

CS152 – Computer Architecture and Engineering

Lecture 5 – Performance and Design Process

2003-09-09

Dave Patterson
(www.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs152/



Review

- **Critical Path** is longest among N parallel paths
- **Setup Time** and **Hold Time** determine how long Input must be stable before and after trigger clock edge
- **Clock skew** is difference between clock edge in different parts of hardware; it affects clock cycle time and can cause hold time, setup time violations
- **FSM** specify control symbolically
 - Moore machine easiest to understand, debug
 - “One hot” reduces decoding for faster FSM
- Die size affects both dies/wafer and yield



Outline this week

- **Performance Review**
 - Latency v. Throughput, CPI, Benchmarks
- **Philosophy of Design**
 - As decomposition (“divide and conquer”)
 - As composition
 - As refinement
- MIPS ALU as example design (if time)
- Online Notebook (next lecture)
 - Capturing design and implementation process, decisions so that can understand evolution of design, fix bugs



Two Notions of “Performance”

Plane	DC to Paris	Top Speed	Passengers	Throughput (pmph)
Boeing 747	6.5 hours	610 mph	470	286,700
BAD/Sud Concorde	3 hours	1350 mph	132	178,200

- **Which has higher performance?**
- Time to deliver 1 passenger?
- Time to deliver 400 passengers?
- In a computer, time for 1 job called **Response Time** or **Execution Time**
- In a computer, jobs per day called **Throughput** or **Bandwidth**



Definitions

- Performance is in units of things per sec
 - bigger is better
- If we are primarily concerned with response time
 - $\text{performance}(x) = \frac{1}{\text{execution_time}(x)}$

“X is n times faster than Y” means

$$n = \frac{\text{Performance}(X)}{\text{Performance}(Y)}$$



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 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 (1/2)

- **CPU execution time for program**
= Clock Cycles for a program
x Clock Cycle Time
- or
= $\frac{\text{Clock Cycles for a program}}{\text{Clock Rate}}$



Measuring Time using Clock Cycles (2/2)

- One way to define clock cycles:
Clock Cycles for program
= **Instructions for a program**
(called "**Instruction Count**")
x **Average Clock cycles Per Instruction**
(abbreviated "**CPI**")
- CPI one way to compare two machines with **same** instruction set, since Instruction Count would be the same



Performance Calculation (1/2)

- CPU execution time for program
= **Clock Cycles for program**
x Clock Cycle Time
- Substituting for clock cycles:
CPU execution time for program
= (**Instruction Count** x **CPI**)
x Clock Cycle Time
= **Instruction Count** x **CPI** x **Clock Cycle Time**



Performance Calculation (2/2)

$$\begin{aligned} \text{CPU time} &= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}} \\ \text{CPU time} &= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}} \\ \text{CPU time} &= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}} \\ \text{CPU time} &= \frac{\text{Seconds}}{\text{Program}} \end{aligned}$$

- **Product of all 3 terms: if missing a term, can't predict time, the real measure of performance**



Administrivia

- HW #1 Due Wed 9/10 **by 5 PM**
 - 3 homework boxes (1 / section) in 283 Soda
- Lab #2 done in pairs since 15 FPGA boards, 33 PCs. Due Monday 9/15
- Form 4 or 5 person teams by Friday 9/12
 - Who have full teams? Needs teammates?
- Office hours in Lab
 - Mon 5 – 6:30 Jack, Tue 3:30-5 Kurt, Wed 3 – 4:30 John
- Dave's office hours Tue 3:30 – 5



Computers in the Real World



◦ **Problem:** IB 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)

http://www.berkeley.edu/news/media/releases/2003/07/28_redwood.shtml



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"



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)



Calculating CPI Another Way

- First calculate CPI for each individual instruction (add, sub, and, etc.)
- Next calculate frequency of each individual instruction
- Finally multiply these two for each instruction and add them up to get final CPI



Example

Op	Freq _i	CPI _i	Prod	(% Time)
ALU	50%	1	.5	(33%)
Load	20%	2	.4	(27%)
Store	10%	2	.2	(13%)
Branch	20%	2	.4	(27%)

Instruction Mix $\frac{1.5}{1.5}$ (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 its hard to do
 - Don't have access to machine to "**benchmark**" before purchase
 - Don't know workload in future



