Review: Combining Combinational Elements + DeMorgan Equivalence

Wire
<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Inverter
<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

NAND Gate

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

NOR Gate

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

DeMorgan's Theorem

Out = A + B = \overline{A} \cdot B

Out = A \cdot B = \overline{A} + B

Review: recall General C/L Cell Delay Model

Combinational Cell (symbol) is fully specified by:
- functional (input -> output) behavior
  - truth-table, logic equation, VHDL
- load factor of each input
- critical propagation delay from each input to each output for each transition
  - \( T_{in}(A, o) = \text{Fixed Internal Delay} + \text{Load-dependent-delay} \times \text{load} \)

Linear model composes

Storage Element’s Timing Model

° Setup Time: Input must be stable BEFORE trigger clock edge
° Hold Time: Input must REMAIN stable after trigger clock edge
° Clock-to-Q time:
  - Output cannot change instantaneously at the trigger clock edge
  - Similar to delay in logic gates, two components:
    - Internal Clock-to-Q
    - Load dependent Clock-to-Q
Critical Path & Cycle Time

- Critical path: the slowest path between any two storage devices
- Cycle time is a function of the critical path
- must be greater than:
  
  \[ \text{Clock-to-Q} + \text{Longest Path through Combination Logic} + \text{Setup} \]

The Design Process

“To Design Is To Represent”

Design activity yields description/representation of an object

- Traditional craftsman does not distinguish between the conceptualization and the artifact
- Separation comes about because of complexity
- The concept is captured in one or more representation languages
- This process is design

Design Begins With Requirements

- Functional Capabilities: what it will do
- Performance Characteristics: Speed, Power, Area, Cost, . . .

Design Process (cont.)

Design Finishes As Assembly

- Design understood in terms of components and how they have been assembled
- Top Down decomposition of complex functions (behaviors) into more primitive functions
- Bottom-up composition of primitive building blocks into more complex assemblies

Design is a “creative process,” not a simple method

Design Refinement

Informal System Requirement

- Initial Specification
- Intermediate Specification
- Final Architectural Description
- Intermediate Specification of Implementation
- Final Internal Specification

Physical Implementation

refinement increasing level of detail
Design as Search

Design involves educated guesses and verification
-- Given the goals, how should these be prioritized?
-- Given alternative design pieces, which should be selected?
-- Given design space of components & assemblies, which part will yield the best solution?

Feasible (good) choices vs. Optimal choices

Performance: Two notions of “performance”

<table>
<thead>
<tr>
<th>Plane</th>
<th>DC to Paris</th>
<th>Speed</th>
<th>Passengers</th>
<th>Throughput (pmph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boeing 747</td>
<td>6.5 hours</td>
<td>610 mph</td>
<td>470</td>
<td>286,700</td>
</tr>
<tr>
<td>BAD/Sud Concorde</td>
<td>3 hours</td>
<td>1350 mph</td>
<td>132</td>
<td>178,200</td>
</tr>
</tbody>
</table>

Which has higher performance?
° Time to do the task (Execution Time)
  -- execution time, response time, latency
° Tasks per day, hour, week, sec, ns.. (Performance)
  -- throughput, bandwidth
Response time and throughput often are in opposition

Measurement and Evaluation

Architecture is an iterative process
-- searching the space of possible designs
-- at all levels of computer systems

Creativity

Cost / Performance Analysis

Good Ideas
Bad Ideas
Mediocre Ideas

What is Time?
° Straightforward definition of time:
  • Total time to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, ...
  • “real time”, “response time” or “elapsed time”
° Alternative: just time processor (CPU) is working only on your program (since multiple processes running at the same time)
  • “CPU execution time” or “CPU time”
  • Often divided into system CPU time (in OS) and user CPU time (in user program)
How to Measure Time?

° User Time $\Rightarrow$ seconds

° CPU Time: Computers constructed using a clock that runs at constant rate
  - These discrete time intervals called clock cycles (or informally clocks or cycles)
  - Length of clock period: clock cycle time (e.g., 250 picoseconds or 250 ps) and clock rate (e.g., 4 gigahertz, or 4 GHz), which is the inverse of the clock period; use these!

