(Quick) REVIEW
FSMs and RTL Design

Graphical and Textual Sequential Circuit Descriptions

- **FSM**
  - Graphical representation
  - Formal method to describe sequential circuits
- **State Table**
  - Textual Representation

How do we implement a sequential circuit?
- Standard Controller architecture
  - Need to store state
  - State register (encoded state)
  - Need to determine next state
  - Current state and external input to combinational logic
  - Need to determine output
  - Current state input to combinational logic

FSM Example: Three-Cycles High Laser Timer

- **State Diagram or Finite-State Machine (FSM)**
  - A way to describe desired behavior of sequential circuit
  - List states, and transitions among states
- **Laser Timer**
  - When button pressed (b=1), turn laser on (x=1) for 3 clock cycles
  - Four states
    - **Off state**
      - Keep laser turned off
      - While b=0 (b'), we are in a wait state
      - When b=1 and rising clock edge (b \cdot \text{clk}\uparrow), transition to On1 state
    - **On1 state**
      - Turns laser on (x=1)
      - On next rising clock edge (\text{clk}\uparrow) transition to On2 state
    - **On2/On3 state**
      - Also turns laser on (x=1)
      - Transitions on next rising clock edge

State Table Example: Laser Timer (cont')

- **State Table**
  - **Inputs**
    - Current state (encoded) - Two bits s1 and s0 encode the current state
    - FSM Input - Input b indicates button press
  - **Outputs**
    - Next state (encoded) - Two bits n1 and n0 encode the next state
    - FSM Output - Output x controls when the laser is on/off
State Table Example: Laser Timer (cont’)

- **State Table**
  - **Next state**
    - Based on current state and FSM input, what is the next state?
  - **FSM Output**
    - Output depends on current state only (Moore FSM)
    - For each state we are currently in, what is the output?

![State diagram]

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1, s0, b, n1, n0, x</td>
<td></td>
</tr>
<tr>
<td><strong>Off</strong></td>
<td>0 0 0 0 1 0</td>
</tr>
<tr>
<td><strong>On1</strong></td>
<td>0 1 0 1 0 1</td>
</tr>
<tr>
<td><strong>On2</strong></td>
<td>1 0 0 1 1 1</td>
</tr>
<tr>
<td><strong>On3</strong></td>
<td>1 1 0 0 0 1</td>
</tr>
</tbody>
</table>

(Condensed) Controller Design Process

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1:</strong></td>
<td>Capture the FSM</td>
</tr>
<tr>
<td></td>
<td>Create an FSM (state diagram) that describes the desired behavior of the circuit</td>
</tr>
<tr>
<td><strong>Step 2:</strong></td>
<td>Create the architecture</td>
</tr>
<tr>
<td></td>
<td>Create the standard architecture by using a state register of appropriate width, and combinational logic with inputs being the state register bits and the FSM inputs, and outputs being the next state bits and the FSM outputs</td>
</tr>
<tr>
<td><strong>Step 3:</strong></td>
<td>Encode the states</td>
</tr>
<tr>
<td></td>
<td>Assign a unique binary number to each state. Each binary number representing a state is known as an encoding. Any encoding will do as long as they are unique.</td>
</tr>
<tr>
<td><strong>Step 4:</strong></td>
<td>Create the state table</td>
</tr>
<tr>
<td></td>
<td>Create a truth table for the combinational logic such that the logic will generate the correct FSM output and next state signals. Ordering the inputs with state bits first makes the truth table describe the state behavior, giving us a state table.</td>
</tr>
<tr>
<td><strong>Step 5:</strong></td>
<td>Implement the combinational logic</td>
</tr>
<tr>
<td></td>
<td>Implement the combinational logic using any method.</td>
</tr>
</tbody>
</table>

Controller Design: Laser Timer

- **Example: Laser Timer**
- **Step 1:** Capture the FSM
  - Already done
- **Step 2:** Create architecture
  - Customize generic controller architecture to our system
    - **State Register**
      - 2-bit state register (for 4 states)
      - s1, s0 – current state bits
      - n1, n0 – next state bits
    - **FSM Input**
      - Button signal b
    - **FSM Output**
      - Laser control x

![Controller design diagram]

