Basic Arithmetic and the ALU

• Earlier in the semester
  • Number representations, 2’s complement, unsigned
  • Addition/Subtraction
  • Add/Sub ALU
    • Full adder, ripple carry, subtraction
    • Carry-lookahead addition
  • Logical operations
    • and, or, xor, nor, shifts
  • Overflow
Basic Arithmetic and the ALU

• Now
  – Integer multiplication
    • Booth’s algorithm
• This is not crucial for the project
Multiplication

• Flashback to 3rd grade
  – Multiplier
  – Multiplicand
  – Partial products
  – Final sum
• Base 10: 8 x 9 = 72
  – PP: 8 + 0 + 0 + 64 = 72
• How wide is the result?
  – log(n x m) = log(n) + log(m)
  – 32b x 32b = 64b result
Array Multiplier

- Adding all partial products simultaneously using an array of basic cells

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c} & & & & & & & & & & & & \\ 1 & 0 & 0 & 0 & 0 & & & & & & & & \\ x & 1 & 0 & 0 & 1 & & & & & & & & \\ 1 & 0 & 0 & 0 & 0 & & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & & & & & & & & \\ 1 & 0 & 0 & 0 & 0 & & & & & & & & \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & & & & & \\ \end{array} \]

Full Adder

\[ S_{in} \quad C_{in} \quad A_i \quad B_j \]

\[ \begin{array}{c|c|c|c|c} & & & & \\ & & \text{Full Adder} & & \\ & C_{out} \quad S_{out} & & \\ & & \text{A}_i, \text{B}_j & & \\ \end{array} \]
16-bit Array Multiplier

- Conceptually straightforward
- Fairly expensive hardware, integer multiplies relatively rare
- Most used in array address calc: replace with shifts

[Source: J. Hayes, Univ. of Michigan]
Instead: Multicycle Multipliers

• Combinational multipliers
  – Very hardware-intensive
  – Integer multiply relatively rare
  – Not the right place to spend resources

• Multicycle multipliers
  – Iterate through bits of multiplier
  – Conditionally add shifted multiplicand
Multiplier

\[
\begin{array}{c}
1000 \\
\times 1001 \\
\hline
1000 \\
0000 \\
0000 \\
1000 \\
\end{array}
\]

Multiplicand

Shift left

64 bits

64-bit ALU

Product

Write

Control test

Multiplier Shift right

32 bits
Multiplied

10000
\times 1001

\underline{10000}
0000
0000
1000
\underline{10010000}

1. Test Multiplier0

2. Shift the Multiplicand register left 1 bit

3. Shift the Multiplier register right 1 bit

32nd repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done

Start

Multiplier0 = 1
Multiplier0 = 0

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

No: < 32 repetitions

Yes: 32 repetitions

Done
Multiplier Improvements

• Do we really need a 64-bit adder?
  – No, since low-order bits are not involved
  – Hence, just use a 32-bit adder
    • Shift product register right on every step
• Do we really need a separate multiplier register?
  – No, since low-order bits of 64-bit product are initially unused
  – Hence, just store multiplier there initially
Multiplier

Multiplicand

32 bits

32-bit ALU

Product

Shift right

Write

Control
test

\[ \begin{array}{c}
1000 \\
\times 1001 \\
\hline
1000 \\
0000 \\
0000 \\
1000 \\
1001000
\end{array} \]
Multiplier

1. Test
Product0 = 1

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

Product0 = 0

2. Shift the Product register right 1 bit

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

Done

\[
\begin{array}{c}
10000 \\
\times 1001 \\
10000 \\
00000 \\
00000 \\
10000 \\
\hline
10010000 \\
\end{array}
\]
Signed Multiplication

• Recall
  – For \( p = a \times b \), if \( a < 0 \) or \( b < 0 \), then \( p < 0 \)
  – If \( a < 0 \) and \( b < 0 \), then \( p > 0 \)
  – Hence \( \text{sign}(p) = \text{sign}(a) \oplus \text{sign}(b) \)

