More Digital Circuits
Signals and Waveforms: Showing Time & Grouping
Signals and Waveforms: Circuit Delay

\[ A = [a_3, a_2, a_1, a_0] \]
\[ B = [b_3, b_2, b_1, b_0] \]

\[ A \xrightarrow{4} C \]

\[ A \xrightarrow{4} \equiv \]

\[ a_0 \rightarrow \]
\[ a_1 \rightarrow \]
\[ a_2 \rightarrow \]
\[ a_3 \rightarrow \]

\[ A \]
2 3 4 5

\[ B \]
3 10 0 1

\[ C \]
5 13 4 6

adder propagation delay
Sample Debugging Waveform
Type of Circuits

• *Synchronous Digital Systems* consist of two basic types of circuits:
  
  • Combinational Logic (CL) circuits
    – Output is a function of the inputs only, not the history of its execution
    – E.g., circuits to add A, B (ALUs)
  
  • Sequential Logic (SL)
    • Circuits that “remember” or store information
    • aka “State Elements”
    • E.g., memories and registers (Registers)
Uses for State Elements

• Place to store values for later re-use:
  • Register files (like x1-x31 in RISC-V)
  • Memory (caches and main memory)

• Help control flow of information between combinational logic blocks

• State elements hold up the movement of information at input to combinational logic blocks to allow for orderly passage
Accumulator Example

Why do we need to control the flow of information?

Want: \( S = 0; \)

\[
\text{for } (i=0; i<n; i++) \\
S = S + X_i
\]

Assume:

- Each \( X \) value is applied in succession, one per cycle
- After \( n \) cycles the sum is present on \( S \)
First Try: Does this work?

No!
Reason #1: How to control the next iteration of the ‘for’ loop?
Reason #2: How do we say: ‘S=0’?
Register Internals

- n instances of a “Flip-Flop”
- Flip-flop name because the output flips and flops between 0 and 1
- D is “data input”, Q is “data output”
- Also called “D-type Flip-Flop”
Flip-Flop Operation

- Edge-triggered d-type flip-flop
- This one is “positive edge-triggered”

- “On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored.”

- Example waveforms:
Flip-Flop Timing

- Edge-triggered d-type flip-flop
  - This one is “positive edge-triggered”
  
  “On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored.”

- Example waveforms (more detail):

![Waveform Diagram]
Camera Analogy Timing Terms

• Want to take a portrait – timing right before and after taking picture

• *Set up time* – don’t move since about to take picture (open camera shutter)

• *Hold time* – need to hold still after shutter opens until camera shutter closes

• *Time click to data* – time from open shutter until can see image on output (viewscreen)
Hardware Timing Terms

• **Setup Time:** when the input must be stable *before* the edge of the CLK

• **Hold Time:** when the input must be stable *after* the edge of the CLK

• **“CLK-to-Q” Delay:** how long it takes the output to change, measured from the edge of the CLK
So How To Build A Flip Flop?
Two "Latches". An example...

- When clk is high...
  - D -> Q
- When clk is low...
  - Q stays with whatever it was
- Chain 2 latches together to create a flip-flop
- Setup time:
  - Need to propagate D to Q on the first latch
- Hold time:
  - Need to make sure the first latch doesn't change before the clock fully switches
- Clk->Q time:
  - Time needed to go through the second latch
Accumulator Timing 1/2

- Reset input to register is used to force it to all zeros (takes priority over D input).
- $S_{i-1}$ holds the result of the $i^{th}$-1 iteration.
- Analyze circuit timing starting at the output of the register.
• reset signal shown.
• Also, in practice X might not arrive to the adder at the same time as $S_{i-1}$
• $S_i$ temporarily is wrong, but register always captures correct value.
• In good circuits, instability never happens around rising edge of clk.
Model for Synchronous Systems

- Collection of Combinational Logic blocks separated by registers
- Feedback is optional
- Clock signal(s) connects only to clock input of registers
- Clock (CLK): steady square wave that synchronizes the system
- Register: several bits of state that samples on rising edge of CLK (positive edge-triggered) or falling edge (negative edge-triggered)
Maximum Clock Frequency

• What is the maximum frequency of this circuit?