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1, s0, b, n1, n0, x</td>
<td></td>
</tr>
<tr>
<td><strong>Off</strong></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td><strong>On1</strong></td>
<td>0 0 1 0 1 0</td>
</tr>
<tr>
<td><strong>On2</strong></td>
<td>0 0 1 1 0 1</td>
</tr>
<tr>
<td><strong>On3</strong></td>
<td>1 0 1 1 1 1</td>
</tr>
</tbody>
</table>

Controller Design: Laser Timer (Cont’)

- **Step 3:** Encode the states
  - Any encoding with each state unique will work
- **Step 4:** Create state table
  - Done this already

![State table]

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1, s0, b, n1, n0, x</td>
<td></td>
</tr>
<tr>
<td><strong>Off</strong></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td><strong>On1</strong></td>
<td>0 0 1 0 1 0</td>
</tr>
<tr>
<td><strong>On2</strong></td>
<td>0 0 1 1 0 1</td>
</tr>
<tr>
<td><strong>On3</strong></td>
<td>1 0 1 1 1 1</td>
</tr>
</tbody>
</table>
Step 5: Implement the combinational logic

\begin{align*}
n_1 &= s_1s_0' + s_1s_0b' + s_1s_0b + s_1s_0b' \\
n_0 &= s_1s_0' + s_1s_0b + s_1s_0b' + s_1s_0b \\
x &= s_1s_0b' + s_1s_0b + s_1s_0'b' + s_1s_0'b + s_1s_0b' + s_1s_0b
\end{align*}

Controller Design: Laser Timer (Cont’)

FSM Formal Definition

- FSM defined by quintuple
  \[ M = (\Sigma, \Gamma, S, \delta, \lambda, s_0) \]
  
  - \( \Sigma \) is the input alphabet
  
  - \( \Gamma \) is the output alphabet
  
  - \( S \) is a finite set of states
  
  - \( \delta \) is the transition function, \( \delta : X \times S \rightarrow S \)
    
    - Given and input and state, what is the next state
  
  - \( \lambda \) is the output function, \( \lambda : S \rightarrow Y \)
    
    - Mealy FSM, \( \lambda : X \times S \rightarrow Y \)
  
  - \( s_0 \) is the initial state

LaserTimer = (\( \Sigma, \Gamma, S, \delta, \lambda, q_0 \)), where

\[ \Sigma = \{0, 1\} \]
\[ \Gamma = \{0, 1\} \]
\[ S = \{\text{Off}, \text{On1}, \text{On2}, \text{On3}\} \]
\[ \delta (\text{Off}, 0) = \text{Off}, \quad \delta (\text{Off}, 1) = \text{On1} \]
\[ \delta (\text{On1}, 0) = \text{On2}, \quad \delta (\text{On1}, 1) = \text{On2} \]
\[ \delta (\text{On2}, 0) = \text{On3}, \quad \delta (\text{On2}, 1) = \text{On3} \]
\[ \delta (\text{On3}, 0) = \text{Off}, \quad \delta (\text{On3}, 1) = \text{Off} \]
\[ \lambda (\text{Off}) = 0, \quad \lambda (\text{On1}) = 1, \quad \lambda (\text{On2}) = 1, \quad \lambda (\text{On3}) = 1 \]
\[ s_0 = \text{Off} \]

\( Y \) is the output alphabet

- \( S \) is a finite set of states

- \( \delta \) is the transition function, \( \delta : X \times S \rightarrow S \)
  
  - Given and input and state, what is the next state

- \( \lambda \) is the output function, \( \lambda : S \rightarrow Y \)
  
  - s_0 is the initial state
Mealy vs. Moore FSM

- Previously associated output with current state
  - Moore FSM
- Another type associates output with both the state and the FSM input (transitions)
  - Mealy FSM
- Example: Soda dispenser
  - Input
    - `enough` - indicates when sufficient money deposited
  - Output
    - `d` - releases a soda
    - `clear` - zeros device counting money deposited

Moore FSM
Inputs: `enough` (bit)
Outputs: `d`, `clear` (bit)

Mealy FSM
Inputs: `enough` (bit)
Outputs: `d`, `clear` (bit)

Why does the timing change?
- More detailed view of FSM implementation architecture

Moore vs. Mealy FSM - Architecture

Moore FSM
Next state logic – function of present state and FSM inputs
Output logic - function of present state only

Mealy FSM
Next state logic – function of present state and FSM inputs
Output logic - function of present state and FSM inputs

Moore FSM Output Condition
- Moore FSM
  - Output logic is combinational – output changes when input changes
  - Only input is state