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



Example Standardized Workload Benchmarks

- **Workstations: Standard Performance Evaluation Corporation (SPEC)**
 - SPEC95: 8 integer (gcc, compress, li, ijpeg, perl, ...) & 10 floating-point (FP) programs (hydro2d, mgrid, applu, turbo3d, ...)
 - SPEC2000: 11 integer (gcc, bzip2, ...) , 18 FP (mgrid, swim, ma3d, ...)
 - www.spec.org
 - Separate average for integer and FP
 - Benchmarks distributed in source code
 - Company representatives select workload
 - Compiler, machine designers target benchmarks, so try to change every 3 years



CS 152 L05 Performance and Design (19)

Patterson Fall 2003 © UCB

Performance Evaluation

- Good products created when have:
 - Good benchmarks
 - Good ways to summarize performance
- Given sales is a function of performance relative to competition, should invest in improving product as reported by performance summary?
- If benchmarks/summary inadequate, then choose between improving product for real programs vs. improving product to get more sales; Sales almost always wins!



CS 152 L05 Performance and Design (20)

Patterson Fall 2003 © UCB

Amdahl's Law

Speedup due to enhancement E:

$$\text{Speedup (E)} = \frac{\text{ExTime w/o E}}{\text{ExTime w/ E}} = \frac{\text{Performance w/ E}}{\text{Performance w/o E}}$$

- 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:

$$\text{Speedup}_{\text{maximum}} = \frac{1}{1 - \text{Fraction}_{\text{affected}}}$$



CS 152 L05 Performance and Design (21)

Patterson Fall 2003 © UCB

Things to Remember

- Latency v. Throughput
- Performance doesn't depend on any single factor: need to know Instruction Count, Clocks Per Instruction and Clock Rate to get valid estimations
- 2 Definitions of times:
 - User Time: time user needs to wait for program to execute (multitasking affects)
 - CPU Time: time spent executing a single program: (no multitasking)
- Amdahl's Law: law of diminishing returns



CS 152 L05 Performance and Design (22)

Patterson Fall 2003 © UCB

Peer Instruction: find the best mismatch!

Performance metric:	Designer choice:
I. Instruction Count	A. Benchmark
II. CPI	B. Compiler
III. Clock Rate	C. HW technology

Match the metric with designer choice least likely to affect it

Doesn't affect?	I.	II.	III.
1.	A	B	C
2.	A	C	B
3.	B	A	C
4.	B	C	A
5.	C	A	B
6.	C	B	A



CS 152 L05 Performance and Design (23)

Patterson Fall 2003 © UCB

Peer Instruction: Amdahl's Law

- Suppose your benchmarks spend 80% of their time on floating point multiply, and your boss tells you the benchmarks must run 5 times faster than it does now. How much faster must you make the Floating Point multiplier?
 1. 4X faster
 2. 5X faster
 3. 8X faster
 4. 10X faster
 5. You get another job, because it can't be done



CS 152 L05 Performance and Design (24)

Patterson Fall 2003 © UCB

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, . . .



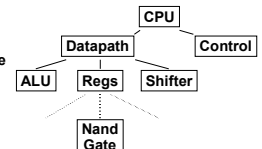
Design Process (cont.)

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

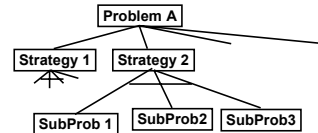
Final Internal Specification

Physical Implementation

refinement
increasing level of detail



Design as Search



BB1 BB2 BB3 BBn

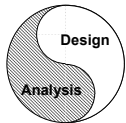
Design involves educated guesses and verification

- 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

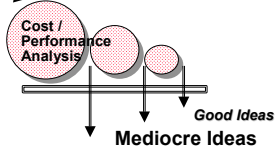


Measurement and Evaluation



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

Creativity



Bad Ideas



Problem: Design a "fast" ALU for the MIPS ISA

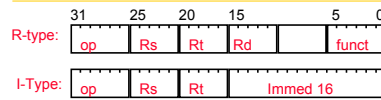
- Requirements?
- Must support the MIPS ISA Arithmetic / Logic operations
- Tradeoffs of cost and speed based on frequency of occurrence, hardware budget



MIPS ALU requirements

- Add, AddU, Sub, SubU, Addl, AddIU
 - => 2's complement adder/sub with overflow detection
- And, Or, Andl, Orl, 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 P&H book chapter 4 supports these ops