Hint:
Frequency = 1/Period

Period = Max Delay = CLK-to-Q Delay + CL Delay + Setup Time
Critical Paths

Timing...

Note: delay of 1 clock cycle from input to output. Clock period limited by propagation delay of adder/shifter.
Pipelining to improve performance

Timing…

• Insertion of register allows higher clock frequency
• More outputs per second (higher bandwidth)
• But each individual result takes longer (greater latency)
Recap of Timing Terms

- **Clock (CLK)** - steady square wave that synchronizes system
- **Setup Time** - when the input must be stable before the rising edge of the CLK
- **Hold Time** - when the input must be stable after the rising edge of the CLK
- **“CLK-to-Q” Delay** - how long it takes the output to change, measured from the rising edge of the CLK

- **Flip-flop** - one bit of state that samples every rising edge of the CLK (positive edge-triggered)
- **Register** - several bits of state that samples on rising edge of CLK or on LOAD (positive edge-triggered)
Administrivia

- Project PAARRTTTTAAYYYY!!!
  - Wednesday, 2nd floor labs, 7-9pm
  - Project 2 due 3/1

- Exam grades will be released tonight with solutions
  - Regrades must be submitted by 23:59 on Sunday

- Project 3 released 3/2...
What is maximum clock frequency? (assume all unconnected inputs come from some register)

- A: 5 GHz
- B: 200 MHz
- C: 500 MHz
- D: 1/7 GHz
- E: 1/6 GHz

Clock->Q  1ns
Setup 1ns
Hold 1ns
AND delay 1ns
Problems With Clocking...

- The clock period **must be** longer than the critical path
  - Otherwise, you will get the wrong answers
  - But it can be even longer than that

- **Critical path:**
  - clk->q time
    - Necessary to get the output of the registers
  - **worst case** combinational logic delay
  - **Setup time** for the next register
  - Must meet all of these to be correct
Hold-Time Violations...

- An alternate problem can occur...
  - Clk->Q + **best case** combinational delay < Hold time...

- What happens?
  - Clk->Q + data propagates...
  - And now you don't hold the input to the flip flop long enough

- Solution:
  - **Add** delay on the best-case path (e.g. two inverters)
Finite State Machines (FSM) Intro

• A convenient way to conceptualize computation over time

• We start at a state and given an input, we follow some edge to another (or the same) state

• The function can be represented with a “state transition diagram”.

• With combinational logic and registers, any FSM can be implemented in hardware.
FSM Example: 3 ones…

Draw the FSM:

Assume state transitions are controlled by the clock:
On each clock cycle the machine checks the inputs and moves to a new state and produces a new output.
Hardware Implementation of FSM

...therefore a register is needed to hold the representation of which state the machine is in. Use a unique bit pattern for each state.

Combinational logic circuit is used to implement a function that maps from present state and input to next state and output.
Specify CL using a truth table

Truth table...

<table>
<thead>
<tr>
<th>PS</th>
<th>Input</th>
<th>NS</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>1</td>
</tr>
</tbody>
</table>
Building Standard Functional Units

- Data multiplexers
- Arithmetic and Logic Unit
- Adder/Subtractor
Data Multiplexer ("Mux") (here 2-to-1, n-bit-wide)
N instances of 1-bit-wide mux

How many rows in TT?

\[ c = \overline{s}ab + \overline{s}ab + s\overline{a}b + sab \]
\[ = \overline{s}(a\overline{b} + ab) + s(\overline{a}b + ab) \]
\[ = \overline{s}(a(\overline{b} + b)) + s((\overline{a} + a)b) \]
\[ = \overline{s}(a(1) + s((1)b) \]
\[ = \overline{s}a + sb \]
How do we build a 1-bit-wide mux?

\[ \overline{s}a + sb \]
4-to-1 multiplexer?

