Design of Digital Systems II

Sequential-Circuit Design with Verilog

Moslem Amiri, Václav Přenosil

Embedded Systems Laboratory
Faculty of Informatics, Masaryk University
Brno, Czech Republic

amiri@mail.muni.cz
prenosil@fi.muni.cz

Fall, 2014

Clocked Circuits

- Majority of Verilog-based digital design is directed to clocked, synchronous systems that use edge-triggered flip-flops
- Like combinational behavior, edge-triggered behavior is specified using `always` blocks
  - Difference between combinational and edge-triggered behavior is in sensitivity list of `always` block
  - Keyword `posedge` or `negedge` is placed in front of signal name to indicate that the statements in block should be executed only at positive (rising) or negative (falling) edge of named signal

Table 1: Behavioral Verilog for a positive-edge-triggered D flip-flop.

```
module VrposDff(CLK, D, Q);
  input CLK, D;
  output Q;
  reg Q;

  always @ (posedge CLK)
    Q <= D;
endmodule
```

Clocked Circuits

Table 2: Behavioral Verilog for a positive-edge-triggered D flip-flop with preset and clear.

```
module Vrposdffpc(CLK, PR_L, CLR_L, D, Q);
  input CLK, PR_L, CLR_L, D;
  output Q;
  reg Q;

  always @ (posedge CLK or negedge PR_L or negedge CLR_L)
    if (PR_L==0) Q<= 1;
    else if (CLR_L==0) Q <= 0;
    else Q <= D;
endmodule
```

- Tab. 2 models a positive edge-triggered D flip-flop with asynchronous active-low preset and clear inputs
  - An edge-sensitivity keyword, `negedge`, is applied to a level, asynchronous input
    - Verilog compilers are set up to recognize this particular representation of edge-triggered-plus-asynchronous behavior, and in synthesis they will pick up right flip-flop component to implement it

Table 3: Two modules for a positive-edge-triggered D ff with a QN output.

```
module VrposDffQN1(CLK, D, Q, QN);
  input CLK, D;
  output Q, QN;
  reg Q, QN;

  always @ (posedge CLK) begin
    Q <= D; QN <= !D;
  end
endmodule

module VrposDffQN2(CLK, D, Q, QN);
  input CLK, D;
  output Q, QN;
  reg Q, QN;

  always @ (posedge CLK)
    Q <= D;
  always @ (Q)
    QN <= !Q;
endmodule
```

- Tab. 3
  - A typical synthesis tool infers two separate D flip-flops from the first module—one for Q and the other for QN
  - For the second module, QN is generated from Q using an inverter

Clocked Circuits

- Always use nonblocking assignments (<=) in sequential `always` blocks
  - In programs with multiple sequential `always` blocks using *blocking* assignments (=), simulation results can vary depending on the order in which the simulator chooses to execute those blocks
  - Using *nonblocking* assignments ensures that righthand sides of all assignments are evaluated before new values are assigned to any of lefthand sides
  - This makes results independent of order in which righthand sides are evaluated

Clocked Circuits

Table 4: Clock generation within a test bench.

```
`timescale 1 ns / 1 ns
module mclkgen(MCLK);
  output MCLK;
  reg MCLK;

initial begin
  MCLK = 1;   // Start clock at 1 at time 0
end

always begin            // 10 ns clock generation
  #6 MCLK = 0;          // 6 ns HIGH
  #4 MCLK = 1;   end;   // 4 ns LOW

endmodule
```

- Tab. 4 shows generation of a 100-MHz clock with a 60% duty cycle
  - At time 0, `MCLK` is set to 1 by `initial` block
  - Then, `always` block waits 6 ns, sets `MCLK` to 0, waits 4 ns, sets `MCLK` to 1, and repeats forever
  - `` `timescale `` directive is used to set up the simulator both a default time unit and a precision of 1 ns

