Overview

- Final Exam (20% of total grade)
  - Dec 12th 3:30-5:00pm in class

- Materials: comprehensive
  - Lecture 1 through 25
  - MP0 through MP3
  - HW1, HW2, HW3
  - Practice exams

- Rules
  - 1 sheet of cheat sheet (both sides), Closed book
  - Scientific calculator only, No other electronics
Lecture 12: Datapath Functional Units

Slides courtesy of Deming Chen

Slides based on the initial set from David Harris
Outline

- Comparators
- Shifters
- Multi-input Adders
- Multipliers

- Readings: 11.3-4; 11.8-9
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)

Diagram showing circuits for detecting all ones and all zeros in a binary number.
Equality Comparator

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

\[ \begin{align*}
A[0] & \quad B[0] \\
\end{align*} \]
Magnitude Comparator

- Compute $B - A$ and look at sign
- $B - A = B + \sim A + 1$
- For unsigned numbers, carry out is sign bit

\[
A = B \quad Z
\]

\[
A \leq B
\]

\[
A \geq B
\]

Diagram of magnitude comparator circuit.
Signed vs. Unsigned

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

<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>$A \neq B$</td>
<td>$\overline{Z}$</td>
<td>$\overline{Z}$</td>
</tr>
<tr>
<td>$A &lt; B$</td>
<td>$C \cdot \overline{Z}$</td>
<td>$\overline{S} \cdot \overline{Z}$</td>
</tr>
<tr>
<td>$A &gt; B$</td>
<td>$\overline{C}$</td>
<td>$S$</td>
</tr>
<tr>
<td>$A \leq B$</td>
<td>$C$</td>
<td>$\overline{S}$</td>
</tr>
<tr>
<td>$A \geq B$</td>
<td>$\overline{C} + Z$</td>
<td>$S + Z$</td>
</tr>
</tbody>
</table>
Shifters

- **Logical Shift:**
  - Shifts number left or right and fills with 0’s
    - $1011 \text{ LSR} \ 1 = 0101 \quad 1011 \text{ LSL} \ 1 = 0110$

- **Arithmetic Shift:**
  - Shifts number left or right. Rt shift sign extends
    - $1011 \text{ ASR} \ 1 = 1101 \quad 1011 \text{ ASL} \ 1 = 0110$

- **Rotate:**
  - Shifts number left or right and fills with lost bits
    - $1011 \text{ ROR} \ 1 = 1101 \quad 1011 \text{ ROL} \ 1 = 0111$
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

- A full adder sums 3 inputs and produces 2 outputs
  - Carry output has twice weight of sum output
- N full adders in parallel are called carry save adder
  - Produce N sums and N carry outs

```
\begin{align*}
X_4 & \quad Y_4 \quad Z_4 \\
C_4 & \quad S_4 \\
X_3 & \quad Y_3 \quad Z_3 \\
C_3 & \quad S_3 \\
X_2 & \quad Y_2 \quad Z_2 \\
C_2 & \quad S_2 \\
X_1 & \quad Y_1 \quad Z_1 \\
C_1 & \quad S_1 \\
X_{N-1} & \quad Y_{N-1} \quad Z_{N-1} \\
C_{N-1} & \quad S_{N-1} \\
\end{align*}
```

n-bit CSA
Multiplication

Example:

\[
\begin{array}{c}
1100 : 12_{10} \\
0101 : 5_{10}
\end{array}
\]

\[
\begin{array}{c}
0101 \\
0101
\end{array}
\]

\[
\begin{array}{c}
1100 \\
1100
\end{array}
\]

\[
0000 \\
1100 \\
0000
\]

\[
00111100 : 60_{10}
\]

- 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:** \( P = \left( \sum_{j=0}^{M-1} y_j 2^j \right) \left( \sum_{i=0}^{N-1} x_i 2^i \right) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_i y_j 2^{i+j} \)

\[
\begin{array}{cccccccc}
\text{y}_5 & \text{y}_4 & \text{y}_3 & \text{y}_2 & \text{y}_1 & \text{y}_0 \\
\text{x}_5 & \text{x}_4 & \text{x}_3 & \text{x}_2 & \text{x}_1 & \text{x}_0 \\
\hline
\text{x}_0\text{y}_5 & \text{x}_0\text{y}_4 & \text{x}_0\text{y}_3 & \text{x}_0\text{y}_2 & \text{x}_0\text{y}_1 & \text{x}_0\text{y}_0 \\
\text{x}_1\text{y}_5 & \text{x}_1\text{y}_4 & \text{x}_1\text{y}_3 & \text{x}_1\text{y}_2 & \text{x}_1\text{y}_1 & \text{x}_1\text{y}_0 \\
\text{x}_2\text{y}_5 & \text{x}_2\text{y}_4 & \text{x}_2\text{y}_3 & \text{x}_2\text{y}_2 & \text{x}_2\text{y}_1 & \text{x}_2\text{y}_0 \\
\text{x}_3\text{y}_5 & \text{x}_3\text{y}_4 & \text{x}_3\text{y}_3 & \text{x}_3\text{y}_2 & \text{x}_3\text{y}_1 & \text{x}_3\text{y}_0 \\
\text{x}_4\text{y}_5 & \text{x}_4\text{y}_4 & \text{x}_4\text{y}_3 & \text{x}_4\text{y}_2 & \text{x}_4\text{y}_1 & \text{x}_4\text{y}_0 \\
\text{x}_5\text{y}_5 & \text{x}_5\text{y}_4 & \text{x}_5\text{y}_3 & \text{x}_5\text{y}_2 & \text{x}_5\text{y}_1 & \text{x}_5\text{y}_0 \\
\hline
\text{p}_{11} & \text{p}_{10} & \text{p}_9 & \text{p}_8 & \text{p}_7 & \text{p}_6 & \text{p}_5 & \text{p}_4 & \text{p}_3 & \text{p}_2 & \text{p}_1 & \text{p}_0
\end{array}
\]

- **Multiplicand**
- **Multiplier**
- **Partial Products**
- **Product**
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

![Booth Hardware Diagram]

- Control lines for each PP
- Booth selectors choose PP bits

Datapath Functional Units
Lecture 13: SRAM

Slides courtesy of Deming Chen

Slides based on the initial set from David Harris
Outline

- Memory Arrays
- SRAM Architecture
  - SRAM Cell
  - Decoders
  - Column Circuitry
  - Multiple Ports
- Serial Access Memories

Readings: Chapter 12.1-12.2.5 and 12.5
Array Architecture

- $2^n$ words of $2^m$ bits each
- If $n \gg m$, fold by $2^k$ into fewer rows of more columns
- Good regularity – easy to design
- Very high density if good cells are used
12T SRAM Cell

- Basic building block: SRAM Cell
  - Holds one bit of information, like a latch
  - Must be read and written
- 12-transistor (12T) SRAM cell
  - Use a simple latch connected to bitline
  - 46 x 75 \( \lambda \) unit cell
6T SRAM Cell

- Cell size accounts for most of array size
  - Reduce cell size at expense of complexity

- 6T SRAM Cell
  - Used in most commercial chips
  - Data stored in cross-coupled inverters

- Read:
  - Precharge bit, bit_b
  - Raise wordline

- Write:
  - Drive data onto bit, bit_b
  - Raise wordline
SRAM Read

- Precharge both bitlines high
- Then turn on wordline
- One of the two bitlines will be pulled down by the cell
- Ex: $A = 0$, $A_b = 1$
  - bit discharges, $bit_b$ stays high
  - But $A$ bumps up slightly
- Read stability
  - $A$ must not flip
  - $N1 \gg N2$

![SRAM Read Diagram]
SRAM Write

- Drive one bitline high, the other low
- Then turn on wordline
- Bitlines overpower cell with new value
- Ex: $A = 0$, $A_b = 1$, bit $= 1$, bit$_b = 0$
  - Force $A_b$ low, then $A$ rises high
- Writability
  - Must overpower feedback inverter
  - $N2 >> P1$
SRAM Sizing

- High bitlines must not overpower inverters during reads
- But low bitlines must write new value into cell
- Cell size is critical: 26 x 45 λ (even smaller in industry)
- Tile cells sharing $V_{DD}$, GND, bitline contacts
Decoders

- A $2^n$ decoder consists of $2^n$ n-input AND gates
  - One needed for each row of memory
  - Build AND from NAND or NOR gates

Static CMOS

Pseudo-nMOS
Decoder Layout

- Decoders must be pitch-matched to SRAM cell
  - Requires very skinny gates

[Diagram of decoder layout with labels for A0, A1, A2, A3, VDD, GND, NAND gate, and buffer inverter]
Dual-Ported SRAM

- Simple dual-ported SRAM
  - Two independent single-ended reads
  - Or one differential write

- Do two reads and one write by time multiplexing
  - Read during ph1, write during ph2
Multi-Ported SRAM

- Adding more access transistors hurts read stability
- Multiported SRAM isolates reads from state node
- Single-ended bitlines save area
Shift Register

- Shift registers store and delay data
- Simple design: cascade of registers
  - Watch your hold times!

Diagram:
```
clk --

Din 8

--

Dout
```
A **tapped delay line** is a shift register with a programmable number of stages.

