Review: recall General C/L Cell Delay Model

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

- Linear model composes

Review: 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
- Typical for class: 1ns Setup, 0.5ns Hold

Review: 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:
  \[ \text{Clock-to-Q + Longest Path through Combination Logic + Setup} \]
**Review: Tricks to Reduce Cycle Time**

- Reduce the number of gate levels
- Use esoteric/dynamic timing methods
- Pay attention to loading
  - One gate driving many gates is a bad idea
  - Avoid using a small gate to drive a long wire
- Use multiple stages to drive large load

![Diagram showing gate levels]

**Review: Karnaugh Map for easier simplification**

<table>
<thead>
<tr>
<th>S₀</th>
<th>S₁</th>
<th>C</th>
<th>S₀ · S₁'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ S₀' = (S₀ · C) + (S₀ · C') \]

![Karnaugh Map diagram]

**Review: DeMorgan’s theorem: Push Bubbles and Morph**

### 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>

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

### 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>1</td>
</tr>
</tbody>
</table>

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

**Review: Integrated Circuit Costs**

Die cost = Wafer cost

\[ \text{Dies per Wafer} \times \text{Die yield} \]

\[ \text{Dies per wafer} = \pi \cdot \left( \frac{\text{Wafer diam}}{2} \right)^2 - \pi \cdot \text{Wafer diam} - \text{Test dies} = \frac{\text{Wafer Area}}{2 \times \text{Die Area}} \]

\[ \text{Die Yield} = \frac{\text{Wafer yield}}{\left(1 + \frac{\text{Defects per unit area} \times \text{Die Area}}{\alpha} \right)^2} \]

\[ \text{Die Cost is goes roughly with (die area)}^3 \text{ or (die area)}^4 \]

2/10/03 ©UCB Spring 2003 CS152 / Kubiatowicz Lec4.5

2/10/03 ©UCB Spring 2003 CS152 / Kubiatowicz Lec4.6

2/10/03 ©UCB Spring 2003 CS152 / Kubiatowicz Lec4.7

2/10/03 ©UCB Spring 2003 CS152 / Kubiatowicz Lec4.8
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

Design as Search

Problem A
Strategy 1
Strategy 2
SubProb 1
SubProb2
SubProb3
BB1
BB2
BB3
BB\eta

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
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

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>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>op</td>
<td>Rs</td>
<td>Rt</td>
<td>Rd</td>
<td>funct</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>

<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>
</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)
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

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;
      . . .
    end
  endmodule

Administrivia

° Note: First homework quiz on Wednesday during class
  • Any of the material from homework 2 is fair game
    • 15-20 minutes at beginning of class

° Hopefully working on Lab 2!
  • Due Thursday – how’s it going?

° 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?

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**

![Diagram of a 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>M0 M1 M2 A B Cin</td>
<td>S Cout</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 0 0 0 0 0</td>
<td>0 0</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

![Diagram of a Full Adder](image)

**Additional operations**

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

![Diagram of a Full Adder with invert and add](image)
Revised Diagram

- LSB and MSB need to do a little extra

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>
</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

![Diagram]

But What about Performance?

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

![Diagram]

Design Trick: Throw hardware at it

Carry Look Ahead (Design trick: peek)

- A B C-out
  - 0 0 0 “kill”
  - 0 1 C-in “propagate”
  - 1 0 C-in “propagate”
  - 1 1 1 “generate”

C0 = Cin

C1 = G0 + C0 • P0

C2 = G1 + G0 • P1 + C0 • P0 • P1

C3 = G2 + G1 • P2 + G0 • P1 • P2 + C0 • P0 • P1 • P2

C4 = ...

Plumbing as Carry Lookahead Analogy

![Diagram]
Cascaded Carry Look-ahead (16-bit): Abstraction

\[ C1 = G0 + C0 \times P0 \]
\[ C2 = G1 + G0 \times P1 + C0 \times P0 \times P1 \]
\[ C3 = G2 + G1 \times P2 + G0 \times P1 + P2 + C0 \times P0 \times P1 \times P2 \]
\[ C4 = \ldots \]

Design Trick: Guess (or “Precompute”)

\[ CP(2n) = 2 \times CP(n) \]
\[ CP(2n) = CP(n) + CP(mux) \]

2nd level Carry, Propagate as Plumbing

\[ P0 \]
\[ G0 \]

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
- [Package Count]
- Area
- Logic Levels
- Fan-in/Fan-out
- Delay
- Power
- Cost
- Design time
- mux design meets at TT
- [Package Count]

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 home 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
Status: R

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

------------------
SIM>stepsize 10nsSIM>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
SIM>b b_in fffffff7
SIM>simtime = 10.0ns  A_IN=FFFFFFFF  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 :)
Add benefit: cool post-design statistics

Sample graph from the Alewife project:

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

Lecture Summary

- An Overview of the Design Process
  - 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 and 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