State-Machine Design with Verilog

- There are many possible coding styles for creating state machines in Verilog, including using no consistent style at all
  - Without discipline of a consistent coding style, it is easy to write syntactically correct code where simulator's operation, synthesized hardware's operation, and what we think the machine should be doing are all different
- In *Verilog state-machine coding style*, code is divided into three parts
  - *State memory*
    - This can be specified in behavioral form using an `always` block that is sensitive to a signal edge
    - Or it can use a structural style with explicit flip-flop instantiations
  - *Next-state (excitation) logic*
    - This is written as a combinational `always` block whose sensitivity list includes machine's current state and inputs
    - This block usually contains a `case` statement that enumerates all values of current state
  - *Output logic*
    - This is another combinational `always` block that is sensitive to current state and inputs
    - It may or may not include a `case` statement, depending on complexity of output function

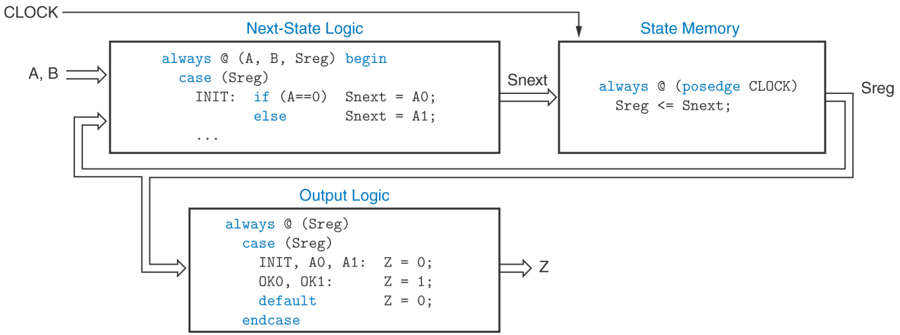


Figure 1: Moore state-machine structure implied by Verilog coding style.

- Detailed coding within each section may very
  - When there is a tight coupling of next-state and output-logic specifications, it may be desirable to combine them into a single combinational `always` block, and into a single `case` statement
  - When pipelined outputs are used, output memory could be specified along with state memory, or a separate process or structural code could be used

A Verilog State-Machine Example

- Design a clocked synchronous state machine with two inputs, `A` and `B`, and a single output `Z` that is 1 if
  - `A` had the same value at each of two previous clock ticks, or
  - `B` has been 1 since the last time that the first condition was true

  Otherwise, output should be 0
- One approach to writing a program is to construct a state and output table by hand and then manually convert it into a corresponding program

Table 5: State and output table for the example state machine.

	A B				
S	***00***	***01***	***11***	***10***	***Z***
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1
	S*				

A Verilog State-Machine Example

Table 6: Verilog program for state-machine example.

```
module VrSMex( CLOCK, A, B, Z );
  input CLOCK, A, B;
  output Z;
  reg Z;
  reg [2:0] Sreg, Snext;          // State register and next state
  parameter [2:0] INIT = 3'b000,  // Define the states
                  A0   = 3'b001,
                  A1   = 3'b010,
                  OK0  = 3'b011,
                  OK1  = 3'b100;

  always @ (posedge CLOCK)  // Create the state memory
    Sreg <= Snext;

  always @ (A, B, Sreg) begin  // Next-state logic
    case (Sreg)
      INIT:    if (A==0)  Snext = A0;
               else       Snext = A1;
      A0:      if (A==0)  Snext = OK0;
               else       Snext = A1;
      A1:      if (A==0)  Snext = A0;
               else       Snext = OK1;
      OK0:     if (A==0)  Snext = OK0;
               else if ((A==1) && (B==0)) Snext = A1;
               else                       Snext = OK1;
      OK1:     if ((A==0) && (B==0))      Snext = A0;
               else if ((A==0) && (B==1)) Snext = OK0;
               else                       Snext = OK1;
      default Snext = INIT;
    endcase
  end

  always @ (Sreg)         // Output logic
    case (Sreg)
      INIT, A0, A1: Z = 0;
      OK0, OK1:     Z = 1;
      default       Z = 0;
    endcase
endmodule
```

	A B				
S	00	01	11	10	Z
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1
	S*				