Measuring Time using Clock Cycles

° CPU execution time for program
  $= \text{Clock Cycles for a program} \times \text{Clock Cycle Time}$

° One way to define clock cycles:
  - Clock Cycles for program
  $= \text{Instructions for a program (called “Instruction Count”)} \times \text{Average Clock cycles Per Instruction (called “CPI”)}$

Performance Calculation

° CPU execution time for program
  $= \text{Clock Cycles for program} \times \text{Clock Cycle Time}$

° Substituting for clock cycles:
  CPU execution time for program
  $= (\text{Instruction Count} \times \text{CPI}) \times \text{Clock Cycle Time}$
  $= \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$

CPU time $= \frac{\text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}}{\text{Seconds}}$

How Calculate the 3 Components?

° Clock Cycle Time: in specification of computer (Clock Rate in advertisements)

° Instruction Count:
  - Count instructions in loop of small program
  - Use simulator to count instructions
  - Hardware counter in spec. register (most CPUs)

° CPI:
  - Calculate: $\frac{\text{Execution Time}}{\text{Clock cycle time}}\times\text{Instruction Count}$
  - Hardware counter in special register (most CPUs)
Example: Calculating CPI by averaging separately

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq&lt;sub&gt;i&lt;/sub&gt;</th>
<th>CPI&lt;sub&gt;i&lt;/sub&gt;</th>
<th>Prod</th>
<th>(% Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>(33%)</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>(27%)</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>2</td>
<td>.2</td>
<td>(13%)</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>(27%)</td>
</tr>
</tbody>
</table>

**Instruction Mix**

1.5 (Where time spent)

- What if Branch instructions twice as fast?

What Programs Measure for Comparison?

- Ideally run typical programs with typical input before purchase, or before even build machine
  - Called a “workload”; For example:
    - Engineer uses compiler, spreadsheet
    - Author uses word processor, drawing program, compression software
- In some situations it’s hard to do
  - Don’t have access to machine to “benchmark” before purchase
  - Don’t know workload in future
- Standardized benchmarks
  - Obviously, apparent speed of processor depends on code used to test it
  - Need industry standards so that different processors can be fairly compared
  - Companies exist that create these benchmarks: “typical” code used to evaluate systems
  - Need to be changed every 2 or 3 years since designers could target these standard benchmarks

Amdahl’s Law: The Law of Diminishing Returns

- Speedup due to enhancement E:
  \[
  \text{Speedup}(E) = \frac{\text{ExTime w/o E}}{\text{ExTime w/ E}} \times \frac{\text{Performance w/ E}}{\text{Performance w/o E}}
  \]

- Suppose that enhancement E accelerates a fraction F of the task by a factor S, and the remainder of the task is unaffected

- Then Maximum benefit:
  \[
  \text{Speedup}_{\text{maximum}} = \frac{1}{1 - \text{Fraction time affected}}
  \]

Administrivia

- Prereq Quizes not graded yet
  - Hopefully very soon
- Lab 1 due tonight by midnight.
  - You will be graded in your lab report on methodology:
    - Not just the fact that you found the bugs!
    - How did you approach the debugging process?
    - Why do you think that you have things completely covered?
- Lab #2/Homework #2 starting today (or tomorrow)
- Office hours in Lab
  - Mon 5 – 6:30 Jack, Tue 3:30-5 Kurt, Wed 3 – 4:30 John
- Kubi’s office hours Monday 3:30 – 5
Computers in the Real World

Problem: Prof. Dawson monitors redwoods by climbing trees, stringing miles of wire, placing printer sized data logger in tree, collect data by climbing trees (300' high)

Solution: CS Prof. Culler proposes wireless “micromotes” in trees. Automatically network together (without wire). Size of film canister, lasts for months on C battery, much less expensive. Read data by walking to base of tree with wireless laptop. “Will revolutionize environmental monitoring”


Problem: Design a “fast” ALU for the MIPS ISA

Requirements?

