Sequential Circuits
Circuits with state

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

February 22, 2018
Combinational circuits
Combinational circuits

\[ A_0, A_1, \ldots, A_{n-1} \rightarrow \text{Mux} \rightarrow \text{Sel} \rightarrow \lg(n) \rightarrow O \]
Combinational circuits

\[
\begin{align*}
\text{Sel} & \quad \text{Mux} \quad \text{Demux} \\
A_0 & \quad A_1 \quad \vdots \quad A_{n-1} & \quad O \\
\text{Sel} & \quad \text{Demux} \\
l_{\text{lg}(n)} & \quad \text{O}_0 \quad \text{O}_1 \quad \text{O}_{n-1}
\end{align*}
\]
Combinational circuits

- Mux
  - Sel
  - $\log(n)$
  - $A_0$, $A_1$, $\ldots$, $A_{n-1}$
  - Output $O$

- Demux
  - Sel
  - $\log(n)$
  - $A$
  - Outputs $O_0$, $O_1$, $\ldots$, $O_{n-1}$

- Decoder
  - $A$
  - $\log(n)$
  - Outputs $O_0$, $O_1$, $\ldots$, $O_{n-1}$
Combinational circuits

- **OpSelect**
  - Add, Sub, ...
  - And, Or, Xor, Not, ...
  - GT, LT, EQ, Zero, ...

- **Result Comp?**

- **Mux**
  - \(A_0, A_1, \ldots, A_{n-1}\)

- **Decoder**
  - \(A\)
  - \(O_0, O_1, \ldots, O_{n-1}\)

- **Demux**
  - \(A\)
  - \(O_0, O_1, \ldots, O_{n-1}\)

- **ALU**
  - \(A\), \(B\)
  - \(Result\)
  - \(Comp?\)
Combinational circuits

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

This circuit can hold a 0 or 1
Simple circuits with feedback, i.e., a cycle

This circuit can hold a 0 or 1
But how do we change its state?
Simple circuits with feedback, i.e., a cycle

This circuit can hold a 0 or 1
But how do we change its state?
Simple circuits with feedback, i.e., a cycle

This circuit can hold a 0 or 1
But how do we change its state?
Simple circuits with feedback, i.e., a cycle

This circuit can hold a 0 or 1
But how do we change its state?
Simple circuits with feedback, i.e., a cycle

This circuit can hold a 0 or 1. But how do we change its state?
Simple circuits with feedback, i.e., a cycle

This circuit can hold a 0 or 1
But how do we change its state?
Simple circuits with feedback, i.e., a cycle

This circuit can hold a 0 or 1
But how do we change its state?

This circuit can hold a 0 or 1
But how do we change its state?
Simple circuits with feedback, i.e., a cycle

This circuit can hold a 0 or 1
But how do we change its state?

This circuit will oscillate between 0 and 1
Simple circuits with feedback, i.e., a cycle

This circuit can hold a 0 or 1
But how do we change its state?

This circuit will oscillate between 0 and 1

Circuits with cycles can hold state
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

Circuits with cycles can hold state

February 22, 2018
L05-3
D Latch: a famous circuit that can hold state

![Diagram of D Latch]

Q

\[ \begin{align*}
\text{D} & \quad 0 \\
\text{C} & \quad 0
\end{align*} \]
D Latch: a famous circuit that can hold state

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

<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>
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 $Q^{t-1}$ represent the value previously held in DL; $Q^t$ represents the current value.
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.
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.
Building circuits with D latches

\[
\begin{array}{c}
\text{DL} \\
D \\
C \\
Q_{\text{int}} \\
Q \\
\text{DL}
\end{array}
\]
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.
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

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
Building circuits with D latches continued

If latches are driven by inverted C signals, one is always holding, and one is always passing.
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?
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 $Q^{int}$ follows the input $D$. 

\[ D \rightarrow \text{DL} \rightarrow Q^{int} \rightarrow \text{DL} \rightarrow Q \]

Building circuits with D latches \textit{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 $Q^{\text{int}}$. 
Building circuits with D latches \textit{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 \textit{rising-edge} of C).
Edge-Triggered D Flip flop
A basic storage element

Suppose C changes periodically (called a Clock signal)
Edge-Triggered D Flip flop
A basic storage element

