September 12, 2023. Apple announces iPhone 15 Pro with A17 Pro

Apple's A17 Pro is a 3nm Chip Powering iPhone 15 Pro, Pro Max. A17 Pro introduces hardware ray tracing in a new GPU, with console-quality games announced for the phone.

Review

• Sequential logic uses flip-flops and (sometimes) latches
  • Flip-flops and latches are inferred in Verilog
    • Always blocks

• Blocking and non-blocking assignments

• Verification effort is equal to the design effort
  • Testbenches, simulation, coverage, assertions

• Practice is the best way to learn a new language…

• Combinational logic block outputs depend only on its inputs

• Boolean algebra can be used for manipulation and simplification of Boolean equations
Combinational Logic (cont’d)
Proving Distributive Law

• $X \cdot (Y+Z) = (X\cdot Y) + (X\cdot Z)$

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$Z$</th>
<th>$(Y+Z)$</th>
<th>$X \cdot (Y+Z)$</th>
<th>$(X\cdot Y)$</th>
<th>$(X\cdot Z)$</th>
<th>$(X\cdot Y) + (X\cdot Z)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Proving Distributive Law

\[ X \cdot (Y+Z) = (X \cdot Y) + (X \cdot Z) \]

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>(Y+Z)</th>
<th>X \cdot (Y+Z)</th>
<th>(X \cdot Y)</th>
<th>(X \cdot Z)</th>
<th>(X \cdot Y) + (X \cdot Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
DeMorgan's Law

- Theorem for complementing a complex function.

\[(x + y)' = x' \cdot y'\]

\[(x \cdot y)' = x' + y'\]
DeMorgan's Law

• Procedure for complementing a complex function.

\[(x + y)' = x' y'\]

\((x y)' = x' + y'\)

\[
\begin{array}{cccccc}
 x & y & x' & y' & (x + y)' & x' y' \\
 0 & 0 & 1 & 1 & 1 & 1 \\
 0 & 1 & 1 & 0 & 0 & 0 \\
 1 & 0 & 0 & 1 & 0 & 0 \\
 1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccc}
 x & y & x' & y' & (x y)' & x' + y' \\
 0 & 0 & 1 & 1 & 1 & 1 \\
 0 & 1 & 1 & 0 & 1 & 1 \\
 1 & 0 & 0 & 1 & 1 & 1 \\
 1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Administrivia

• Discussion on Friday 8am cancelled

• Midterm 1: Thursday 10/5, 7-8:30pm

• Lab 2 finished
  • If you couldn’t attend the lab, please get checked off in office hours this week

• Lab 3 is this week
  • Cycle wraps up on Monday
  • It is a more complex lab

• Homework 2 due this Friday
  • Homework 3 posted on Thursday
Finite-State Machines
Sequential logic

• Combinational logic:
  • Memoryless: the outputs only dependent on the current inputs.

• Sequential logic:
  • Memory: the outputs depend on both current and previous values of the inputs.
    • Distill the prior inputs into a smaller amount of information, i.e., states.
  • State: the information about a circuit
    • Influences the circuit’s future behavior
    • Stored in Flip-flops and Latches
  • Finite State Machines:
    • Useful representation for designing sequential circuits
    • As with all sequential circuits: output depends on present and past inputs
    • We will first learn how to design by hand then how to implement in Verilog.
FSM Example

• Cat Brain (Simplified…)
  • Inputs:
    • Feeding
    • Petting
  • Outputs:
    • Eyes: open or close
    • Mouth: open or close
  • States:
    • Eating
    • Sleeping
    • Annoyed…

Cat Brain FSM

- Feeding
- Petting
- Eyes
- Mouth
FSM State Transition Diagram

• States:
  • Circles
• Outputs:
  • Labeled in each state
  • Arcs
• Inputs:
  • Arcs

<Diagram>

- **Eat**
  - Eyes: T
  - Mouth: T
- **Sleep**
  - Eyes: F
  - Mouth: F
- **Annoyed**
  - Eyes: T
  - Mouth: F

Arrows indicate transitions between states with labels indicating inputs or outputs.
### FSM Symbolic State Transition Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Input</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eat</td>
<td>Feeding</td>
<td>Eat</td>
</tr>
<tr>
<td>Eat</td>
<td>Petting</td>
<td>Sleep</td>
</tr>
<tr>
<td>Sleep</td>
<td>Feeding</td>
<td>Sleep</td>
</tr>
<tr>
<td>Sleep</td>
<td>Petting</td>
<td>Annoyed</td>
</tr>
<tr>
<td>Annoyed</td>
<td>Feeding</td>
<td>Eat</td>
</tr>
<tr>
<td>Annoyed</td>
<td>Petting</td>
<td>Annoyed</td>
</tr>
</tbody>
</table>
FSM Encoded State Transition Table