- Set number of stages with delay controls to mux
  - Ex: 0 – 63 stages of delay
Serial In Parallel Out

- 1-bit shift register reads in serial data
  - After N steps, presents N-bit parallel output

```
clk

Sin

P0  P1  P2  P3
```
Parallel In Serial Out

- Load all N bits in parallel when shift = 0
  - Then shift one bit out per cycle
Queues

- Queues allow data to be read and written at different rates.
- Read and write each use their own clock, data
- Queue indicates whether it is full or empty
- Build with SRAM and read/write counters (pointers)
**FIFO, LIFO Queues**

- *First In First Out (FIFO)*
  - Initialize read and write pointers to first element
  - Queue is EMPTY
  - On write, increment write pointer
  - If write almost catches read, Queue is FULL
  - On read, increment read pointer

- *Last In First Out (LIFO)*
  - Also called a *stack*
  - Use a single *stack pointer* for read and write
Lecture 14: CAMs, ROMs, PLAs

Slides courtesy of Deming Chen

Slides based on the initial set from David Harris
Outline

- Content-Addressable Memories
- Read-Only Memories
- Programmable Logic Arrays

- Readings: 12.4-12.7
CAMs

- Extension of ordinary memory (e.g. SRAM)
  - Read and write memory as usual
  - Also *match* to see which words contain a *key*
10T CAM Cell

- Add four match transistors to 6T SRAM
  - $56 \times 43 \lambda$ unit cell
CAM Cell Operation

- Read and write like ordinary SRAM
- For matching:
  - Leave wordline low
  - Precharge matchlines
  - Place key on bitlines
  - Matchlines evaluate
- Miss line
  - Pseudo-nMOS NOR of match lines
  - Goes high if no words match
Read-Only Memories

- Read-Only Memories are nonvolatile
  - Retain their contents when power is removed
- Mask-programmed ROMs use one transistor per bit
  - Presence or absence determines 1 or 0
ROM Example

- 4-word x 6-bit ROM
  - Represented with dot diagram
  - Dots indicate 1’s in ROM

Words:
- Word 0: 010101
- Word 1: 011001
- Word 2: 100101
- Word 3: 101010

Looks like 6 4-input pseudo-nMOS NORs
Flash Programming

- Charge on floating gate determines $V_t$
- Logic 1: negative $V_t$
- Logic 0: positive $V_t$
- Cells erased to 1 by applying a high body voltage so that electrons tunnel off floating gate into substrate
- Programmed to 0 by applying high gate voltage

0 V | 20 V
---|---
20 V | 20 V

- Erase
- Inhibit Erase

0 V | 0 V | 0 V | 0 V
---|---|---|---
20 V | 20 V | 20 V | 10 V

- Program 0
- Do Not Program
- Inhibit Program
NAND Flash

- High density, low cost / bit
  - Programmed one page at a time
  - Erased one block at a time
- Example:
  - 4096-bit pages
  - 16 pages / 8 KB block
  - Many blocks / memory
Building Logic with ROMs

- Use ROM as lookup table containing truth table
  - n inputs, k outputs requires \(2^n\) words x \(k\) bits
  - Changing function is easy – reprogram ROM

- Finite State Machine
  - n inputs, k outputs, s bits of state
  - Build with \(2^{n+s}\) x \((k+s)\) bit ROM and \((k+s)\) bit reg

![Diagram of ROM Array and Finite State Machine](image-url)
ROM Implementation

- 16-word x 5 bit ROM

```
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
```

4:16 DEC

L, R  `---`  TL, TR, F

S' 1:0

S 1
S 0
L R
PLAs

- A *Programmable Logic Array* performs any function in sum-of-products form.
- *Literals*: inputs & complements
- *Products / Minterms*: AND of literals
- *Outputs*: OR of Minterms

- Example: Full Adder

\[
s = \overline{a}b\overline{c} + \overline{a}bc + \overline{a}\overline{b}c + abc
\]

\[
c_{out} = ab + bc + ac
\]
NOR-NOR PLAs

- ANDs and ORs are not very efficient in CMOS
- Dynamic or Pseudo-nMOS NORs are very efficient
- Use DeMorgan’s Law to convert to all NORs
PLAs vs. ROMs

- The OR plane of the PLA is like the ROM array
- The AND plane of the PLA is like the ROM decoder
- PLAs are more flexible than ROMs
  - No need to have $2^n$ rows for $n$ inputs
  - Only generate the minterms that are needed
  - Take advantage of logic simplification
Lecture 15: System Modeling and Verilog

Slides courtesy of Deming Chen
Lecture 16: Circuit Pitfalls and Design for Test

Slides courtesy of Deming Chen

Slides based on the initial set from David Harris
Outline

- Variation
- Reliability
- Circuit Pitfalls
- Testing
- Fault Models
- Observability and Controllability
- Design for Test
- Boundary Scan

- Readings: 7.1-7.3; 15.1-15.7
Variation

- Process
  - Threshold
  - Channel length
  - Interconnect dimensions

- Environment
  - Voltage
  - Temperature

- Aging / Wearout
Process Variation

- **Threshold Voltage**
  - Depends on placement of dopants in channel
  - Standard deviation inversely proportional to channel area

- **Channel Length**
  - Systematic *across-chip linewidth variation* (ACLV)
  - Random line edge roughness (LER)

- **Interconnect**
  - Etching variations affect w, s, h

[Bernstein06]

Circuit Pitfalls  CMOS VLSI Design 4th Ed.
Spatial Distribution

- Variations show spatial correlation
  - Lot-to-lot (L2L)
  - Wafer-to-wafer (W2W)
  - Die-to-die (D2D) / inter-die
  - Within-die (WID) / intradie
- Closer transistors match better

[Image of spatial distribution graph]

Courtesy M. Pelgrom
Environmental Variation

- **Voltage**
  - $V_{DD}$ is usually designed +/- 10%
  - Regulator error
  - On-chip droop from switching activity

- **Temperature**
  - Ambient temperature ranges
  - On-die temperature elevated by chip power consumption

[Courtesy IBM]

<table>
<thead>
<tr>
<th>Type</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commercial</td>
<td>0 °C</td>
<td>70 °C</td>
</tr>
<tr>
<td>Industrial</td>
<td>−40 °C</td>
<td>85 °C</td>
</tr>
<tr>
<td>Military</td>
<td>−55 °C</td>
<td>125 °C</td>
</tr>
</tbody>
</table>

[Harris01b]
Aging

- Transistors change over time as they wear out
  - Hot carriers
  - Negative bias temperature instability
  - Time-dependent dielectric breakdown
- Causes threshold voltage changes
Process Corners

- Model extremes of process variations in simulation
- Corners
  - Typical (T)
  - Fast (F)
  - Slow (S)
- Factors
  - nMOS speed
  - pMOS speed
  - Wire
  - Voltage
  - Temperature

<table>
<thead>
<tr>
<th>Corner</th>
<th>Voltage</th>
<th>Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>1.98</td>
<td>0 °C</td>
</tr>
<tr>
<td>T</td>
<td>1.8</td>
<td>70 °C</td>
</tr>
<tr>
<td>S</td>
<td>1.62</td>
<td>125 °C</td>
</tr>
</tbody>
</table>
Corner Checks

- Circuits are simulated in different corners to verify different performance and correctness specifications.

<table>
<thead>
<tr>
<th>Corner</th>
<th>nMOS</th>
<th>pMOS</th>
<th>Wire</th>
<th>$V_{DD}$</th>
<th>Temp</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>S</td>
<td>S</td>
<td>Timing specifications (binned parts)</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>Timing specifications (conservative)</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>Race conditions, hold time constraints, pulse collapse, noise</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>?</td>
<td>F</td>
<td>S</td>
<td>S</td>
<td>Dynamic power</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>S</td>
<td>Subthreshold leakage noise and power, overall noise analysis</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>F</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>Races of gates against wires</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>S</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>Races of wires against gates</td>
</tr>
<tr>
<td>S</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>Pseudo-nMOS and ratioed circuits noise margins, memory read/write, race of pMOS against nMOS</td>
</tr>
<tr>
<td>F</td>
<td>S</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>Ratioed circuits, memory read/write, race of nMOS against pMOS</td>
</tr>
</tbody>
</table>
Noise

- Sources
  - Power supply noise / ground bounce
  - Capacitive coupling
  - Charge sharing
  - Leakage
  - Noise feedthrough

- Consequences
  - Increased delay (for noise to settle out)
  - Or incorrect computations
Reliability

- Hard Errors
  - Oxide wearout
  - Interconnect wearout
  - Overvoltage failure
  - Latchup
- Soft Errors
- Characterizing reliability
  - Mean time between failures (MTBF)
    - # of devices x hours of operation / number of failures
  - Failures in time (FIT)
    - # of failures / thousand hours / million devices
Accelerated Lifetime Testing

- Expected reliability typically exceeds 10 years
- But products come to market in 1-2 years
- Accelerated lifetime testing required to predict adequate long-term reliability

![Graph showing DC lifetime 10% ion vs. Vdd stress (Arnaud08)]
Electromigration

- “Electron wind” causes movement of metal atoms along wires
- Excessive electromigration leads to open circuits
- Most significant for unidirectional (DC) current
  - Depends on current density $J_{dc}$ (current / area)
  - Exponential dependence on temperature

- Black’s Equation: $MTTF \propto \frac{E_a}{e^{kT}} \frac{e^{kT}}{J_{dc}}^n$

- Typical limits: $J_{dc} < 1 - 2$ mA / $\mu$m$^2$
Self-Heating

- Current through wire resistance generates heat
  - Oxide surrounding wires is a thermal insulator
  - Heat tends to build up in wires
  - Hotter wires are more resistive, slower

- Self-heating limits AC current densities for reliability
  - Typical limits: $J_{\text{rms}} < 15 \text{ mA/\mu m}^2$
Overvoltage Failure

- High voltages can blow out tiny transistors
- Electrostatic discharge (ESD)
  - Kilovolts from static electricity when the package pins are handled
- Oxide breakdown
  - In a 65 nm process, $V_g \approx 3$ V causes arcing through thin gate oxides
- Punchthrough
  - High $V_{ds}$ causes depletion region between source and drain to touch, leading to high current flow and destructive overheating
Circuit Pitfalls

- Detective puzzle
  - Given circuit and symptom, diagnose cause and recommend solution
  - All these pitfalls have caused failures in real chips
Bad Circuit 1

- Circuit
  - 2:1 multiplexer

- Symptom
  - Mux works when selected D is 0 but not 1.
  - Or fails at low V_{DD}.
  - Or fails in SFSF corner.

