Combinational circuits

Such circuits have no cycles (feedback) or state elements
Flip flop: The basic building block of Sequential Circuits

Data is sampled at the rising edge of the clock

Flip-flops with Write Enables

Data is captured only if EN is on
Registers

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

Register file: A group of registers with a common clock, input and output port(s)

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>q1q0</td>
<td>q1q0</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>

\[ q_{1t+1} = \neg \text{inc} \cdot q_{0t} + \text{inc} \cdot \neg q_{0t} \]

\[ q_{1t+1} = \neg \text{inc} \cdot q_{1t} + \text{inc} \cdot \neg q_{1t} \cdot q_{0t} + \text{inc} \cdot q_{1t} \cdot \neg q_{0t} \]
Modulo-4 counter circuit

\[
\begin{align*}
q_0^{t+1} &= \sim\text{inc} \cdot q_0^t + \text{inc} \cdot \sim q_0^t \\
q_1^{t+1} &= \sim\text{inc} \cdot q_1^t + \text{inc} \cdot \sim q_1^t \cdot q_0^t + \text{inc} \cdot q_1^t \cdot \sim q_0^t
\end{align*}
\]

"Optimized" logic
\[
\begin{align*}
q_0^{t+1} &= \text{inc} \oplus q_0^t \\
q_1^{t+1} &= (\text{inc} == 1) \ ? \ q_0^t \oplus q_1^t : q_1^t
\end{align*}
\]

Finite State Machines (Sequential Ckts)

- A computer (in fact all digital hardware) is an FSM
- Neither State tables nor diagrams are suitable for describing very large digital designs
  - large circuits must be described in a modular fashion
    -- as a collection of cooperating FSMs
- BSV is a modern programming language to describe cooperating FSMs
  - We will give various examples of FSMs in BSV
modulo4 counter in BSV

```
module moduloCounter(Counter);
    Reg#(Bit#(2)) cnt <- mkReg(0);
    method Action inc;
        cnt <= (~cnt[1] & cnt[0] | cnt[1] & ~cnt[0]);
    endmethod
    method Bit#(2) read;
        return cnt;
    endmethod
endmodule
```

Can be replaced by `cnt+1`

Interface

- Modulo counter has the following interface, i.e., type

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

- An interface can have many different implementations
  - For example, the numbers may be represented as Gray code
FIFO Interface

```verbatim
interface Fifo #(numeric type size, type t);
  method Bool notFull;
  method Bool notEmpty;
  method Action enq(t x);
  method Action deq;
  method t first;
endinterface
```

- `enq` should be called only if `notFull` returns True;
- `deq` and `first` should be called only if `notEmpty` returns True

One-Element FIFO Implementation

```verbatim
module mkCFFifo (Fifo #(1, t));
  Reg#(t) d <- mkRegU;
  Reg#(Bool) v <- mkReg(False);
  method Bool notFull;
    return !v;
  endmethod
  method Bool notEmpty;
    return v;
  endmethod
  method Action enq(t x);
    v <= True; d <= x;
  endmethod
  method Action deq;
    v <= False;
  endmethod
  method t first;
    return d;
  endmethod
endmodule
```
Two-Element FIFO

module mkCFFifo #(Fifo#(2, t));
  Reg#(t) da <- mkRegU();
  Reg#(Bool) va <- mkReg(False);
  Reg#(t) db <- mkRegU();
  Reg#(Bool) vb <- mkReg(False);
  method Bool notFull; return !vb; endmethod
  method Bool notEmpty; return va; endmethod
  method Action enq(t x);
    if (va) begin db <= x; vb <= True; end
    else begin da <= x; va <= True; end
  endmethod
  method Action deq;
    if (vb) begin da <= db; vb <= False; end
    else begin va <= False; end
  endmethod
  method t first; return da; endmethod
endmodule

Assume, if there is only one element in the FIFO it resides in da

parallel composition of actions

no change in fifo interface

Switch

if (inQ.first.color == Red) begin
  redQ.enq(inQ.first.value); inQ.deq;
end else begin
  greenQ.enq(inQ.first.value); inQ.deq;
end

let x = inQ.first;
if (x.color == Red) redQ.enq(x.value);
else greenQ.enq(x.value);
inQ.deq;

parallel composition of actions. Effect of inQ.deq is not visible to inQ.first

The code does not test for empty inQ or full redQ or full greenQ conditions!
Switch with empty/full tests on queues

\[
\text{if (inQ.isEmpty) begin}
\text{if (inQ.first.color == Red) begin}
\text{if (redQ.isFull) begin}
\text{redQ.enq(inQ.first.value); inQ.deq;}
\text{end}
\text{end else begin}
\text{if (greenQ.isFull) begin}
\text{greenQ.enq(inQ.first.value); inQ.deq;}
\text{end}
\text{end}
\text{inQ.deq;}
\text{end}
\]

What's wrong if the \texttt{deq} is moved here?

Switch with counters

\[
\text{if (inQ.first.color == Red) begin}
\text{redQ.enq(inQ.first.value); inQ.deq;}
\text{redC <= redC+1;}
\text{end else begin}
\text{greenQ.enq(inQ.first.value); inQ.deq;}
\text{greenC <= greenC+1;}
\text{end}
\]

Ignoring full/empty conditions
Shared counters

if (inQA.first.color == Red) begin
    redQA.enq(inQA.first.value);
inQA.deq; redC <= redC+1;
end else begin
    greenQA.enq(inQA.first.value);
inQA.deq; greenC <= greenC+1;
end;
if (inQB.first.color == Red) begin
    redQB.enq(inQB.first.value);
inQB.deq; redC <= redC+1;
end else begin
    greenQB.enq(inQB.first.value);
inQB.deq; greenC <= greenC+1;
end

What is wrong with this code?

Double write error

Double-write problem

- Parallel composition is illegal if a double-write possibility exists
- If the BSV compiler cannot prove that the predicates for writes into a register or a method call are mutually exclusive, it rejects the program
Observations

- These programs are not very complex and yet it would have been tedious to express these programs in a state table or as a circuit directly.
- BSV method calls are not available in Verilog/VHDL, and thus such programs sometimes require tedious programming.
- Even the meaning of double-write errors is not standardized across tool implementations in Verilog.