Mealy FSM Output Condition
- Mealy FSM
  - Again output logic is combinational – output changes when input changes
  - Input is both state and FSM input
RTL Design Method

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1:</strong></td>
<td>Capture the high-level FSM</td>
</tr>
<tr>
<td><strong>Step 2:</strong></td>
<td>Create a datapath to carry out the data operations on the high-level state machine</td>
</tr>
<tr>
<td><strong>Step 3:</strong></td>
<td>Connect the datapath to the controller block. Connect external Boolean inputs and output to the controller block</td>
</tr>
<tr>
<td><strong>Step 4:</strong></td>
<td>Derive the controller's FSM</td>
</tr>
</tbody>
</table>

Laser-Based Distance Measurer

**Step 1: Capture a high-level state machine**

- **Object of interest**
- **2D = T sec * 3*10^8 m/sec**

- **Example of how to create a high-level state machine to describe desired processor behavior**
- **Laser-based distance measurement – pulse laser, measure time T to sensor reflection**
  - Laser light travels at speed of light, 3*10^8 m/sec
  - Distance is thus \( D = \frac{T \text{ sec} \times 3 \times 10^8 \text{ m/sec}}{2} \)

**Laser-Based Distance Measurer**

**Step 1 : Capture a high-level state machine**

- **Inputs/outputs**
  - \( B \): bit input, from button to begin measurement
  - \( L \): bit output, activates laser
  - \( S \): bit input, senses laser reflection
  - \( D \): 16-bit output, displays computed distance

- **Step 1: Create high-level state machine**
  - Begin by declaring inputs and outputs
  - Create initial state, name it **S0**
    - Initialize laser to off \((L=0)\)
    - Initialize displayed distance to 0 \((D=0)\)
Laser-Based Distance Measurer

Step 1: Capture a high-level state machine

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)
Local Registers: Dctr (16 bits)

- Add another state, call S1, that waits for a button press
  - B' – stay in S1, keep waiting
  - B – go to a new state S2

Q: What should S2 do?
A: Turn on the laser

- Stay in S3 until sense reflection (S)
- To measure time, count cycles for which we are in S3
  - To count, declare local register Dctr
  - Increment Dctr each cycle in S3
  - Initialize Dctr to 0 in S1. S2 would have been O.K. too

- Once reflection detected (S), go to new state S4
  - Calculate distance
  - Assuming clock frequency is \(3 \times 10^8\), Dctr holds number of meters, so \(D = \frac{Dctr}{2}\)
  - After S4, go back to S1 to wait for button again
Digital Design
Copyright © 2006
Frank Vahid

Laser-Based Distance Measurer
Step 2: Create a Datapath

- Datapath must
  - Implement data storage
  - Implement data computations

- Look at high-level state machine, do three substeps
  a) Make data inputs/outputs be datapath inputs/outputs
  b) Instantiate declared registers into the datapath (also instantiate a register for each data output)
  c) Examine every state and transition, and instantiate datapath components and connections to implement any data computations

**Instantiate:** to introduce a new component into a design.

Digital Design
Copyright © 2006
Frank Vahid

Laser-Based Distance Measurer
Step 3: Connecting the Datapath to a Controller

- Laser-based distance measurer example
- Easy – just connect all control signals between controller and datapath
Laser-Based Distance Measurer

Step 4: Deriving the Controller’s FSM

- FSM has same structure as high-level state machine
  - Inputs/outputs all bits now
  - Replace data operations by bit operations using datapath

**Inputs:** B, S (1 bit each)
**Outputs:** L, D (16 bits)
Local Registers: Dctr (16 bits)

### Local Registers: Dctr (16 bits)

- **S0:** L = 0, Dreg_clr = 1
  - (laser off)
- **S1:** L = 1, Dreg_clr = 1
  - (clear D reg)
- **S2:** L = 0, Dreg_ld = 1
  - (load D reg with Dctr/2)
- **S3:** L = 1, Dctr_cnt = 1
  - (laser off)
- **S4:** L = 0, Dctr_cnt = 0
  - (stop counting)

### Inputs/Outputs

- **B**
- **S**

### Outputs

- **L**
- **D**

### Transition Diagram

- **Transition Diagram:**
  - From B, S (1 bit each)
  - To L, D (16 bits)
  - Inputs/outputs all bits now
  - Replace data operations by bit operations using datapath