Must support the Arithmetic / Logic operations

Tradeoffs of cost and speed based on frequency of occurrence, hardware budget

We will optimize time to do ALU ops for today

MIPS ALU requirements

- Add, AddU, Sub, SubU, AddI, AddIU
  - => 2’s complement adder/sub with overflow detection
- And, Or, AndI, OrI, Xor, Xori, Nor
  - => Logical AND, logical OR, XOR, nor
- SLTI, SLTIU (set less than)
  - => 2’s complement adder with inverter, check sign bit of result
- ALU from from CS 150 / P&H book chapter 4 supports these ops

MIPS arithmetic instruction format

<table>
<thead>
<tr>
<th>Type</th>
<th>op</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>op</td>
<td>Rs</td>
</tr>
<tr>
<td>I-type</td>
<td>op</td>
<td>Rs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>op</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>10</td>
<td>xx</td>
</tr>
<tr>
<td>ADDIU</td>
<td>11</td>
<td>xx</td>
</tr>
<tr>
<td>SLTI</td>
<td>12</td>
<td>xx</td>
</tr>
<tr>
<td>SLTIU</td>
<td>13</td>
<td>xx</td>
</tr>
<tr>
<td>ANDI</td>
<td>14</td>
<td>xx</td>
</tr>
<tr>
<td>ORI</td>
<td>15</td>
<td>xx</td>
</tr>
<tr>
<td>Xori</td>
<td>16</td>
<td>xx</td>
</tr>
<tr>
<td>LUI</td>
<td>17</td>
<td>xx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>op</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>00</td>
<td>40</td>
</tr>
<tr>
<td>ADDU</td>
<td>00</td>
<td>41</td>
</tr>
<tr>
<td>SUB</td>
<td>00</td>
<td>42</td>
</tr>
<tr>
<td>SUBU</td>
<td>00</td>
<td>43</td>
</tr>
<tr>
<td>AND</td>
<td>00</td>
<td>44</td>
</tr>
<tr>
<td>OR</td>
<td>00</td>
<td>45</td>
</tr>
<tr>
<td>XOR</td>
<td>00</td>
<td>46</td>
</tr>
<tr>
<td>NOR</td>
<td>00</td>
<td>47</td>
</tr>
</tbody>
</table>

Signed arithmetic generate overflow, no carry
Design Trick: divide & conquer

- Trick #1: Break the problem into simpler problems, solve them and glue together the solution
- Example: assume the immediates have been taken care of before the ALU
  - 10 operations (4 bits)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 00 |   |   |   |   |   |   |   |   | add |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 01 |   |   |   |   |   |   |   |   | addU|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 02 |   |   |   |   |   |   |   |   | sub |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 03 |   |   |   |   |   |   |   |   | subU|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 04 |   |   |   |   |   |   |   |   | and |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 05 |   |   |   |   |   |   |   |   | or  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 06 |   |   |   |   |   |   |   |   | xor |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 07 |   |   |   |   |   |   |   |   | nor |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 08 |   |   |   |   |   |   |   |   | slt |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 09 |   |   |   |   |   |   |   |   | sltU|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Refined Requirements

1. Functional Specification
   - inputs: 2 x 32-bit operands A, B, 4-bit mode
   - outputs: 32-bit result S, 1-bit carry, 1 bit overflow
   - operations: add, addu, sub, subu, and, or, xor, nor, slt, sltU

2. Block Diagram (schematic symbol, Verilog description)

Behavioral Representation: verilog

```verilog
module ALU(A, B, m, S, c, ovf);
input [0:31] A, B;
input [0:3] m;
output [0:31] S;
output c, ovf;
reg [0:31] S;
reg c, ovf;
always @(A, B, m) begin
  case (m)
    0: S = A + B;
    //...
  endcase
endmodule
```

Design Decisions

- Simple bit-slice
  - big combinational problem
  - many little combinational problems
  - partition into 2-step problem
- Bit slice with carry look-ahead
  - ...
**Refined Diagram: bit-slice ALU**

![Refined Diagram: bit-slice ALU](image)

**7-to-2 Combinational Logic**

- start turning the crank . . .