MIPS arithmetic instruction format



Type	op	funct	Type	op	funct	Type	op	funct
ADDI	10	xx	ADD	00	40		00	50
ADDIU	11	xx	ADDU	00	41		00	51
SLTI	12	xx	SUB	00	42		00	52
SLTIU	13	xx	SUBU	00	43		00	53
ANDI	14	xx	AND	00	44			
ORI	15	xx	OR	00	45			
XORI	16	xx	XOR	00	46			
LUI	17	xx	NOR	00	47			

- Signed arithmetic generate overflow, no carry

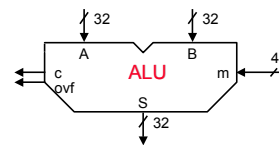
Design Trick: divide & conquer

- 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
12	slt
13	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

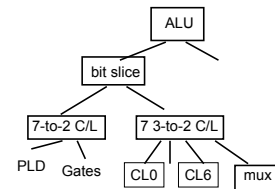
```

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;

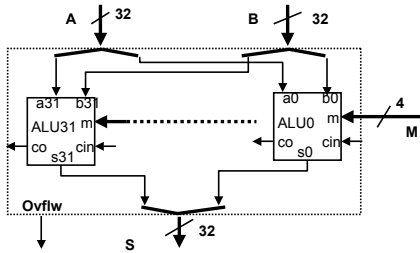
always @(A, B, m) begin
    case (m)
        0: S = A + B;
        . . .
    end
end
endmodule
    
```

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



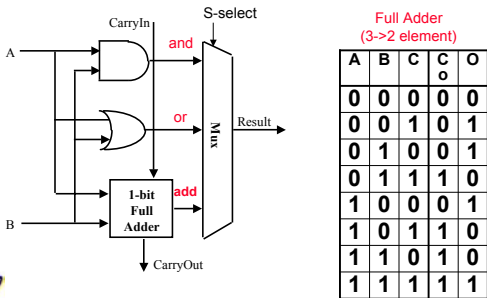
7-to-2 Combinational Logic

start turning the crank . . .

Function	Inputs						Outputs		K-Map	
	M0	M1	M2	M3	A	B	Cin	S		Coout
0	add	0	0	0	0	0	0	0	0	

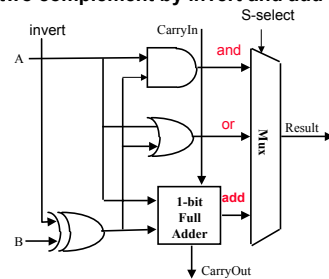
Seven plus a MUX ?

- 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



Additional operations

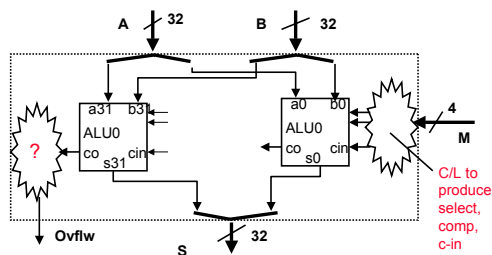
- $A - B = A + (-B) = A + \bar{B} + 1$
- form two complement by invert and add one



Set-less-than? – left as an exercise

Revised Diagram

LSB and MSB need to do a little extra

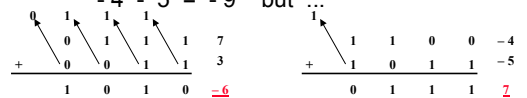


Overflow

Decimal	Binary	Decimal	2's Complement
0	0000	0	0000
1	0001	-1	1111
2	0010	-2	1110
3	0011	-3	1101
4	0100	-4	1100
5	0101	-5	1011
6	0110	-6	1010
7	0111	-7	1001
		-8	1000

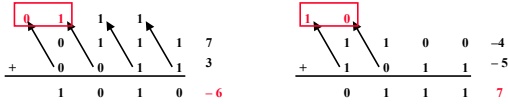
Examples: $7 + 3 = 10$ but ...

$-4 - 5 = -9$ but ...