• Hence
  – Convert multiplier, multiplicand to positive number with \((n-1)\) bits
  – Multiply positive numbers
  – Compute sign, convert product accordingly

• Or,
  – Perform sign-extension on shifts for prev. design
  – Right answer falls out
Booth’s Encoding

- Recall grade school trick
  - When multiplying by 9:
    - Multiply by 10 (easy, just shift digits left)
    - Subtract once
  - E.g.
    - $123454 \times 9 = 123454 \times (10 - 1) = 1234540 - 123454$
    - Converts addition of six partial products to one shift and one subtraction
- Booth’s algorithm applies same principle
  - Except no ‘9’ in binary, just ‘1’ and ‘0’
  - So, it’s actually easier!
Booth’s Encoding

• Search for a run of ‘1’ bits in the multiplier
  – E.g. ‘0110’ has a run of 2 ‘1’ bits in the middle
  – Multiplying by ‘0110’ (6 in decimal) is equivalent to multiplying by 8 and subtracting twice, since $6 \times m = (8 - 2) \times m = 8m - 2m$

• Hence, iterate right to left and:
  – Subtract multiplicand from product at first ‘1’
  – Add multiplicand to product after last ‘1’
  – Don’t do either for ‘1’ bits in the middle
## Booth’s Algorithm

<table>
<thead>
<tr>
<th>Current bit</th>
<th>Bit to right</th>
<th>Explanation</th>
<th>Example</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Begins run of ‘1’</td>
<td>00001111000</td>
<td>Subtract</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Middle of run of ‘1’</td>
<td>00001111000</td>
<td>Nothing</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>End of a run of ‘1’</td>
<td>000001111000</td>
<td>Add</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Middle of a run of ‘0’</td>
<td>00001111000</td>
<td>Nothing</td>
</tr>
</tbody>
</table>
Booth’s Encoding

• Really just a new way to encode numbers
  – Normally positionally weighted as $2^n$
  – With Booth, each position has a sign bit
  – Can be extended to multiple bits

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>Binary</td>
</tr>
<tr>
<td>+1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td></td>
<td>1-bit Booth</td>
</tr>
<tr>
<td>+2</td>
<td></td>
<td>-2</td>
<td></td>
<td></td>
<td>2-bit Booth</td>
</tr>
</tbody>
</table>


2-bits/cycle Booth Multiplier

• For every pair of multiplier bits
  – If Booth’s encoding is ‘-2’
    • Shift multiplicand left by 1, then subtract
  – If Booth’s encoding is ‘-1’
    • Subtract
  – If Booth’s encoding is ‘0’
    • Do nothing
  – If Booth’s encoding is ‘1’
    • Add
  – If Booth’s encoding is ‘2’
    • Shift multiplicand left by 1, then add
# 2 bits/cycle Booth’s

<table>
<thead>
<tr>
<th>Current</th>
<th>Previous</th>
<th>Operation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>+0; shift 2</td>
<td>[00] =&gt; +0, [00] =&gt; +0; 2x(+0)+(+0)=+0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>+M; shift 2</td>
<td>[00] =&gt; +0, [01] =&gt; +M; 2x(+0)+(+M)=+M</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>+M; shift 2</td>
<td>[01] =&gt; +M, [10] =&gt; -M; 2x(+M)+(-M)=+M</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>+2M; shift 2</td>
<td>[01] =&gt; +M, [11] =&gt; +0; 2x(+M)+(+0)=+2M</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>-2M; shift 2</td>
<td>[10] =&gt; -M, [00] =&gt; +0; 2x(-M)+(+0)=-2M</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>-M; shift 2</td>
<td>[10] =&gt; -M, [01] =&gt; +M; 2x(-M)+(+M)=-M</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>-M; shift 2</td>
<td>[11] =&gt; +0, [10] =&gt; -M; 2x(+0)+(-M)=-M</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>+0; shift 2</td>
<td>[11] =&gt; +0, [11] =&gt; +0; 2x(+0)+(+0)=+0</td>
</tr>
</tbody>
</table>

