Building a Better Beta

Quiz reviews: Tue & Wed (7:30, 6-120)  Quiz 4 on Friday.
Plan of Attack

1. Load /lab5 and /lab2 modules, adjust cycle time (slide 7)
2. Improve FA module (slide 16), adjust cycle time
3. Use only two main memory ports (slide 3)
4. Use only seven 32-bitmuxes for SHIFT (slide 4)
5. Don’t use RA2SEL from CTL! (slide 12), adjust cycle time
6. Reorganize instruction fetch (slide 14), adjust cycle time
7. Upgrade to carry-select adder (slide 17), adjust cycle time
8. Deal with heavily-loaded signals (slide 13), adjust cycle time
Smaller: Sharing Memory Ports

Goal: use a single read/write port for LD/LDR and ST

This is the component you are looking for

\[ \begin{align*}
\text{LD/LDR} & : \quad \text{id}[31:0] \\
\text{ST} & : \quad \text{id}[31:0]
\end{align*} \]

\[ \begin{align*}
\text{LD/LDR} & : \quad \text{id}[31:0] \\
\text{ST} & : \quad \text{id}[31:0]
\end{align*} \]

When \( e = 0 \):

\[ \begin{cases} 
0 \\
1 \\
\text{No connection}
\end{cases} \]
Smaller: Use One Shift Mux-Tree

Observation: Right shift looks like a “mirrored” left shift

Idea: To shift right: flip A bits, shift left, flip output

Original 2-shifter design: $5 + 5 + 1 = 11$ 32-bit muxes
Improved “flipper” design: $1 + 5 + 1 = 7$ 32-bit muxes
Smaller: Incrementer

Original FA equations from Lab 1, Problem 8:

\[ S = (A \oplus B) \oplus C_{IN} \]
\[ C_{OUT} = A \cdot B + A \cdot C_{IN} + B \cdot C_{IN} \]

In an incrementer, the B input is 0 and we set \( C_{IN} = 1 \) for the low-order bit in order to add 1 to the A value:

\[ S = A \oplus C_{IN} \]
\[ C_{OUT} = A \cdot C_{IN} \]

Note that PC_INC[1:0] are always 0 and PC_INC[31] isn’t needed since we use PC[31] instead. So you really just need to have incrementing logic for PC[30:2].

Same idea applies to PC_OFFSET[30:2].
Smaller: General Hints

• Don’t use logic where clever wiring will do the job!
  – Multiplying by 4 can be done by shifting signals two places to the left

• Only build the logic you need
  – If you need an adder (eg, for PC_INC or PC_OFFSET) build a copy of your ripple-carry adder from ARITH. Do not instantiate ARITH, or even worse, a whole ALU!
  – Low-order 2 bits of PC, PC_INC, PC_OFFSET are always 0, so leave them out of muxes, adders, regs, etc.
  – Bit [31] from PC+4 logic isn’t used (since we use PC[31] instead). So PC+4 logic only needs to produce PC_INC[30:2]. Ditto for PC_OFFSET.
Faster: Adjusting the Clock Cycle Time

1. Run simulation using pre-supplied 100ns clock
2. Click “Stats” button in waveform window
3. How much faster can $t_{CLK}$ be? $100 - 86 = \sim 14$ns
4. Go to Test tab for /project/test, adjust .cycle line
   .cycle CLK=1 tran 1n assert inputs tran 6n CLK=0 tran 7n
Faster: What Determines $t_{CLK}$?

CLK↑

New PC

PC+4

Fetch Inst.

+OFFSET

RA2SEL mux

Control Logic

Read Regs

ASEL mux

BSEL mux

ALU

=0?