- Principle: Threshold drop
  - X never rises above V_{DD}-V_{t}
  - V_{t} is raised by the body effect
  - The threshold drop is most serious as V_{t} becomes a greater fraction of V_{DD}.

- Solution: Use transmission gates, not pass transistors
Bad Circuit 2

- Circuit
  - Latch
    \[ D \rightarrow X \rightarrow Q \]

- Symptom
  - Load a 0 into \( Q \)
  - Set \( \phi = 0 \)
  - Eventually \( Q \) spontaneously flips to 1

- Principle: Leakage
  - \( X \) is a dynamic node holding value as charge on the node
  - Eventually subthreshold leakage may disturb charge

- Solution: Staticize node with feedback
  - Or periodically refresh node (requires fast clock, not practical processes with big leakage)
Bad Circuit 3

- **Circuit**
  - Pseudo-nMOS OR

- **Symptom**
  - When only one input is true, $Y = 0$.
  - Perhaps only happens in SF corner.

- **Principle: Ratio Failure**
  - nMOS and pMOS fight each other.
  - If the pMOS is too strong, nMOS cannot pull $X$ low enough.

- **Solution:** Check that ratio is satisfied in all corners
Bad Circuit 4

- Circuit
  - Latch

- Symptom
  - Q stuck at 1.
  - May only happen for certain latches where input is driven by a small gate located far away.

- Principle: Ratio Failure (again)
  - Series resistance of D driver, wire resistance, and tgate must be much less than weak feedback inverter.

- Solutions: Check relative strengths
  - Avoid unbuffered diffusion inputs where driver is unknown
Bad Circuit 5

- **Circuit**
  - Latch

```
\begin{circuitikz}

\draw (0,0) node[and gate, below] (A) {}; 
\draw (0,2) node[vcc] (B) {}; 
\draw (2,0) node{\textit{weak}} (C); 
\draw (B) -- (A) -- (C); 
\node at (1,-0.5) {D}; \node at (1,2.5) {V_{DD}}; \node at (3,-0.5) {Q}; \node at (3,2.5) {V_{DD}}; \node at (0,-2) {GND}; \node at (2,-2) {V_{DD}}; 
\end{circuitikz}
```

- **Symptom**
  - Q changes while latch is opaque
  - Especially if D comes from a far-away driver

- **Principle:** Diffusion Input Noise Sensitivity
  - If D < -V_t, transmission gate turns on
  - Most likely because of power supply noise or coupling on D

- **Solution:** Buffer D locally

```
\begin{circuitikz}

\draw (0,0) node{\textit{weak}} node[and gate, below] (A) {}; 
\draw (0,2) node[vcc] (B) {}; 
\draw (2,0) node{\textit{weak}} (C); 
\draw (B) -- (A) -- (C); 
\node at (1,-0.5) {D}; \node at (1,2.5) {V_{DD}}; \node at (3,-0.5) {Q}; \node at (3,2.5) {V_{DD}}; \node at (0,-2) {V_{DD}}; \node at (2,-2) {V_{DD}}; 
\end{circuitikz}
```
Testing

- Testing is one of the most expensive parts of chips
  - Logic verification accounts for > 50% of design effort for many chips
  - Debug time after fabrication has enormous opportunity cost
  - Shipping defective parts can sink a company

- Example: Intel FDIV bug (1994)
  - Logic error not caught until > 1M units shipped
  - Recall cost $450M (!!!)
Logic Verification

- Does the chip simulate correctly?
  - Usually done at HDL level
  - Verification engineers write test bench for HDL
    - Can’t test all cases
    - Look for corner cases
    - Try to break logic design

- Ex: 32-bit adder
  - Test all combinations of corner cases as inputs:
    - 0, 1, 2, $2^{31}$-1, -1, $-2^{31}$, a few random numbers

- Good tests require ingenuity
Stuck-At Faults

- How does a chip fail?
  - Usually failures are shorts between two conductors or opens in a conductor
  - This can cause very complicated behavior

- A simpler model: Stuck-At
  - Assume all failures cause nodes to be “stuck-at” 0 or 1, i.e. shorted to GND or $V_{DD}$
  - Not quite true, but works well in practice
Test Pattern Generation

- Manufacturing test ideally would check every node in the circuit to prove it is not stuck.
- Apply the smallest sequence of test vectors necessary to prove each node is not stuck.

- Good observability and controllability reduces number of test vectors required for manufacturing test.
  - Reduces the cost of testing
  - Motivates design-for-test
Lecture 17: Design for Low Power

Slides courtesy of Deming Chen

Slides based on the initial set from David Harris
Outline

- Power and Energy
- Dynamic Power
- Static Power

- Readings: 5.1-5.3
Power and Energy

- Power is drawn from a voltage source attached to the $V_{DD}$ pin(s) of a chip.

- Instantaneous Power: $P(t) = I(t)V(t)$

- Energy:
  $$E = \int_0^T P(t)dt$$

- Average Power:
  $$P_{avg} = \frac{E}{T} = \frac{1}{T} \int_0^T P(t)dt$$
Charging a Capacitor

- When the gate output rises
  - Energy stored in capacitor is
    \[ E_C = \frac{1}{2} C_L V_{DD}^2 \]
  - But energy drawn from the supply is
    \[ E_{VDD} = \int_0^\infty I(t) V_{DD} dt = \int_0^\infty C_L \frac{dV}{dt} V_{DD} dt \]
    \[ = C_L V_{DD} \int_0^{V_{DD}} dV = C_L V_{DD}^2 \]
  - Half the energy from \( V_{DD} \) is dissipated in the pMOS transistor as heat, other half stored in capacitor

- When the gate output falls
  - Energy in capacitor is dumped to GND
  - Dissipated as heat in the nMOS transistor
Switching Power

\[ P_{\text{switching}} = \frac{1}{T} \int_{0}^{T} i_{\text{DD}}(t) V_{\text{DD}} \, dt \]

\[ = \frac{V_{\text{DD}}}{T} \int_{0}^{T} i_{\text{DD}}(t) \, dt \]

\[ = \frac{V_{\text{DD}}}{T} \left[ T f_{\text{sw}} C V_{\text{DD}} \right] \]

\[ = C V_{\text{DD}}^2 f_{\text{sw}} \]
Activity Factor

- Suppose the system clock frequency = $f$
- Let $f_{sw} = \alpha f$, where $\alpha = \text{activity factor}$
  - If the signal is a clock, $\alpha = 1$
  - If the signal switches once per cycle, $\alpha = \frac{1}{2}$

- Dynamic power:
  $$P_{\text{switching}} = \alpha CV_{DD}^2 f$$
Power Dissipation Sources

- $P_{\text{total}} = P_{\text{dynamic}} + P_{\text{static}}$
- Dynamic power: $P_{\text{dynamic}} = P_{\text{switching}} + P_{\text{shortcircuit}}$
  - Switching load capacitances
  - Short-circuit current
- Static power: $P_{\text{static}} = (I_{\text{sub}} + I_{\text{gate}} + I_{\text{junct}} + I_{\text{contention}})V_{DD}$
  - Subthreshold leakage
  - Gate leakage
  - Junction leakage
  - Contention current
Clock Gating

- The best way to reduce the activity is to turn off the clock to registers in unused blocks
  - Saves clock activity ($\alpha = 1$)
  - Eliminates all switching activity in the block
  - Requires determining if block will be used
Capacitance

- Gate capacitance
  - Fewer stages of logic
  - Small gate sizes

- Wire capacitance
  - Good floorplanning to keep communicating blocks close to each other
  - Drive long wires with inverters or buffers rather than complex gates
Voltage / Frequency

- Run each block at the lowest possible voltage and frequency that meets performance requirements.

- Voltage Domains
  - Provide separate supplies to different blocks
  - Level converters required when crossing from low to high $V_{DD}$ domains

- Dynamic Voltage Scaling
  - Adjust $V_{DD}$ and $f$ according to workload
Static Power

- Static power is consumed even when chip is quiescent.
  - Leakage draws power from nominally OFF devices
  - Ratioed circuits burn power in fight between ON transistors
Subthreshold Leakage

- For $V_{ds} > 50$ mV

$$I_{sub} \approx I_{off} \times 10^{\frac{V_{gs} + \eta(V_{ds} - V_{DD}) - k \gamma V_{sb}}{S}}$$

- $I_{off}$ = leakage at $V_{gs} = 0$, $V_{ds} = V_{DD}$

Typical values in 65 nm:

- $I_{off} = 100$ nA/µm @ $V_t = 0.3$ V
- $I_{off} = 10$ nA/µm @ $V_t = 0.4$ V
- $I_{off} = 1$ nA/µm @ $V_t = 0.5$ V

- $\eta = 0.1$
- $k_\gamma = 0.1$
- $S = 100$ mV/decade
Leakage Control

- Leakage and delay trade off
  - Aim for low leakage in sleep and low delay in active mode

- To reduce leakage:
  - Increase $V_t$: *multiple* $V_t$
    - Use low $V_t$ only in critical circuits
  - Increase $V_s$: *stack effect*
    - *Input vector control* in sleep
  - Decrease $V_b$
    - *Reverse body bias* in sleep
    - Or forward body bias in active mode
Gate Leakage

- Extremely strong function of $t_{ox}$ and $V_{gs}$
  - Negligible for older processes
  - Approaches subthreshold leakage at 65 nm and below in some processes
- An order of magnitude less for pMOS than nMOS
- Control leakage in the process using $t_{ox} > 10.5$ Å
  - High-k gate dielectrics help
  - Some processes provide multiple $t_{ox}$
    - e.g. thicker oxide for 3.3 V I/O transistors
- Control leakage in circuits by limiting $V_{DD}$
Power Gating

