SystemVerilog

• IEEE 1800™ SystemVerilog is the industry's first unified hardware description and verification language (HDVL) standard.

• SystemVerilog is a major extension of the established IEEE 1364™ Verilog language.

• It was developed originally by Accellera to dramatically improve productivity in the design of large gate-count, IP-based, bus-intensive chips.

• SystemVerilog is targeted primarily at the chip implementation and verification flow, with powerful links to the system-level design flow.

• SystemVerilog has been adopted by 100's of semiconductor design companies and supported by more than 75 EDA, IP and training solutions worldwide.

• **Se : www.systemverilog.org**

Combinational Logic with Gate Models

```
module Nand3 (output wire z, input
wire w, x, y);

  assign #5ps z = ~(w & x & y);

endmodule
```

Table 3.1 Arithmetic and Logical Operators in Order of Decreasing Precedence

Arithmetic	Bitwise	Logical
+ – (unary)	~	!
* / %		
+ –		
	<< >>	
	&	
	^ ^~	
	|	
		&&
		||

- Compilation: vlog –sv +acc –work <mylib> <design>.v
  or : vlog +acc –work <mylib> <design>**.sv**

- Simulation: vsim –lib <mylib> <design>

Basic Gate Models

- Low level primities includes: and, or, nand, nor, xor, xnor, not.

```
module FullAdder1(output wire S, Co,
                  input  wire a, b, Ci);

// See Table 3.2 p. 59 in Zwolinski.

  logic na, nb, nc, d, e, f, g, h, i, j;

  not n0 (na, a);
  not n1 (nb, b);
  not n2 (nc, Ci);
  and a0 (d, na, nb, Ci);
  and a1 (e, na, b, nc);
  and a2 (f, a, b, Ci);
  and a3 (g, a, nb, nc);
  or  o0 (S, d, e, f, g);
  and a4 (h, b, Ci);
  and a5 (i, a, b);
  and a6 (j, a, Ci);
  or  o1 (Co, h, i, j);

endmodule
```

```
module FullAdder2(output wire S, Co,
                  input  wire a, b, Ci);

  logic res1, res2, res3;

  xor g0(res1, a, b);
  xor g1(S, res1, Ci);

  and g2(res2, res1, Ci);
  and g3(res3, a, b);
  or  g4(Co, res3, res2);

endmodule
```

Parameters and Delay

```
module Nand3 #(parameter delay=5ps) (output wire z, input wire w, x, y);
  assign #(delay) z = ~(w & x & y);
Endmodule

Nand3  #(8ps)  g1(a, b, c);
```

```
module Nand3 (output wire z, input wire w, x, y);
  assign #(5ps:7ps:9ps, 6ps:8ps:10ps) z = ~(w & x & y);
endmodule
```

Combinational Building Blocks: Multiplexor

```
module mux4 (output logic y,
             input logic a, b, c, d, s0, s1);

always_comb
  if (s0)
    if (s1)
       y = d;
    else
       y = c;
  else
    if (s1)
       y = b;
    else
       y = a;

endmodule
```

Decoders

```
module decoder (output logic [3:0] y,
                input logic [1:0] a);

always_comb
  case (a)
    0 : y = 1;
    1 : y = 2;
    2 : y = 4;
    3 : y = 8;
    default : y = 'x; // Not for synthesize!!
  endcase
endmodule



module decoderN #(parameter N = 3)
  (output logic [(1<<N)-1:0] y, input logic [N-1:0] a);

  always_comb
    y = 1'b1 << a;

endmodule
```

Decoders continue

```
module decoderlogN #(parameter N = 8)
  (output logic [N-1:0] y,
   input logic [clog2(N)-1:0] a);

  function int clog2(input int n);
    begin
    clog2 = 0;
    // n--;
    n = n-1;
    while (n > 0)
      begin
           // clog2++;
            clog = clog+1;
           // n >>= 1;
            n = n>>1;
      end
    end
  endfunction


  always_comb
    y = 1'b1 << a;

endmodule
```

Priority Encoder

```
module encoder (output logic [1:0] y, logic valid,
                input logic [3:0]a);

always_comb
  unique casez (a)
    4b'1??? : {y,valid} = 3'b111;
    4b'01?? : {y,valid} = 3'b101;
    4b'001? : {y,valid} = 3'b011;
    4b'0001 : {y,valid} = 3'b001;
    default : {y,valid} = 3'b000;
  endcase

endmodule
```

Table 4.2 Priority Encoder

Inputs				Outputs		
A3	A2	A1	A0	Y1	Y0	Valid
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	-	0	1	1
0	1	-	-	1	0	1
1	-	-	-	1	1	1