<table>
<thead>
<tr>
<th>Function</th>
<th>Inputs</th>
<th>Outputs</th>
<th>K-Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Seven plus a MUX?**

- Design trick 2: take pieces you know (or can imagine) and try to put them together
- Design trick 3: solve part of the problem and extend

**Additionla operations**

- $A - B = A + (-B) = A + \overline{B} + 1$
  - form two complement by invert and add one

- Set-less-than? – left as an exercise
**Revised Diagram**

- **LSB and MSB need to do a little extra**

![Diagram of ALU with overflow detection logic](image)

**Overflow**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>2’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>-8</td>
<td>1000</td>
</tr>
</tbody>
</table>

- **Examples:**
  - 7 + 3 = 10 but ...
  - -4 - 5 = -9 but ...

**Overflow Detection**

- **Overflow:** the result is too large (or too small) to represent properly
  - Example: -8 ≤ 4-bit binary number ≤ 7
- **When adding operands with different signs, overflow cannot occur!**
- **Overflow occurs when adding:**
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is positive
- **On your own: Prove you can detect overflow by:**
  - Carry into MSB ≠ Carry out of MSB

**Overflow Detection Logic**

- **Carry into MSB ≠ Carry out of MSB**
  - For a N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]
More Revised Diagram

° LSB and MSB need to do a little extra

Critical Path of n-bit Ripple-carry adder is n*CP

Design Trick: Throw hardware at it

But What about Performance?

Carry Look Ahead (Design trick: peek)

Plumbing as Carry Lookahead Analogy
Cascaded Carry Look-ahead (16-bit): Abstraction

Design Trick: Guess (or “Precompute”)

Carry Skip Adder: reduce worst case delay

Just speed up the slowest case for each block
Exercise: optimal design uses variable block sizes
Additional MIPS ALU requirements

- Mult, MultU, Div, DivU (next lecture) => Need 32-bit multiply and divide, signed and unsigned
- Sll, Srl, Sra (next lecture) => Need left shift, right shift, right shift arithmetic by 0 to 31 bits
- Nor (leave as exercise to reader) => logical NOR or use 2 steps: (A OR B) XOR 1111....1111

Elements of the Design Process

- Divide and Conquer (e.g., ALU)
  - Formulate a solution in terms of simpler components.
  - Design each of the components (subproblems)
- Generate and Test (e.g., ALU)
  - Given a collection of building blocks, look for ways of putting them together that meets requirement
- Successive Refinement (e.g., carry lookahead)
  - Solve "most" of the problem (i.e., ignore some constraints or special cases), examine and correct shortcomings.
- Formulate High-Level Alternatives (e.g., carry select)
  - Articulate many strategies to "keep in mind" while pursuing any one approach.
- Work on the Things you Know How to Do
  - The unknown will become “obvious” as you make progress.

Summary of the Design Process

Hierarchical Design to manage complexity

- Top Down vs. Bottom Up vs. Successive Refinement
- Importance of Design Representations:
  - Block Diagrams
  - Decomposition into Bit Slices
  - Truth Tables, K-Maps
  - Circuit Diagrams
  - Other Descriptions: state diagrams, timing diagrams, reg xfer, . . .
- Optimization Criteria:
  - Gate Count
  - Area
  - Logic Levels
  - Fan-in/Fan-out
  - Delay
  - Power
  - Pin Out
  - Cost
  - Design time

Why should you keep a design notebook?

- Keep track of the design decisions and the reasons behind them
  - Otherwise, it will be hard to debug and/or refine the design
  - Write it down so that can remember in long project: 2 weeks ->2 yrs
  - Others can review notebook to see what happened
- Record insights you have on certain aspect of the design as they come up
- Record of the different design & debug experiments
  - Memory can fail when very tired
- Industry practice: learn from others mistakes
Why do we keep it on-line?

- You need to force yourself to take notes
  - Open a window and leave an editor running while you work
    1) Acts as reminder to take notes
    2) Makes it easy to take notes
  - 1) + 2) ⇒ will actually do it
- Take advantage of the window system's "cut and paste" features
- It is much easier to read your typing than your writing
- Also, paper log books have problems
  - Limited capacity ⇒ end up with many books
  - May not have right book with you at time vs. networked screens
  - Can use computer to search files/index files to find what looking for

How should you do it?

