Five classic components

I am like a control tower
I am like a pack of file folders
I am like a conveyor belt + service stations
I exchange information with outside world
Binary arithmetic

- (Sounds scary)

- So far we studied
  - Instruction set architecture basic
  - MIPS architecture & assembly language

- We will review binary arithmetic algorithms and their implementations

- Binary arithmetic will form the basis for CPU’s datapath design

Binary number representations

- We looked at how to represent a number (in fact the value represented by a number) in binary
  - Unsigned numbers – everything is positive

- We will deal with more complicated cases
  - Negative numbers
  - Real numbers (a.k.a. floating-point numbers)

- The first question: How do we represent a negative number in decimal?
Method 1: sign-magnitude

- This is the same method we use for decimal numbers
- \{\text{sign bit, absolute value (magnitude)}\}
  - Sign bit
    - 0 – positive, 1 – negative
  - Examples, assume 4-bit representation
    - 0000
    - 0011
    - 1001
    - 1111
    - 1000
- Properties
  - Two 0s – a positive 0 and a negative 0?
  - There are equal # of positive and negative numbers

Method 2: one’s complement

- \((2^N - 1) - \text{number}\)
  - \(A + (-A) = 2^N - 1\)
  - Given a number \(A\), its negation is done by \((1111…1111 - A)\)
  - In fact, simple bit-by-bit inversion will give the same-magnitude number with a different sign
  - Examples, assume 4-bit representation
    - 0000
    - 0011
    - 1001
    - 1111
    - 1000
- Properties
  - There are two 0s
  - There are equal # of positive and negative numbers
Method 3: two’s complement

- \((2^N - \text{number})\)
  - \(A + (-A) = 2^N\)
  - Given a number \(A\), its negation is done by \((1111...1111 - A) + 1\)
  - In fact, simple bit-by-bit inversion followed by adding 1 will give the same-magnitude number with a different sign
  - Examples, assume 4-bit representation
    - 0000
    - 0011
    - 1001
    - 1111
    - 1000
  - Properties
    - There is a single 0
    - There are unequal # of positive and negative numbers

Summary

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

- Issues
  - # of zeros
  - Balance
  - Arithmetic algorithm implementation
When \( N = 32 \)

- If we use the two’s complement method
  - \( 0000 0000 0000 0000 0000 0000 0000 0000 = 0 \)
  - \( 0000 0000 0000 0000 0000 0000 0000 0001 = +1 \)
  - \( 0000 0000 0000 0000 0000 0000 0000 0010 = +2 \)
  - ...
  - \( 0111 1111 1111 1111 1111 1111 1111 1110 = +2,147,483,646 \)
  - \( 0111 1111 1111 1111 1111 1111 1111 1111 = +2,147,483,647 \)
  - \( 1000 0000 0000 0000 0000 0000 0000 0000 = -2,147,483,648 \)
  - \( 1000 0000 0000 0000 0000 0000 0000 0001 = -2,147,483,647 \)
  - \( 1000 0000 0000 0000 0000 0000 0000 0010 = -2,147,483,646 \)
  - ...
  - \( 1111 1111 1111 1111 1111 1111 1111 1101 = -3 \)
  - \( 1111 1111 1111 1111 1111 1111 1111 1110 = -2 \)
  - \( 1111 1111 1111 1111 1111 1111 1111 1111 = -1 \)

**Two’s complement operations**

- **Negating**
  - Invert all bits and add 1
  - This operation is equivalent to subtracting the number from \( 2^N \) (why?)

- **Converting an N-bit number into an M-bit number, \( M > N \)**
  - E.g., MIPS 16-bit immediate (it’s a signed number) converted into a 32-bit number
  - This operation is called *sign-extension*
  - E.g., from 4-bit encoding to 8-bit encoding
    - \( 0010 \Rightarrow \ldots \ldots \ldots \ldots \)
    - \( 1010 \Rightarrow \)
  - What is *zero-extension*?
Addition

- We are quite familiar with adding two numbers in decimal
  - What about adding two binary numbers?

