Design Tradeoffs

Door #1: Performance
- Reducing power
- Improving speed
- Reducing area
- Increasing throughput

Door #2: Size

Door #3: Power

Today’s handouts:
- Lecture slides
Notes:
- Lab #2 due Thursday
Optimizing Your Design

There are a large number of implementations of the same functionality -- each represents a different point in the area-time-power space

Optimization metrics:
1. Area of the design
2. Throughput
3. Latency
4. Power consumption
5. Energy of executing a task
6. ...

©Advanced Micro Devices (with permission)  vs.  Justin14 (CC BY-SA 4.0)
CMOS Static Power Dissipation

1. Tunneling current through gate oxide: SiO$_2$ is a very good insulator, but when very thin (< 20Å) electrons can tunnel across.

2. Current leakage from drain to source even though MOSFET is “off” (aka sub-threshold conduction)
   - Leakage gets larger as difference between $V_{TH}$ and “off” gate voltage (eg, $V_{OL}$ in an nFET) gets smaller.
   - Significant as $V_{TH}$ has become smaller.
   - Fix: 3D FINFET wraps gate around inversion region
CMOS Dynamic Power Dissipation

$V_{IN}$ moves from L to H to L

$t_{CLK}=1/f_{CLK}$

$V_{OUT}$ moves from H to L to H

$C$ discharges and then recharges:

$I = C \frac{dV_{OUT}}{dt} \Rightarrow P = C \frac{dV_{OUT}}{dt} V_{OUT}$

Power dissipated to discharge $C$:

$P_{NFET} = f_{CLK} \int_{0}^{t_{CLK}/2} i_{NFET} V_{OUT} \, dt$

$= f_{CLK} \int_{0}^{t_{CLK}/2} -C \frac{dV_{OUT}}{dt} V_{OUT} \, dt$

$= f_{CLK} C \int_{V_{DD}}^{0} -V_{OUT} \, dV_{OUT}$

$= f_{CLK} C \frac{V_{DD}^2}{2}$

Power dissipated to recharge $C$:

$P_{PFET} = f_{CLK} \int_{t_{CLK}/2}^{t_{CLK}} i_{PFET} V_{OUT} \, dt$

$= f_{CLK} \int_{t_{CLK}/2}^{t_{CLK}} C \frac{dV_{OUT}}{dt} V_{OUT} \, dt$

$= f_{CLK} C \int_{V_{DD}}^{V_{DD}} V_{OUT} \, dV_{OUT}$

$= f_{CLK} C \frac{V_{DD}^2}{2}$

$E = \int_{t} P(t) \, dt$
CMOS Dynamic Power Dissipation

Power dissipated

\[ \text{Power dissipated} = CV^2_{DD} \text{ per node} \]
\[ = NCV^2_{DD}f \text{ per chip} \]

where

- \( f \) = frequency of charge/discharge
- \( N \) = number of changing nodes/chip

**“Back of the envelope”: trends**

- \( f \sim 1 \text{GHz} = 1 \times 10^9 \text{ cycles/sec} \)
- \( N \sim 1 \times 10^8 \text{ changing nodes/cycle} \)
- \( C \sim 1 \text{fF} = 1 \times 10^{-15} \text{ farads/node} \)
- \( V \sim 1 \text{V} \)

\[ \Rightarrow 100 \text{ Watts} \]
How Can We Reduce Power?

What if we could eliminate unnecessary transitions? When the output of a CMOS gate doesn’t change, the gate doesn’t dissipate much power!
Fewer Transitions $\rightarrow$ Lower Power

Signals in this region make transitions only when ALU is doing a shift operation.

Variations: Dynamically adjust $t_{CLK}$ or $V_{DD}$ (either overall or in specific regions) to accommodate workload.

Must computation consume energy? See § 6.5 of Notes.
Improving Speed: Adder Example

Worse-case path: carry propagation from LSB to MSB, e.g., when adding 11...111 to 00...001.

$$t_{PD} = (N-1)*(t_{PD,NAND3} + t_{PD,NAND2}) + t_{PD,XOR} \approx \Theta(N)$$

$$\Theta(N)$$ is read “order N” and tells us that the latency of our adder grows in proportion to the number of bits in the operands.
“Order Of” notation:

"g(n) is of order f(n)"  \( g(n) = \Theta(f(n)) \)

g(n)=\Theta(f(n)) \text{ if there exist } C_2 \geq C_1 > 0 \text{ such that for all but finitely many integral } n \geq 0

\[
C_1 \cdot f(n) \leq g(n) \leq C_2 \cdot f(n)
\]

Example:

\[ n^2 + 2n + 3 = \Theta(n^2) \]

since

\[ n^2 < n^2 + 2n + 3 < 2n^2 \]

“almost always”
Carry Select Adders

Hmm. Can we get the high half of the adder working in parallel with the low half?