Ripple Adder with Generate Statement

```
module fulladder (output logic sum, cout, input logic a, b, cin);
  always_comb
    begin
      sum = a ^ b ^ cin;
      cout = a & b | a & cin | b & cin;
    end
endmodule

module ripple #(parameter N = 4)
  (output logic [N-1:0] Sum, output logic Cout,
   input logic [N-1:0] A, B, input logic Cin);

  logic [N-1:1] Ca;
  genvar i;

  fulladder f0 (Sum[0], Ca[1], A[0], B[0], Cin);

  generate for (i = 1; i < N-1; i++)
    begin : f_loop
    fulladder fi (Sum[i], Ca[i+1], A[i], B[i], Ca[i]);
    end
  endgenerate

  fulladder fN (Sum[N-1], Cout, A[N-1], B[N-1], Ca[N-1]);

endmodule
```

Ripple adder with Task sub-routine

```
module ripple_task #(parameter N = 4)
  (output logic [N-1:0] Sum, output logic Cout,
   input logic [N-1:0] A, B, input logic Cin);

  logic [N-1:1] Ca;
  genvar i;

task automatic fulladder (output logic sum, cout,
                  input logic a, b, cin);

    begin
    sum = a ^ b ^ cin;
    cout = a & b | a & cin | b & cin;
    end

endtask

always_comb
  fulladder (Sum[0], Ca[1], A[0], B[0], Cin);

  generate for (i = 1; i < N-1; i++)
    begin : f_loop
    always_comb
      fulladder (Sum[i], Ca[i+1], A[i], B[i], Ca[i]);
    end
  endgenerate

always_comb
  fulladder fN (Sum[N-1], Cout, A[N-1], B[N-1], Ca[N-1]);

endmodule
```

Parity Checker

```
module parity_loop #(parameter N = 4)
       (output logic even, input logic [N-1:0] a);

  always_comb
    begin
    even = '1;
    for (int i = 0; i < N; i++)
      even = even ^ a[i];
    end

endmodule



module parity #(parameter N = 4)
       (output logic even, input logic [N-1:0] a);

  always_comb
    even = ~^a;

endmodule
```

Multi-Valued Logic – Three-State Buffers

```
module threestate (output wire y,
                   input logic a, enable);

  assign y = enable ? a : 'z;

endmodule
```

“Don’t do this at home” ☺:

```
module threemux4 (output wire y,
            input logic a, b, c, d, s0, s1);

  assign y = (~s0 && ~s1) ? a : 'z;
  assign y = (s0 && ~s1)  ? b : 'z;
  assign y = (~s0 && s1)  ? c : 'z;
  assign y = (s0 && s1)   ? d : 'z;

endmodule
```

Basic Self-Checking Testbench

```
module TestAdder;

  logic a, b, Ci, S, Co; // Note: Changed from "wire"
  // logic answ_Co, answ_S;
  logic error;

  // FullAdder f0 (S, Co, a, b, Ci);
  FullAdder f0 (.*);  // Allowed when formal and actual parameters are exactly equal

  initial
  begin
    a = '0; b = '0; Ci = '0;
    #10ns a = '1;
    #10ns a = '0; b = '1;
    #10ns a = '1;
    #10ns a = '0; b = '0; Ci= '1;
    #10ns a = '1;
    #10ns a = '0; b = '1;
    #10ns a = '1; Ci = '0;
    #10ns Ci ='1;
    #10ns a = '0; b = '0; Ci = '0;
  end

  always @(Co, S)
  begin
    error = (a+b+Ci) != {Co,S};
  end

endmodule
```

Asynchron DFF

```
module dffr (output logic q,
             input  logic d, clk, n_reset);
  always_ff @(posedge clk, negedge n_reset) // removing “negedge n_reset”
                                            // makes synchron reset
    if (~n_reset)
      q <= '0;  // In Verilog:  q <= 1’b0;
    else
      q <= d;
endmodule


module dffrs (output logic q,
              input  logic d, clk, n_reset, n_set);

  always_ff @(posedge clk, negedge n_reset, negedge n_set)
    if (~n_set)
      q <= '1;
    else if (~n_reset)
      q <= '0;
    else
      q <= d;
endmodule
```

Multiple bit register

```
module dffn #(parameter N = 8) (output logic [N-1:0]q,
                                input  logic [N-1:0] d,
                                input  logic clk, n_reset);

always_ff @(posedge clk, negedge n_reset)
  if (~n_reset)
    q <= '0;
  else
    q <= d;
endmodule
```

Universal Shift Register

```
module usr #(parameter N = 8) (output logic [N-1:0]q,
                               input logic [N-1:0] a,
                               input logic [1:0] s,
                               input logic lin, rin, clk, n_reset);

always_ff @(posedge clk, negedge n_reset)
  if (~n_reset)
    q <= '0;
  else
    unique case (s)
      2'b11: q <= a;
      2'b10: q <= {q[N-2:0], lin};
      2'b01: q <= {rin, q[N-1:1]};
      default:;
    endcase

endmodule
```

Table 5.5 Universal Shift Register

S_1S_0	Action
00	Hold
01	Shift right
10	Shift left
11	Parallel load

Binary Counter

```
module bincounter #(parameter N = 8)
                   (output logic [N-1:0] count,
                    input logic n_reset, clk);

always_ff @(posedge clk, negedge n_reset)
  if (~n_reset)
    count <= 0;
  else
    count <= count + 1; // The counter will ”wrap round” to zero.

endmodule
```

- Notice that it is illegal to use the ”++” operator (i.e. count++) as this is used in blocking assignment (i.e. count=count++) and can cause erroneous simulated behavior!