- Turn OFF power to blocks when they are idle to save leakage
  - Use virtual $V_{DD}$ ($V_{DDV}$)
  - Gate outputs to prevent invalid logic levels to next block

- Voltage drop across sleep transistor degrades performance during normal operation
  - Size the transistor wide enough to minimize impact

- Switching wide sleep transistor costs dynamic power
  - Only justified when circuit sleeps long enough
Lecture 18: VLSI Design Styles

Slides courtesy of Deming Chen
Outline

- Overview
  - Microprocessor/DSP
  - Programmable Logic
  - Gate Arrays and Sea of Gates
  - Cell-based Design
  - Full Custom Design
  - System on a Chip
  - Intellectual Property (IP)

- Reading
  - Part of the lecture is coming from Text 14.3
Different design methodologies provide varying degrees of freedom, cost and performance:

- Microprocessor/DSP
- Programmable Logic (FPGA and CPLD)
- Gate array, sea of gates, and structured ASIC
- Standard Cell
- Custom
- System-on-chip (core-based)
Microprocessor/DSP

- Standard microprocessor or digital signal processor (DSP) are very practical and usually offer great flexibility
- Systems can be upgraded in the field through software patches
- Off-the-shelf processors with wide range of clock speeds, memory sizes, and analog I/O capability
- Embedded processors are available for system-on-chip design methodology
- May not be most efficient in terms of cost, speed and power dissipation
### Gaps Are Widening

#### Design Productivity Gap
- Increasing complexity of designs
- Reduced time-to-market

#### Verification/Predictability Gap
- Delayed final tapeout

#### Quality Gap
- RTL design focusing on limited architecture alternatives

---

**Technology Scaling**

- **Design Productivity Gap**
  - gates/cm²: Moore’s Law (59%*)
  - Design Productivity (20-25%*)

- **Productivity/Complexity**

---

**Source:**
- Pittsburgh Digital Greenhouse
- Semico Research Corp.
- EETimes.com

---

*Compound Annual Growth Rate*
**The Trend: High-level Synthesis (HLS)**

- **System level**: Design spec. in high-level languages
  - SW/HW Co-design

- **Behavior level**: C, C++, SystemC

- **RT level**: (VHDL, Verilog)

- **High-level Synthesis**
  - 10X code reduction
  - 1000X simulation time reduction
  [Source: NEC]

- **Logic Synthesis**

- **Gate level**: (netlist)

- **Place & Route**

- **Chips**

**Levels of Abstraction**
Cell-Based Design

- Design composed out of a set of pre-designed blocks, called *standard cells*
  - Cells often simple gates/latches, can be more complex
  - Cells pre-tested and pre-characterized
  - Companies offer standard cell libraries for use with their fabrication and CAD technologies
    - Amortize the effort of designing the cell library over all the designs that use it
Organize cells into rows to make placement easier

- Require that all cells have the same height
Full-Custom Design

- What we’ve been doing so far
- Designer specifies size, position, connections of every device in a circuit, down to the mask level
- Gives highest performance and device density
- Disadvantages:
  - Very tedious and complex
  - Long design times
  - Higher design cost
  - Error prone
- Nowadays, this design style is used only for parts of state-of-the-art designs where performance and area are pushed to the limits (to achieve an A grade).
System-on-a-Chip (Platform-Based)

- **Motivation**: Even with synthesis from VHDL/Verilog, design effort for modern chips is very large.

- **Observation**: Many VLSI systems incorporate similar components
  - ALUs
  - Memory Interfaces
  - RAM blocks
  - DSP modules

- **Idea**: Provide a set of *cores* (Intellectual Properties) that perform common functions, let designers use them
  - Eliminate the need to re-design
  - Becomes worth the effort to heavily optimize cores
Intellectual Property (IP)

- Building block components (roughly equivalent terms)
  - Macros, cores, IPs, virtual components (VCs)

- Examples
  - Microprocessor core, A/D converter, Digital filter, Audio compression algorithm

- Three types of IP blocks
  - Hard (least flexible)
  - Firm
  - Soft (most flexible)
Fixed Schematics and Layout

Schematic of a NAND gate

Layout of a NOR gate
Lecture 19: VLSI CAD Tools

Slides courtesy of Deming Chen, Leslie Hwang, Ashutosh Dhar
Outline

- Overview
  - Design Languages
  - CAD Tools
    - Design Capture Tools
    - Synthesis Tools
    - Verification Tools
  - Vivado Design Suite from Xilinx as a case study

- Reading
  - Part of the lecture comes from Text 14.4
As we have seen, a design system is a set of computer-aided design (CAD) tools, used to simplify the design and verification tasks.
Design Flow

- **Design flow** is a set of procedures designers progress from specification for a chip to the final chip implementation in error-free way.

  ASIC design: flows partitioned between different EDA companies.

  Behavioral level: operation of the system without implementation.

  RTL: logic and memory elements.

  Structural level: gates and registers.

  Synthesis: transformation to physical (layout) description.
A design can be described at different levels of abstraction (high-low):

- Architectural level
- Finite State Machine level
- Logic level
- Symbolic level, e.g., PLA
- Transistor, or switch, level
- Sticks level
- Mask level
Architectural Level

- Procedural (Architectural Level) Languages
  - Primitives are *functions* and *variables*
  - Gives an *algorithmic* description of the design
    - Important special case is Register Transfer Languages (RTL)
    - *Registers* hold the variables
    - *Arithmetic units* perform the operations on the variables

```verilog
module mux_without_default (I0, I1, I2, I3, Sel, F);
  input I0, I1, I2, I3;
  input [1:0] Sel;  // Sel0, Sel1
  output F;
  reg F;
  always @(I0 or I1 or I2 or I3 or Sel) begin
    case (Sel)
      0 : F = I0;
      1 : F = I1;
      2 : F = I2;
      3 : F = I3;
    endcase
  endmodule
```
Finite State Machine Languages
- A symbolic description of states and state transitions
- A behavioral description
- Not structural, not geometrical
- E.g. microprocessor controller
- Can be compiled into PLAs or logic
  - The compiler must make structural and geometrical decisions.
Logic Level

- **Logic Languages**
  - The primitives are *logic gates* and *nodes*
    - Higher level and more versatile than PLA
  - Describes *logical structure*, but no spatial relationships
  - Can be transformed, or *compiled*, into a switch language
Transistor, Switch and Symbolic Level

Transistor, or Switch, Languages
- The primitives are *transistors* and *nodes*
- Specifies *electrical topology*
- No spatial relationships
- Not easily transformed to a geometry specification
- Easily *extracted* from layout
- Suitable for simulation

Symbolic Languages
- Slightly higher level than transistors
  - E.g. PLA dot diagram
  - Usually *specialized* for specific layout
Stick Level

- Sticks Languages
  - These are *virtual* or *symbolic* layout descriptions
    - A popular example is the LAVA language
  - Transistors and vias are represented as points on a grid
  - Wires are represented as zero-width lines
  - Gives the *relative positions* of circuit elements
Geometry or Mask Level

- Geometry Languages
  - Use colored rectangles or shapes as their primitives
    - A popular example is the Caltech Intermediate Form (CIF)
  - One can directly fabricate a chip from its description

Command
- Polygon with a path
- Box with length, width, center, and direction
- Round flash with diameter and center
- Wire with width and path

Form
- P path
- B integer integer point point
- R integer point
- W integer path
- L shortname
- DS integer integer integer
- DF
- DD integer
- C integer transformation
- digit userText
  ( commentText )
- E

CIF example

FIGURE B.2 A sample CIF "wire" statement. The statement is: W25 100 200 100 100 200 200 300 200;
Modern design systems use a variety of CAD tools, including:

- **Design-capture tools**
  - These tools help translate an idea into a high-level design description

- **Synthesis tools**
  - These tools translate a higher-level description into a lower-level one

- **Verification tools**
  - These tools verify that a lower-level implementation is equivalent to a higher-level one

Typically, the design process is not completely automated. Instead, the various CAD tools are used at the designer’s discretion to facilitate and validate the design tasks. CAD tools need to understand both the language and the primitives that the language refers to.
Design Capture Tools

- Design-capture tools are the first tools to be used.
- They provide an interface between what the designer has in mind and the high-level design description.
- There are two general methods of design capture:
  - Textually: body of code
  - Schematically: graphical

Definition of 1-bit Full adder will be as follow:

```verilog
module FullAdder(A,B,Cin,Cout,S);
  input  A, B, Cin;
  output Cout, S;
  // Design Body
endmodule
```

![Diagram of 1-bit Full adder](image)
Formal Verification

- It aims to directly prove the equivalence of two design representations.
  - For instance, graph isomorphism is applied to check that a transistor netlist is identical to an extracted netlist.
- These methods are mature and well developed
- The main challenge is scalability
- Many commercial tools available
- SoC verification is still a challenge
At a slightly higher level, *Boolean equivalence* is used to check if a gate-level implementation performs required Boolean function, as specified in the RTL description (equivalence checkers).

- Binary Decision Diagrams (BDDs) extracted and compared.
Simulation Tools

- Simulators are probably the most often used design tools.
- A simulator uses mathematical models to represent the behavior of circuit components.
- Given specific input signals, the simulator solves for the signals inside the circuit.
- Simulators come in a wide variety depending on the level of accuracy and the simulation speed desired:
  - circuit simulation
  - switch-level simulation
  - logic simulation
  - functional simulation
### MP3 Automation Tools

- **Logic Synthesis**: Schematic automation
  - Tool: Design Vision by Synopsys

<table>
<thead>
<tr>
<th>Convert from</th>
<th>Convert to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavioral HDL</td>
<td>Schematic, structural RTL</td>
</tr>
<tr>
<td><code>controller.v</code></td>
<td><code>controller_synth.v</code></td>
</tr>
</tbody>
</table>

- **Place & Route**: Layout automation
  - Tool: Encounter by Cadence
  - Industry-standard file formats
    - Floorplan: DEF format (.def)
    - Layout: GDSII format (.gds)

<table>
<thead>
<tr>
<th>Convert from</th>
<th>Convert to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthesized RTL</td>
<td>Layout</td>
</tr>
<tr>
<td><code>controller_synth.v</code></td>
<td><code>controller.def, controller.gds</code></td>
</tr>
</tbody>
</table>
Lecture 20: High-level Synthesis (1)

Slides courtesy of Deming Chen

Some slides are from Prof. S. Levitan of U. of Pittsburgh
Outline

- High-level synthesis introduction
- High-level synthesis operations
  - Scheduling
    - ASAP and ALAP
    - List scheduling
  - Allocation
  - Binding
- Loop unrolling and pipelining
What is High-level Synthesis?

