CSE 31
Computer Organization

Lecture 12 – Logical Operators
Instruction Format

Announcement

- Project #1
  - Due at 11:59pm on 3/22, Friday (no more late submission)
  - You must demo your submission to your TA during week of 4/1, in lab.
- Lab #6 this week
  - Due at 11:59pm on the same day of your next lab
  - You must demo your submission to your TA within 14 days
- HW #3 in CatCourses
  - Due Monday (3/18) at 11:59pm
- Reading assignment
  - Chapter 4.1 – 4.9of zyBooks
    - Make sure to do the Participation Activities
    - Due Wednesday (3/20) at 11:59pm

Bitwise Operations

- So far, we've done arithmetic (**`add`**, **`sub`**,**`addi`**), mem access (**`lw`** and **`sw`**), & branches and jumps.
- All of these instructions view contents of register as a single quantity (e.g., signed or unsigned int)
- New Perspective: View register as 32 raw bits rather than as a single 32-bit number
  - Since registers are composed of 32 bits, wish to access individual bits (or groups of bits) rather than the whole.
- Introduce two new classes of instructions
  - Logical & Shift Ops

Logical Operators (1/3)

- Two basic logical operators:
  - AND: outputs 1 only if all inputs are 1
  - OR: outputs 1 if at least one input is 1
- Truth Table: standard table listing all possible combinations of inputs and resultant output

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 (2/3)

- 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)
- In general, can define them to accept > 2 inputs, but in the case of MIPS assembly, these accept exactly 2 inputs and produce 1 output
  - Again, rigid syntax, simpler hardware

Logical Operators (3/3)

- 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 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.
  - C: Bitwise AND is **&** (e.g., `z = x & y;`)
  - C: Bitwise OR is **|** (e.g., `z = x | y;`)

Uses of Logical Operators (1/3)

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

```
1011 0110 1010 0100 0011 1101 1001 1010
0000 0000 0000 0000 0000 1111 1111 1111
```

mask:

  - The result of `and`ing these:

```
0000 0000 0000 0000 0000 1101 1001 1010
```

mask last 12 bits

Uses of Logical Operators (2/3)

- 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 to all `0`s).
- Thus, the **`and`** operator can be used to set certain portions of a bitstring to `0`s, while leaving the rest alone.
  - In particular, if the first bitstring in the above example were in **`$t0`**, then the following instruction would mask it:

```
andi   $t0,$t0,0xFFF
```

Uses of Logical Operators (3/3)

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

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

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

Shift Instructions (review) (1/4)

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

1001 0010 0011 0100 0101 0110 0111 1000

0000 0000 1001 0010 0011 0100 0101 0110

  - Example: shift left by 8 bits

0001 0010 1011 0100 0101 0110 0111 1000

1011 0100 0101 0110 0111 1000 0000 0000

Shift Instructions (2/4)

- 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)

- 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 (3/4)

- Example: shift right arithmetic (sra) by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- Example: shift right arithmetic (sra) by 8 bits

1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110

Shift Instructions (4/4)

- 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 (rounds towards $-\infty$)
  - remember to use `sra`

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 logical, **srl**,for division by powers of 2 of unsigned numbers (**unsigned int**)
  - Use shift right arithmetic, **sra**,for division by powers of 2 of signed numbers (**int**)
- New Instructions:

  and, andi, or, ori, sll, srl, sra
- That's all you need to know about MIPS!

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

```
Multiplicand        1000      8
  Multiplier       x1001      9
                    1000
                   0000
                  0000
                +1000
                 01001000
```

- m bits x n bits = m + n bit product

Integer Multiplication (2/3)

- In MIPS, we multiply registers, so:
  - 32-bit value x 32-bit value = 64-bit value
- Syntax of Multiplication (signed):
  - `mult` register1, register2 No destination register!
  - Multiplies 32-bit values in those registers & puts 64-bit product in special result regs:
    - puts product upper half in **hi**, lower half in **lo**
  - hi and lo are 2 registers separate from the 32 general purpose registers
  - Use mfhi register and mflo register to move from **hi**, **lo** to another register

Integer Multiplication (3/3)

- Example:
  - in C: `a = b * c;`
  - in MIPS:
    - let `b` be `$s2`; let `c` be `$s3`; and let `a` be `$s0` and `$s1` (since it may be up to 64 bits)

```
mult $s2,$s3      # b*c
mfhi $s0          # upper half of
                  # product into $s0
mflo $s1          # lower half of
                  # product into $s1
```

- Note: Often, we only care about the lower half of the product.

Integer Division (1/2)

- Paper and pencil example (unsigned):

```
                 1001      Quotient
            ____________
Divisor 1000|1001010       Dividend
           -1000
              10
              101
              1010
             -1000
                10 Remainder
                 (or Modulo result)
```

