Administrativia

- New homework to be posted this weekend
  - Last one to be graded
- Project Phase 3 Launched Next We
  - Some insights today
Class Material

- Last lecture
  - Adders
- Today’s lecture
  - Multipliers
  - Introduction to Memory
- Reading (Ch 11)
**Binary Multiplication**

\[
\begin{array}{c}
1 & 0 & 1 & 0 & 1 & 0 \\
\times & 1 & 0 & 1 & 1 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
+ 1 & 0 & 1 & 0 & 1 & 0 \\
\hline
1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
\end{array}
\]

- **Multiplicand**
- **Multiplier**
- **Partial products**
- **Result**

---

**Binary Multiplication**

\[
Z = \tilde{X} \times Y = \sum_{k=0}^{M-N-1} Z_k 2^k
\]

\[
- \left( \sum_{i=0}^{M-1} X_i 2^i \right) \left( \sum_{j=0}^{N-1} Y_j 2^j \right)
\]

\[
\left( \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} X_i Y_j 2^{i+j} \right)
\]

**with**

\[
X = \sum_{i=0}^{M-1} X_i 2^i
\]

\[
Y = \sum_{j=0}^{N-1} Y_j 2^j
\]
The Array Multiplier

The M-by-N Array Multiplier: Critical Path

\[ t_{\text{mul}} \approx [(M-1) + (N-2)] \cdot t_{\text{carry}} + (N-1) \cdot t_{\text{sum}} + t_{\text{und}} \]
Transmission-Gate Full Adder

Balanced $t_{\text{sum}}$ and $t_{\text{carry}}$

Carry-Save Multiplier

\[ t_{\text{multi}} = t_{\text{and}} + (N - 1) \cdot t_{\text{carry}} + t_{\text{merge}} \]
Multiplier Floorplan

Wallace-Tree Multiplier
Wallace-Tree Multiplier

Partial products

First stage

Bit position

Second stage

Final adder

(a)

(b)

(c)

(d)
Multipliers – Summary

- Optimization constraints different than in binary adder
  - Once again:
    - Need to identify critical path
    - And find ways to use parallelism to reduce it

- Other possible techniques
  - Logarithmic versus linear (Wallace Tree Mult)
  - Data encoding (Booth)
  - Pipelining

First glimpse at system level optimization

The Binary Shifter

![Binary Shifter Diagram]
The Barrel Shifter

Area Dominated by Wiring

4x4 Barrel Shifter

Width_{barrel} \sim 2 \, \text{p.m} \, M
Logarithmic Shifter

0-7 bit Logarithmic Shifter

\[ \text{width}_{\text{log}} \approx p_m \cdot \left( 2K + \left( 1 + 2 + \ldots + 2^{K-1} \right) \right) = p_m \cdot \left( 2^K + 2K - 1 \right) \]
Arithmetic - Summary

Semiconductor Memory
Why Memory?

Intel 45nm Core 2

Semiconductor Memory Classification

<table>
<thead>
<tr>
<th>Read-Write Memory</th>
<th>Non-Volatile Read-Write Memory</th>
<th>Read-Only Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Access</td>
<td>Non-Random Access</td>
<td>EPROM</td>
</tr>
<tr>
<td>SRAM</td>
<td></td>
<td>E²PROM</td>
</tr>
<tr>
<td>DRAM</td>
<td>FIFO</td>
<td>FLASH</td>
</tr>
<tr>
<td></td>
<td>LIFO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Register</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CAM</td>
<td>Mask-Programmed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Programmable (PROM)</td>
</tr>
</tbody>
</table>
Random Access Memories (RAM)

- **STATIC (SRAM)**
  - Data stored as long as supply is applied
  - Larger (6 transistors/cell)
  - Fast
  - Differential (usually)

- **DYNAMIC (DRAM)**
  - Periodic refresh required
  - Smaller (1-3 transistors/cell)
  - Slower
  - Single Ended