WDSEL mux

Fetch data

PCSEL mux

PC setup

RF setup

Mem setup

CLK↑
The timing analyzer computes $\Sigma t_{PD}$ along every register-to-register path. It then reports the 10 longest paths. Not every reported path is possible since the analyzer doesn’t know that certain combinations of control signals aren’t used (e.g., ASEL=1 and ALUFN=SHR).
Timing Analyzer Output

+ 0.000ns = 0.000ns clk↑
+ 0.255ns = 0.255ns ia[11] [beta.pc_1.dreg_1[11] dreg]
+ 4.005ns = 4.260ns id[31] [main memory]
+ 2.025ns = 6.284ns beta.ra2sel [beta.ctl_1.ctlrom memory]
+ ...  \[ \sum t_{PD} \]  \[ \Sigma t_{PD} \]  output sig. [Component name & type]

Example: the last line in the trace tells us

- the signal id[31] is an input to the ctlrom
- the signal beta.ra2sel is an output of the ctlrom
- the t_{PD} of ctlrom is 2.025ns
  -- this includes delay to due capacitance on output
- the cumulative t_{PD} along the path is 6.284.ns
Timing Analyzer Diagnosis

ROM access before regfile access

Heavily-loaded signal

Slow ripple-carry adder
Faster: RA2SEL

RA2SEL must be 1 for ST.

<table>
<thead>
<tr>
<th>2:0</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>5:3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RA2SEL must be 0 for these opcodes.

Using the opcode bits, what’s the simplest equation for RA2SEL that produces the desired values?
Faster: Heavily-loaded Signals

Signal connects to many inputs ⇒ lots of capacitance to charge/discharge ⇒ long $t_{PD}$ for component that drives signal

Not a good approach (still many connections to drive!):

Now the top signal is fast

For small $t_{CLK}$ need this signal to be fast!
Faster: Reorganize instruction fetch

Idea: overlap fetching inst with fetching data!
Faster: Reorganize Instruction Fetch

What we used originally when fetching instructions.
Faster: Shorter $C_{IN}$-to-$C_{OUT}$ $t_{PD}$

Original FA equations from Lab 1, Problem 8:

$$S = (A \oplus B) \oplus C_{IN}$$
$$C_{OUT} = A \cdot B + A \cdot C_{IN} + B \cdot C_{IN}$$

Improved FA equations from L08, Slide 12:

$$P = A \oplus B$$
$$S = P \oplus C_{IN}$$
$$C_{OUT} = A \cdot B + P \cdot C_{IN}$$

Remember to use inverting logic here, i.e., NAND/NAND instead of AND/OR

<table>
<thead>
<tr>
<th>Function</th>
<th>$t_{PD}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND2</td>
<td>120ps</td>
</tr>
<tr>
<td>OR3</td>
<td>210ps</td>
</tr>
<tr>
<td>NAND2</td>
<td>30ps</td>
</tr>
</tbody>
</table>
Faster: Carry-select Adder

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

Design goal: have these two sets of signals arrive simultaneously at each carry-select mux.
Faster: Fast/Small CMP Module

Comparison Equation for LSB CFN[1:0]

\[
\begin{align*}
A = B & \quad \text{LSB} = Z & 01 \\
A < B & \quad \text{LSB} = N \oplus V & 10 \\
A \leq B & \quad \text{LSB} = Z + (N \oplus V) & 11
\end{align*}
\]

\[
\text{LSB} = (Z \cdot \text{CFN}[0]) + ((N \oplus V) \cdot \text{CFN}[1])
\]

Remember to implement AND-OR as NAND/NAND!
Faster: Shorten Critical Path for LD

Direct access to ARITH output

Move ASEL mux here

Fast path for MRD

MUX2: $t_{PD} = 120\text{ps}$, size = $27\mu^2$

MUX4: $t_{PD} = 190\text{ps}$, size = $66\mu^2$
Faster: 2-stage Pipeline (step 1)

The annul_if signal should be 1 when reset=1 or when pcsel[2:0]!=0, which happens when we're changing the PC from sequential execution. Get this working first -- it won't be faster, but it sets the stage for a speed improvement in the next step.
Faster: 2-stage Pipeline (step 2)

Plan: LD/LDR/ST take two cycles in EXE stage. 1\textsuperscript{st} to compute address, 2\textsuperscript{nd} for memory access
1. Convert PC\textsuperscript{IF}, PC\textsuperscript{EXE}, IR\textsuperscript{exe} to load-enabled registers controlled by NOT\_STALL
2. STALL=1 if first cycle of LD/LDR/ST in EXE stage

During ST: Only 1 during 2\textsuperscript{nd} cycle