- Input is a high level, algorithmic description
  - Control structures (if/then, loop, subroutines)
  - Concurrent and sequential semantics
  - Abstract data types
  - Logical and arithmetic operators
  - AND

- A set of constraints
  - Speed, power, area, interconnect style, coding style
  - A library of pre-specified components

- Output is an RTL description for further synthesis and optimization phases
Synthesis Steps

Behavioral Description

CDFG Generation

Resource Allocation

Scheduling

Netlisting (scheduled)

Register Allocation

Binding

Data Path and State Machine Extraction

Netlisting (final)
Resource Allocation

- Deciding how many and which kinds of resources will be used in a given implementation
- This has a major impact on final design
  - Number of operation units (multiple adders?) set the maximum parallelism that the architecture can provide
  - Reuse of overloaded operators (e.g. an adder/subtractor unit) provides smallest designs
  - Choice of buses or muxes provides parallelism vs. size
  - Choice of registers, multi-ported register files or RAM also limits parallelism in data movement
Scheduling

- As soon as possible (ASAP)
- As late as possible (ALAP)
- List scheduling
ASAP Schedule
(unconstrained)

\[ Y = ((a*b)+c)+(d*e)-(f+g) \]

The start time for each operation is the least one allowed by the dependencies.
ASAP Algorithm

\[
\text{ASAP} (G(V, E)) \{ \\
\quad \text{Schedule all the nodes driven only by PIs to cycle 1,} \\
\quad \text{for all such } v_i \text{ nodes, } t_i \text{ (starting time)} = 1; \\
\text{Repeat} \{ \\
\quad \text{Select a vertex } v_i \text{ whose predecessors are all scheduled;} \\
\quad \text{Schedule } v_i \text{ by setting } t_i = \text{MAX}(t_j + d_j); \\
\quad (v_j, v_i) \in E \\
\} \\
\text{Until all the nodes are scheduled;} \\
\text{Return the schedule in a vector;}
\}
\]
The end time of each operation is the latest one allowed by the dependencies and the latency constraint.

\[ Y = ((a*b)+c)+(d*e)-(f+g) \]
Mobility (or Slack)

Y = ((a*b)+c)+(d*e)-(f+g)

Mobility is the difference of the start times computed by the ALAP and ASAP.
List Scheduling (1)

- Priority based on mobility (other metrics possible)
- Resource constraints: one adder, one multiplier
- Schedule ready nodes

Prioritized Ready List

<table>
<thead>
<tr>
<th>Operation</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1(mul)</td>
<td>0</td>
</tr>
<tr>
<td>op2(mul)</td>
<td>1</td>
</tr>
<tr>
<td>op3(add)</td>
<td>2</td>
</tr>
</tbody>
</table>

Clock cycle 1: 2 3 4
List Scheduling (2)

Clock cycle 1  2  3  4

Prioritized Ready List

op2(mul) 1
op4(add) 0

Schedule op4 and op2

op2
op4
List Scheduling (3)

Clock cycle 1  2  3  4

Prioritized Ready List

<table>
<thead>
<tr>
<th>Schedule</th>
<th>op5(add)0</th>
</tr>
</thead>
</table>

MUL → ADD
ADD
MUL
ADD

ECE 425
CMOS VLSI Design 4th Ed.
Formulation can be extended to handle multi-cycle operations.
List Scheduling Algorithm

\[ \text{LIST}_L(G(V, E), a) \{ \]
\[ l = 1; \]
\[ \text{repeat} \{ \]
\[ \quad \text{for each resource type } k = 1, 2, \ldots, n_{\text{res}} \{ \]
\[ \quad \quad \text{Determine candidate operations } U_{l,k}; \]
\[ \quad \quad \text{Determine unfinished operations } T_{l,k}; \]
\[ \quad \quad \text{Select } S_k \subseteq U_{l,k} \text{ vertices, such that } |S_k| + |T_{l,k}| \leq a_k; \]
\[ \quad \quad \text{Schedule the } S_k \text{ operations at step } l \text{ by setting } \]
\[ \quad \quad \quad t_i = l \text{ for all } i : v_i \in S_k; \]
\[ \quad \} \]
\[ l = l + 1; \]
\[ \} \]
\[ \text{until (all nodes are scheduled); } \]
\[ \text{return } (t) \]
\[ \} \]
An example CDFG: schedule it using list scheduling
Register Allocation

Clock cycle 1

Registers Implied

Clock cycle 2

Clock cycle 3

y = a + b + c + d
Lifetime Analysis

Registers 1 & 2 can be shared

Clock cycle

1

2

(R3 needed for output latch)

3

a

b

c

d

y

R1

R2

R3

(R3 needed for output latch)
y = (a+b+c) \times (d+e)

Clock cycle

1  2  3
**Binding 1**

**Results**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op1</td>
<td>Add1</td>
</tr>
<tr>
<td>Op2</td>
<td>Add2</td>
</tr>
<tr>
<td>Op3</td>
<td>Add1</td>
</tr>
</tbody>
</table>

- **Operation Table**
  - **Op1** is bound to **Add1**
  - **Op2** is bound to **Add2**
  - **Op3** is bound to **Add1**

- **Diagram**
  - The diagram shows a network of operations with inputs and outputs labeled **a**, **b**, **c**, **d**, **e**, **op1**, **op2**, **op3**, and **y**.
  - The operations include addition and multiplication.
  - The binding of operations to specific nodes is indicated in the diagram.

- **Table Description**
  - The table provides a clear mapping of operations to their corresponding bindings, aiding in understanding the functional flow of the diagram.
Binding 2

Results

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op1</td>
<td>Add1</td>
</tr>
<tr>
<td>Op2</td>
<td>Add2</td>
</tr>
<tr>
<td>Op3</td>
<td>Add2</td>
</tr>
</tbody>
</table>
Rolled vs. Unrolled Loops

- Unrolled loops
  - Perform everything in parallel
  - Not as much chance for scheduler optimization
  - Generally lead to faster designs

- Rolled loops
  - Schedule faster
  - May improve resource sharing
  - Generally extend latency
Pipelining and Multi-Cycle

- Given a synchronous system, clock must run at the speed of slowest functional unit and longest wire delay
- Complex operations (like multiply) can slow down whole design
  - Need to Pipeline or Multi-Cycle
- If you have many slow paths – rethink the architecture
Pipelining: split up logic and use faster clocks
Multi-Cycle: skip clock edges and use faster clocks

- combinational
  - enable
  - latch
Lecture 21: High-level Synthesis (2)

Slides courtesy of Deming Chen
Outline

- Binding for DFG
  - Left-edge algorithm
  - Network flow algorithm
- Binding to reduce interconnects
- Simultaneous scheduling and binding
- A case study: FCUDA
**Left edge algorithm for binding**

```
LEFT_EDGE(I) {
    Sort elements of I in a list L in ascending order of \( l_i \); /* left edge*/
    \( c = 0 \);
    while (some variable/operation has not been bound) do {
        \( S = 0; \ r = 0; \)
        while (there is an element in L whose left edge coordinate is larger than \( r \))
            do {
                \( s = \) first element in the list L with \( l_s > r \);
                \( S = S \cup \{s\}; \)
                \( r = r_s; \)
                delete \( s \) from L;
            }
        \( c = c + 1; \)
        bind \( S \) into one resource;
    }
}
```

**Optimal for interval graphs** - An interval graph is a graph whose vertices can be put in one-to-one correspondence with a set of intervals, so that two vertices are adjacent if and only if the corresponding intervals intersect.
Compatibility (Comparability)
Graph for functional unit

- Given a DFG $G$, build a compatibility graph, $G_c = (V_c, A_c)$
- $V_c$: all the operations in $G$ (For FU binding)
- $A_c$: all the edges between compatible operations in $V_c$
  - $a_c = (v_i, v_j)$ iff $T_D(v_i) < T_B(v_j)$
- $W_{ij}$: weight of $a_c$, the cost of binding $v_i$ and $v_j$ into a single FU
  - switching activity

<table>
<thead>
<tr>
<th>Op</th>
<th>$T_B$</th>
<th>$T_D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Life Times

$G$ (additions)

$G_c$
Compatibility (Comparability) Graph for variables

A *compatibility* graph, $G_c = (V_c, A_c)$ for $G$

$V_c:$ all the variables in $G$  (For register binding)

$A_c:$ all the edges between *compatible* variables in $V_c$, e.g., $a_c = (v_i, v_j)$ iff $DT(v_i) < BT(v_j)$

$w_{ij}:$ weight of $a_c$, the cost of binding $v_i$ and $v_j$ into a single register
Cost function to calculate $W_{ij}$

- Use the number of multiplexers to reflect the connection requirement
  - the more MUXes, the more connections, and the larger cost
- A MUX occurs in two situations:
  - before a port $p$ of a functional unit
  - before a register $R$