---

Video Compression – Sum of Absolute Differences

- Need to quickly determine whether two frames are similar enough to just send difference for second frame
  - Compare corresponding 16x16 "blocks"
    - Treat 16x16 block as 256-byte array
    - Compute the absolute value of the difference of each array item
    - Sum those differences – if above a threshold, send complete frame for second frame; if below, can use difference method (using another technique, not described)

### Compare

- Each is a pixel, assume represented as 1 byte
- (actually, a color picture might have 3 bytes per pixel, for intensity of red, green, and blue components of pixel)

### Video Compression – Sum of Absolute Differences

- **Want fast sum-of-absolute-differences (SAD) component**
  - When go=1, sums the differences of element pairs in arrays A and B, outputs that sum

### Diagram

- **Diagram:**
  - 256-byte array A
  - 256-byte array B
  - SAD
  - Integrate

---

Video Compression – Sum of Absolute Differences

- Video is a series of frames (e.g., 30 per second)
- Most frames similar to previous frame
  - Compression idea: just send difference from previous frame

### Diagram

- **Diagram:**
  - Digitized frame 1
  - Digitized frame 2
  - Difference of 2 from 1
  - 0.01 Mbyte
  - Just send difference

### Diagram

- **Diagram:**
  - Digitized frame 1
  - Digitized frame 2
  - Digitized frame 1
  - Digitized frame 2
  - 1 Mbyte
  - 1 Mbyte
  - 1 Mbyte
  - 1 Mbyte

### Diagram

- **Diagram:**
  - Digitized frame 1
  - Digitized frame 2
  - Digitized frame 1
  - Digitized frame 2
  - 1 Mbyte
  - 1 Mbyte
  - 1 Mbyte
  - 1 Mbyte

---

Video Compression – Sum of Absolute Differences

- **Video Compression – Sum of Absolute Differences**
  - Need to quickly determine whether two frames are similar enough to just send difference for second frame
  - Compare corresponding 16x16 "blocks"
    - Treat 16x16 block as 256-byte array
    - Compute the absolute value of the difference of each array item
    - Sum those differences – if above a threshold, send complete frame for second frame; if below, can use difference method (using another technique, not described)

### Diagram

- **Diagram:**
  - Digitized frame 1
  - Digitized frame 2
  - Digitized frame 1
  - Digitized frame 2
  - 1 Mbyte
  - 1 Mbyte
  - 1 Mbyte
  - 1 Mbyte

### Diagram

- **Diagram:**
  - Digitized frame 1
  - Digitized frame 2
  - Digitized frame 1
  - Digitized frame 2
  - 1 Mbyte
  - 1 Mbyte
  - 1 Mbyte
  - 1 Mbyte

### Diagram

- **Diagram:**
  - Digitized frame 1
  - Digitized frame 2
  - Digitized frame 1
  - Digitized frame 2
  - 1 Mbyte
  - 1 Mbyte
  - 1 Mbyte
  - 1 Mbyte
Video Compression – Sum of Absolute Differences

Step 1: Create high-level state machine
- S0: wait for go
- S1: initialize sum and index
- S2: check if done (i >= 256)
- S3: add difference to sum, increment index
- S4: done, write to output sad_reg

Inputs: A, B (256 byte memory); go (bit)
Outputs: sad (32 bits)
Local registers: sum, sad_reg (32 bits); i (9 bits)

Step 2: Create datapath

Inputs: A, B (256 byte memory); go (bit)
Outputs: sad (32 bits)
Local registers: sum, sad_reg (32 bits); i (9 bits)

Step 3: Connect to controller

Step 4: Replace high-level state machine by FSM

Comparing software and custom circuit
SAD
Circuit: Two states (S2 & S3) for each i, 256 fs → 512 clock cycles
Software: Loop (for i = 1 to 256), but for each i, must move memory to local registers, subtract, compute absolute value, add to sum, increment i – say about 6 cycles per array item → 256*6 = 1536 cycles
Circuit is about 3 times (300%) faster
RTL Design Pitfalls and Good Practice

- Common pitfall: Assuming register is update in the state it’s written

- Example
  - Final value of Q? Final state?
  - Answers may surprise you
    - Value of Q unknown
    - Final state is C, not D

- Why?
  - State A: R=99 and Q=R happen simultaneously
  - State B: R not updated with R+1 until next clock cycle, simultaneously with state register being updated