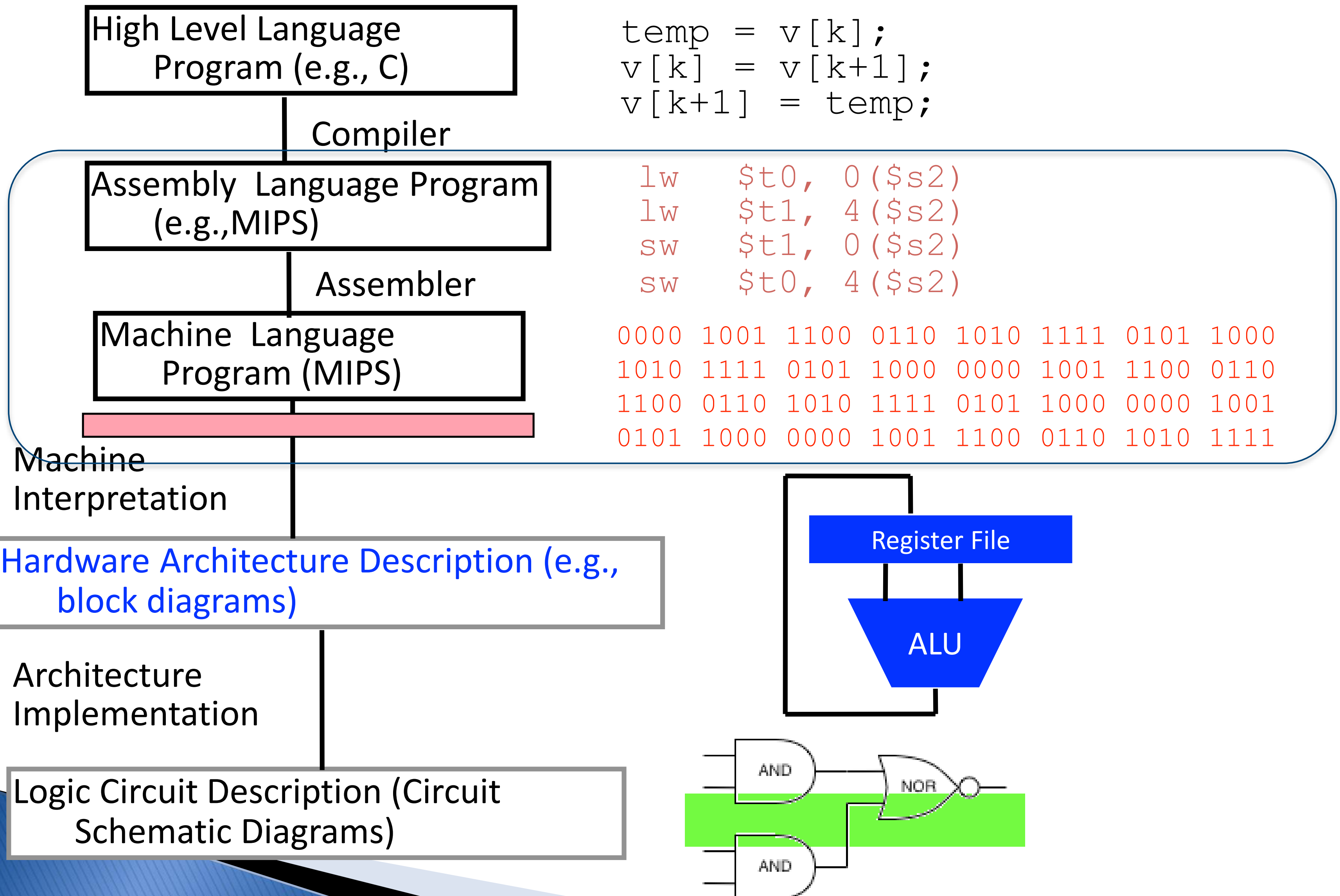
- Dividend = Quotient x Divisor + Remainder

Integer Division (2/2)

- Syntax of Division (signed):
  - `div` register1, register2
  - Divides 32-bit register1 by 32-bit register2
  - Puts remainder of division in `hi`, quotient in `lo`
- Implements C division (/) and modulo (%)
- Example in C: `a = c / d;` `b = c % d;`
- in MIPS: `a↔$s0;b↔$s1;c↔$s2;d↔$s3`

```
div  $s2,$s3     # lo=c/d, hi=c%d
mflo $s0         # get quotient
mfhi $s1         # get remainder
```

Levels of Representation (abstractions)



Big Idea: Stored-Program Concept

- Where are programs stored when they are being run?
  - How are they stored in memory?
- Computers built on 2 key principles:
  - Instructions are represented as bit patterns - can think of these as numbers.
  - Therefore, entire programs can be stored in memory to be read or written just like data.
- Simplifies SW/HW of computer systems:
  - Memory technology for data also used for programs

Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
  - both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- One register keeps address of instruction being executed: "Program Counter" (PC)
  - Basically a pointer to memory: Intel calls it Instruction Address Pointer, a better name

Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for Macintoshes and PCs
- New machines want to run old programs (“binaries”) as well as programs compiled to new instructions
  - Leads to “backward compatible” instruction set evolving over time
  - Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium 4); could still run program from 1981 PC today

Instructions as Numbers (1/2)

- Currently all data we work with is in words (32-bit blocks):
  - Each register is a word.
  - **`lw`** and **`sw`** both access memory one word at a time.
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so "**`add $t0,$0,$0`**" is meaningless.
  - MIPS wants simplicity: since data is in words, make instructions into words too!

Instructions as Numbers (2/2)

- One word is 32 bits, so divide instruction word into "fields".
- Each field tells processor something about the instruction.
- We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  - I-format
  - J-format
  - R-format
  - What do these letters (I, J, R) stand for?

Instruction Formats

- I-format: used for instructions with immediates, `lw` and `sw` (since offset counts as an immediate), and branches (`beq` and `bne`),
  - (but not the shift instructions; later)
- J-format: used for `j` and `jal`
- R-format: used for all other instructions
- Why 3 different formats?
  - It will soon become clear why the instructions have been partitioned in this way.

R-Format Instructions (1/5)

- Define "fields" of the following number of bits each: 6 + 5 + 5 + 5 + 5 + 6 = 32

6	5	5	5	5	6

- For simplicity, each field has a name:

opcode	rs	rt	rd	shamt	funct

- Important: On these slides and in book, each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer.
  - Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63.