Sequential Circuits
Circuits with state

Silvina Hanono Wachman
Computer Science & Artificial Intelligence Lab
M.I.T.

Reminders:
Quiz 1 tonight, 7:30-9:30PM
Last names A-S in 34-101
T-Z in 32-155
Good Luck!
Combinational circuits

Combinational 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 can hold a 0 or 1

But how do we change its state?

This circuit will oscillate between 0 and 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 \( Q^t \) represent the current value held in DL; \( Q^{t+1} \) represents the next value.
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)
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
 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 ports
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
An example

Modulo-4 counter

| Prev State | NextState
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>q1q0</td>
<td>inc = 0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Finite State Machine (FSM) representation

\[
q_0^{t+1} = \neg \text{inc} \cdot q_0^t + \text{inc} \cdot \neg q_0^t \\
= \text{inc} \oplus q_0^t
\]

\[
q_1^{t+1} = \neg \text{inc} \cdot q_1^t + \text{inc} \cdot \neg q_1^t \cdot q_0^t + \text{inc} \cdot q_1^t \cdot \neg q_0^t \\
= \neg \text{inc} \cdot q_1^t + \text{inc} \cdot (q_1^t \oplus q_0^t)
\]
Circuit for the modulo counter using D flip-flops with enables

\[ q_0^{t+1} = \sim \text{inc} \cdot q_0^t + \text{inc} \cdot \sim q_0^t \]
\[ q_1^{t+1} = \sim \text{inc} \cdot q_1^t + \text{inc} \cdot (q_1^t \oplus q_0^t) \]

- Use two D flip-flops with enables to store \( q_0 \) and \( q_1 \)
- Notice, the state of a flip-flop changes only when its is enable is true, and thus, the next-state input to a flip-flop matters only when its enable is true
  \[ q_0\text{en} = \text{inc}; \]
  \[ q_1\text{en} = \text{inc} \]
  \[ q_0^{t+1} = (\sim \text{inc} \cdot q_0^t + \text{inc} \cdot \sim q_0^t) \cdot q_0\text{en} \]
  \[ q_1^{t+1} = (\sim \text{inc} \cdot q_1^t + \text{inc} \cdot (q_1^t \oplus q_0^t)) \cdot q_1\text{en} \]

- simplifying the next-state equations
  \[ q_0^{t+1} = \sim q_0^t \]
  \[ q_1^{t+1} = q_1^t \oplus q_0^t \]
In our designs all registers are connected to the same clock input and therefore, we will not draw the clock wires in future.
Finite State Machines (FSMs)

- Synchronous Sequential Circuits are a method to implement FSMs in hardware
- FSMs are a much studied mathematical object like the Boolean Algebra
  - FSMs are used extensively in software as well
  - A computer (in fact any digital hardware) is an FSM, though we don’t think of it as such!
Sequential Circuit as a module with Interface

- A module has internal state and an interface
- The internal state can be read and manipulated only by its interface methods
- An *action* method specifies which state elements are to be modified; it has an *enable* wire which must be true to execute the action
- Actions are *atomic* -- either all the specified state elements are modified or none of them are modified (no partially modified state is visible)
- We refer to the *interface* of a module as its type

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

*A module in Bluespec is like a class definition in Java or C++*
interface Counter;
    method Action inc;
    method Bit#(2) read;
endinterface

module mkCounter (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

State specification

Initial value

Action to specify how the value of the cnt is to be set

q0^{t+1} = ~q0^{t}
q1^{t+1} = q1^{t} \oplus q0^{t}
Modulo-4 counter

The generated circuit

module mkCounter(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
Summary

- Sequential circuits have state and are built using registers to hold state
- A register has an enable signal and its state can be changed only if its enable is true
- A sequential circuit is represented by a module in Bluespec and has a well-defined set of interface methods to read and modify its state
  - A register is a primitive module type in Bluespec
- A module can be instantiated repeatedly to create objects, i.e., sequential circuits, of that type

Next more examples...
Take-home problems

- Write a module that can increment, decrement, or not change the value of a counter.
- Does this module need a busy method?
A method for computing GCD

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

\[
\begin{align*}
a & : 15 \\
 & : 9 \\
 & : 3 \\
 & : 6 \\
 & : 3 \\
 & : 0
\end{align*}
\]

\[
\begin{align*}
b & : 6 \\
 & : 6 \\
 & : 3 \\
 & : 3 \\
 & : 3
\end{align*}
\]

\textbf{answer} 3

\[
\begin{align*}
\text{def } \text{gcd}(a, b): \\
\text{if } a \text{ == 0: return } b \quad \# \text{ stop} \\
\text{elif } a \text{ >= } b: \text{ return gcd}(a-b, b) \quad \# \text{ subtract} \\
\text{else: return gcd } (b,a) \quad \# \text{ swap}
\end{align*}
\]
GCD module

interface GCD;
    method Action start (Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
    method Bool busy;
    method Bool ready;
endinterface

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

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 = busy_flag;
method Bool ready = x==0;
endmodule

Assume b != 0

start should be called only if the module is not busy;
getResult should be called only when ready is true.
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 there is a possibility of a double write

Stay tuned...