An Arithmetic Logic Unit (ALU)

The ALU is the ‘brawn’ of the computer

- What does it do?

- How wide does it need to be?

- What outputs do we need for MIPS?

A simple 32-bit ALU

ADMIN

- Course paper descriptions – due Fri Feb 22 via plain text email (not MS Word)
  - See details online
ALU Control and Symbol

<table>
<thead>
<tr>
<th>ALU Control Lines</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0010</td>
<td>Add</td>
</tr>
<tr>
<td>0110</td>
<td>Subtract</td>
</tr>
<tr>
<td>0111</td>
<td>Set on less than</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>

Multiplication: Simple Implementation

Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- Example: grade-school algorithm
  \[ 0010 \text{ (multiplicand)} \times 1011 \text{ (multiplier)} \]

- Multiply \( m \times n \) bits, How wide (in bits) should the product be?
Using Multiplication

- Product requires 64 bits
  - Use dedicated registers
  - HI – more significant part of product
  - LO – less significant part of product
- MIPS instructions
  - mult $s2, $s3
  - multu $s2, $s3
  - mfhi $t0
  - mflo $t1
- Division
  - Can perform with same hardware! (see book)
  - div $s2, $s3  Lo = $s2 / $s3
  - Hi = $s2 mod $s3
  - divu $s2, $s3

Floating Point

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 3.15576 \times 10^{32}
- Representation:
  - sign, exponent, significand:
    - $(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent(some power)}}$
  - Significand always in normalized form:
    - Yes:
    - No:
  - more bits for significand gives more
  - more bits for exponent increases

IEEE754 Standard

Single Precision (float): 8 bit exponent, 23 bit significand

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent (8 Bits)</th>
<th>Significand (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>123456789...17</td>
<td>18 19 20</td>
</tr>
</tbody>
</table>

Double Precision (double): 11 bit exponent, 52 bit significand

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent (11 Bits)</th>
<th>Significand (20 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>123456789...17</td>
<td>18 19 20</td>
</tr>
<tr>
<td>More</td>
<td>Significand (32 more bits)</td>
<td></td>
</tr>
</tbody>
</table>

IEEE 74 – Optimizations

- Significand
  - What’s the first bit?
  - So...

- Exponent is “biased” to make sorting easier
  - Smallest exponent represented by:
  - Largest exponent represented by:
  - Bias values
    - 127 for single precision
    - 1023 for double precision

- Summary: $(-1)^{\text{sign}} \times (1+\text{significand}) \times 2^{\text{exponent – bias}}$
Example #1:

- Represent $-5.75_{10}$ in binary, single precision form:

  - **Strategy**
    - Transfer into binary notation (fraction)
    - Normalize significand (if necessary)
    - Compute exponent
      - $(\text{Real exponent}) = (\text{Stored exponent}) - \text{bias}$
    - Apply results to formula
      $(-1)_{\text{sign}} \times (1+\text{significand}) \times 2^{\text{exponent} - \text{bias}}$

Example #1:

- Represent $-9.75_{10}$ in binary single precision:
  - $-9.75_{10} =$

  - Compute the exponent:
    - Remember $(2^{\text{exponent} - \text{bias}})$
    - $\text{Bias} = 127$

  - Formula $(-1)_{\text{sign}} \times (1+\text{significand}) \times 2^{\text{exponent} - \text{bias}}$

Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
  - other complexities
- Implementing the standard can be tricky

MIPS Floating Point Basics

- Floating point registers
  $f0$, $f1$, $f2$, ..., $f31$
  Used in pairs for double precision ($f0$, $f1$) ($f2$, $f3$), ...$
  f0$ not always zero

- Register conventions:
  - Function arguments passed in
  - Function return value stored in
  - Where are addresses (e.g. for arrays) passed?

- Load and store:
  lwc1 $f2$, 0($sp$)
  swc1 $f4$, 4($t2$)
MIPS FP Arithmetic

- Addition, subtraction: add.s, add.d, sub.s, sub.d
  
  add.s $f1, $f2, $f3
  add.d $f2, $f4, $f6

- Multiplication, division: mul.s, mul.d, div.s, div.d
  
  mul.s $f2, $f3, $f4
  div.s $f2, $f4, $f6

MIPS FP Control Flow

- Pattern of a comparison: c.___.s (or c.___.d)
  
  c.lt.s $f2, $f3
  c.ge.d $f4, $f6

- Where does the result go?

  Branching:
  
  bc1t label10
  bc1f label20

Example #1

- Convert the following C code to MIPS:
  
  float max (float A, float B) {
    if (A <= B) return A;
    else return B;
  }

Example #2

- Convert the following C code to MIPS:
  
  void setArray (float F[], int index, float val) {
    F[index] = val;
  }
Chapter Three Summary

- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist
  - two’s complement
  - IEEE 754 floating point
- Computer instructions determine “meaning” of the bit patterns
- Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation).

- We are (almost!) ready to move on (and implement the processor)

Chapter Goals

- Introduce 2’s complement numbers
  - Addition and subtraction
  - Sketch multiplication, division
- Overview of ALU (arithmetic logic unit)
- Floating point numbers
  - Representation
  - Arithmetic operations
  - MIPS instructions