- If we use the two’s complement method to represent binary numbers, addition can be done in a straightforward way

Examples (4-bit representation)
- 0010 (2) + 0011 (3) = 0101 (5)
- 1111 (-1) + 0100 (4) = 0011 (3)
- 0111 (7) + 0010 (2) =

- Let’s talk about overflow now…

Overflow

- Because we use a limited number of digits to represent a number, the result of an operation may not fit

- No overflow when
  - We add two numbers with different signs
  - We subtract a number from another number having the same sign

- Overflow detection
  - Adding two positive numbers yields a negative number
  - Adding two negative numbers yields a positive number
  - How about subtraction?
What happens on overflow?

- The CPU can
  - Generate an exception
  - Set a flag in the status register
  - Do nothing

- Languages may have different notions about overflow

- Do we have overflows in the case of unsigned, always positive numbers?
  - Example: addu, addiu, subu

MIPS example

- I looked at the MIPS32 instruction set manual
- ADD, ADDI instructions generate an exception on overflow
- ADDU, ADDIU are silent
Subtraction

- We know how to add
- We know how to negate a number
- We will use the above two known operations to perform subtraction
- \( A - B = A + (-B) \)
- The hardware used for addition can be extended to handle subtraction!
1-bit adder

- We will look at a single-bit adder
  - Will build on this adder to design a 32-bit adder

- 3 inputs
  - A: 1st input
  - B: 2nd input
  - C_in: carry input

- 2 outputs
  - S: sum
  - C_out: carry out

N-bit adder

- An N-bit adder can be constructed with N single-bit adders
  - A carry out generated in a stage is propagated to the next (“ripple-carry adder”)

- 3 inputs
  - A: N-bit, 1st input
  - B: N-bit, 2nd input
  - C_in: carry input

- 2 outputs
  - S: N-bit sum
  - C_out: carry out
**N-bit ripple-carry adder**

A “truth” table will tell us about the operation of a single-bit adder.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Multiplication

- More complicated than addition
  - A straightforward implementation will involve addition and shifting

- A “more complex operation” implies
  - More area (on silicon) and/or
  - More time (more clock cycles or longer clock cycle time)

- Let’s begin from a simple, straightforward method
  - For now, consider only unsigned numbers!

Straightforward algorithm

\[
\begin{array}{c}
01010010 \text{ (multiplicand)} \\
\times \quad 01101101 \text{ (multiplier)} \\
\hline
\end{array}
\]
Hardware design 1

“Straightforward but naïve design”

Hardware design 2

32-bit ALU instead of 64-bit ALU!
Hardware design 3

Example

- Let’s do 0010 × 0110 (2 × 6), unsigned

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Multiplicand</th>
<th>Implementation 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Step</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>initial values</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1: 0 -&gt; no op</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: shift right</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>1: 1 -&gt; product = product + multiplicand</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: shift right</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>1: 1 -&gt; product = product + multiplicand</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: shift right</td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>1: 0 -&gt; no op</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: shift right</td>
</tr>
</tbody>
</table>
Booth’s encoding

- Three symbols to represent a binary number: \( \{1, 0, -1\} \)
- Examples (8-bit encoding)
  - \(-1\)
    - 11111111 (two’s complement)
    - 0000000-1 (Booth’s encoding)
  - 14
    - 00011110 (two’s complement)
    - 000100-10 (Booth’s encoding)

- Bit transitions in number (in two’s complement encoding) show how Booth’s encoding works
  - 0 to 0 (from right to left): 0
  - 0 to 1: -1
  - 1 to 1: 0
  - 1 to 0: 1

Booth’s encoding

- Key point
  - A “1” in the multiplier implies an addition operation
  - If you have many “1”s – that means many addition operations

- Booth’s encoding is useful because it can reduce the number of addition operations you have to perform

- With Booth’s encoding, partial results are obtained by
  - Adding multiplicand
  - Adding 0
  - Subtracting multiplicand
**Booth’s algorithm in action**

- Let’s do $0010 \times 1101 \ (2 \times -3)$

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Multiplicand</th>
<th>Booth’s algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0010</td>
<td>Step</td>
</tr>
<tr>
<td>0</td>
<td>initial values</td>
<td>0000 1101 0</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>10 -&gt; product = product – multiplicand</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shift right</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>01 -&gt; product = product + multiplicand</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shift right</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>10 -&gt; product = product – multiplicand</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shift right</td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>11 -&gt; no op</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shift right</td>
</tr>
</tbody>
</table>

---

**Booth’s algorithm in action**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Multiplicand</th>
<th>Booth’s algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0010</td>
<td>Step</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>