---

Case 1

- $F_1$ to $R$
- $F_2$ to $R$
- $MUX$ before $F_3$ and $F_4$

Case 2

- $MUX$ before $F_3$ and $F_4$
Cost function to calculate $W_{ij}$

Cost of binding $v_i$ and $v_j$ together (Case 2):

$$W_{ij} = -(N_{mux} + \partial \cdot T_{rf} + \beta \cdot T_{fu}) - L$$

- $N_{mux}$: the number of MUXes saved (or MUXes wasted) by binding $v_i$ and $v_j$ into a single register (Case 2) than not binding them into a single register (Case 1)
- $T_{rf}$: the total number of connections between register $R$ and the successor functional units
- $T_{fu}$: the total number of successor functional units involved during this tempted binding of $v_i$ and $v_j$
- $L$: a large positive constant
Obtaining the register binding solution

- Solution obtaining procedure
  - Build a network flow graph (a decent algorithm course would cover this topic)
  - Find the minimum cost $k$-flow in the network
  - Find all the edges with unit flow
    - edge connecting the vertices of variables $v_i$ and $v_j$: $v_i$ and $v_j$ should be bound together
    - edge connecting the vertices of a single variable $v_i$: $v_i$ occupies a register just by itself

- Features of the solution
  - Guarantee of optimal number of registers
  - Minimization of the numbers of MUXes and MUX inputs

Lecture 22: Logic Synthesis (1)

Slides courtesy of Deming Chen

Some slides Courtesy of Prof. J. Cong of UCLA
Outline

- Reading
  - Synthesis and optimization of digital circuits, G. De Micheli, 1994, Section 2.5-2.5.1

- Overview
  - Boolean algebra
  - Boolean functions
  - Boolean space
  - Boole’s expansion
Logic Synthesis and Optimization

- Determine microscopic structure of the circuit
- Explore (area-delay) trade-off
  - Combinational circuits:
    * I/O delay
  - Sequential circuits:
    * cycle-time
  - Enhance circuit testability
Example: Design Space

- Implement \( f = pqr \) with:
  - 2-input or 3-input AND gates.
- Suppose area and delay proportional to number of inputs
Logic Synthesis Problems

- Implementation styles:
  - Two-level (e.g. PLA macrocells).
  - Multi-level (e.g. cell-based, array-based).
    * Technology-independent optimization
    * Technology mapping

- Operations
  - Combinational.
  - Sequential:
    * Synchronous
    * Asynchronous
Combinational Logic Design

Background

- Boolean algebra
  - Quintuple (B, +, ·, 0, 1)
  - Binary Boolean algebra B={0, 1}
- Boolean function
  - N-input, single output: $f: B^n \rightarrow B$.
  - N-input, multiple output: $f: B^n \rightarrow B^m$.
  - Incompletely specified:
    * don’t care symbol ‘*’.
    * $f: B^n \rightarrow \{0, 1, *\}^m$. 
Some Properties of Boolean Algebraic Systems

- $a + (b + c) = (a + b) + c$  
  **Associativity**
- $a(bc) = (ab)c$  
  **Associativity**
- $a + a = a$  
  **Idempotence**
- $aa = a$  
  **Idempotence**
- $a + (ab) = a$  
  **Absorption**
- $a(a + b) = a$  
  **Absorption**
- $(a + b)' = a'b'$  
  **De Morgan**
- $(ab)' = a' + b'$  
  **De Morgan**
- $(a')' = a$  
  **Involution**
- $a + a'b = a + b$
The *don’t care* Conditions

- We don’t care about the value of the function.
- Related to the environment:
  - Input patterns that never occur (input controllability don’t-cares).
  - Input patterns such that some output is never observed (output observability don’t cares).
- Very important for synthesis and optimization
Definitions

- **Scalar function**
  - **ON-set:** subset of the domain such that $f$ is true.
  - **OFF-set:** subset of the domain such that $f$ is false.
  - **DC-set:** subset of the domain such that $f$ is a *don’t care*.

- **Multiple-output function:**
  - Defined for each component.
Cubical Representation

\[\begin{array}{ccc}
000 & 010 & 100 \\
001 & 101 & 110 \\
111 & & \\
\end{array}\]

\[\begin{array}{ccc}
a'b'c & a'b'c' & abc' \\
 abc & ab'c' & ab'c \\
& a'bc & a'bc' \\
\end{array}\]
Definitions (Cont’d)

- **Boolean variables.**
- **Boolean literal:**
  - an instance of a variable or of its complement.
- **Product or cube:**
  - product of literals.
- **Implicant:**
  - product implying a value of a function (usually TRUE).
    - Hypercube in the Boolean space.
- **Minterm:**
  - product of all input variables implying a value of a function (usually TRUE).
    - Vertex in the Boolean space.
Tabular Representations

- **Truth table:**
  - List of all minterms of a function

- **Implicant table or cover**
  - List of implicants of a function sufficient to define function.

- Remark:
  - Implicant tables are smaller in size.
## Example of Tabular Representations

- **Example of Truth Table**

  \[ x = ab + a'c; \quad y = ab + bc + ac \]

<table>
<thead>
<tr>
<th>abc</th>
<th>( xy )</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>001</td>
<td>10</td>
</tr>
<tr>
<td>010</td>
<td>00</td>
</tr>
<tr>
<td>011</td>
<td>11</td>
</tr>
<tr>
<td>100</td>
<td>00</td>
</tr>
<tr>
<td>101</td>
<td>01</td>
</tr>
<tr>
<td>110</td>
<td>11</td>
</tr>
<tr>
<td>111</td>
<td>11</td>
</tr>
</tbody>
</table>

- **Example of implicant table**

  \[ x = ab + a'c; \quad y = ab + bc + ac \]

<table>
<thead>
<tr>
<th>abc</th>
<th>( Xy )</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>10</td>
</tr>
<tr>
<td>*11</td>
<td>11</td>
</tr>
<tr>
<td>101</td>
<td>01</td>
</tr>
<tr>
<td>11*</td>
<td>11</td>
</tr>
</tbody>
</table>

---

ECE 425  
CMOS VLSI Design 4th Ed.  
Slide 165
Cubical Representation of Minterms and Implicants

\[ f_1 = a'b'c' + a'b'c + ab'c + abc + abc' \] (simplify)

\[ f_2 = a'b'c + ab'c \] (simplify)
Cofactor Functions

- Function \( f(x_1, x_2, ...x_i, ... x_n) \).

- **Cofactor** of \( f \) with respect to variable \( x_i \):
  \[
  f_{x_i} = f(x_1, x_2, ...1, ..., x_n)
  \]

- **Cofactor** of \( f \) with respect to variable \( x'_i \):
  \[
  f_{x'_i} = f(x_1, x_2, ..., 0, ..., x_n)
  \]

- **Boole’s expansion theorem**:
  \[
  f(x_1, x_2, ..., x_i, ..., x_n) = x_i \cdot f_{x_i} + x'_i \cdot f_{x'_i}
  \]
Unate Functions

- Function $f( x_1, x_2, \ldots x_i, \ldots x_n )$.
- Positive unate in $x_i$ when (set of minterms of $f_{x_i}$ includes the set of minterms of $f_{x_i'}$)

$$f_{x_i} \geq f_{x_i'} \quad \text{or} \quad f_{x_i} \supseteq f_{x_i'}$$

- Negative unate in $x_i$ when

$$f_{x_i} \leq f_{x_i'} \quad \text{or} \quad f_{x_i} \subseteq f_{x_i'}$$

- A function is positive / negative unate when positive / negative unate in all its variables.
More on Boolean Functions

- Function \( f( x_1, x_2, \ldots x_i, \ldots x_n ) \).
- **Boolean difference** of \( f \) w. r. to variable \( x_i \):
  \[
  \frac{\partial f}{\partial x_i} = f_{x_i} \oplus f_{x'_i}
  \]
  (whether \( f \) is sensitive to changes in input \( x_i \). When it is 0, the function does not depend on \( x_i \))
- **Consensus** of \( f \) w. r. to variable \( x_i \):
  \[
  C_{x_i} = f_{x_i} \cdot f_{x'_i}
  \]
  (the part of on-set that is independent of \( x_i \))
- **Smoothing** of \( f \) w. r. to variable \( x_i \):
  \[
  S_{x_i} = f_{x_i} + f_{x'_i}
  \]
  (deleting all appearances of \( x_i \))
Lecture 23: Logic Synthesis (2)

Slides courtesy of Deming Chen

Some slides Courtesy of Prof. J. Cong of UCLA
Outline

- Reading
  - Synthesis and optimization of digital circuits,
    G. De Micheli, 1994, Section 2.5.2

- Overview
  - BDD (binary decision diagram)
  - OBDD (ordered BDD)
  - ROBDD (reduced OBDD)
  - ITE (if-then-else operator)
  - Introduction of mapping
Binary Decision Diagrams

- Efficient representation of logic functions
  - Proposed by Lee and Akers.
  - Popularized by Bryant (canonical form)
- Used for Boolean manipulations
- Applicable to other domains:
  - Set and relation representation
  - Simulations, finite-system analysis

**Definition:**
- A BDD is a tree or a rooted DAG with a decision at each vertex.
Ordered BDD

- Each decision is the evaluation of a boolean variable
- The tree (or dag) can be levelized so that each level corresponds to a Boolean variable
- Definition
  - Rooted DAG
  - Each non-leaf vertex (v) has
    - A pointer index to a variable
    - Two children: low(v) and high(v)
  - Each leaf vertex (v) has a value (1 or 0)
- Ordering
  - index(v) < index(low(v))
  - index(v) < index(high(v))
Reduced OBDDs

- No redundancy
  - no vertex with \( \text{low}(v) = \text{high}(v) \)
  - no pair \( \{u,v\} \) with isomorphic subgraphs rooted at \( u, v \)
- Reduction can be achieved in polynomial time
- ROBDDs can be such by construction
- ROBDDs (canonical) allow us to
  - Verify logic equivalence in constant time
  - perform logic operations in time proportional to the graph size (vertex cardinality)
  - Although in some cases this graph approach is of exponential complexity, many cases result in very compact representations
- Drawback: ROBDD size depends on variable ordering
OBDD Reduction Algorithm

