10. Datapath Design

• Last module:
  – Adder circuits
  – Simple adders
  – Fast addition

• This module
  – Comparators
  – Shifters
  – Multi-input Adders
  – Multipliers
Floating Point (IEEE 754-1985)

Special values NaN, $\infty$, $-\infty$

Rounds to nearest by default, but three other rounding modes
“Halfway” result found to nearest even floating point number
“Denormal” numbers to represent results of computations
$< 1.0 \times 2^{E_{\text{min}}}$

Sophisticated facilities for handling exceptions
Area @ 55nm

- FP-ADD: 29067; 4 cycles
- FP-MUL: 48595; 7 cycles
- FP-DIV: 11850; 61 cycles
- INT-ALU: 4963; 1 cycle
- INT-MUL: 32802; 5 cycle
# Latency

<table>
<thead>
<tr>
<th>processor</th>
<th>ALU</th>
<th>FP add</th>
<th>FP mult</th>
<th>FP div single</th>
<th>FP div double</th>
</tr>
</thead>
<tbody>
<tr>
<td>ULTRA-Sparc 3</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>17(15)</td>
<td>20(18)</td>
</tr>
<tr>
<td>Pentium 3</td>
<td>1</td>
<td>3(1)</td>
<td>5(2)</td>
<td>17(17)</td>
<td>32(32)</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>1</td>
<td>5(1)</td>
<td>7(2)</td>
<td>23(23)</td>
<td>38(38)</td>
</tr>
<tr>
<td>Itanium</td>
<td>1</td>
<td>5(1)</td>
<td>5(1)</td>
<td>30+(11)*</td>
<td>40+(13)*</td>
</tr>
<tr>
<td>AMD Athlon</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>16(13)</td>
<td>20(17)</td>
</tr>
<tr>
<td>Power3</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>17(13)</td>
<td>21(17)</td>
</tr>
<tr>
<td>Motorola G4</td>
<td>1</td>
<td>5(1)</td>
<td>5(1)</td>
<td>21(21)</td>
<td>35(35)</td>
</tr>
<tr>
<td>Alpha 21064</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>34(34)</td>
<td>63(63)</td>
</tr>
<tr>
<td>Alpha 21164</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>19(19)</td>
<td>31(31)</td>
</tr>
<tr>
<td>Alpha 21264/21364</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>12(9)</td>
<td>15(12)</td>
</tr>
<tr>
<td>R80000</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>14(11)</td>
<td>20(17)</td>
</tr>
<tr>
<td>R120000</td>
<td>1</td>
<td>2(1)</td>
<td>2(1)</td>
<td>14(12)</td>
<td>21(19)</td>
</tr>
</tbody>
</table>
## Energy

### Datapath Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy (fJ)</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit addition</td>
<td>520</td>
<td>1×</td>
</tr>
<tr>
<td>16-bit multiply</td>
<td>2,200</td>
<td>4.2×</td>
</tr>
<tr>
<td>32-bit pipeline register</td>
<td>330</td>
<td>0.63×</td>
</tr>
</tbody>
</table>

### Embedded RISC Processor

<table>
<thead>
<tr>
<th>Component</th>
<th>Energy (fJ)</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register File (32 entries 2R+1W)</td>
<td>250</td>
<td>0.48×</td>
</tr>
<tr>
<td>32-bit read</td>
<td>470</td>
<td>0.90×</td>
</tr>
<tr>
<td>Data Cache (2 KB 4-way set associative)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit load</td>
<td>3,540</td>
<td>6.8×</td>
</tr>
<tr>
<td>32-bit store</td>
<td>3,530</td>
<td>6.8×</td>
</tr>
<tr>
<td>miss</td>
<td>1,410</td>
<td>2.7×</td>
</tr>
<tr>
<td>Instruction Cache (2 KB 4-way set associative)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit fetch</td>
<td>3,500</td>
<td>6.8×</td>
</tr>
<tr>
<td>miss</td>
<td>1,410</td>
<td>2.7×</td>
</tr>
<tr>
<td>128-bit refill</td>
<td>9,710</td>
<td>19×</td>
</tr>
</tbody>
</table>
Comparators

- 0’s detector: \( A = 00\ldots000 \)
- 1’s detector: \( A = 11\ldots111 \)
- Equality comparator: \( A = B \)
- Magnitude comparator: \( A < B \)
1’s & 0’s Detectors

- 1’s detector: N-input AND gate
- 0’s detector: NOTs + 1’s detector (N-input NOR)
Equality Comparator

- Check if each bit is equal (XNOR, or "equality gate")
- 1’s detect on bitwise equality

\[
\begin{align*}
A[3] & \quad (B[3]) \\
A[2] & \quad (B[2]) \\
A[1] & \quad (B[1]) \\
A[0] & \quad (B[0]) \\
A = B &
\end{align*}
\]
Magnitude Comparator

- Compute B-A and look at sign
- B-A = B + ~A + 1
- For unsigned numbers, carry out is sign bit
Signed vs. Unsigned

