COMPUTER ARCHITECTURE

COMPUTER ARITHMETIC

Dr. Bill Yi
Santa Clara University

(Based on text: David A. Patterson & John L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3rd Ed., Morgan Kaufmann, 2007)
(Also based on presentation: Dr. Nam Ling, COEN210 Lecture Notes)
COURSE CONTENTS

- Introduction
- Instructions
- Computer Arithmetic
- Performance
- Processor: Datapath
- Processor: Control
- Pipelining Techniques
- Memory
- Input/Output Devices
Arithmetic Logic Unit (ALU)
Fast Adder
Multiplication
Floating Point
Additional MIPS instructions
Foundation Knowledge

- Decimal, Binary, Octal, & Hexadecimal Numbers
- Signed & Unsigned Numbers
- 2’s Complement Representation
- 2’s Complement Negation, Addition, & Subtraction
- Overflow
- Sign Extension
- ASCII vs Binary
- Boolean Algebra
- Logic Design
- Assembly Language
Numbers

- Bits are just bits (no inherent meaning)
  - Conventions define relationship between bits and numbers
- Binary numbers (base 2)
  - 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  - decimal: 0 . . . 2^n – 1
- Of course it gets more complicated:
  - Numbers are finite (overflow)
  - Fractions and real numbers
  - Negative numbers
  - E.g., no MIPS subi instruction; addi can add a negative number
- How do we represent negative numbers?
  - i.e., which bit patterns will represent which numbers?
**Possible Representations**

Three representations

<table>
<thead>
<tr>
<th>Sign Magnitude:</th>
<th>One's Complement</th>
<th>Two's Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 = +0</td>
<td>000 = +0</td>
<td>000 = +0</td>
</tr>
<tr>
<td>001 = +1</td>
<td>001 = +1</td>
<td>001 = +1</td>
</tr>
<tr>
<td>010 = +2</td>
<td>010 = +2</td>
<td>010 = +2</td>
</tr>
<tr>
<td>011 = +3</td>
<td>011 = +3</td>
<td>011 = +3</td>
</tr>
<tr>
<td>100 = -0</td>
<td>100 = -3</td>
<td>100 = -4</td>
</tr>
<tr>
<td>101 = -1</td>
<td>101 = -2</td>
<td>101 = -3</td>
</tr>
<tr>
<td>110 = -2</td>
<td>110 = -1</td>
<td>110 = -2</td>
</tr>
<tr>
<td>111 = -3</td>
<td>111 = -0</td>
<td>111 = -1</td>
</tr>
</tbody>
</table>

- **Issues:** balance, number of zeros, ease of operations
- Which one is best? Why?
32 bit signed numbers:

0000 0000 0000 0000 0000 0000 0000 0000\textsubscript{two} = 0\textsubscript{ten}
0000 0000 0000 0000 0000 0000 0000 0001\textsubscript{two} = +1\textsubscript{ten}
0000 0000 0000 0000 0000 0000 0000 0010\textsubscript{two} = +2\textsubscript{ten}
...
0111 1111 1111 1111 1111 1111 1111 1110\textsubscript{two} = +2,147,483,646\textsubscript{ten}
0111 1111 1111 1111 1111 1111 1111 1111\textsubscript{two} = +2,147,483,647\textsubscript{ten}
1000 0000 0000 0000 0000 0000 0000 0000\textsubscript{two} = −2,147,483,648\textsubscript{ten}
1000 0000 0000 0000 0000 0000 0000 0001\textsubscript{two} = −2,147,483,647\textsubscript{ten}
1000 0000 0000 0000 0000 0000 0000 0010\textsubscript{two} = −2,147,483,646\textsubscript{ten}
...
1111 1111 1111 1111 1111 1111 1111 1101\textsubscript{two} = −3\textsubscript{ten}
1111 1111 1111 1111 1111 1111 1111 1110\textsubscript{two} = −2\textsubscript{ten}
1111 1111 1111 1111 1111 1111 1111 1111\textsubscript{two} = −1\textsubscript{ten}

- maxint: $+2,147,483,647\textsubscript{ten}$
- minint: $−2,147,483,648\textsubscript{ten}$
Two’s Complement Operations