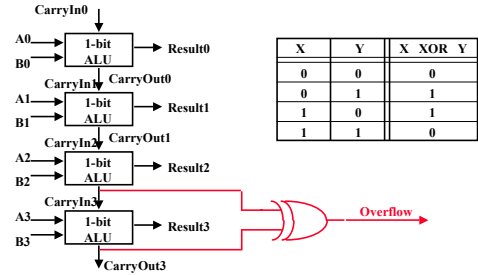
Overflow Detection

- ° **Overflow:** the result is too large (or too small) to represent properly
 - Example: $-8 \leq 4\text{-bit binary number} \leq 7$
- ° 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 \neq Carry out of MSB



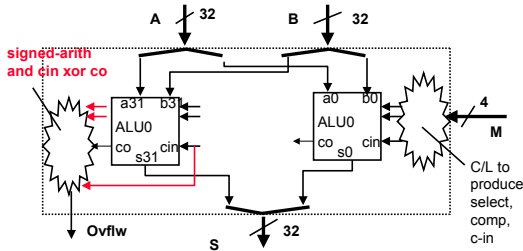
Overflow Detection Logic

- ° Carry into MSB \neq Carry out of MSB
 - For a N -bit ALU: $\text{Overflow} = \text{CarryIn}[N - 1] \text{ XOR } \text{CarryOut}[N - 1]$



More Revised Diagram

- ° LSB and MSB need to do a little extra

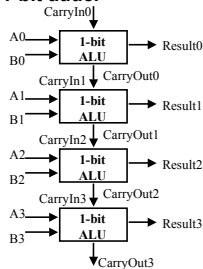


Peer Instruction: Which is good design advice?

1. Wait until you know everything before you start ("Be prepared")
2. The best design is a one-pass, top down process ("Plan Ahead")
3. Start simple, measure, then optimize ("Less is more")
4. Don't be biased by the components you already know ("Start with a clean slate")

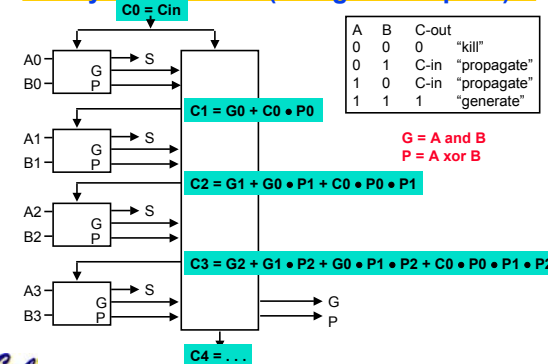
But What about Performance?

- ° Critical Path of n -bit Ripple-carry adder is $n \cdot \text{CP of 1-bit adder}$

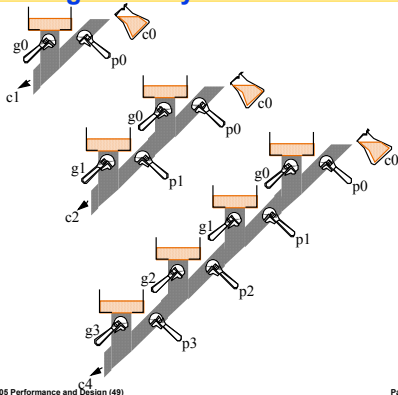


Design Trick:
Throw hardware at it

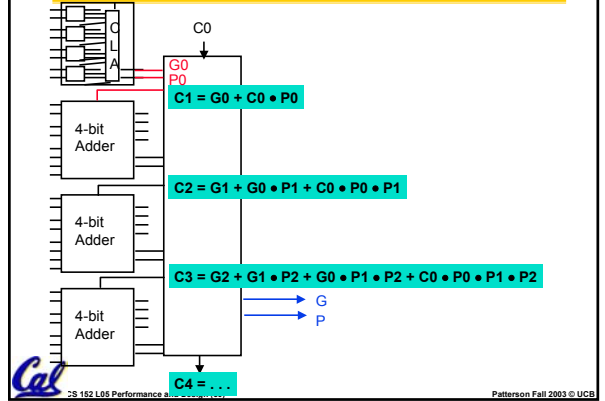
Carry Look Ahead (Design trick: peek)