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eat</td>
<td>00</td>
</tr>
<tr>
<td>Sleep</td>
<td>01</td>
</tr>
<tr>
<td>Annoyed</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Current State</th>
<th>Input</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>S0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ S0' = \overline{S1S0X} + \overline{S1S0\overline{X}} = S1(\overline{S0X} + S0\overline{X}) = S1(S0+X) \]

\[ S1' = \overline{S1S0X} + S1\overline{S0X} = (S1\oplus S0)X \]
# FSM Output Table

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eat</td>
<td>00</td>
</tr>
<tr>
<td>Sleep</td>
<td>01</td>
</tr>
<tr>
<td>Annoyed</td>
<td>10</td>
</tr>
</tbody>
</table>

## Current State Outputs

<table>
<thead>
<tr>
<th>Current State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1 S0 E M</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

\[
E = \overline{S1S0} + S1\overline{S0} = \overline{S0}
\]

\[
M = \overline{S1} \overline{S0}
\]
\[ S_1' = X(S_0 \oplus S_1) \]
\[ S_0' = \overline{S_1}(S_0 \oplus X) \]
\[ E = \overline{S_0} \]
\[ M = \overline{S_1}S_0 \]
FSM Design Process

- Specify circuit function
- Draw state transition diagram
- Write down symbolic state transition table
- Write down encoded state transition table
- Derive logic equations
- Derive circuit diagram
  - Register to hold state
  - Combinational logic for next state and outputs
FSM State Encoding

• Binary encoding:
  • i.e., for four states, 00, 01, 10, 11

• One-hot encoding
  • One state bit per state
  • Only one state bit TRUE at once
  • i.e., for four states, 0001, 0010, 0100, 1000
  • Requires more flip-flops
  • Often next state and output logic can be simpler
Moore and Mealy FSMs
Moore’s vs Mealy’s FSMs

• Next state is always determined by current state and inputs

• Differ in output logic:
  • Moore FSM: outputs depend only on current state
  • Mealy FSM: outputs depend on current state and inputs
Example: Edge Detector

• **Input:**
  • A bit stream that is received one bit at a time.

• **Output:**
  • 0/1

• **Circuit:**
  • Asserts its output to be true when the input bit stream changes from 0 to 1.
State Transition Diagram Solution A

<table>
<thead>
<tr>
<th>Input</th>
<th>Current State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Zero (00)</td>
<td>Zero</td>
<td>0</td>
</tr>
</tbody>
</table>
State Transition Diagram Solution A

<table>
<thead>
<tr>
<th>Input</th>
<th>Current State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Zero (00)</td>
<td>Zero</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Zero (00)</td>
<td>Change</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>Change (01)</td>
<td>Zero</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Change (01)</td>
<td>One</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>One (11)</td>
<td>Zero</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>One (11)</td>
<td>One</td>
<td>0</td>
</tr>
</tbody>
</table>
State Transition Diagram Solution A

CS = \{CS1, CS0\}

<table>
<thead>
<tr>
<th>Input</th>
<th>Current State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Zero (00)</td>
<td>Zero</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Zero (00)</td>
<td>Change</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>Change (01)</td>
<td>Zero</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Change (01)</td>
<td>One</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>One (11)</td>
<td>Zero</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>One (11)</td>
<td>One</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ NS_1 = \text{IN AND CS0} \]

\[ NS_0 = \text{IN} \]

\[ \text{OUT} = \text{NOT (CS1) AND CS0} \]
State Transition Diagram Solution B

<table>
<thead>
<tr>
<th>Input</th>
<th>Current State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Zero (0)</td>
<td>Zero</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Zero (0)</td>
<td>One</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>One (1)</td>
<td>Zero</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>One (1)</td>
<td>One</td>
<td>0</td>
</tr>
</tbody>
</table>
Edge Detection Timing Diagrams

- **Solution A (Moore)**: both edges of output follow the clock.
- **Solution B (Mealy)**: output rises with input rising edge and is not synchronous wrt the clock, output falls synchronously with next clock edge.
FSM Comparison

**Solution A**

**Moore Machine**

- output function only of current state
- maybe *more* states (why?)
- **synchronous** outputs
  - Input glitches not sent to output
  - one cycle “delay”
  - full cycle of stable output

**Solution B**

**Mealy Machine**

- output function of both current = & input
- maybe fewer states
- **asynchronous** outputs
  - if input glitches, so does output
  - output immediately available
  - output may not be stable long enough to be useful (below):

If output of Mealy FSM goes through combinational logic before being registered, the CL might delay the signal and it could be missed by the clock edge (or violate setup time requirement)
Quiz: Which of the diagrams are Moore machines?