- Negating a two’s complement number: invert all bits and add 1
  - Remember: “negate” and “invert” are quite different!
- Converting $n$ bit numbers into numbers with more than $n$ bits:
  - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
  - Copy the most significant bit (the sign bit) into the other bits
    - 0010 -> 0000 0010
    - 1010 -> 1111 1010
  - “sign extension” ($lbu$ vs. $lb$)
**Additional MIPS Instructions**

- **Character transfer**
  
  \[
  \text{lbu } \$s1, 100(\$s2) \quad \# \; \$s1 \leftarrow \text{memory } [\$s2+100] \text{ (load byte unsigned)} \\
  \text{sb } \$s1, 100(\$s2) \quad \# \; \text{memory } [\$s2+100] \leftarrow \$s1 \text{ (store byte)}
  \]

- **Conditions**

  \[
  \text{sltu } \$s2, \$s3, \$s4 \quad \# \; \text{if } (\$s3) < (\$s4) \text{ then } \$s2 \leftarrow 1; \\
  \quad \# \; \text{else } \$s2 \leftarrow 0 \text{ (set on less than, unsigned numbers)} \\
  \quad \# \; \text{Note that slt works on } 2' \text{ complement numbers}
  \]

- **Arithmetic on unsigned numbers**

  \[
  \text{addu } \$s1, \$s2, \$s3 \quad \# \; \$s1 \leftarrow \$s2 + \$s3 \text{ (no overflow detection)} \\
  \text{subu } \$s1, \$s2, \$s3 \quad \# \; \$s1 \leftarrow \$s2 - \$s3 \text{ (no overflow detection)} \\
  \text{addiu } \$s1, \$s2, 100 \quad \# \; \$s1 \leftarrow \$s2 + 100 \text{ (no overflow detection)}
  \]

- **MIPS detects overflow with an exception (interrupt), which is an unscheduled procedure call. MIPS includes a register, called exception program counter (EPC) to contain the address of the instruction that caused the exception**

  \[
  \text{mfc0 } \$s1, \$epc \quad \# \; \$s1 \leftarrow \$epc \text{ (move from special registers)}
  \]
Additional MIPS Instructions

- **Shift operations**
  sll $t2, $s0, 8  #  $t2 ← $s0<<8 (shift left by constant)
  srl $s1, $s2, 10  #  $s1 ← $s2>>10 (shift right by constant)
  Fill the emptied bits with 0’s

- **Logical operations**
  and $s1, $s2, $s3  #  $s1 ← $s2 and $s3 (bit-by-bit and)
  or $s1, $s2, $s3  #  $s1 ← $s2 or $s3 (bit-by-bit or)
  andi $s1, $s2, 100  #  $s1 ← $s2 and 100
  ori $s1, $s2, 100  #  $s1 ← $s2 or 100
**ALU: Arithmetic Logic Unit**

- Performs arithmetic (e.g. add) & logical operations (e.g. and) in CPU

<table>
<thead>
<tr>
<th>Control</th>
<th>Funct</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>and</td>
<td>A and B</td>
</tr>
<tr>
<td>001</td>
<td>or</td>
<td>A or B</td>
</tr>
<tr>
<td>010</td>
<td>add</td>
<td>A + B</td>
</tr>
<tr>
<td>110</td>
<td>sub</td>
<td>A - B</td>
</tr>
<tr>
<td>111</td>
<td>slt</td>
<td>1 if A&lt;B</td>
</tr>
</tbody>
</table>
ALU Building Blocks

- 1-bit adder
  - $c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$
  - $\text{sum} = a \oplus b \oplus c_{in}$
    - Note: Cin is carryin, cout is carryout

- Gates, multiplexor
A 1-bit ALU that performs AND, OR, and addition (shown below)
Building a 32-bit ALU (shown right)
Two's complement approach: just negate $b$ and add.

How do we negate?

$$a - b = a + (-b) = a + (\bar{b} + 1)$$

By selecting $\text{Binvert} = 1$, and setting $\text{CarryIn} = 1$ in the least significant bit of ALU, we get 2's complement subtraction $a - b$. 