Suppose C changes periodically (called a Clock signal)
Edge-Triggered D Flip flop
A basic storage element

D
C
→
Q

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
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
Edge-Triggered D Flip flop
A basic storage element

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

The building block of Sequential Circuits

Data is captured only if EN is on
D Flip-flop with Write Enable

The building block of Sequential Circuits

Data is captured only if EN is on
D Flip-flop with Write Enable

The building block of Sequential Circuits

Data is captured only if EN is on

<table>
<thead>
<tr>
<th>EN</th>
<th>D</th>
<th>$Q^t$</th>
<th>$Q^{t+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>
D Flip-flop with Write Enable

The building block of Sequential Circuits

Data is captured only if EN is on

<table>
<thead>
<tr>
<th>EN</th>
<th>D</th>
<th>Q^t</th>
<th>Q^{t+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>
D Flip-flop with Write Enable

The building block of Sequential Circuits

Data is captured only if EN is on

<table>
<thead>
<tr>
<th>EN</th>
<th>D</th>
<th>Q^t</th>
<th>Q^{t+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>
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
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
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.
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.
Registers

Register: A group of flip-flops with a common clock and enable
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>inc = 0</th>
<th>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>

February 22, 2018

An example

Modulo-4 counter

<table>
<thead>
<tr>
<th>Prev State</th>
<th>Next State inc = 0</th>
<th>Next State inc = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1q0</td>
<td></td>
<td></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>

Finite State Machine (FSM) representation
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>

Finite State Machine (FSM) representation

$q_0^{t+1} = \neg{\text{inc}} \cdot q_0^t + \text{inc} \cdot \neg{q_0^t}$
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>inc = 0</td>
<td>inc = 1</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>

\[
q^{t+1} = \sim\text{inc} \cdot q^t + \text{inc} \cdot \sim q^t
= \text{inc} \oplus q^t
\]

Finite State Machine (FSM) representation

February 22, 2018
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></td>
<td></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>

Finite State Machine (FSM) representation

$$q^{t+1}_0 = \sim inc \cdot q^t_0 + inc \cdot \sim q^t_0$$

$$q^{t+1}_1 = \sim inc \cdot q^t_1 + inc \cdot \sim q^t_1 \cdot q^t_0 + inc \cdot q^t_1 \cdot \sim q^t_0$$
An example

Modulo-4 counter

<table>
<thead>
<tr>
<th>Prev State</th>
<th>NextState</th>
<th>NextState</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1q0</td>
<td>inc = 0</td>
<td>inc = 1</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>

$q_0^{t+1} = \neg \text{inc} \cdot q_0^t + \text{inc} \cdot \neg q_0^t$

$= \text{inc} \oplus q_0^t$

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

$= (\text{inc} == 1) \ ? q_0^t \oplus q_1^t : q_1^t$
Finite State Machines (FSM) and Sequential Circuits

FSMs are a mathematical object like the Boolean Algebra
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
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
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
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
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

February 22, 2018
Sequential circuits are described as modules in BSV

interface Counter;
  method Action inc;
  method Bit#(2) read;
endinterface
Sequential circuits are described as modules in BSV

- A module has internal state

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

Modulo-4 counter
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

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

Modulo-4 counter

inc → read

2
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

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

Modulo-4 counter

2

inc read

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)

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

![Diagram of a Modulo-4 counter](http://csg.csail.mit.edu/6.S084)
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++
Modulo-4 counter in BSV

interface Counter;
    method Action inc;
    method Bit#(2) read;
endinterface
Modulo-4 counter in BSV

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
Modulo-4 counter in BSV

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

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
Modulo-4 counter in BSV

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

dmodule 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
```

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

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

An action to specify how the value of the cnt is to be set
Modulo-4 counter in BSV

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

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

<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>
Modulo-4 counter

The generated circuit

```verilog
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
```

February 22, 2018

Modulo-4 counter

The generated circuit

```verilog
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
```

February 22, 2018

The generated circuit

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
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
Modulo-4 counter

The generated circuit

```verilog
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
```

February 22, 2018

Modulo-4 counter

The generated circuit

```verilog
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
```

[Diagram of the modulo-4 counter circuit]

February 22, 2018

Modulo-4 counter

The generated circuit