- For signed numbers, comparison is harder
  - C: carry out
  - Z: zero (all bits of A-B are 0)
  - N: negative (MSB of result)
  - V: overflow (inputs had different signs, output sign ≠ B)

### Table 10.4  Magnitude comparison

<table>
<thead>
<tr>
<th>Relation</th>
<th>Unsigned Comparison</th>
<th>Signed Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = B</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>Z̅</td>
<td>Z̅</td>
</tr>
<tr>
<td>A &lt; B</td>
<td>C + Z̅</td>
<td>(N ⊕ V) + Z̅</td>
</tr>
<tr>
<td>A &gt; B</td>
<td>C̅</td>
<td>(N ⊕ V)</td>
</tr>
<tr>
<td>A ≤ B</td>
<td>C</td>
<td>(N ⊕ V̅)</td>
</tr>
<tr>
<td>A ≥ B</td>
<td>C̅ + Z</td>
<td>(N ⊕ V) + Z</td>
</tr>
</tbody>
</table>
Shifters

• Logical Shift:
  – Shifts number left or right and fills with 0’s
    • 1011 LSR 1 = 0101    1011 LSL1 = 0110

• Arithmetic Shift:
  – Shifts number left or right. Rt shift sign extends
    • 1011 ASR1 = 1101    1011 ASL1 = 0110

• Barrel Shift (Rotate):
  – Shifts number left or right and fills with lost bits
    • 1011 ROR1 = 1101    1011 ROL1 = 0111
Funnel Shifter

- A funnel shifter can do all six types of shifts
- Selects N-bit field Y from 2N-bit input
  - Shift by k bits (0 ≤ k < N)

\[
\begin{array}{c}
2N-1 \\
N-1 \\
0 \\
\hline
\quad B \\
\quad C \\
Y
\end{array}
\]

offset + N-1

offset
### Funnel Shifter Operation

<table>
<thead>
<tr>
<th>Shift Type</th>
<th>$B$</th>
<th>$C$</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Right</td>
<td>0...0</td>
<td>$A_{N-1}...A_0$</td>
<td>$k$</td>
</tr>
<tr>
<td>Logical Left</td>
<td>$A_{N-1}...A_0$</td>
<td>0...0</td>
<td>$N-k$</td>
</tr>
<tr>
<td>Arithmetic Right</td>
<td>$A_{N-1}...A_{N-1}$</td>
<td>$A_{N-1}...A_0$</td>
<td>$k$</td>
</tr>
<tr>
<td>(sign extension)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Left</td>
<td>$A_{N-1}...A_0$</td>
<td>0</td>
<td>$N-k$</td>
</tr>
<tr>
<td>Rotate Right</td>
<td>$A_{N-1}...A_0$</td>
<td>$A_{N-1}...A_0$</td>
<td>$k$</td>
</tr>
<tr>
<td>Rotate Left</td>
<td>$A_{N-1}...A_0$</td>
<td>$A_{N-1}...A_0$</td>
<td>$N-k$</td>
</tr>
</tbody>
</table>

- Computing $N-k$ requires an adder
Simplified Funnel Shifter

- Optimize down to 2N-1 bit input

<table>
<thead>
<tr>
<th>Shift Type</th>
<th>Z</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Right</td>
<td>0..0, ( A_{N-1}...A_0 )</td>
<td>( k )</td>
</tr>
<tr>
<td>Logical Left</td>
<td>( A_{N-1}...A_0, 0..0 )</td>
<td>( \overline{k} )</td>
</tr>
<tr>
<td>Arithmetic Right</td>
<td>( A_{N-1}...A_{N-1}, A_{N-1}...A_0 )</td>
<td>( k )</td>
</tr>
<tr>
<td>Arithmetic Left</td>
<td>( A_{N-1}...A_0, 0..0 )</td>
<td>( \overline{k} )</td>
</tr>
<tr>
<td>Rotate Right</td>
<td>( A_{N-2}...A_0, A_{N-1}...A_0 )</td>
<td>( k )</td>
</tr>
<tr>
<td>Rotate Left</td>
<td>( A_{N-1}...A_0, A_{N-1}..A_1 )</td>
<td>( \overline{k} )</td>
</tr>
</tbody>
</table>
Funnel Shifter Design 1

- N N-input multiplexers
  - Use 1-of-N hot select signals for shift amount
  - nMOS pass transistor design ($V_t$ drops!)
Funnel Shifter Design 2

- Log N stages of 2-input MUXes
  - No select decoding needed
Multi-input Adders

- Suppose we want to add $k$ $N$-bit words
  - Ex: $0001 + 0111 + 1101 + 0010 = 10111$

- Straightforward solution: $k-1$ $N$-input CPAs
  - Large and slow
Carry Save Addition

- Full adder sums 3 inputs, produces 2 outputs
  - Carry output has twice weight of sum output
- N full adders in parallel: carry save adder
  - Produce N sums and N carry outs
CSA Application

- Use k-2 stages of CSAs
  - Keep result in carry-save redundant form
- Final CPA computes actual result
Multiplication