![ALU Subtraction Diagram]
To support set-on-less-than instruction \((slt)\)
- \(slt\) is an arithmetic instruction
- produces a 1 if \(rs < rt\) and 0 otherwise
- use subtraction: \((a-b) < 0\) implies \(a < b\)
- use a Set & a Less signal to indicate result

To support test for equality \((beq)\)
- use subtraction: \((a-b) = 0\) implies \(a = b\)
A 1-bit ALU that performs AND, OR, add, subtract:

Less is used for *slt* instruction (see 32-bit ALU next slide)

The ALU for the most significant bit:

*Set* is used for *slt* instruction, it is connected to *Less* of lsb (see 32-bit ALU next slide)

*Overflow* detection needed on msb
A 32-bit ALU

A 32-bit ALU that performs AND, OR, add, & subtract

For subtract, set \( Binvert = 1 \) and \( CarryIn = 1 \) (for add or logical operations, both set to 0)

Can combine \( Binvert \) & \( CarryIn \) to \( Bnegate \)

\( Set \) and \( Less \), together with subtraction, can be used for \( slt \)
Add a zero detector to test for zero results or equality (e.g. in beq instruction)

Control lines (Operation) (3-bit):

000 = and
001 = or
010 = add
110 = subtract
111 = slt

bit1 & bit0 to multiplexors in ALU
bit2 to Bnega

• Note: zero is a 1 when the result is zero!
ALU Design: Summary

- Select **building blocks**: adders, gates
- Use **multiplexors** to select the output we want
- Perform subtraction using two’s complement
- Replicate a 1-bit ALU to produce a 32-bit ALU --> regularity
- Need circuit to detect conditions e.g. zero result, overflow, sign, carry out
- Shift instructions: Done *outside* the ALU by *barrel shifter*, which can shift from 1 to 31 bits in no more time than it takes to add two 32 bit numbers using carry lookahead adders
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the “critical path” or the “deepest level of logic”)
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)
Ripple carry adder is just too slow:

- The sequential chain reaction is too slow for time-critical hardware
- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products
  - Can you see the ripple? How could you get rid of it?
**Carry Lookahead Adder**

- **Carry lookahead adder (CLA):** an approach in-between our two extremes
- **Motivation:**
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? $g_i = a_i \cdot b_i$
  - When would we propagate the carry? $p_i = a_i + b_i$
  - Did we get rid of the ripple?

\[
\begin{align*}
    c_{i+1} &= g_i + p_i c_i \\
    c_1 &= g_0 + p_0 c_0 \\
    c_2 &= g_1 + p_1 g_0 + p_1 p_0 c_0 \\
    c_3 &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \\
    c_4 &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0
\end{align*}
\]

- **Carry lookahead!**
Building Bigger Adders

- Can’t build a 16 bit adder using the $g_i$ & $p_i$ CLA method --> too big
- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again! (see left figure)
More complicated than addition
  - accomplished via shifting and addition
More time and more chip area
Let's look at pencil-paper algorithm (both unsigned numbers)

\[
\begin{array}{c}
0111 & \text{(multiplicand)} \\
\times & 1011 & \text{(multiplier)} \\
\hline
0111 \\
0111 \\
0111 \\
0000 \\
0111 \\
\hline
01001101 & \text{(product)}
\end{array}
\]

- \text{$n$}-bit multiplicand \text{x} \text{$m$}-bit multiplier gives \text{$n+m$} bit product
- Negative numbers: convert and multiply in positive forms, then adjust sign of results
Multiplier: First Version