Outline

1) Visit OBDD bottom up
2) Label each vertex \( v \) with an identifier \( id(v) \)
3) For each \( v \) in the subset of vertices with index \( i \) (level \( i \))
   a) If \( id(low(v)) = id(high(v)) \), we set \( id(v) = id(low(v)) \) (\( v \) is redundant)
   b) If there are \( u \) & \( v \) such that
      \( id(low(v)) = id(low(u)) \) and
      \( id(high(v)) = id(high(u)) \) then we set \( id(v) = id(u) \) (isomorphic)
4) In the remaining cases, different \( id \) given to each vertex at level \( i \)
5) Terminate at the root.
The \textit{ite} Operator

- Apply operators to ROBDDs
  - Three boolean functions: $f,g,h$ with top variable $x$

  - \textit{ite}(f,g,h)
    - If $f$ then $g$ else $h$

    - $fg + f'\ h$

    - Recursive Property:
      \[
      \text{ite}(f,g,h) = \text{ite}(x, \text{ite}(f_x, g_x, h_x), \text{ite}(f'_x, g'_x, h'_x))
      \]
## ITE - Boolean operators

<table>
<thead>
<tr>
<th></th>
<th>ite(0,0,0)</th>
<th></th>
<th>ite(1,1,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>ite(f,1,0)</td>
<td>g</td>
<td>ite(g,1,0)</td>
</tr>
<tr>
<td>f'</td>
<td>ite(f,0,1)</td>
<td>g'</td>
<td>ite(g,0,1)</td>
</tr>
<tr>
<td>f·g</td>
<td>ite(f,g,0)</td>
<td>(f·g)'</td>
<td>ite(f,g',1)</td>
</tr>
<tr>
<td>f'·g</td>
<td>ite(f,0,g)</td>
<td>f·g'</td>
<td>ite(f,g',0)</td>
</tr>
<tr>
<td>f+g</td>
<td>ite(f,1,g)</td>
<td>(f+g)'</td>
<td>ite(f,0,g')</td>
</tr>
<tr>
<td>f'+g</td>
<td>ite(f,g,1)</td>
<td>f+g'</td>
<td>ite(f,1,g')</td>
</tr>
<tr>
<td>f ⊕ g</td>
<td>ite(f,g',g)</td>
<td>f xnor g</td>
<td>ite(f,g,g')</td>
</tr>
</tbody>
</table>
The *ITE* algorithm

- Used to construct the ROBDD of a function.
- Evaluate the ite(f,g,h) operator recursively – bottom up
- Keep OBDDs in reduced form
- Use two tables (per function):
  - Unique table: represents ROBDD
  - Computed table: stores previous info
- Smart implementations of *ITE* have linear time complexity in the product of the ROBDD sizes.
ITE (if-then-else) Algorithm

ITE(f,g,h) /* one call of ITE */
    if (terminal case)
        return (r=trivial result);
    else {
        if (computed table has entry {(f,g,h),r})
            return (r from computed table)
        else {
            x = top var of f,g,h
            t = ite(f\_x, g\_x, h\_x)
            e = ite(f\_x', g\_x', h\_x')
            if (t==e) /* isomorphic */
                return (t);
            r = find\_or\_add\_unique\_table(x,t,e);
            update computed table with {(f,g,h),r}
            return (r);
        }
    }

Technology Mapping

- Technology mapping converts a given Boolean circuit (a netlist) into a functionally equivalent network comprised only of LUTs or PLAs
- Technology mapping is a crucial optimization step in the programmable logic design flow
- Direct impact on
  - Delay (number of levels of logic)
  - area/power (number of LUTs or PLAs)
  - Interconnects (number of edges)
Definitions and Motivation

- **DAG:** Boolean network
- **Cone C_v:** sub-network rooted on node v
- **K-feasible cone:** \(|\text{input}(C_v)| \leq K\)
- **Fanin Cone F_v:** the largest C_v
- **K-feasible cut:** a K-feasible C_v
- **Unit delay model:**
  - One LUT contributes one unit delay
  - No edge delay

Diagram:
- **Pls**
- **F_v**
- **3-feasible cone C_v**
- **Delay of 2**
Combine sub-cuts on the inputs of the gate
Process each gate in topological order from PIs to POs
Cut Selection – Mapping Generation

- From POs to PIs
- Critical paths
  - optimal delay + best area available
- Non-critical paths
  - relaxed delay + better area

LUT roots in list L
- L: f, g
- L: g, e, d
- L: e, d
- L: b
Lecture 24:
Partitioning and Floorplan

Slides courtesy of Deming Chen

Some slides Courtesy of Prof. J. Cong of UCLA
Outline

- Now we get into physical design
  - Circuit partitioning
  - Circuit clustering
  - Floorplanning

- We will talk about several typical algorithms

- Next lecture will discuss placement and routing

Circuit Partitioning

Formulation

Bi-partitioning formulation:

Minimize interconnections between partitions

- Minimum cut: \( \min c(x, x') \)
- minimum bisection: \( \min c(x, x') \) with \( |x| = |x'| \)
- minimum ratio-cut: \( \min \frac{c(x, x')}{|x||x'|} \)
Circuit Partitioning Formulation (Cont’d)

General multi-way partitioning formulation:

Partitioning a network $N$ into $N_1, N_2, \ldots, N_k$ such that

1. Each partition has an area constraint

$$\sum_{v \in N_i} a(v) \leq A_i$$

2. Each partition has an I/O constraint

$$c(N_i, N - N_i) \leq I_i$$

Minimize the total interconnection:

$$\sum_{N_i} c(N_i, N - N_i)$$
Importance of Circuit Partitioning

- Divide-and-conquer methodology
  - The most effective way to solve problems of high complexity
  - E.g.: min-cut based placement, partitioning-based test generation,…

- System-level partitioning for multi-chip designs
  - Inter-chip interconnection delay dominates system performance.

- Circuit emulation/parallel simulation
  - Partition large circuit into multiple FPGAs or multiple special-purpose processors

- Parallel CAD development
  - Task decomposition and load balancing

- In deep-submicron designs, partitioning defines local and global interconnect, and has significant impact on circuit performance
Iterative Partitioning Algorithms

1. Greedy Iterative improvement method
   [Kernighan-Lin 1970]
   [Fiduccia-Mattheyses 1982]
   [krishnamurthy 1984]

2. Simulated Annealing
   [Kirkpartrick-Gelatt-Vecchi 1983]
   [Greene-Supowit 1984]
Kernighan-Lin’s Algorithm

1. Pair-wise exchange of nodes to reduce cut size
2. Allow cut size to increase temporarily within a pass
3. Compute the gain of a swap
   Repeat
   Perform a feasible swap of max gain
   Mark swapped nodes “locked”;
   Update swap gains;
   Until no feasible swap;
   Find min cutsize in the resulting sequence $c_1, c_2, \ldots, c_m$
   Make corresponding swaps permanent.

Start another pass if current pass reduces the cut size (usually converge after a few passes)
Assume edge weight = 1

- Decrease of cutsize measured by a gain value $D(i)$ of $V_i$
  - $D(i) = \text{outedge}(i) - \text{inedge}(i)$
- When $v_i$ and $v_j$ are exchanged, the decrease of cutsize is $D(i) + D(j)$
- If the two nodes share an edge, the gain is $D(i) + D(j) - 2$

- Can be extended to non-integer edges
Fiduccia-Mattheyses’ Improvement

1. Each pass in KL-algorithm takes $O(n^3)$ ($n$: #modules).

2. FM-algorithm takes $O(n)$ time per pass

3. Key ideas in FM-algorithms
   - Cell move instead of cell swapping
     ⇒ the cell with max gain is moved
   - Maintain a list of gain buckets
     ⇒ update gain values of affected cells after each move

4. Further improvement by Krishnamurthy
   Look-ahead in gain computation
Circuit Clustering Formulation

- Motivation:
  - Reduced the size of flat netlists
  - Identify natural circuit hierarchy

- Objectives:
  - Maximize the connectivity of each cluster
  - Minimize the size, delay (or simply depth), and density of clustered circuits
Lawler’s Labeling Algorithm
[Lawler-Levitt-Turner 1969]

1. Assumption: Cluster size $\leq K$; Intra-cluster delay $= 0$; Inter-cluster delay $= 1$
2. Objective: Find a clustering of minimum delay
3. Algorithm:

Phase 1: Label all nodes in topological order
- For each PI node $V$, $L(v) = 0$;
- For each non-PI node $v$
  - $p =$ Maximum label of predecessors of $v$
  - $X_p =$ set of predecessors of $v$ with label $p$
  - if $|X_p| < K$ then $L(v) = p$ else $L(v) = p + 1$

Phase 2: Form clusters
- Start from PO to generate necessary clusters
- Nodes with the same label form a cluster
Floorplan

- Floorplan is a very important step for design planning
- Floorplan may be performed before synthesis of each block (at RTL level)
- Slicing floorplan allows efficient representation and optimization
  - Module orientation and sizing problem can be solved easily
  - Good results using SA
- Non-slicing floorplan is harder to solve
  - Generalized cutlines
  - Sequence pair representation
Problem formulation

Given, for each block $B_i$

- Area of $B_i$
  \[ w_i h_i = A_i \]

- Aspect ratio of $B_i$
  \[ s_i \leq \frac{w_i}{h_i} \leq r_i \]
  (continuous or discrete)

- Connectivity

Determine: for each block
location $(x_i, y_i)$, and
dimension $(w_i, h_i)$

\[ \text{min} \]

total area
interconnections
Slicing Floorplan and General Floorplan

Slicing floorplan

non-slicing floorplan

Slicing Tree
Generic Simulated Annealing Algorithm

1. Get an initial solution \( S \)

2. Get an initial temperature \( T > 0 \)

3. While not yet “frozen” do the following:
   
   3.1 For \( 1 \leq i \leq L \), do the following:
      
      3.1.1 Pick a random neighbor \( S' \) of \( S \)
      
      3.1.2 Let \( \Delta = \text{cost}(s') - \text{cost}(s) \)
      
      3.1.3 If \( \Delta \leq 0 \) (downhill move),
         
         Set \( S = S' \)
      
      3.1.4 If \( \Delta > 0 \) (uphill move)
         
         set \( S = S' \) with probability \( e^{-\Delta/T} \)

3.2 Set \( T' = rT \) (reduce temperature)

4. Return \( S \)
Representation of Solutions

\[ 16 + 35 \times 2 + 7 = 4 \]

\[ 1 + 6 + 2 + 7 + 4 \]

\[ 3 + 5 \]
Neighborhood Structure

Chain: $*$*$*$*..... or $*$ $*$*$*$*.....