*1 bit Booth Table*
Booth’s Example

• Negative multiplicand:
  
  \(-6 \times 6 = -36\)

  1010 x 0110, 0110 in Booth’s encoding is +0-0

  Hence:

<table>
<thead>
<tr>
<th>1111 1010</th>
<th>x 0</th>
<th>0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0100</td>
<td>x (-1)</td>
<td>0000 1100</td>
</tr>
<tr>
<td>1110 1000</td>
<td>x 0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1101 0000</td>
<td>x +1</td>
<td>1101 0000</td>
</tr>
</tbody>
</table>

Final Sum: 1101 1100 (-36)
Booth’s Example

- Negative multiplier: 
  \[-6 \times -2 = 12\]
  
  \[1010 \times 1110, 1110 \text{ in Booth’s encoding is 00-0}\]

  Hence:

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Multiplier</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 1010</td>
<td>x 0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1111 0100</td>
<td>x -1</td>
<td>0000 1100</td>
</tr>
<tr>
<td>1110 1000</td>
<td>x 0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1101 0000</td>
<td>x 0</td>
<td>0000 0000</td>
</tr>
</tbody>
</table>

Final Sum: 0000 1100 (12)
Summary

• Integer multiply
  – Combinational
  – Multicycle
  – Booth’s algorithm
Basic Arithmetic and the ALU

• Integer division
  – Restoring, non-restoring

• These are not crucial for the project
Integer Division

• Again, back to 3rd grade ($74 \div 8 = 9 \text{ rem } 2$)

```
  1 0 0 1  Quotient
1 0 0 0 0   Dividend
  1 0 0 0 0
- 1 0 0 0 0
    1 0

  1 0 1
1 0 1 0
- 1 0 0 0 0
    1 0  Remainder
```
Integer Division

• How does hardware know if division fits?
  – Condition: if remainder ≥ divisor
  – Use subtraction: (remainder − divisor) ≥ 0
• OK, so if it fits, what do we do?
  – Remainder_{n+1} = Remainder_n − divisor
• What if it doesn’t fit?
  – Have to restore original remainder
• Called restoring division
Integer Division

1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register

Remainder ≥ 0

Test Remainder

Remainder < 0

2a. Shift the Quotient register to the left, setting the new rightmost bit to 1

2b. Restore the original value by adding the Divisor register to the Remainder register and place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0

3. Shift the Divisor register right 1 bit

33rd repetition?

No: < 33 repetitions

Yes: 33 repetitions

Done

**Example:**

Divisor: 1000

Dividend: 10010

Quotient: 1001

Remainder: 10
Integer Division

\[
\begin{array}{c}
1001 \quad \text{Quotient} \\
\hline
1000 & 1010 \quad \text{Dividend} \\
\hline
1000 & 101 \\
101 & 1010 \\
-1000 & \\
\hline
10 \quad \text{Remainder}
\end{array}
\]
Division Improvements

• Skip first subtract
  – Can’t shift ‘1’ into quotient anyway
  – Hence shift first, then subtract
    • Undo extra shift at end

• Hardware similar to multiplier
  – Can store quotient in remainder register
  – Only need 32b ALU
    • Shift remainder left vs. divisor right
Improved Divider

1. Shift the Remainder register left 1 bit

2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register

3a. Shift the Remainder register to the left, setting the new rightmost bit to 1

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0

32nd repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done. Shift left half of Remainder right 1 bit
Improved Divider

32-bit ALU

Remainder

Shift right
Shift left
Write

Control test

Divisor

32 bits

64 bits

32-bit ALU
Further Improvements

• Division still takes:
  – 2 ALU cycles per bit position
    • 1 to check for divisibility (subtract)
    • One to restore (if needed)