Example: 0010 x 0011 = 0000 0110

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Step</th>
<th>Multiplier</th>
<th>Multiplicand</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial values</td>
<td>0011</td>
<td>0000 0010</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1</td>
<td>1: Prod = Prod + Mcand</td>
<td>0011</td>
<td>0000 0010</td>
<td>0000 0010</td>
</tr>
<tr>
<td></td>
<td>Shift left Mcand</td>
<td>0011</td>
<td>0000 0100</td>
<td>0000 0010</td>
</tr>
<tr>
<td></td>
<td>Shift right Mplier</td>
<td>0001</td>
<td>0000 0100</td>
<td>0000 0010</td>
</tr>
<tr>
<td>2</td>
<td>1: Prod = Prod + Mcand</td>
<td>0001</td>
<td>0000 0100</td>
<td>0000 0110</td>
</tr>
<tr>
<td></td>
<td>Shift left Mcand</td>
<td>0001</td>
<td>0000 1000</td>
<td>0000 0110</td>
</tr>
<tr>
<td></td>
<td>Shift right Mplier</td>
<td>0000</td>
<td>0000 1000</td>
<td>0000 0110</td>
</tr>
<tr>
<td>3</td>
<td>0: no op</td>
<td>0000</td>
<td>0000 1000</td>
<td>0000 0110</td>
</tr>
<tr>
<td></td>
<td>Shift left Mcand</td>
<td>0000</td>
<td>0001 0000</td>
<td>0000 0110</td>
</tr>
<tr>
<td></td>
<td>Shift right Mplier</td>
<td>0000</td>
<td>0001 0000</td>
<td>0000 0110</td>
</tr>
<tr>
<td>4</td>
<td>0: no op</td>
<td>0000</td>
<td>0010 0000</td>
<td>0000 0110</td>
</tr>
<tr>
<td></td>
<td>Shift left Mcand</td>
<td>0000</td>
<td>0010 0000</td>
<td>0000 0110</td>
</tr>
<tr>
<td></td>
<td>Shift right Mplier</td>
<td>0000</td>
<td>0010 0000</td>
<td>0000 0110</td>
</tr>
</tbody>
</table>
Multiplier: Second Version

Example: $0010 \times 0011 = 0000\ 0110$

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Step</th>
<th>Multiplier</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial values</td>
<td>0011</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1</td>
<td>1: Prod = Prod’ + Mcand’</td>
<td>0011</td>
<td>0010 0000</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod, Multiplier</td>
<td>0001</td>
<td>0001 0000</td>
</tr>
<tr>
<td>2</td>
<td>1: Prod = Prod’ + Mcand’</td>
<td>0001</td>
<td>0011 0000</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod, Multiplier</td>
<td>0000</td>
<td>0001 1000</td>
</tr>
<tr>
<td>3</td>
<td>0: no op</td>
<td>0000</td>
<td>0001 1000</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod, Multiplier</td>
<td>0000</td>
<td>0000 1100</td>
</tr>
<tr>
<td>4</td>
<td>0: no op</td>
<td>0000</td>
<td>0000 1100</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod, Multiplier</td>
<td>0000</td>
<td>0000 0110</td>
</tr>
</tbody>
</table>

1. Test Multiplier0
   - Multiplier0 = 1
     - Shift the Multiplier register right 1 bit
   - Multiplier0 = 0

2. Shift the Product register right 1 bit
3. Shift the Multiplier register right 1 bit

32nd repetition?
- No: < 32 repetitions
- Yes: 32 repetitions

Done
Multipler: Final Version

Example: 0010 x 0011 = 0000 0110

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Step</th>
<th>Multiplicand</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial values</td>
<td>0010</td>
<td>0000 0011</td>
</tr>
<tr>
<td>1</td>
<td>1: Prod = Prod + Mcand</td>
<td>0010</td>
<td>0010 0011</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod</td>
<td></td>
<td>0001 0001</td>
</tr>
<tr>
<td>2</td>
<td>1: Prod = Prod + Mcand</td>
<td>0010</td>
<td>0011 0001</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod</td>
<td></td>
<td>0001 1000</td>
</tr>
<tr>
<td>3</td>
<td>0: no op</td>
<td>0010</td>
<td>0001 1000</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod</td>
<td></td>
<td>0000 1100</td>
</tr>
<tr>
<td>4</td>
<td>0: no op</td>
<td>0010</td>
<td>0000 1100</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod</td>
<td></td>
<td>0000 0110</td>
</tr>
</tbody>
</table>