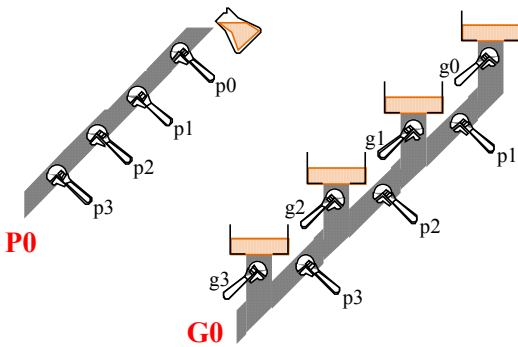
Plumbing as Carry Lookahead Analogy



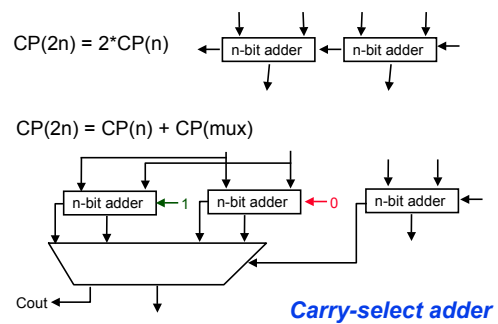
Cascaded Carry Look-ahead (16-bit): Abstraction



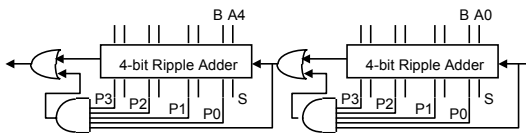
2nd level Carry, Propagate as Plumbing



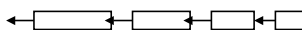
Design Trick: Guess (or "Precompute")



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 (earlier lecture) => Need 32-bit multiply and divide, signed and unsigned
- Sll, Srl, Sra => 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.

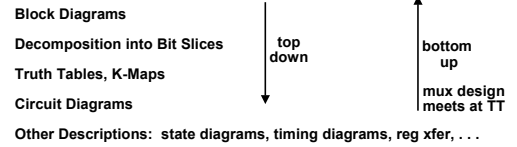


Summary of the Design Process

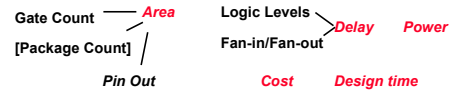
Hierarchical Design to manage complexity

Top Down vs. Bottom Up vs. Successive Refinement

Importance of Design Representations:



Optimization Criteria:



Peer Instruction: Match for Design Principle?

- | | |
|---|--|
| I. Composition | A. Design 1-bit ALU slice before 32-bit ALU |
| II. Divide and Conquer | B. Replace ripple carry with carry lookahead |
| III. Start simple, then optimize critical paths | C. Use Mux to join AND, OR gates with Adder |

Best match	I.	II.	III.
1.	A	B	C
2.	A	C	B
3.	B	A	C
4.	B	C	A
5.	C	A	B
6.	C	B	A



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:

~cs152/handouts/online_notebook_example.html



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 laid 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)



2nd page of On-line notebook (Thursday 9/7/95)

```
+ =====
Thu Sep  7 14:02:21 PDT 1995

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.

Thu Sep  7 16:15:32 PDT 1995
- =====
```



3rd page of On-line notebook (Monday 9/11/95)

```
+ =====
Mon Sep 11 12:01:45 PDT 1995

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 (NK5.67e/NK3.0S)
       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 ffffffff7 are equal.

Can you take a look at this?
Bart
-----
```



4th page of On-line notebook (9/11/95 contd)

```
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 ffffffff7\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:)
```



5th page of On-line notebook (9/11/95 contd)

```
On second inspection of the whole layout, I think I can
remove one level of gates in the design and make it go faster.
But who cares! the comparator is not in the critical path
right now, the delay through the ALU is dominating the critical
path, so unless the ALU gets a lot faster, we can live with
a less than optimal comparator.

I e-mailed the group that the bug has been fixed
```

```
Mon Sep 11 14:03:41 PDT 1995
- =====
```

- Perhaps later critical path changes;
- What was idea to make comparator faster?
- Check on-line notebook!

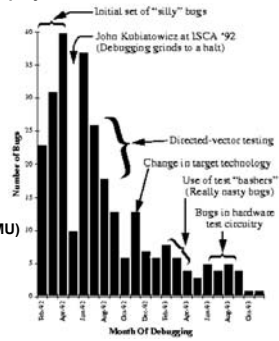


Added benefit: cool post-design statistics

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**
 - 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