• Can reduce to 1 cycle per bit
  – Called non-restoring division
  – Avoids restore of remainder when test fails
Non-restoring Division

• Consider remainder to be restored:
  \[ R_i = R_{i-1} - d < 0 \]
  – Since \( R_i \) is negative, we must restore it, right?
  – Well, maybe not. Consider next step \( i+1 \):
  \[ R_{i+1} = 2 \times (R_i) - d = 2 \times (R_i - d) + d \]

• Hence, we can compute \( R_{i+1} \) by not restoring \( R_i \), and adding \( d \) instead of subtracting \( d \)
  – Same value for \( R_{i+1} \) results

• Throughput of 1 bit per cycle
### NR Division Example

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Step</th>
<th>Divisor</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial values</td>
<td>0010</td>
<td>0000 0111</td>
</tr>
<tr>
<td></td>
<td>Shift rem left 1</td>
<td>0010</td>
<td>0000 1110</td>
</tr>
<tr>
<td>1</td>
<td>2: Rem = Rem - Div</td>
<td>0010</td>
<td>1110 1110</td>
</tr>
<tr>
<td></td>
<td>3b: Rem &lt; 0 (add next), sll 0</td>
<td>0010</td>
<td>1101 1100</td>
</tr>
<tr>
<td>2</td>
<td>2: Rem = Rem + Div</td>
<td>0010</td>
<td>1111 1100</td>
</tr>
<tr>
<td></td>
<td>3b: Rem &lt; 0 (add next), sll 0</td>
<td>0010</td>
<td>1111 1000</td>
</tr>
<tr>
<td>3</td>
<td>2: Rem = Rem + Div</td>
<td>0010</td>
<td>0001 1000</td>
</tr>
<tr>
<td></td>
<td>3a: Rem &gt; 0 (sub next), sll 1</td>
<td>0010</td>
<td>0011 0001</td>
</tr>
<tr>
<td>4</td>
<td>Rem = Rem - Div</td>
<td>0010</td>
<td>0001 0001</td>
</tr>
<tr>
<td></td>
<td>Rem &gt; 0 (sub next), sll 1</td>
<td>0010</td>
<td>0010 0011</td>
</tr>
<tr>
<td></td>
<td>Shift Rem right by 1</td>
<td>0010</td>
<td>0001 0011</td>
</tr>
</tbody>
</table>
Summary

• Integer dividers covered
  – Multicycle restoring
  – Non-restoring

• Other approaches
  – SRT division [Sweeney, Robertson, Tocher] uses lookup tables
    • Famous Intel fdiv bug caused by incomplete table
  – Newton-Raphson method
    • Estimate reciprocal, iterate to refine, multiply
  – Beyond the scope of this course
Basic Arithmetic and the ALU

• Now
  – Floating point representation
  – Floating point addition, multiplication

• These are not crucial for the project
Floating Point

- Want to represent larger range of numbers
  - Fixed point (integer): $-2^{n-1} \ldots (2^{n-1} - 1)$
- How? Sacrifice precision for range by providing exponent to shift relative weight of each bit position
- Similar to scientific notation:
  - $3.14159 \times 10^{23}$
- Cannot specify every discrete value in the range, but can span much larger range
Floating Point

• Still use a fixed number of bits
  – Sign bit $S$, exponent $E$, significand $F$
  – Value: $(-1)^S \times F \times 2^E$

• IEEE 754 standard

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Exponent</th>
<th>Significand</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single precision</td>
<td>32b</td>
<td>8b</td>
<td>23b</td>
<td>$2 \times 10^{+/-38}$</td>
</tr>
<tr>
<td>Double precision</td>
<td>64b</td>
<td>11b</td>
<td>52b</td>
<td>$2 \times 10^{+/-308}$</td>
</tr>
</tbody>
</table>
Floating Point Exponent