1. Test Product0
   - Product0 = 1
   - Product0 = 0

1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register

2. Shift the Product register right 1 bit

32nd repetition?
- No: < 32 repetitions
- Yes: 32 repetitions

Done
Signed Multiplication

- Negative numbers: convert and multiply in positive forms, then adjust sign of results
- It turns out that our final version multiplier works for signed numbers; however, sign extension must be applied in shifting signed numbers
Multiplication: Booth Algorithm

- If multiplier contains long strings of 1s: slow down the process
- Booth: replace each string of 1s with an initial subtract when we first see a 1 and then later add when we see the bit after the last 1
  - Example: replace 0011110 by 01000(-1)0
  - For long strings of 1s, Booth algorithm is faster (reduce the number of additions)
- Booth algorithm handles signed numbers as well (in the same way)
- Booth algorithm:

<table>
<thead>
<tr>
<th>Current bit</th>
<th>Bit to the right</th>
<th>Explanation</th>
<th>Example</th>
<th>Replace current bit by</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Beginning of string of 1s</td>
<td>0001111000</td>
<td>-1 (i.e. a subtraction)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Middle of string of 1s</td>
<td>0001111000</td>
<td>0 (i.e. no arith. op)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>End of string of 1s</td>
<td>0001111000</td>
<td>1 (i.e. an addition)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Middle of run of 0s</td>
<td>0001111000</td>
<td>0 (i.e. no arith. op)</td>
</tr>
</tbody>
</table>
Booth Algorithm: Example 1

- Both operands positive: 0010 x 0110 = 0000 1100
  - Append a 0 to the right of multiplier at start
  - Examine 2 bits each stage and perform sub/add/no_op according to Booth algorithm
  - Multiplicand subtracts/adds to the most significant 4 bits of product
  - Note the sign extension in right shift (arithmetic right shift)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Step</th>
<th>Multiplicand</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial values</td>
<td>0010</td>
<td>0000 0110 0</td>
</tr>
<tr>
<td>1</td>
<td>00: no op</td>
<td>0010</td>
<td>0000 0110 0</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod</td>
<td>0010</td>
<td>0000 0011 0</td>
</tr>
<tr>
<td>2</td>
<td>10: Prod = Prod - Mcand</td>
<td>0010</td>
<td>1110 0011 0</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod</td>
<td>0010</td>
<td>1111 0001 1</td>
</tr>
<tr>
<td>3</td>
<td>11: no op</td>
<td>0010</td>
<td>1111 0001 1</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod</td>
<td>0010</td>
<td>1111 1000 1</td>
</tr>
<tr>
<td>4</td>
<td>01: Prod = Prod + Mcand</td>
<td>0010</td>
<td>0001 1000 1</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod</td>
<td>0010</td>
<td>0000 1100 0</td>
</tr>
</tbody>
</table>
Booth Algorithm: Example 2

- One operand positive, one negative: 0010 x 1101 = 1111 1010 --> same procedure (nice!):
  - Append a 0 to the right of multiplier at start
  - Examine 2 bits each stage and perform sub/add/no_op according to Booth algorithm
  - Multiplicand subtracts/adds to the most significant 4 bits of product
  - Note the sign extension in right shift (arithmetic right shift)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Step</th>
<th>Multiplicand</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial values</td>
<td>0010</td>
<td>0000 1101 0</td>
</tr>
<tr>
<td>1</td>
<td>10: Prod = Prod - Mcand</td>
<td>0010</td>
<td>1110 1101 0</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod</td>
<td>0010</td>
<td>1111 0110 1</td>
</tr>
<tr>
<td>2</td>
<td>01: Prod = Prod + Mcand</td>
<td>0010</td>
<td>0001 0110 1</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod</td>
<td>0010</td>
<td>0000 1011 0</td>
</tr>
<tr>
<td>3</td>
<td>10: Prod = Prod - Mcand</td>
<td>0010</td>
<td>1110 1011 0</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod</td>
<td>0010</td>
<td>1111 0101 1</td>
</tr>
<tr>
<td>4</td>
<td>11: no op</td>
<td>0010</td>
<td>1111 0101 1</td>
</tr>
<tr>
<td></td>
<td>Shift right Prod</td>
<td>0010</td>
<td>1111 1010 1</td>
</tr>
</tbody>
</table>
Special purpose registers $Hi$ and $Lo$, each 32 bits $\rightarrow$ 64 bits to contain product