Two copies of the high half of the adder: one assumes a carry-in of “0”, the other carry-in of “1”.

Once the low half computes the actual value of the carry-in to the high half, use it select the correct version of the high-half addition.

\[ t_{PD} = 16 \times t_{PD, CI \rightarrow CO} + t_{PD, MUX2} \approx \text{half of } 32 \times t_{PD, CI \rightarrow CO} \]

Aha! Apply the same strategy to build 16-bit adders from 8-bit adders. And 8-bit adders from 4-bit adders, and so on. Resulting \( t_{PD} \) for \( N \)-bit adder is \( \Theta(\log N) \).
32-bit Carry Select Adder

Practical Carry-select addition: choose block sizes so that trial sums and carry-in from previous stage arrive simultaneously at MUX.

Design goal: have these two sets of signals arrive simultaneously at each carry-select mux.

Select input is heavily loaded, so buffer for speed.
Let’s see if we can improve the speed by rewriting the equations for $C_{OUT}$:

$$C_{OUT} = AB + AC_{IN} + BC_{IN}$$

$$= AB + (A + B)C_{IN}$$

$$= G + P \cdot C_{IN} \quad \text{where } G = AB \text{ and } P = A + B$$

Actually, $P$ is usually defined as $P = A \oplus B$ which won’t change $C_{OUT}$ but will allow us to express $S$ as a simple function of $P$ and $C_{IN}$:

$$S = P \oplus C_{IN}$$
Carry Look-ahead Adders (CLA)

We can build a hierarchical carry chain by generalizing our definition of the Carry Generate/Propagate (GP) Logic. We start by dividing our addend into two parts, a higher part, H, and a lower part, L. The GP function can be expressed as follows:

\[
G_{HL} = G_H + P_H G_L \\
P_{HL} = P_H P_L
\]

- **Generate a carry out if the high part generates one, or if the low part generates one and the high part propagates it.**
- **Propagate a carry if both the high and low parts propagate theirs.**

Hierarchical building block
We can build a tree of GP units to compute the generate and propagate logic for any sized adder. Assuming N is a power of 2, we’ll need N-1 GP units.

This will let us to quickly compute the carry-ins for each FA!
8-bit CLA (carry generation)

Now, given a the value of the carry-in of the least-significant bit, we can generate the carries for every adder.

\[ c_H = G_L + P_L c_{in} \]
\[ c_L = c_{in} \]

Notice that the inputs on the left of each \( C \) blocks are the same as the inputs on the right of each corresponding \( GP \) block.
8-bit CLA (complete)

\[ t_{PD} = \Theta(\log N) \]

Notice that we don’t need the carry-out output of the adder any more.

To learn more, look up Kogge-Stone adders on Wikipedia.
### Binary Multiplication*

*Actually unsigned binary multiplication*

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>(A_3)</th>
<th>(A_2)</th>
<th>(A_1)</th>
<th>(A_0)</th>
<th>(B_3)</th>
<th>(B_2)</th>
<th>(B_1)</th>
<th>(B_0)</th>
</tr>
</thead>
</table>
| \(A_3B_0\) | \(A_2B_0\) | \(A_1B_0\) | \(A_0B_0\) | \(A_3B_1\) | \(A_2B_1\) | \(A_1B_1\) | \(A_0B_1\) | \(A_3B_2\) | \(A_2B_2\) | \(A_1B_2\) | \(A_0B_2\)
| \(A_3B_3\) | \(A_2B_3\) | \(A_1B_3\) | \(A_0B_3\)

**Hey, that looks like an AND gate**

\(AB_i\) called a “partial product”

Multiplying \(N\)-digit number by \(M\)-digit number gives \((N+M)\)-digit result

Easy part: forming partial products (just an AND gate since \(B_i\) is either 0 or 1)

Hard part: adding \(M\) \(N\)-bit partial products
Combinational Multiplier

Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$
Hardware = $\Theta(N^2)$
2’s Complement Multiplication

