CS 61C:
Great Ideas in Computer Architecture
*Finite State Machines*

Instructors:
Krste Asanovic & Vladimir Stojanovic

http://inst.eecs.berkeley.edu/~cs61c/sp15
Levels of Representation/Interpretation

High Level Language Program (e.g., C) → Compiler → Assembly Language Program (e.g., MIPS) → Assembler → Machine Language Program (MIPS) → Machine Interpretation

**Compiler**

- temp = v[k];
- v[k] = v[k+1];
- v[k+1] = temp;

**Assembler**

- lw $t0, 0($2)
- lw $t1, 4($2)
- sw $t1, 0($2)
- sw $t0, 4($2)

**Machine Language Program (MIPS)**

- Anything can be represented as a *number*, i.e., data or instructions

- 0000 1001 1100 0110 1010 1111 0101 1000
- 1010 1111 0101 1000 0000 1001 1100 0110
- 1100 0110 1010 1111 0101 1000 0000 1001
- 0101 1000 0000 1001 1100 0110 1010 1111

**Machine Interpretation**

- Hardware Architecture Description (e.g., block diagrams)

**Architecture Implementation**

- Logic Circuit Description (Circuit Schematic Diagrams)
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 $1-$31 in MIPS)
  – 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’?
Second Try: How About This?

Register is used to hold up the transfer of data to adder

Square wave clock sets when things change

Rough timing ...

High (1)
LOAD/CLK
Low (0)

High (1)
Low (0)

High (1)
Low (0)

Time

Rounded Rectangle per clock means could be 1 or 0

Xi must be ready before clock edge due to adder delay
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)
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):

[Diagram of flip-flop timing with waveforms showing input data, setup time, hold time, and clk-to-q delay]
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
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 $(i-1)$th 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.

![Diagram of accumulator timing](image)
Maximum Clock Frequency

• What is the maximum frequency of this circuit?

Max Delay = \text{CLK-to-Q Delay} + \text{CL Delay} + \text{Setup Time}
Critical Paths

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

- 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)
Clickers/Peer Instruction

What is maximum clock frequency?

- 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
Administrivia

• Project 1-1 due 3/01
• Midterm is next Thursday 2/26, in class
  – Covers up to and including the previous lecture
  – 1 handwritten, double sided, 8.5”x11” cheat sheet
  – We’ll give you MIPS green sheet
• Review Sessions:
  – TA: Monday 2/23, 7-9pm, 10 Evans
  – HKN: Saturday 2/21, 1-4pm, 100 Genetics Plant Biology
Finite State Machines (FSM) Intro

• You have seen FSMs in other classes.
• Same basic idea.
• 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...

FSM to detect the occurrence of 3 consecutive 1's in the input.

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.

Combination logic circuit is used to implement a function 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>
Moving between Representations

- Use this table and techniques we learned last time to transform between alternative views of same logic function
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}a\overline{b} + \overline{s}ab + s\overline{ab} + sab
= \overline{s}(a\overline{b} + ab) + s(\overline{ab} + 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?

\[ \overline{s}a + s\overline{b} \]
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?

Hint: NCAA tourney!

Ans: Hierarchically!
• 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 $S=10$, $R=A \text{ AND } B$
when $S=11$, $R=A \text{ OR } B$
Our simple ALU
In the News: Microsoft, Google beat Humans at Image Recognition (EE Times)

- On ImageNet benchmark image database, systems from Microsoft and Google performed better than humans at recognizing images.
- Both companies used deep artificial neural networks to train on image database.
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

\[
\begin{array}{ccc}
\text{a}_3 & \text{a}_2 & \text{a}_1 \\
\hline
\text{b}_3 & \text{b}_2 & \text{b}_1 \\
\hline
\text{s}_3 & \text{s}_2 & \text{s}_1 \\
\end{array}
\]

\[
\begin{array}{c|cc}
\text{a}_0 & \text{b}_0 & \text{s}_0 & \text{c}_1 \\
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}
\]

\[
s_0 = \\
c_1 =
\]
Adder/Subtractor – One-bit adder

\[
\begin{array}{cccc}
\begin{array}{ccc}
 a_3 & a_2 & a_1 \\
 b_3 & b_2 & b_1
\end{array} \\
\begin{array}{ccc}
 a_0 & s_3 & s_2 \\
 b_0 & s_1 & s_0
\end{array}
\end{array}
\]

\[
\begin{array}{cccc|cc}
 a_i & b_i & c_i & s_i & c_{i+1} \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 1 & 1 & 0 \\
 0 & 1 & 0 & 1 & 0 \\
 0 & 1 & 1 & 0 & 1 \\
 1 & 0 & 0 & 1 & 0 \\
 1 & 0 & 1 & 0 & 1 \\
 1 & 1 & 0 & 0 & 1 \\
 1 & 1 & 1 & 1 & 1
\end{array}
\]
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 ⇒ 1 N-bit adder

What about overflow?
Overflow = $c_n$?
Extremely Clever Subtractor

XOR serves as conditional inverter!

<table>
<thead>
<tr>
<th></th>
<th></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>
In Conclusion

• Finite State Machines have clocked state elements plus combinational logic to describe transition between states

• Standard combinational functional unit blocks built hierarchically from subcomponents