Instructions (pseudoinstructions)
- **mult**: signed multiplication
  - mult $s2, s3$  #$Hi, Lo \leftarrow s2 \times s3$
- **multu**: unsigned multiplication
- **mflo**: to fetch integer 32-bit product (move from Lo)
  - mflo $s1$  #$s1 \leftarrow Lo$
- **mfhi**: used with **mflo** to fetch 64-bit product to registers
- MIPS multiply instructions ignore overflow
Floating Point Numbers

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., 0.00000001; very large numbers, e.g., $3.15576 \times 10^9$
  - Hence we need floating point representation, in addition to fixed point representation that we have already studied

- Floating point representation:
  - sign, exponent, significand (fraction): $(-1)^\text{sign} \times \text{fraction} \times 2^{\text{exponent}}$
  - more bits for fraction gives more accuracy
  - more bits for exponent increases range

- IEEE 754 floating point standard (in virtually every computer):
  - single precision: 1 bit sign, 8 bit exponent, 23 bit fraction (24 bit significand = 1 + fraction) ("1" is implied)
  - double precision: 1 bit sign, 11 bit exponent, 52 bit fraction (53 bit significand)
IEEE 754 Floating-Point Standard

- Single precision:

```
 31  30  23  22  0
  s  8 bit exponent E  23 bit fraction
```

- Double precision:

```
 31  30  20  19  0
  s  11 bit exponent E  20 bit fraction
```

32 bit fraction (continued)
Leading “1” bit of significand is implicit (normalized form), hence significand is actually 24 bits (or 53 bits in double precision)

Fractional portion of the significand represents a fraction between 0 and 1 (each bit from left to right has weight $2^{-1}$, $2^{-2}$, ...)

Since 0 has no leading 1, it is given the reserved exponent value 0 so that hardware won’t attach a leading 1 to it

Exponent is “biased” (biased notation) to make sorting easier

- all 0s is smallest exponent (most negative) all 1s is largest (most positive)
- bias of 127 for single precision and 1023 for double precision

Summary: $(-1)^{\text{sign}} \times (1 + \text{fraction}) \times 2^{\text{exponent} - \text{bias}}$

Example: $-0.75_{\text{ten}}$

- decimal: $-0.75 = -3/4 = -3/2^2$
- binary: $-0.11 = -1.1 \times 2^{-1}$
- floating point: $126-127=-1 \Rightarrow \text{exponent} = 126 = 01111110$
- sign: 1 (negative)
- IEEE single precision: $10111110100000000000000000000000$
Example: 5.0
- decimal: 5
- binary: \( 101 = 1.01 \times 2^2 \)
- floating point: \( 129 - 127 = 2 \Rightarrow \text{exponent} = 129 = 10000001 \)
- sign: 0 (positive)
- IEEE single precision: 0100 0000 1010 0000 0000 0000 0000 0000
Floating-Point Addition

- Steps (example: $0.5 + -0.4375 \Rightarrow 1.000_2 \times 2^{-1} + (-1.110_2 \times 2^{-2})$) (assume 4 bits of precision in significand for the sake of simplicity)
  - **Step 1: Shift right** the significand of the number with the lesser exponent until its exponent matches the larger number
    - $-1.110_2 \times 2^{-2} \Rightarrow -0.111_2 \times 2^{-1}$
  - **Step 2: Add** the significands
    - $1.000_2 \times 2^{-1} + (-0.111_2 \times 2^{-1}) = 0.001_2 \times 2^{-1}$
  - **Step 3: Normalize** the sum, checking for overflow or underflow
    - $1.000_2 \times 2^{-4}$
    - since $127 \geq 4 \geq -126$, there is no overflow or underflow (recall that the pattern of all 0s in exponent is reserved and used to represent 0, pattern of all 1s in exponent represents values & situations outside the scope of normal floating point numbers; hence for single precision, max exponent is $127$ and min exponent is $-126$)
    - the biased exponent would be $-4 + 127 = 123$
  - **Step 4: Round** the sum
    - the sum already fits in 4 bits, so no change to the bits due to rounding
    - final result is $1.000_2 \times 2^{-4}$ (which is 0.0625)
Floating-Point Adder Hardware

