Announcements

• Homework #7 “due” Thursday
  - Ensure you start studying for the midterm!
  - No new homework this week
  - (Don’t forget about project – next checkpoint due 1 week after midterm)

• Midterm next Thurs. 6pm **sharp**

• Elad will hold a review session next Mon. evening
  - Time/location TBD
Asynchronous Inputs and Metastability

Asynchronous Inputs to Synchronous Systems

• Many synchronous systems need to interface to asynchronous input signals
  - Examples:

  - “Brute force” solution:
“Synchronizer” Circuit

“Synchronizer” Circuit: What Not To Do
Source of Synchronizer Failures

Synchronizer Failure & Metastability

Transfer function:

- $V_{out} = \tau(\neg V_{in})$
- $V_{out} = \neg(\tau V_{in})$

where $\tau$ is the metastability time constant.
Synchronizer Failure & Metastability

- If the system uses a synchronizer output while the output is still in the indeterminate (metastable) state → synchronizer failure.
- Initial versions of several commercial ICs have suffered from metastability problems:
  - AMD9513 system timing controller
  - AMD9519 interrupt controller
  - Zilog Z-80 Serial I/O interface
  - Intel 8048 microprocessor
  - AMD 29000 microprocessor

Synchronizer Failure Solution

- Only real "solution": WAIT LONG ENOUGH
  - "Long enough", according to Wakerly, is so that the mean time between synchronizer failures is several orders of magnitude longer than the designer’s expected length of employment!
- In practice, can reduce probability of synchronizer failure to incredibly small values
Reliable Synchronizer Design

- The probability that a flip-flop stays in the metastable state decreases exponentially with time.
- Therefore, any scheme that delays using the signal can be used to decrease the probability of failure.
- In practice, delaying the signal by a cycle is usually sufficient:

```
<table>
<thead>
<tr>
<th>ASYNCIN</th>
<th>D</th>
<th>META</th>
<th>FF1</th>
<th>FF2</th>
<th>SYNCIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(asynchronous input)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(synchronous output)</td>
</tr>
<tr>
<td>CLOCK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(system clock)</td>
</tr>
</tbody>
</table>
```

- If the clock period is greater than metastability resolution time plus FF2 setup time, FF2 gets a synchronized version of ASYNCIN.
- Multi-cycle synchronizers [using counters or more cascaded flip-flops] are even better – but often overkill.

Now on to FSMs
**FSM Implementation**

- FFs form state register
- number of states $\leq 2^{\text{number of flip-flops}}$
- CL (combinational logic) calculates next state and output
- Remember: The FSM follows exactly one edge per cycle.

So far we have learned how to implement in Verilog. Now we will learn how to design “by hand” at the gate level.

---

**Parity Checker Example**

A string of bits has “even parity” if the number of 1’s in the string is even.

- Design a circuit that accepts a bit-serial stream of bits and outputs a 0 if the parity thus far is even and outputs a 1 if odd:

`Parity Checker` input: 0 0 1 1 1 0 1

Next we take this example through the “formal design process”. But first, can you guess a circuit that performs this function?
Parity Checker Example

A string of bits has "even parity" if the number of 1's in the string is even.

- Design a circuit that accepts a bit-serial stream of bits and outputs a 0 if the parity thus far is even and outputs a 1 if odd:

We will next we take this example through the "formal design process". But first, can you guess a circuit that performs this function?

Formal Design Process

"State Transition Diagram"

- circuit is in one of two "states".
- transition on each cycle with each new input, over exactly one arc (edge).
- Output depends on which state the circuit is in.
Formal Design Process

State Transition Table:

<table>
<thead>
<tr>
<th>present state</th>
<th>OUT</th>
<th>IN</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEN</td>
<td>0</td>
<td>0</td>
<td>EVEN</td>
</tr>
<tr>
<td>EVEN</td>
<td>0</td>
<td>1</td>
<td>ODD</td>
</tr>
<tr>
<td>ODD</td>
<td>1</td>
<td>0</td>
<td>ODD</td>
</tr>
<tr>
<td>ODD</td>
<td>1</td>
<td>1</td>
<td>EVEN</td>
</tr>
</tbody>
</table>

Invent a code to represent states:
Let 0 = EVEN state, 1 = ODD state

<table>
<thead>
<tr>
<th>present state (ps)</th>
<th>OUT</th>
<th>IN</th>
<th>next state (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Derive logic equations from table (how?):