$16 + 35 \star 2 + \star 74 + \star$

The moves:

M1: swap adjacent operands
M2: Complement some chain of operators
M3: swap 2 adjacent operand and operator

Representative work: D. F. Wong and C. L. Liu, A new algorithm for floorplan design, DAC’86.
Lecture 25: Placement and Routing

Slides courtesy of Deming Chen

Some slides courtesy of Prof. J. Cong and N. Carter
Outline

- Overview
  - The placement problem
  - Partitioning-based placement
  - The routing problem
  - Global routing and detailed routing

The Placement Problem

- *Placement* is to find locations for the cells in a netlist such that they
  - Fit within the available space
  - The router can make all of the necessary connections between cells
- Placement and routing are inherently connected
  - Standard CAD tools do them in separate passes to simplify the algorithms
  - Algorithms that do placement and routing simultaneously to improve performance are an active research area
What Makes a Placement Good?

- **Routability** -- if you can’t route the required connections, design won’t work
- **Performance** -- placement affects wire delays, which affects performance
- **Density** -- what percentage of the available area can the placer use while still maintaining routability
  - ~70% for current tools, depending on design
- **Physical Issues** -- May want to distribute power dissipation/heat generation/clock load across the chip
- Fortunately, designs that have good routability often have good performance, because placing connected modules close together improves both metrics
Placement Methods

- Constructive methods.
  - Cluster growth algorithm
  - Force-directed method
  - Algorithm by Goto
  - Min-cut based method

- II. Iterative improvement approaches
  - Pairwise exchange
  - Simulated annealing - Timberwolf
  - Genetic algorithm

- III. Analytical methods
Interconnection Cost

(a) Steiner Tree
Rectilinear Length = 14

(b) Steiner Tree with Trunk
Rectilinear Length = 15

(c) Minimum Spanning Tree
Rectilinear Length = 16

(d) Chain
Rectilinear Length = 17

(e) Complete Graph
Rectilinear Length = 42

Approximation: half perimeter of the bounding box
Min-Cut Based Placement

“A procedure for placement of standard-call VLSI circuits”

A.E. Dunlop, B.W. Kernighan, IEEE Trans. on CAD, vol CAD-4 No.1, Jan 1985

Min-cut with terminal propagation

\[
\text{minimize} \quad \frac{1}{2} n \quad \frac{1}{2} n
\]
Min-Cut Based Placement

(Cont’d)

This process continues until there are only a few cells in each group ($\approx 6$)

- Each group has $\leq 6$ cells

Assign cells in each group close together in the same row or nearly in adjacent rows

Group: smallest partition
Terminal Propagation

We should use the fact that s is in L₁!

p will stay in R₁ for the rest of partitioning
Terminal Propagation

When not to use p to bias partitioning

In this case, p should not be used to bias the solution in either direction.

In general:

Do not use p

Net s has cells in many groups:

Minimum cost rectilinear sterner tree

P₂ should be ignored!

too close to the partition line
Terminal Propagation (Cont’d)

Terminal propagation reduce overall area by ~30%

Creating Rows

Row 1 \{ \[ C_1 \] C_2 \] C_3 \} cells in $C_1 \rightarrow$ row1
Row 2 \{ \}
cells in $C_3 \rightarrow$ row1
Row 3 \{ \}
cells in $C_2$
Row 4 \{ \}

Choose $\alpha$ and $\beta$ preferably to balance row to balance row length (During re-arrangement)
Creating Rows

Partitioning of circuit into 32 groups. Each group is either assigned to a single row or divided into 2 rows.
The Routing Problem

- Given a set of cells, a set of pin locations on each cell, and a set of nets connecting cells, connect the cell pins with wires to implement the nets
  - May also want to achieve other goals:
    - Minimize overall area
    - Minimize length of each wire
- This is a hard problem -- routing can consume more area than actual cells
- The routing problem on modern chips involves managing several metal layers and assigning wires to them
Global Routing Formulation

**Given**
(i) Placement of blocks/cells
(ii) channel capacities

**Determine**
Routing topology of each net

**Optimize**
(i) max # nets routed
(ii) min routing area
(iii) min total wirelength

In general cell design or standard cell design, we are able to move blocks or cell rows, so we can guarantee connections of all the nets.

In gate array design, exceeding channel capacity is not allowed.
Routing

- First, divide the chip into regions of the type that the router can handle.
- Then, use a *global router* to assign each net to a set of regions.
- Finally, use a *detailed router* to lay out wires within the regions assigned to each net.
Many different types of routers have been developed.
Global Routing

- Can work on channel-graph
- Edges in the channel graph represent channels or subsections of channels
  - Weights on edges represent the capacity of the channel
- Vertices represent intersections between channels
- Goal of global router is to assign wires to channels without exceeding any capacities
  - Detailed router worries about wire position within each channel
Global Routing Algorithms

- The problem is to find a suitable path for each connection through the weighted graph.
- Ideally, we would like to find minimum-cost trees, but:
  - This is \textit{NP}-complete.
  - Does not model net interactions, required to control congestion.
- Instead, heuristic techniques are used, usually consisting of two phases:
  - An Initial Global Route Technique: use shortest path algorithms to find some routing (may be congested).
  - An Iterative Improvement Global Route Technique: select and re-route connections to reduce congestion.
Detailed Routers

- Once nets have been assigned to routing regions, use a detailed router to find exact wire locations.
- We’ll look at three detailed routers:
  - Maze routers
  - Line-probe routers
  - Channel Routers

- Maze and line-probe routers can handle arbitrary geometries.
- Channel routers can only handle rectangular channels.
Maze Routers

- Oldest kind of router, first used for printed circuit boards
- Operate on a grid graph, where the surface is represented as an array of cells

Section of a Printed Circuit Board

Corresponding Gridded Model
Maze Routers -- the Lee Algorithm

- Always finds a shortest path if one exits
- Starting at the source node, explore out in a breadth-first fashion, keeping a list of the leaf nodes of the exploration tree
- At each step, replace each leaf node with its neighbors and update their distance to the source node
- Stop when you see the target cell (vertex)
- Block cells used by this wire and go on to the next
Lee Algorithm Illustration
Line-Probe Routers

- First proposed by Mikami/Tabuchi and Hightower (independently)
- Model routing surface as a list of lines
- Basic idea: project line-probes from starting point until you encounter an obstruction
Line-Probe Routers

- Specifically:
  - Project probes in both the horizontal and vertical directions.
  - If obstructed, choose *escape points* along the current probes.
  - Project new segments from the escape points.

- This continues until two probes originating from the root and the target intersect.

- The two algorithms (Mikami and Hightower) differ in the way they choose escape points:
  - Mikami generates additional escape points at every grid point.
    - It is breadth-first and guarantees a solution, if one exists
  - Hightower adds a single escape point, appropriately chosen.
    - It does not guarantee a solution, even if one exists.
Line-Probe Routers

- The added overhead of Mikami’s algorithm can be seen in this example:

Mikami Algorithm

Hightower Algorithm

- The main advantage of line-probe routers, over maze routers, is their reduced memory requirements.
Channel Routers

- By requiring that the routing region be a rectangle, we can reduce routing effort.
- A channel is a rectangular routing region with no obstacles and with pins only on two parallel sides.
- We’ll represent channels as follows:
Channel Routers

- Within a channel, the routing problem now becomes simpler: ensure that the number of wires in the channel doesn’t exceed its capacity at any point.

- One simple algorithm: the left-edge algorithm
  1. Imposes simplifying restrictions
  2. Each net can only use one trunk section
  3. Trunks must be in one routing layer, branches in another
  4. With these constraints, the algorithm determines the minimum number of tracks required to route a set of wires.
Left-Edge Algorithm

1. Sort trunks by their left-edge coordinate

2. Place trunks in sequence, using free space before adding tracks

3. Add branches to the placed trunks
Left-Edge Algorithm

- Has problems if there are multiple terminals at one horizontal position
  - In PCB environment, could handle this by *jogging* one of the conflicting segments

- This doesn’t work in an IC environment, making the problem NP-complete
  - Practical Left-Edge algorithms use heuristics that don’t guarantee finding the absolute minimum number of channels
Question 7: List Scheduling
Practice Final Exam

- Question 4: Left-Edge Channel Routing
  - Routed result
  - Channel density

![Diagram showing channel routing and density](image)
Textbook Chapters

- Weste and Harris
  - 5.1-5.3
  - 7.1-7.3
  - 11.3-4, 11.8-9
  - 12.1-12.2.5, 12.4-12.7
  - 14.3-4
  - 15.1-15.7

- De Micheli
  - 2.5-2.5.2
  - 5

- Sherwani
  - 5.1-5.4, 6.1
  - 7-7.1, 7.4, 8-8.4, 9.1.3, 9.4.2
Textbook Exercises

- Weste and Harris
  - 5.1, 5.11
  - 11.2
  - 12.1, 12.4, 12.10
  - 15.4, 15.7

- De Micheli
  - 2.4, 2.6, 2.7
  - 5.1

- Sherwani
  - 7.2