How many rows in TT?

\[ e = \overline{s_1 s_0} a + \overline{s_1 s_0} b + s_1 \overline{s_0} c + s_1 s_0 d \]
Another way to build 4-1 mux?

Ans: Hierarchically!

Hint: NCAA tourney!
Arithmetic and Logic Unit

• Most processors contain a special logic block called the “Arithmetic and Logic Unit” (ALU)
• We’ll show you an easy one that does ADD, SUB, bitwise AND, bitwise OR

when S=00, R=A+B
when S=01, R=A-B
when S=10, R=A AND B
when S=11, R=A OR B
Our simple ALU
How to design Adder/Subtractor?

• Truth-table, then determine canonical form, then minimize and implement as we’ve seen before

• Look at breaking the problem down into smaller pieces that we can cascade or hierarchically layer
Adder/Subtractor – One-bit adder LSB...

\[
\begin{array}{cccc|c|cc}
\quad & a_3 & a_2 & a_1 & a_0 \\
+ & b_3 & b_2 & b_1 & b_0 \\
\hline
s_3 & s_2 & s_1 & s_0 & \\
\end{array}
\]

<table>
<thead>
<tr>
<th>a_0</th>
<th>b_0</th>
<th>s_0</th>
<th>c_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
s_0 = \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad c_1 =
\]
Adder/Subtractor – One-bit adder (1/2)…

\[
\begin{array}{c c c | c c}
\text{a}_i & \text{b}_i & \text{c}_i & \text{s}_i & \text{c}_{i+1} \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{align*}
s_i & = \\
c_{i+1} & =
\end{align*}
\]
Adder/Subtractor – One-bit adder (2/2)

\[ s_i = \text{XOR}(a_i, b_i, c_i) \]
\[ c_{i+1} = \text{MAJ}(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i \]
N 1-bit adders $\Rightarrow$ 1 N-bit adder

What about overflow?

Overflow = $c_n$?
Extremely Clever Adder/Subtractor: "Invert and add one"

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>XOR(x,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</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>

\[ \text{XOR serves as conditional inverter!} \]
iClicker Question

Convert the truth table to a boolean expression (no need to simplify):

A: \( F = xy + x(\neg y) \)

B: \( F = xy + (\neg x)y + (\neg x)(\neg y) \)

C: \( F = (\neg x)y + x(\neg y) \)

D: \( F = xy + (\neg x)y \)

E: \( F = (x+y)(\neg x+\neg y) \)
In Conclusion

- Finite State Machines have clocked state elements plus combinational logic to describe transition between states
- Clocks synchronize D-FF change (Setup and Hold times important!)
- Standard combinational functional unit blocks built hierarchically from subcomponents
Finally: Nick's Thoughts on Project 2

• There is a **lot of room** for optimization that we **do not do**...
  • Preamble/postamble saves a ton of registers that may not be used
  • Intermediate values, local variables and arguments are just all passed on the stack
    • Arguments and local variables addressed by the frame pointer, intermediates off the stack pointer
• Doing this **right and fast** is hard, annoying, and tedious
  • So we don't have you do it: We want just correctness, not performance...
  • So instead we are actually compiling how a basic CISC compiler does it: A "stack machine"
• But what would it take to get performance?
  • A preview of CS164
The Problem: 

**Register Allocation**

- **One Big RISC idea:**
  - Let us make the compiler writer's job much more annoying in return for simpler hardware with better performance

- **Every value in the function has a "Lifespan"**
  - From when it is first needed (and initialized) to when it is no longer needed
  - If the lifespan doesn't cross function calls:
    - Can use temporary registers
  - If the lifespan does cross function calls:
    - Must either use saved registers or allocate the data on the stack
The Register Allocator

• For each thing that needs to be allocated...
  • Assign each variable to a register or stack space

• Requirement:
  • No elements can share the same register if their lifetimes overlap

• Optimization:
  • Minimize the number of stack entries and saved registers needed to map all registers

• Of course this is NP-complete
  • No known polynomial time solution
  • But plenty of "good enuf" heuristics

• After allocating registers, can then create the preamble/postamble