- Compare exponents
- Shift smaller number right
- Add
- Normalize
- Round
Floating-Point Multiplication

- Steps (example: $0.5 \times -0.4375 \Rightarrow (1.000_2 \times 2^{-1}) \times (-1.110_2 \times 2^{-2})$)
  - assume 4 bits of precision in significands for the sake of simplicity

- Step 1: Add exponents without bias, then bias the result
  - $-1 + (-2) = -3 \Rightarrow -3 + 127 = 124$

- Step 2: Multiply the significands
  - $1.000_2 \times 1.110_2 = 1.110000_2 \Rightarrow$ limit to 4 bits, so $1.110_2$
  - Hence product is $1.110_2 \times 2^{-3}$

- Step 3: Normalize the product, checking for overflow or underflow
  - $1.110_2 \times 2^{-3}$ already normalized
  - since $127 \geq -3 \geq -126$, or $254 \geq 124$ (biased form) $\geq 1$, there is no overflow or underflow

- Step 4: Round the product
  - the sum already fits in 4 bits, so no change to the bits due to rounding

- Step 5: Produce the sign bit, based on the signs of the operands
  - since the signs of operands differ, sign of product should be negative
  - final result is $-1.110_2 \times 2^{-3}$ (which is $-0.21875$)
MIPS supports IEEE 754 single & double precision formats
MIPS has a separate set of floating-point registers $f0 - f31$, each 32 bits (a double precision is an even-odd pair using even register number as name)

Instructions
- **add.s, add.d**: floating-point addition (single, double)
  - add.s $f2, f4, f6  #$f2 ← $f4 + $f6
- **sub.s, sub.d**: floating-point subtraction
- **mul.s, mul.d, div.s, div.d**: floating-point multiplication & division
- **c.x.s, c.x.d, bclt, bclf**: floating-point compare, branch
- **lwc1, swc1**: load/store 32-bit floating-point number
  - lwc1 $f1, 100($s2)  #$f1 ← Memory[$s2+100]
Floating Point Complexities

- Operations are somewhat more complicated than those of fixed point
- In addition to overflow we can have “underflow”
- Accuracy 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” (NaN)
  - other complexities
- Implementing the standard can be tricky
  - the Pentium bug (floating-point divide flaw) in 1994 (discovered by a math professor) – cost Intel $500 million!
  - another bug in floating-point-to-integer store instruction in Pentium Pro & Pentium II in 1997 – this time Intel corrected it quickly
  - IEEE 754 / 854 (practically the same)
## Data Types

<table>
<thead>
<tr>
<th>C type</th>
<th>Data Transfers</th>
<th>Operations (examples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>lw, sw, lui</td>
<td>add, sub, addi, and, mult</td>
</tr>
<tr>
<td>unsigned int</td>
<td>lw, sw, lui</td>
<td>addu, subu, addiu, and, multu</td>
</tr>
<tr>
<td>char</td>
<td>lb, sb, lui</td>
<td>addu, subu, addiu, and, or</td>
</tr>
<tr>
<td>float</td>
<td>lwc1, swc1</td>
<td>add.s, sub.s, mul.s</td>
</tr>
<tr>
<td>double</td>
<td>lwc1, swc1</td>
<td>add.d, sub.d, mul.d</td>
</tr>
</tbody>
</table>
### Special Symbols in Floating Point

#### Single Precision

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Fraction</th>
<th>Object represented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>± denormalized number</td>
</tr>
<tr>
<td>1 - 254</td>
<td>anything</td>
<td>± floating point number</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>± infinity</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>NaN (not a Number)</td>
</tr>
</tbody>
</table>
Chapter Summary

- Additional MIPS instructions
- The design of an ALU
- Carry lookahead adder
- Multiplication
- Computer arithmetic is constrained by limited precision (accuracy vs range)
- 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).