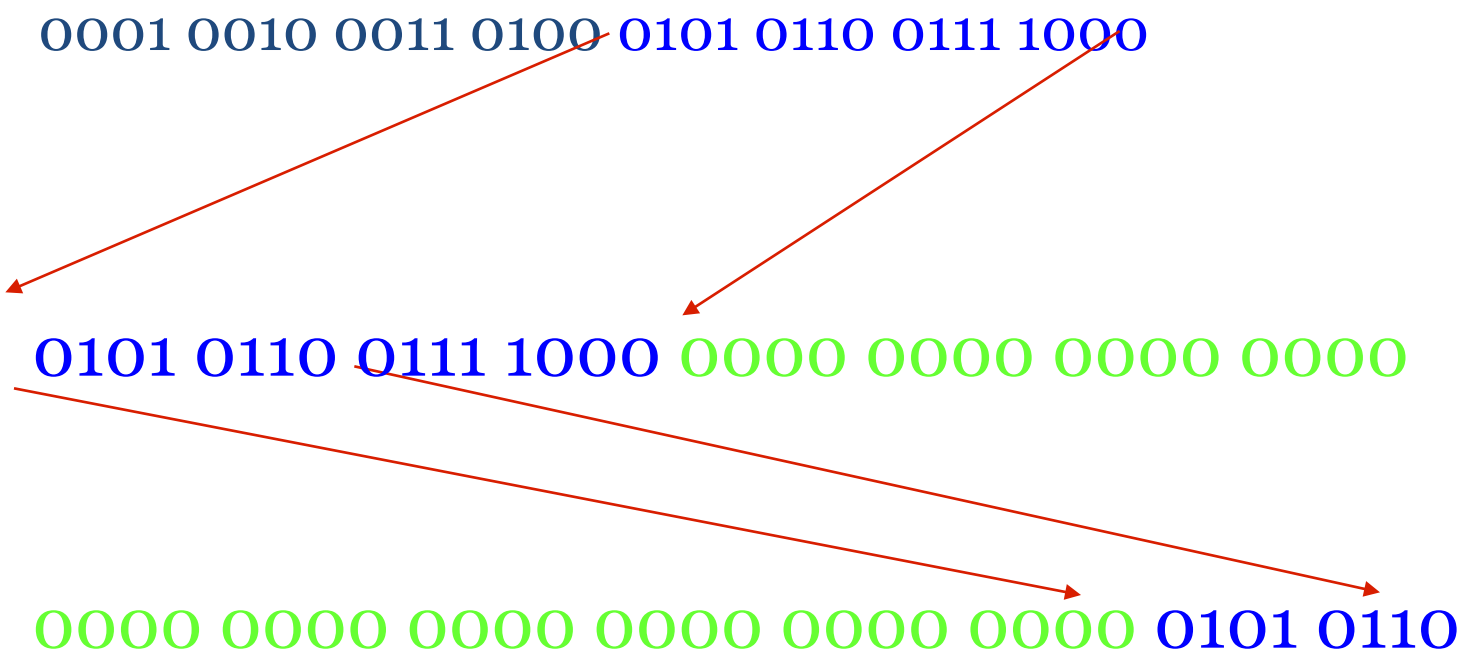
```
andi    $t0,$t0,0xFF00
```

but then we still need to shift to the right by 8 bits…

# Uses for Shift Instructions

- Could use instead:

```
sll   $t0,$t0,16
srl   $t0,$t0,24
```



# Uses for Shift Instructions

- In binary:
  - Multiplying by 2 is same as shifting left by 1:
    - $11_2 \times 10_2 = 110_2$
    - $1010_2 \times 10_2 = 10100_2$
  - Multiplying by 4 is same as shifting left by 2:
    - $11_2 \times 100_2 = 1100_2$
    - $1010_2 \times 100_2 = 101000_2$
  - Multiplying by $2^n$ is same as shifting left by n

# Uses for Shift Instructions

- Since shifting is faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

  `a *= 8;` (in C)

  would compile to:

  `sll  $s0,$s0,3` (in MIPS)

- Likewise, shift right to divide by powers of 2
  - remember to use `sra`

# MIPS Native Instructions

- Instructions that have **direct** hardware implementation, implement in **1 cycle**
- As opposed to ***pseudo instructions*** which are translated into multiple native instructions
- Native:
  - Add, addi, add, sub, lw, sw, and, andi, or, ori, slt, sll, srl, beq, bne, j, jr, jal
- Pseudo:
  - Multi, div, li, bge, ble

# A Short Summary

- Logical and Shift Instructions
  - Operate on bits individually, unlike arithmetic, which operate on entire word.
  - Use to isolate fields, either by masking or by shifting back and forth.
  - Use shift left logical, `sll,` for multiplication by powers of 2
  - Use shift right arithmetic, `sra,` for division by powers of 2.
- New Instructions:

  `and,andi, or,ori, sll,srl,sra`

# Assembly Program of the Day

- Exit the program via system call

```
# Daniel J. Ellard -- 02/21/94
# add.asm-- A program that computes the sum of 1 and 2,
# leaving the result in register $t0.
# Registers used:
# t0 - used to hold the result.
# t1 - used to hold the constant 1.
# v0 - syscall parameter.

            . text
main:                           # SPIM starts execution at main.
            li $t1, 1           # load 1 into $t1.
            add $t0, $t1, 2     # compute the sum of $t1 and 2, and
                                # put it into $t0.
            li $v0, 10          # syscall code 10 is for exit.
            syscall             # make the syscall.
```