A. AC
B. BD
C. AD
D. BC

www.yellkey.com/town
FSMs in Verilog
Implement FSM with Verilog

• Specify circuit function
• Draw state transition diagram
• Write down symbolic state transition table
• Assign encodings (bit patterns) to symbolic states
• Code as Verilog behavioral description
  • Use parameters to represent encoded states
  • Use separate always blocks for register assignment and combinational logic block
  • Use case statement for combinational logic.
    • Within each case section (state), assign outputs and next state based on inputs
    • Moore: outputs only dependent on states not on inputs
Finite State Machine in Verilog

**State Transition Diagram**

- **IDLE**
  - Input: in = 0
  - Output: out = 0

- **S0**
  - Input: in = 0
  - Output: out = 0

- **S1**
  - Input: in = 1
  - Output: out = 1

Holds a symbol to keep track of which bubble the FSM is in.

**Circuit Diagram**

- **State Register**
  - Clock (clk)
  - Reset (rst)

- **Combinational Logic**
  - Input: in
  - Output: out

CL functions to determine output value and next state based on input and current state.

- out = f(in, current state)
- next state = f(in, current state)
Finite State Machine in Verilog

```
module FSM1(
    input clk, rst,
    input in,
    output out
)

// Defined state encoding:
parameter IDLE = 2'b00;
parameter S0 = 2'b01;
parameter S1 = 2'b10;
reg out;
reg [1:0] current_state, next_state;

// always block for state register
always @(posedge clk)
    if (rst) current_state <= IDLE;
    else current_state <= next_state;
```

A separate always block should be used for combination logic part of FSM. Next state and output generation. (Always blocks in a design work in parallel.)
// always block for combinational logic portion
always @(current_state or in)
case (current_state)
// For each state def output and next
    IDLE   : begin
        out = 1'b0;
        if (in == 1'b1) next_state = S0;
        else next_state = IDLE;
    end
    S0     : begin
        out = 1'b0;
        if (in == 1'b1) next_state = S1;
        else next_state = IDLE;
    end
    S1     : begin
        out = 1'b1;
        if (in == 1'b1) next_state = S1;
        else next_state = IDLE;
    end
    default: begin
        next_state = IDLE;
        out = 1'b0;
    end
endcase
endmodule

For each state define:
- Output value(s)
- State transition

Use "default" to cover unassigned state. Usually unconditionally transition to reset state.

Each state becomes a case clause.
Finite State Machine in Verilog (cont.)

```verilog
always @(*) begin
  next_state = IDLE;
  out = 1'b0;
  case (state)
    IDLE : if (in == 1'b1) next_state = S0;
    S0  : if (in == 1'b1) next_state = S1;
    S1  : begin
      out = 1'b1;
      if (in == 1'b1) next_state = S1;
    end
    default: ;
  endcase
end
endmodule
```

* for sensitivity list

Nominal values: used unless specified below.

Within case only need to specify exceptions to the nominal values.

Note: The use of "blocking assignments" allow signal values to be "rewritten" (evaluated immediately), simplifying the specification.
Edge Detector Example

Mealy Machine

```verilog
always @(posedge clk)
  if (rst) ps <= ZERO;
  else ps <= ns;
always @(ps in)
  case (ps)
    ZERO: if (in) begin
      out = 1'b1;
      ns = ONE;
    end
    else begin
      out = 1'b0;
      ns = ZERO;
    end
    ONE: if (in) begin
      out = 1'b0;
      ns = ONE;
    end
    else begin
      out = 1'b0;
      ns = ZERO;
    end
    default: begin
      out = 1'bx;
      ns = default;
    end
```

Moore Machine

```verilog
always @(posedge clk)
  if (rst) ps <= ZERO;
  else ps <= ns;
always @(ps in)
  case (ps)
    ZERO: begin
      out = 1'b0;
      if (in) ns = CHANGE;
      else ns = ZERO;
    end
    CHANGE: begin
      out = 1'b1;
      if (in) ns = ONE;
      else ns = ZERO;
    end
    ONE: begin
      out = 1'b0;
      if (in) ns = ONE;
      else ns = ZERO;
    end
    default: begin
      out = 1'bx;
      ns = default;
    end
```

EECS151 L07 FSMs
Summary

• Finite state machines: Common example of sequential logic
  • Moore’s machine: Output depends only on the current state
  • Mealy’s machine: Output depends on the current state and the input

• Large state machines can be factored

• Common Verilog patterns for FSMs

• Common job interview questions 😊