Step 1: two’s complement operands so high order bit is $-2^{N-1}$. Must sign extend partial products and **subtract** the last one

\[
\begin{array}{c}
X3 & X2 & X1 & X0 \\
\hline
Y3 & Y2 & Y1 & Y0 \\
\end{array}
\]

\[
\begin{array}{c}
X3Y0 & X3Y0 & X3Y0 & X3Y0 & X2Y0 & X1Y0 & X0Y0 \\
+ & X3Y1 & X3Y1 & X3Y1 & X2Y1 & X1Y1 & X0Y1 \\
+ & X3Y2 & X3Y2 & X3Y2 & X2Y2 & X1Y2 & X0Y2 \\
- & X3Y3 & X3Y3 & X2Y3 & X1Y3 & X0Y3 \\
\end{array}
\]

\[
\begin{array}{c}
Z7 & Z6 & Z5 & Z4 & Z3 & Z2 & Z1 & Z0 \\
\end{array}
\]

Step 2: don’t want all those extra additions, so add a carefully chosen constant, remembering to subtract it at the end. Convert subtraction into add of (complement + 1).

\[
\begin{array}{c}
X3Y0 & X3Y0 & X3Y0 & X3Y0 & X2Y0 & X1Y0 & X0Y0 \\
+ & 1 \\
+ & X3Y1 & X3Y1 & X3Y1 & X2Y1 & X1Y1 & X0Y1 \\
+ & 1 \\
+ & X3Y2 & X3Y2 & X3Y2 & X2Y2 & X1Y2 & X0Y2 \\
+ & 1 \\
+ & X3Y3 & X3Y3 & X2Y3 & X1Y3 & X0Y3 \\
+ & 1 \\
- & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{c}
B = \overline{B} + 1 \\
\end{array}
\]

Step 3: add the ones to the partial products and propagate the carries. All the sign extension bits go away!

\[
\begin{array}{c}
X3Y0 & X2Y0 & X1Y0 & X0Y0 \\
+ & X3Y1 & X2Y1 & X1Y1 & X0Y1 \\
+ & X3Y2 & X2Y2 & X1Y2 & X0Y2 \\
+ & X3Y3 & X2Y3 & X1Y3 & X0Y3 \\
+ & 1 \\
- & 1 & 1 & 1 & 1 \\
\end{array}
\]

Step 4: finish computing the constants…

\[
\begin{array}{c}
X3Y0 & X2Y0 & X1Y0 & X0Y0 \\
+ & X3Y1 & X2Y1 & X1Y1 & X0Y1 \\
+ & X3Y2 & X2Y2 & X1Y2 & X0Y2 \\
+ & X3Y3 & X2Y3 & X1Y3 & X0Y3 \\
+ & 1 \\
\end{array}
\]

Result: multiplying 2’s complement operands takes just about same amount of hardware as multiplying unsigned operands!
2’s Complement Multiplier
Increase Throughput With Pipelining

Before pipelining: Throughput = \( \sim 1/(2N) = \Theta(1/N) \)

After pipelining: Throughput = \( \sim 1/N = \Theta(1/N) \)
“Carry-save” Pipelined Multiplier

Observation: Rather than propagating the carries to the next column, they can instead be forwarded onto the next column of the following row.

Latency = $\Theta(N)$
Throughput = $\Theta(1)$
Hardware = $\Theta(N^2)$
Assume the multiplicand (A) has N bits and the multiplier (B) has M bits. If we only want to invest in a single N-bit adder, we can build a sequential circuit that processes a single partial product at a time and then cycle the circuit M times:

\[
\begin{align*}
\text{Init: } P &\leftarrow 0, \text{ load } A&B \\
\text{Repeat } M \text{ times } \{ & \\
& P \leftarrow P + (B_{\text{LSB}}==1 \ ? A : 0) \\
& \text{shift } S_N, P, B \text{ right one bit} \\
\} \\
\text{Done: } \text{(N+M)-bit result in } P, B
\end{align*}
\]

Latency = \Theta(N)  
Throughput = \Theta(1/N)  
Hardware = \Theta(N)

\( T_{PD} = \Theta(1) \) for carry-save (see previous slide), but adds \Theta(N) cycles & \Theta(N) hardware
Summary

• Power dissipation can be controlled by dynamically varying $T_{CLK}$, $V_{DD}$ or by selectively eliminating unnecessary transitions.

• Functions with $N$ inputs have minimum latency of $O(\log N)$ if output depends on all the inputs. But it can take some doing to find an implementation that achieves this bound.

• Performing operations in “slices” is a good way to reduce hardware costs (but latency increases)

• Pipelining can increase throughput (but latency increases)

• Asymptotic analysis only gets you so far – factors of 10 matter in real life and typically $N$ isn’t a parameter that’s changing within a given design.