A Verilog State-Machine Example

- First `always` block (state memory creation) in Tab. 6
  - During synthesis, positive-edge-triggered D flip-flops are inferred for `Sreg`
  - A synchronous or asynchronous RESET signal is easily provided as shown in Tab. 7

Table 7: Synchronous and asynchronous reset for state machines in Verilog.

```
// State memory with active-high synchronous reset
always @ (posedge CLOCK) // Create state memory
    if (RESET==1) Sreg <= INIT; else Sreg <= Snext;

// State memory with active-high asynchronous reset
always @ (posedge CLOCK or posedge RESET) // Create state memory
    if (RESET==1) Sreg <= INIT; else Sreg <= Snext;
```

A Verilog State-Machine Example

- Second `always` block (next-state logic) in Tab. 6
  - `default` case at the end handles unused states, but not uncovered input combinations in other cases
  - In each case, an `if` statement and a final `else` is used to ensure that a value is always assigned to `Snext`
    - If there were any state/input combinations in which no value was assigned to `Snext`, Verilog compiler would infer an unwanted latch for `Snext`
  - Variations
    - To establish a default next state for machine if `case` fails to cover all state/input combinations, precede `case` statement with "`Snext = INIT`"
    - When most transitions stay in current state, `case` statement can be preceded by "`Snext = Sreg`"
    - Preceding with "`Snext = 3'bx`" is another variation which is useful in simulation to detect unspecified state/input combinations

A Verilog State-Machine Example

- Third `always` block (output logic) in Tab. 6
  - Handles machine's single Moore output, `Z`, which is set to a value as a function of current state
  - To define Mealy output, output should be a function of inputs as well as state in each enumerated case
    - Inputs should also be added to sensitivity list of `always` block
- **Pipelined outputs**
  - In design of high-speed circuits, it is often necessary to ensure that state-machine outputs are available as early as possible and do not change during each clock period
  - One way to get this behavior is to encode state so that state variables themselves serve as outputs
    - Called *output-coded state assignment*
    - It yields a Moore machine in which output logic is wires
  - Another approach is to design state machine so that outputs during one clock period depend on state and inputs during *previous* clock period
    - Called **pipelined outputs**
    - Obtained by attaching another stage of memory (flip-flops) to a machine's outputs

A Verilog State-Machine Example

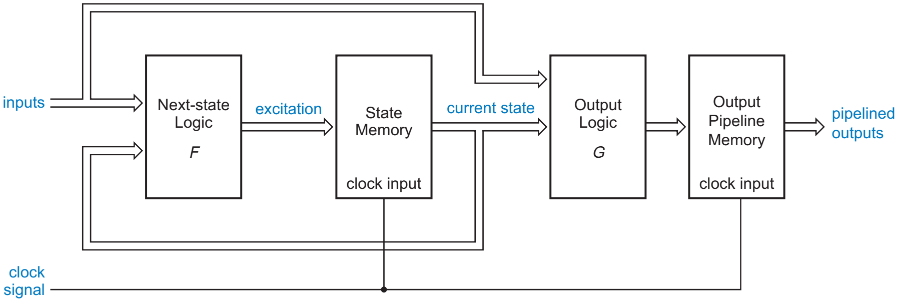


Figure 2: Mealy machine with pipelined outputs.

- In our example Verilog state machine, the module defines a Moore-type state machine with the structure shown in Fig. 1
  - We can convert the machine to have pipelined outputs with the structure shown in Fig. 3
  - To do this, we need only to declare a "next-output" variable `Zn` and replace the original Verilog state-memory and output code with code shown in Tab. 8