- Keep it simple
  - DON'T make it so elaborate that you won't use (fonts, layout, ...)
- Separate the entries by dates
  - type "date" command in another window and cut&paste
- Start day with problems going to work on today
- Record output of simulation into log with cut&paste; add date
  - May help sort out which version of simulation did what
- Record key email with cut&paste
- Record of what works & doesn't helps team decide what went wrong after you left
- Index: write a one-line summary of what you did at end of each day

On-line Notebook Example

- Refer to the handout “Example of On-Line Log Book” on cs 152 handouts page

1st page of On-line notebook (Index + Wed. 9/6/95)

* Index

Wed Sep  6 00:47:28 PDT 1995 - Created the 32-bit comparator component
Thu Sep  7 14:02:21 PDT 1995 - Tested the comparator
Mon Sep 11 12:01:45 PDT 1995 - Investigated bug found by Bart in comp32 and fixed it
*===================================================================

Wed Sep  6 00:47:28 PDT 1995

Goal: Layout the schematic for a 32-bit comparator

I've layed out the schematics and made a symbol for the comparator.
I named it comp32. The files are

~/wv/proj1/sch/comp32.sch
~/wv/proj1/sch/comp32.sym

Wed Sep  6 02:29:22 PDT 1995

- Add 1 line index at front of log file at end of each session: date+summary
- Start with date, time of day + goal
- Make comments during day, summary of work
- End with date, time of day (and add 1 line summary at front of file)
Goal: Test the comparator component

I've written a command file to test comp32. I've placed it in ~/wv/proj1/diagnostics/comp32.cmd.

I ran the command file in viewsim and it looks like the comparator is working fine. I saved the output into a log file called ~/wv/proj1/diagnostics/comp32.log

Notified the rest of the group that the comparator is done.

---

Goal: Investigate bug discovered in comp32 and hopefully fix it

Bart found a bug in my comparator component. He left the following e-mail.

From bart@simpsons.residence Sun Sep 10 01:47:02 1995
Received: by wayne.manor (NX5.67e/NX3.0S) id AA00334; Sun, 10 Sep 95 01:47:01 -0800
Date: Wed, 10 Sep 95 01:47:01 -0800
From: Bart Simpson <bart@simpsons.residence>
To: bruce@wayne.manor, old_man@gokuraku, hojo@sanctuary
Subject: [cs152] bug in comp32

Hey Bruce,
I think there's a bug in your comparator. The comparator seems to think that ffffffff and fffffff7 are equal.
Can you take a look at this?
Bart

---

I verified the bug. Here's a viewsim of the bug as it appeared.
(equal should be 0 instead of 1)

---

SIM>stepsize 10ns
SIM>v a_in A[31:0]
SIM>v b_in B[31:0]
SIM>w a_in b_in equal
SIM>a a_in ffffffff\h
SIM>a b_in fffffff7\h
SIM>sim time = 10.0ns A_IN=FFFFFFFF\H B_IN=FFFFFFF7\H EQUAL=1 Simulation stopped at 10.0ns.
---

Ah. I've discovered the bug. I mislabeled the 4th net in the comp32 schematic.

I corrected the mistake and re-checked all the other labels, just in case.

I re-ran the old diagnostic test file and tested it against the bug Bart found. It seems to be working fine. Hopefully there aren't any more bugs :)
Added benefit: cool post-design statistics

Sample graph from the Alewife project:

- For the Communications and Memory Management Unit (CMMU)
- These statistics came from on-line record of bugs

Lecture Summary

- An Overview of the Design Process
  - Know your metrics of success! (Performance, etc)
  - Design is an iterative process, multiple approaches to get started
  - Do NOT wait until you know everything before you start

- Example: Instruction Set drives the ALU design
  - Divide an Conquer
  - Take pieces you know and put them together
  - Start with a partial solution and extend

- Optimization: Start simple and analyze critical path
  - For adder: the carry is the slowest element
  - Logarithmic trees to flatten linear computation
  - Precompute: Double hardware and postpone slow decision

- On-line Design Notebook
  - Open a window and keep an editor running while you work; cut & paste
  - Refer to the handout as an example
  - Former CS 152 students (and TAs) say they use on-line notebook for programming as well as hardware design; one of most valuable skills