Determining Clock Frequency

- Designers of digital circuits often want fastest performance
  - Means want high clock frequency

- Frequency limited by *longest register-to-register delay*\
  - Known as **critical path**
  - If clock is any faster, incorrect data may be stored into register
  - Longest path on right is 2 ns
    - Ignoring wire delays, and register setup and hold times, for simplicity

Critical Path

- Example shows four paths
  - a to c through + (2 ns)
  - a to d through + and * (7 ns)
  - b to d through + and * (7 ns)
  - b to d through * (5 ns)

- Longest path is thus 7 ns
- Fastest frequency
  - $1 / 7 \text{ ns} = 142 \text{ MHz}$

\* register-to-register or input-to-register delay
Critical Path Considering Wire Delays

- Real wires have delay too
  - Must include in critical path
- Example shows two paths
  - Each is $0.5 + 2 + 0.5 = 3$ ns
- Trend
  - 1980s/1990s: Wire delays were tiny compared to logic delays
  - But wire delays not shrinking as fast as logic delays
    - Wire delays may even be greater than logic delays!
- Must also consider register setup and hold times, also add to path
- Then add some time to the computed path, just to be safe
  - e.g., if path is 3 ns, say 4 ns instead

Sequential Logic Design

Non-Ideal Flip-Flop Behavior

- Can't change flip-flop input too close to clock edge
  - Setup time: time that D must be stable before edge
    - Else, stable value not present at internal latch
  - Hold time: time that D must be held stable after edge
    - Else, new value doesn’t have time to loop around and stabilize in internal latch

Sequential Logic Design

Metastability

- Violating setup/hold time can lead to bad situation known as metastable state
  - Metastable state: Any flip-flop state other than stable 1 or 0
    - Eventually settles to one or other, but we don’t know which
  - For internal circuits, we can make sure observe setup time
  - But what if input comes from external (asynchronous) source, e.g., button press?
- Partial solution
  - Insert synchronizer flip-flop for asynchronous input
    - Special flip-flop with very small setup/hold time
  - Doesn’t completely prevent metastability

A Circuit May Have Numerous Paths

- Paths can exist
  - In the datapath
  - In the controller
  - Between the controller and datapath
  - May be hundreds or thousands of paths
- Timing analysis tools that evaluate all possible paths automatically very helpful
Earlier sum-of-absolute-differences example
- Started with high-level state machine
- C code is an even better starting point -- easier to understand

C code
```c
int SAD (byte A[256], byte B[256]) // not quite C syntax
{
    uint sum; short uint i;
    sum = 0;
    while (i < 256) {
        sum = sum + abs(A[i] - B[i]);
        i = i + 1;
    }
    return sum;
}
```

Conversion from C to High-Level State Machine
- Convert each C construct to equivalent states and transitions
  - Assignment statement
    - Becomes one state with assignment
  - If-then statement
    - Becomes state with condition check, transitioning to "then" statements if condition true, otherwise to ending state
    - "then" statements would also be converted to states
  - If-then-else statement
    - Becomes state with condition check, transitioning to "then" statements if condition true, or to "else" statements if condition false
  - While loop statement
    - Becomes state with condition check, transitioning to while loop's statements if true, then transitioning back to condition check

Behavioral-Level Design: C to Gates
- Start with C (or Similar Language)
  - Replace first step of RTL design method by two steps
    - Capture in C, then convert C to high-level state machine
    - How convert from C to high-level state machine?

Step 1A: Capture in C
Step 1B: Convert to high-level state machine
Simple Example of Converting from C to High-Level State Machine

- Simple example: Computing the maximum of two numbers
  - Convert if-then-else statement to states (b)
  - Then convert assignment statements to states (c)

Inputs: uint X, Y
Outputs: uint Max

if (X > Y) {
  Max = X;
} else {
  Max = Y;
}

Example: Converting Sum-of-Absolute-Differences C code to High-Level State Machine

- From high-level state machine, follow RTL design method to create circuit
- Thus, can convert C to gates using straightforward automatable process
  - Not all C constructs can be efficiently converted
  - Use C subset if intended for circuit
  - Can use languages other than C, of course

```c
main()
{
  uint sum; short uint i;
  while (1) {
    sum = 0;
    i = 0;
    while (!go);
    while (i < 256) {
      i = i + 1;
    }
    sad = sum;
  }
}
```