Sequential Circuits
Circuits with state

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology
Combinational circuits

Such circuits have no cycles (feedback) or state elements
Simple circuits with feedback, i.e., a cycle

Circuits with cycles can hold state

Generally behavior is difficult to analyze and requires paying attention to propagation delays

This circuit will oscillate between 0 and 1

But how do we change its state?

This circuit can hold a 0 or 1
D Latch: a famous circuit that can hold state

if C=0, the value of D passes to Q
if C=1, the value of Q holds

Let $Q^{t-1}$ represents the value previously held in DL; $Q^t$ represents the current value.

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>$Q^{t-1}$</th>
<th>$Q^t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$X$</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$X$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$X$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$X$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Building circuits with D latches

- If two latches are driven by the same C signal, they *pass* signals at the same time and *hold* signals at the same time.
- The composed latches look just like a single D latch (assuming signals aren’t changing too fast)
Building circuits with D latches *continued*

- If latches are driven by inverted C signals, one is always holding, and one is always passing.

- How does this circuit behave?
  - When $C = 0$, $Q$ holds its old value, but $Q^{\text{int}}$ follows the input $D$.
  - When $C = 1$, $Q^{\text{int}}$ holds its old value, but $Q$ follows $Q^{\text{int}}$.
  - $Q$ doesn’t change when $C = 0$ or $C = 1$, but it changes its value when $C$ transitions from 0 to 1 (a *rising-edge* of $C$).
Edge-Triggered D Flip flop
A basic storage element

Suppose C changes periodically (called a Clock signal)

Data is sampled at the rising edge of the clock and must be stable at that time.
D Flip-flop with Write Enable

The building block of Sequential Circuits

Data is captured only if EN is on

No need to show the clock explicitly
Clocked Sequential Circuits

- In this class we will deal with only clocked sequential circuits.
- We will also assume that all flip flops are connected to the same clock.
- To avoid clutter, the clock input will be implicit and not shown in diagrams.
- Clock inputs are not needed in BSV descriptions unless we design multi-clock circuits.

February 22, 2018
Registers

Register: A group of flip-flops with a common clock and enable

Register file: A group of registers with a common clock, a shared set of input and output port
An example

Modulo-4 counter

<table>
<thead>
<tr>
<th>Prev State</th>
<th>nextState, inc = 0</th>
<th>nextState, inc = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1q0</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>00</td>
</tr>
</tbody>
</table>

$q_0^{t+1} = \overline{inc} \cdot q_0^t + inc \cdot \overline{q_0^t}$
$q_1^{t+1} = \overline{inc} \cdot q_1^t + inc \cdot \overline{q_1^t} \cdot q_0^t + inc \cdot q_1^t \cdot \overline{q_0^t}$

= $(inc == 1) \ ? q_0^t \oplus q_1^t : q_1^t$

Finite State Machine (FSM) representation
Finite State Machines (FSM) and Sequential Circuits

- FSMs are a mathematical object like the Boolean Algebra
  - A computer (in fact any digital hardware) is an FSM
- Synchronous Sequential Circuits is a method to implement FSMs in hardware

- Large circuits need to be described as a collection of cooperating FSMs
  - State diagrams and next-state tables are not suitable for such descriptions
Sequential circuits are described as modules in BSV

- A module has internal state
- The internal state can only be read and manipulated by the (interface) methods
- An action specifies which state elements are to be modified
- Actions are *atomic* -- either all the specified state elements are modified or none of them are modified (no partially modified state is visible)

```
interface Counter;
    method Action inc;
    method Bit#(2) read;
endinterface
```

A module in BSV is like a class definition in Java or C++
module moduloCounter(Counter);
    Reg#(Bit#(2)) cnt <- mkReg(0);
    method Action inc;
        cnt <= {cnt[1]^cnt[0],~cnt[0]};
    endmethod
    method Bit#(2) read;
        return cnt;
    endmethod
endmodule

interface Counter;
    method Action inc;
    method Bit#(2) read;
endinterface

An action to specify how the value of the cnt is to be set

<table>
<thead>
<tr>
<th>q1q0^t</th>
<th>q1q0^(t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
</tr>
</tbody>
</table>

State specification

Initial value
Modulo-4 counter

The generated circuit

```verilog
define module moduloCounter(Counter);
    Reg#(Bit#(2)) cnt <- mkReg(0);
    method Action inc;
        cnt <= {cnt[1]^cnt[0],~cnt[0]};
    endmethod
    method Bit#(2) read;
        return cnt;
    endmethod
endmodule
```

A hardware module for computing GCD

Euclid’s algorithm for computing the Greatest Common Divisor (GCD):

\[
\begin{array}{cccc}
15 & 6 & & \\
9  & 6 & subtract & \\
3  & 6 & subtract & \\
6  & 3 & swap & \\
3  & 3 & subtract & \\
0  & 3 & subtract & answer
\end{array}
\]

```python
def gcd(a, b):
    if a == 0: return b  # stop
    elif a >= b: return gcd(a-b, b) # subtract
    else: return gcd(b,a) # swap
```

GCD module

GCD can be started if the module is not busy; Results can be read when ready

```vhdl
interface GCD;
    method Action start (Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
    method Bool busy;
    method Bool ready;
endinterface
```
module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);
rule gcd;
    if (x >= y) begin x <= x - y; end //subtract
    else if (x != 0) begin x <= y; y <= x; end //swap
endrule
method Action start(Bit#(32) a, Bit#(32) b);
    x <= a; y <= b; busy_flag <= True;
endmethod
method ActionValue#(Bit#(32)) getResult;
    busy_flag <= False; return y;
endmethod
method Bool busy; return busy_flag;
endmethod
method Bool ready; return (busy_flag && (x==0));
endmethod
endmodule
A module may contain rules

```plaintext
rule gcd;
    if (x >= y) begin x <= x - y; end //subtract
    else if (x != 0) begin x <= y; y <= x; end //swap
endrule
```

- A rule is a collection of actions, which invoke methods
- All actions in a rule execute in parallel
- A rule can execute any time and when it executes all of its actions must execute
Parallel Composition of Actions & Double-Writes

rule one;
  y <= 3; x <= 5; x <= 7; endrule

rule two;
  y <= 3; if (b) x <= 7; else x <= 5; endrule

rule three;
  y <= 3; x <= 5; if (b) x <= 7; endrule

Parallel composition, and consequently a rule containing it, is illegal if a double-write possibility exists.

The BSV compiler rejects a program if it there is any possibility of a double write in a rule or method.

Double write
No double write
Possibility of a double write