• Exponent specified in *biased* or *excess* notation

• Why?
  – To simplify sorting
  – Sign bit is MSB to ease sorting
  – 2’s complement exponent:
    • Large numbers have positive exponent
    • Small numbers have negative exponent
  – Sorting does not follow naturally
Excess or Biased Exponent

<table>
<thead>
<tr>
<th>Exponent</th>
<th>2’s Compl</th>
<th>Excess-127</th>
</tr>
</thead>
<tbody>
<tr>
<td>-127</td>
<td>1000 0001</td>
<td>0000 0000</td>
</tr>
<tr>
<td>-126</td>
<td>1000 0010</td>
<td>0000 0001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>+127</td>
<td>0111 1111</td>
<td>1111 1110</td>
</tr>
</tbody>
</table>

- **Value**: \((-1)^S \times F \times 2^{(E-bias)}\)
  - SP: bias is 127
  - DP: bias is 1023
Floating Point Normalization

• S,E,F representation allows more than one representation for a particular value, e.g.
  \[ 1.0 \times 10^5 = 0.1 \times 10^6 = 10.0 \times 10^4 \]
  – This makes comparison operations difficult
  – Prefer to have a single representation
• Hence, normalize by convention:
  – Only one digit to the left of the floating point
  – In binary, that digit must be a 1
    • Since leading ‘1’ is implicit, no need to store it
    • Hence, obtain one extra bit of precision for free
FP Overflow/Underflow

• FP Overflow
  – Analogous to integer overflow
  – Result is too big to represent
  – Means exponent is too big

• FP Underflow
  – Result is too small to represent
  – Means exponent is too small (too negative)

• Both can raise an exception under IEEE754
## IEEE754 Special Cases

<table>
<thead>
<tr>
<th>Single Precision</th>
<th>Double Precision</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent</td>
<td>Significand</td>
<td>Exponent</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>0</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>1-2046</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>2047</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>2047</td>
</tr>
</tbody>
</table>
FP Rounding

• Rounding is important
  – Small errors accumulate over billions of ops

• FP rounding hardware helps
  – Compute extra guard bit beyond 23/52 bits
  – Further, compute additional round bit beyond that
    • Multiply may result in leading 0 bit, normalize shifts guard bit into product, leaving round bit for rounding
  – Finally, keep sticky bit that is set whenever ‘1’ bits are “lost” to the right
    • Differentiates between 0.5 and 0.500000000001
Floating Point Addition

• Just like grade school
  – First, align decimal points
  – Then, add significands
  – Finally, normalize result

• Example

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9.997 x 10²</td>
<td>9.997000 x 10²</td>
</tr>
<tr>
<td>4.631 x 10⁻¹</td>
<td>0.004631 x 10²</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>10.001631 x 10²</td>
</tr>
<tr>
<td>Normalized</td>
<td>1.0001631 x 10³</td>
</tr>
</tbody>
</table>
FP Adder

- Compare exponents
- Shift smaller number right
- Add
- Normalize
- Round
FP Multiplication

• Sign: \( P_s = A_s \ xor \ B_s \)

• Exponent: \( P_E = A_E + B_E \)
  – Due to bias/excess, must subtract bias
    \[
    e = e_1 + e_2 \\
    E = e + 1023 = e_1 + e_2 + 1023 \\
    E = (E_1 - 1023) + (E_2 - 1023) + 1023 \\
    E = E_1 + E_2 - 1023
    \]

• Significand: \( P_F = A_F \ x B_F \)
  – Standard integer multiply (23b or 52b + g/r/s bits)
  – Use Wallace tree of CSAs to sum partial products
FP Multiplication

- Compute sign, exponent, significand
- Normalize
  - Shift left, right by 1
- Check for overflow, underflow
- Round
- Normalize again (if necessary)
Summary

- Floating point representation
  - Normalization
  - Overflow, underflow
  - Rounding
- Floating point add
- Floating point multiply