```verilog
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
```

Modulo-4 counter

The generated circuit

```
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):

15 6
A hardware module for computing GCD

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

\[
\begin{array}{c|c}
15 & 6 \\
9 & 6 \\
\end{array}
\]

subtraction
A hardware module for computing GCD

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

subtract

subtract
A hardware module for computing GCD

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

\[
\begin{array}{c|c|c}
15 & 6 & \\
9 & 6 & \text{subtract} \\
3 & 6 & \text{subtract} \\
6 & 3 & \text{swap}
\end{array}
\]
A hardware module for computing GCD

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

\[
\begin{array}{ll}
15 & 6 \\
9 & 6 \\
3 & 6 \\
6 & 3 \\
3 & 3 \\
\end{array}
\]

- subtract
- subtract
- swap
- subtract
A hardware module for computing GCD

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Subtract
Swap
Subtract
Subtract
A hardware module for computing GCD

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

$\begin{array}{c|c} 15 & 6 \\ 9 & 6 \\ 3 & 6 \\ 6 & 3 \\ 3 & 3 \\ 0 & 3 \end{array}$

answer

```
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.
GCD module

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

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);
module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);
rule gcd;

    endrule
method Action start(Bit#(32) a, Bit#(32) b);

    endmethod
method ActionValue#(Bit#(32)) getResult;

    endmethod
method Bool busy;

    endmethod
endmethod
endmodule
module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);
rule gcd;

endrule
method Action start(Bit#(32) a, Bit#(32) b);
    x <= a; y <= b; busy_flag <= True;
endmethod
method ActionValue#(Bit#(32)) getResult;
endmethod
method Bool busy;
endmethod
endmodule
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;
        endrule
    method Action start(Bit#(32) a, Bit#(32) b);
        x <= a; y <= b; busy_flag <= True;
    endmethod
    method ActionValue#(Bit#(32)) getResult;
        endmethod
    method Bool busy;
        endmethod
    method Bool ready;
        endmethod
endmodule
module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);
    rule gcd;

    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;
endmethod
method Bool ready;
endmethod
endmodule
module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);
rule gcd;
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;
endmethod
endmodule
module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);
rule gcd;
    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

GCD in BSV

Assume b /= 0
**GCD in BSV**

```haskell
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
```

Assume \( b \neq 0 \)
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; return busy_flag;
endmethod
method Bool ready; return (busy_flag && (x==0));
endmethod
endmodule
Rule

A module may contain rules
Rule

A module may contain rules

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

A module may contain rules

```cpp
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
Rule

A module may contain rules

```verilog
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
Rule

A module may contain rules

```vhdl
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
A module may contain rules

```verbatim
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 atomically.
Parallel Composition of Actions & Double-Writes

```plaintext
rule one;
y <= 3; x <= 5; x <= 7; endrule
```
Parallel Composition of Actions & Double-Writes

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

Double write
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
Parallel Composition of Actions & Double-Writes

**rule one;**
\[ y \leq 3; \ x \leq 5; \ x \leq 7; \ \text{endrule} \quad \text{Double write} \]

**rule two;**
\[ y \leq 3; \ \text{if (b)} \ x \leq 7; \ \text{else} \ x \leq 5; \ \text{endrule} \quad \text{No double write} \]
Parallel Composition of Actions & Double-Writes

rule one;
  y <= 3; x <= 5; x <= 7; endrule  \text{Double write}

rule two;
  y <= 3; \text{if (b)} x <= 7; \text{else} x <= 5; endrule  \text{No double write}

rule three;
  y <= 3; x <= 5; \text{if (b)} x <= 7; endrule
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 of Actions & Double-Writes

**Rule one:**
\[
y \leq 3; \ x \leq 5; \ x \leq 7; \ \text{endrule}
\]

**Rule two:**
\[
y \leq 3; \ \text{if} \ (b) \ x \leq 7; \ \text{else} \ x \leq 5; \ \text{endrule}
\]

**Rule three:**
\[
y \leq 3; \ x \leq 5; \ \text{if} \ (b) \ x \leq 7; \ \text{endrule}
\]

- Double write
- No double write
- Possibility of a double write

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