A Verilog State-Machine Example

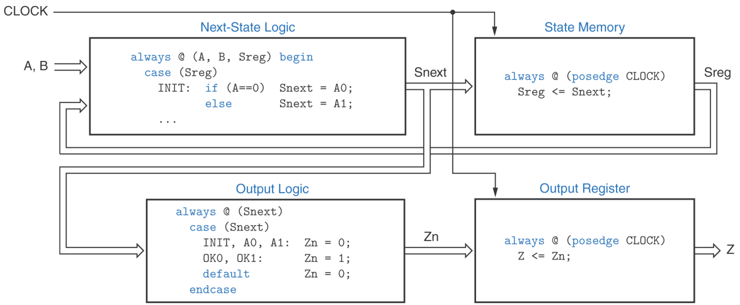
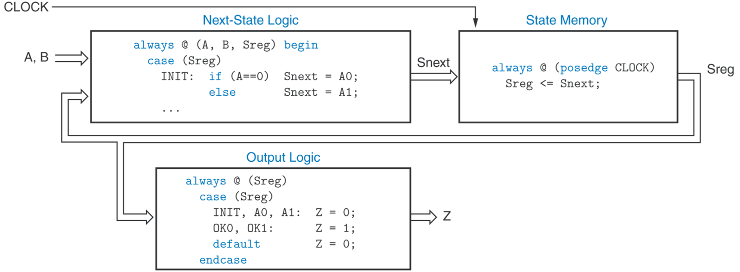


Figure 3: Verilog state machine with pipelined outputs.



A Verilog State-Machine Example

Table 8: Verilog pipelined output code.

```
always @ (posedge CLOCK)   // Create output register -- this code
  Z <= Zn; // could be combined with state-memory code if desired

always @ (Snext)           // Output logic
  case (Snext)
    INIT, A0, A1: Zn = 0;
    OK0, OK1:     Zn = 1;
    default       Zn = 0;
  endcase
```

- New machine's behavior is indistinguishable from that of original machine, except for timing
  - We have reduced propagation delay from `CLOCK` to `Z` by producing `Z` directly on a register output
  - But we have increased setup-time requirements of `A` and `B` to `CLOCK`
    - In addition to their propagation delay through next-state logic, changes in `A` and `B` must also get through output logic in time to meet setup time requirement of output flip-flop's D input

A Verilog State-Machine Example

- Direct Verilog coding without a state table
  - It is possible to write a Verilog program directly, without writing out a state table by hand
  - Design a clocked synchronous state machine with two inputs, `A` and `B`, and a single output `Z` that is 1 if
    - `A` had the same value at each of two previous clock ticks, or
    - `B` has been 1 since the last time that the first condition was true

    Otherwise, output should be 0
  - We need to have a register that keeps track of `A` (`LASTA`)
  - Three states must be defined
    - `INIT`
    - `LOOKING`: still looking for a match
    - `OK`: got a match or `B` has been 1 since last match

A Verilog State-Machine Example

Table 9: Simplified Verilog state-machine design.

```
module VrSMexa( CLOCK, RESET, A, B, Z );
  input CLOCK, RESET, A, B;
  output Z;
  wire Z;              // declared as wire for continuous assignment
  reg LASTA;                         // LASTA holds last value of A
  reg [1:0] Sreg, Snext;             // State register and next state
  parameter [1:0] INIT    = 2'b00, // Define the states
                  LOOKING = 2'b10,
                  OK      = 2'b11;

  always @ (posedge CLOCK) begin // State memory (with sync. reset)
    if (RESET==1) Sreg <= INIT; else Sreg <= Snext;
    LASTA <= A;
  end

  always @ (A, B, LASTA, Sreg) begin  // Next-state logic
    case (Sreg)
      INIT:    Snext = LOOKING;
      LOOKING: if (A==LASTA)          Snext = OK;
               else                   Snext = LOOKING;
      OK:      if (B==1 || A==LASTA) Snext = OK;
               else                   Snext = LOOKING;
      default                         Snext = INIT;
    endcase
  end

  assig  Z = (Sreg==OK) ? 1 : 0;      // Output logic
endmodule
```

- Tab. 9
  - First `always` block creates both state memory and `LASTA` register
  - Second one creates next-state logic using our simplified approach
  - `Z` output is a simple combinational decode of `OK` state, so it is created using a continuous-assignment statement

More Verilog State-Machine Examples

- Ones-counting machine
  - Design a clocked synchronous state machine with two inputs, X and Y, and one output, Z
  - Output should be 1 if the number of 1 inputs on X and Y since reset is a multiple of 4, and 0 otherwise
- We can write a Verilog module for this problem directly, without constructing a state table