• Example:

\[
\begin{array}{c}
1100 \quad : \quad 12_{10} \\
0101 \quad : \quad 5_{10} \\
\hline
1100 \\
0000 \\
1100 \\
0000 \\
\hline
00111100 \quad : \quad 60_{10}
\end{array}
\]

- multiplicand
- multiplier
- partial products
- product

• M x N-bit multiplication
  - Produce N M-bit partial products
  - Sum these to produce M+N-bit product
General Form

- **Multiplicand:** \( Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0) \)
- **Multiplier:** \( X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0) \)
- **Product:** \( P = \left( \sum \right) \sum \sum \sum \)
Dot Diagram

• Each dot represents a bit
Array Multiplier

\[ y_3 y_2 y_1 y_0 \]
\[ x_0 x_1 x_2 x_3 \]
\[ p_0 p_1 p_2 p_3 p_4 p_5 p_6 p_7 \]

CSA Array

CPA

\[ \text{critical path} \]

\[ \text{Cin} \]

\[ \text{Cout} \]

\[ \text{Sout} \]

\[ A \]

\[ B \]

\[ \text{Sin} \]

\[ \text{Cin} \]

\[ \text{Cout} \]

\[ \text{Sout} \]
Rectangular Array

- Squash array to fit rectangular floorplan

![Rectangular Array Diagram]
Fewer Partial Products

- Array multiplier requires N partial products.
- If we looked at groups of r bits, we could form N/r partial products.
  - Faster and smaller?
  - Called radix-$2^r$ encoding
- Ex: $r = 2$: look at pairs of bits
  - Form partial products of 0, Y, 2Y, 3Y
  - First three are easy, but 3Y requires adder 😞
Booth Encoding

- Instead of 3Y, try \(-Y\), then increment next partial product to add 4Y
- Similarly, for 2Y, try \(-2Y + 4Y\) in next partial product

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Partial Product</th>
<th>Booth Selects</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_{2i+1})</td>
<td>(x_{2i})</td>
<td>(x_{2i-1})</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Booth Hardware

- Booth encoder generates control lines for each PP
  - Booth selectors choose PP bits
Advanced Multiplication

• Signed vs. unsigned inputs
• Higher radix Booth encoding
• Array vs. tree CSA networks
• Serial Multiplication
Serial Multiplication

- Lower area at expense of speed
  - signal processing on bit streams
- Delay for $n \times n$ multiply
  - $2n$ bit product with $2n$ bit delay
  - Additional $n$-bit delay to shift $n$ bits
  - Total delay of $3n$ bits
- **Pipelined multiplier**: possible to produce a new $2n$ bit product every $2n$ bit times after initial $n$ bit delay
  - Only interest in high-order bits: $n$ bit delay for $n$ bit product
Pipeline multiplier accumulates partial product sums starting with the least significant partial product (result is n-bit number which is truncated to n-1 bits before the next partial product).
Division

• To divide A by B
  – Shift P and A one bit left
  – Subtract B from P, put the result back
  – If result is negative, additional steps, set low order bits of A to 0, otherwise to 1
  – “restoring” or “non-restoring” division to fix negative result
SRT Division

- Divide A by B (n-bits) (view numbers as fractions between $\frac{1}{2}$ and 1)

1. If B has k leading 0s when expressed using n bits, shift all registers by k bits

2. For i = 0 to (n-1)
   1. If top 3 bits of P equal, set $q_i = 0$, shift (P,A) one bit left
   2. If top 3 bits of P not all equal, and P negative, set $q_i = -1$, (written as $\overline{1}$, shift (P,A) one bit left and add B
   3. Otherwise, set $q_i = 1$, shift (P,A) one bit left, subtract B

3. If the final remainder is negative, correct by adding B, correct quotient by subtracting 1; finally, shift remainder k bits right

4. Radix-4 SRT algorithm used in Pentium chip
Iterative Division

- Newton’s iteration: finding the 0 of a function
  - starting from a guess for the 0, approximate function by its tangent at the guess, form new guess based on where tangent has a 0

- Goldschmidt’s method
  - To compute $a/b$, iteratively, multiply both numerator and denominator by $r$ with $r \cdot b = 1$
  - Set $x_0 = a$, $y_0 = b$ and write $b = 1 - \delta$ where $|\delta| < 1$
  - If we pick $r_0 = 1 + \delta$, then $y_1 = r_0 y_0 = 1 - \delta^2$
  - Next, pick $r_1 = 1 + \delta^2$, etc., and $y_i \rightarrow 1$
  - Used in the TI 8847 chip
Sign Extension

- Partial products can be negative
  - Require sign extension, which is cumbersome
  - High fanout on most significant bit
Simplified Sign Extension

- Sign bits are either all 0’s or all 1’s
  - Note that all 0’s is all 1’s + 1 in proper column
  - Use this to reduce loading on MSB
Even Simpler Sign Extension

- No need to add all the 1’s in hardware
  - Precompute the answer!