Random Access Chip Architecture

- **Conceptual: linear array**
  - Each box holds some data
  - But this does not lead to a nice layout shape
  - Too long and skinny

- **Create a 2-D array**
  - Decode Row and Column address to get data
**Memory Architecture: Decoders**

Intuitive architecture for $N \times M$ memory
Too many select signals:
$N$ words $== N$ select signals

Decoder reduces the number of select signals
$K = \log_2 N$

**Basic Memory Array**

**CORE:**
- keep square within a 2:1 ratio
- rows are word lines
- columns are bit lines
- data in and out on columns

**DECODERS:**
- needed to reduce total number of pins; $N+M$ address lines for $2^{N+M}$ bits of storage
  Ex: if $N+M=20 \rightarrow 2^{20} = 1$Mb

**MULTIPLEXING:**
- used to select one or more columns for input or output of data
**Row Decoders**

Collection of $2^m$ complex logic gates
Organized in regular and dense fashion

(N)AND Decoder

$$WL_0 = A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9$$

$$WL_{511} = A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9$$

NOR Decoder

$$WI_0 = \overline{A_0} + A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8 + A_9$$

$$WL_{511} = \overline{A_0} + \overline{A_1} + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8 + A_9$$

Decoder Design Example

- Look at decoder for 256x256 memory block (8KBytes)
Problem Setup

- Goal: Build fastest possible decoder with static CMOS logic

- What we know
  - Basically need 256 AND gates, each one of them drives one word line

Problem Setup (1)

- Each word line has 256 cells connected to it
- Total output load is $256 \times C_{cell} + C_{wire}$
- Assume that decoder input capacitance is $C_{address} = 4 \times C_{cell}$
- Each address drives $2^8/2$ AND gates
  - $A_0$ drives $\frac{1}{2}$ of the gates, $A_0_b$ the other $\frac{1}{2}$ of the gates
- Neglecting $C_{wire}$, the fan-out on each one of the 16 address wires is:

$$BF = \frac{C_{load}}{C_{in}} = \frac{(2^8 / 2)(2^8 C_{cell})}{4C_{cell}} = 2^{13}$$
**Decoder Fan-Out**

- FB of at least $2^{13}$ means that we will want to use more than $\log_4(2^{13}) = 6.5$ stages to implement the AND8

- Need many stages anyways
  - So what is the best way to implement the AND gate?
  - Will see next that it’s the one with the most stages and least complicated gates

---

**8-Input AND**

![Diagram of 8-input AND gate]

- $LE=10/3$, $\Pi L = 10/3$, $P = 8 + 1$
- $LE=2$, $\Pi L = 10/3$, $P = 4 + 2$
- $LE=4/3$, $\Pi L = 5/3$, $P = 2 + 2 + 2 + 1$
- $LE=4/3$, $\Pi L = 5/3$, $P = 4/3 + 1$
- $LE=80/27$, $\Pi L = 5/3$, $P = 2 + 2 + 2 + 1$
8-Input AND

- Using 2-input NAND gates
  - 8-input gate takes 6 stages
- Total LE is \((4/3)^3 \approx 2.4\)
- So PE is \(2.4 \times 2^{13}\) – optimal N of ~7.1

Decoder So Far

- 256 8-input AND gates
  - Each built out of tree of NAND gates and inverters
- Issue:
  - Every address line has to drive 128 gates (and wire) right away
  - Can’t build gates small enough - Forces us to add buffers just to drive address inputs
Look Inside Each AND8 Gate

- Use a single gate for each of the shared terms
  - E.g., from \(A_0, A_1, \overline{A_0}, \overline{A_1}\), generate four signals: \(A_0A_1, \overline{A_0}A_1, A_0\overline{A_1}, A_0A_1\)

- In other words, we are decoding smaller groups of address bits first
  - And using the “predecoded” outputs to do the rest of the decoding

Predecoders
Predecoder and Decoder

\[ A_0, A_1 \quad A_2, A_3 \quad A_4, A_5 \]

\[ \ldots \]