Table 10: Verilog module for a ones-counting machine.

```
module Vronescnt( CLOCK, RESET, X, Y, Z );
  input CLOCK, RESET, X, Y;
  output Z;
  wire Z;             // declared as wire for continuous assignment
  reg [1:0] COUNT, NEXTCNT;     // Current and next count, modulo 4
  parameter [1:0] ZERO = 2'b00;

  always @ (posedge CLOCK)      // State memory (with sync. reset)
    if (RESET==1) COUNT <= ZERO; else COUNT <= NEXTCNT;

  always @ (COUNT, X, Y)        // Counting logic
    NEXTCNT = COUNT + {1'b0, X} + {1'b0, Y};

  assign Z = (COUNT==ZERO) ? 1 : 0;     // Output logic
endmodule
```

More Verilog State-Machine Examples

- In synthesis, counting logic in Tab. 10 does not necessarily yield a compact or speedy circuit
  - With a simple tool, it could yield two 2-bit adders connected in series
  - A good tool may be able to synthesize a more compact incrementer for each of the two additions
  - Another approach is to replace `NEXTCNT always` block with the one shown in Tab. 11
    - Formulating the choices in a `case` statement makes for a faster circuit, allowing the two adders or incrementers to operate in parallel
    - A multiplexer can be used to select one of outputs according to choices
  - To ensure that only one full adder is synthesized, we use code shown in Tab. 12

Table 11: Alternative Verilog counting logic for ones-counting machine.

```
always @ (COUNT, X, Y) case ({X, Y})    // Counting logic
  2'b01, 2'b10: NEXTCNT = COUNT + 2'b01;
  2'b11:        NEXTCNT = COUNT + 2'b10;
  default       NEXTCNT = COUNT;
endcase
```

Table 12: Fastest and smallest Verilog counting logic for ones-counting machine.

```
always @ (COUNT, X, Y) begin  // Counting logic
  XY1s[0] = X ^ Y;
  XY1s[1] = X & Y;
  NEXTCNT = COUNT + XY1s;
end
```

- Tab. 12
  - A 2-bit `reg` variable `XY1s` is declared
  - The two equations create a half adder
  - Only one full adder is synthesized

More Verilog State-Machine Examples

- Combination-lock state machine
  - Design a clocked synchronous state machine with one input, `X`, and one output, `UNLK`
    `UNLK` output should be 1 if and only if `X` is 0 and the sequence of inputs received on `X` at preceding seven clock ticks was 0110111
  - Output of the machine at any time is completely determined by its inputs during current and preceding seven clock ticks
    - Thus, we use a "finite-memory" approach to design this machine
    - With this approach, we explicitly keep track of past seven inputs and then form output as a combinational function of these and current inputs
  - Tab. 13
    - This module keeps track of last seven values of `X` using a "shift register"
    - Bits are shifted left on each clock tick
    - Next-state logic is put into the same `always` block as state memory
    - `XHISTORY` is initialized to all 1s at reset so the user does not get benefit of a "free" 0 right after reset to begin combination pattern

Table 13: Verilog module for finite-memory design of combination-lock state machine.

```
module Vrcomblock( CLOCK, RESET, X, UNLK );
  input CLOCK, RESET, X;
  output UNLK;
  wire UNLK;           // declared as wire for continuous assignment
  reg [7:1] XHISTORY; // 7-tick history of X
  parameter [7:1] COMBINATION = 7'b0110111;

  always @ (posedge CLOCK)   // State memory (with sync. reset)
    if (RESET==1) XHISTORY <= 7'b1111111;
    else XHISTORY <= {XHISTORY[6:1], X}; // next-state logic

  // Output logic -- Detect combination
  assign UNLK = (XHISTORY==COMBINATION && X==0) ? 1 : 0;
endmodule
```

References

John F. Wakerly, *Digital Design: Principles and Practices (4th Edition)*, Prentice Hall, 2005.