- \( \text{OUT} = \text{PS} \)
- \( \text{NS} = \text{PS} \oplus \text{IN} \)

Logic equations from table:

- \( \text{OUT} = \text{PS} \)
- \( \text{NS} = \text{PS} \oplus \text{IN} \)

- Circuit Diagram:
  - XOR gate for ns calculation
  - DFF to hold present state
  - no logic needed for output in this example.
Formal Design Process

Review of Design Steps:

1. Specify **circuit function** (English)
2. Draw **state transition diagram**
3. Write down **symbolic state transition table**
4. Write down **encoded state transition table**
5. Derive **logic equations**
6. Derive **circuit diagram**
   - Register to hold state
   - Combinational Logic for Next State and Outputs

State Encoding

- In general:
  - # of possible FSM state = $2^{\# \text{ of FFs}}$
  - Example:
    - state1 = 01, state2 = 11, state3 = 10, state4 = 00
- However, often more than $\log_2(\# \text{ of states})$ FFs are used, to simplify logic at the cost of more FFs.
- Extreme example is one-hot state encoding.
State Encoding

- One-hot encoding of states.
- One FF per state.

Ex: 3 States,

STATE1: 001
STATE2: 010
STATE3: 100

- Why one-hot encoding?
  - Simple design procedure.
    - Circuit matches state transition diagram (example next page).
    - Often can lead to simpler and faster "next state" and output logic.
- Why not do this?
  - Can be costly in terms of FFs for FSMs with large number of states.
  - FPGAs are "FF rich", therefore one-hot state machine encoding is often a good approach.

One-hot encoded FSM

- Even Parity Checker Circuit:

  - FFs must be initialized for correct operation (only one 1)

  Circuit generated through direct inspection of the STD.

- In General:

  - state FF Input
  - to other state FF logic and/or output
Another Ex: One-Hot_encoded Combination Lock

FSM Implementation Notes

• General FSM form:

• All examples so far generate output based only on the present state:

• Commonly named Moore Machine
  (If output functions include both present state and input then called a Mealy Machine)
Finite State Machines

• Example: Edge Detector

Bit are received one at a time (one per cycle), such as: 000111010 → time

Design a circuit that asserts its output for one cycle when the input bit stream changes from 0 to 1.

Try two different solutions.

State Transition Diagram Solution A

ZERO

OUT=0

IN=0

IN=1

CHANGE

OUT=1

IN=1

ONE

OUT=0

IN=1

ZERO

IN PS NS OUT

0 00 00 0

1 00 01 0

CHANGE

0 01 00 1

1 01 11 1

ONE

0 11 00 0

1 11 11 0
Solution A, circuit derivation

<table>
<thead>
<tr>
<th>ZERO</th>
<th>PS</th>
<th>NS</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 1</td>
<td>0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1</td>
<td>1 1</td>
<td>1</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>0 1</td>
<td>1 1</td>
<td>0</td>
</tr>
</tbody>
</table>

Solution B

Output depends not only on PS but also on input, IN

<table>
<thead>
<tr>
<th>IN</th>
<th>PS</th>
<th>NS</th>
<th>OUT</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>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Let ZERO = 0, ONE = 1

NS = IN, OUT = IN PS'
Edge detector timing diagrams

- Solution A: output follows the clock
- Solution B: output changes with input rising edge and is asynchronous wrt the clock.

FSM Comparison

**Solution A**

*Moore Machine*
- output function only of PS
- maybe more states (why?)
- synchronous outputs
  - no glitches
  - one cycle “delay”
  - full cycle of stable output

**Solution B**

*Mealy Machine*
- output function of both PS & 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.
Final Notes on Moore versus Mealy

1. A given state machine could have both Moore and Mealy style outputs. Nothing wrong with this, but you need to be aware of the timing differences between the two types.

2. The output timing behavior of the Moore machine can be achieved in a Mealy machine by “registering” the Mealy output values:

General FSM Design Process with Verilog

Design Steps:  Implementation

1. Specify circuit function (English)
2. Draw state transition diagram
3. Write down symbolic state transition table
4. Assign encodings (bit patterns) to symbolic states
5. Code as Verilog behavioral description

✓ Use parameters to represent encoded states.
✓ Use separate always blocks for register assignment and CL logic block.
✓ Use case for CL block. Within each case section assign all outputs and next state value based on inputs.  Note: For Moore style machine make outputs dependent only on state not dependent on inputs.