Read-Only Memory (ROM)

```
module sevensegrom(output logic [6:0] data,
                   input  logic [3:0] address);

  logic [6:0] rom [0:15] = {7'b1110111,  //0
                            7'b0010010,  //1
                            7'b1011101,  //2
                            7'b1011011,  //3
                            7'b0111010,  //4
                            7'b1101011,  //5
                            7'b1101111,  //6
                            7'b1010010,  //7
                            7'b1111111,  //8
                            7'b1111011,  //9
                            7'b1101101,  //E 10
                            7'b1101101,  //E 11
                            7'b1101101,  //E 12
                            7'b1101101,  //E 13
                            7'b1101101,  //E 14
                            7'b1101101}; //E 15
  always_comb
    data = rom[address];

endmodule
```

Random Access Memory (RAM)

```
module RAM16x8(inout wire  [7:0] Data,
               input logic [3:0] Address,
               input logic n_CS, n_WE, n_OE);
  logic [7:0] mem [0:15];

  assign Data = (~n_CS & ~n_OE) ? mem[Address] : 'z;

  always_latch
    if (~n_CS & ~n_WE & n_OE)
      mem[Address] <= Data;
endmodule

module SyncRAM #(parameter M = 4, N = 8)
  (output logic [N-1:0] Qout,
   input  logic [M-1:0] Address,
   input  logic [N-1:0] Data,
   input  logic WE, Clk);

  logic [N-1:0] mem [0:(1 << M)-1]; // Equal to: logic [N-1:0] mem [0:(2**M)-1];

always_comb
  Qout = mem[Address];

always_ff @(posedge Clk)
  if (~WE)
    mem[Address] <= Data;
endmodule
```

Quiz: Fulladder in always statement

```
module syncfulladder(output logic S, Co,
                     input  logic n_reset, clk, a, b, Ci);

  always_ff @(posedge clk, negedge n_reset)
    if (~n_reset)
      ?? <= '0;
    else
      ?? <= ??;

endmodule
```

Quiz: Solution

```
module syncfulladder(output logic S, Co,
                     input  logic n_reset, clk, a, b, Ci);

  always_ff @(posedge clk, negedge n_reset)
    if (~n_reset)
      {Co,S} <= '0;
    else
//      {Co,S}<= {1'b0,a}+{1'b0,b}+{1'b0,Ci};
      {Co,S} <= a+b+Ci;

endmodule
```

Testbenches for Sequential Circuits: Clock Generation

```
module Clockgen;
  timeunit 1ns;         // Sets the time unit in the module
  timeprecision 100ps; // Sets the simulation precision within the module
  parameter ClockFreq_MHz = 100.0; // 100 MHz
  parameter Mark = 45.0; // Mark length %
  // Mark time in ns
  parameter ClockHigh = (Mark*10)/ClockFreq_MHz;

  // Space time in ns
  parameter ClockLow = ((100 - Mark)*10)/ClockFreq_MHz;

  logic clock;

  initial
  begin
    clock = '0;
    forever
    begin
      #ClockLow clock = '1;  // ClockLow may be set as parameter
      #ClockHigh clock = '0; // ClockHigh may be set as parameter
    end
  end

endmodule
```

Testbench Example part 1: Added FFs to Fulladder

```
module syncfulladder(output logic S, Co,
                     input logic n_reset, clk, a, b, Ci);
  logic na, nb, nc, d, e, f, g, h, i, j, S_tmp, Co_tmp;

  not n0 (na, a);
  not n1 (nb, b);
  not n2 (nc, Ci);
  and a0 (d, na, nb, Ci);
  and a1 (e, na, b, nc);
  and a2 (f, a, b, Ci);
  and a3 (g, a, nb, nc);
  or o0 (S_tmp, d, e, f, g);
  and a4 (h, b, Ci);
  and a5 (i, a, b);
  and a6 (j, a, Ci);
  or o1 (Co_tmp, h, i, j);

  always_ff @(posedge clk, negedge n_reset)
    if (~n_reset)
      begin
        S  <= '0;
        Co <= '0;
      end
    else
      begin
        S  <= S_tmp;
        Co <= Co_tmp;
      end

endmodule
```

Testbench example part 2: Clocked Testbench

```
module tb_syncfulladder;

  timeunit 1ns;
  timeprecision 100ps;

  parameter ClockFreq_MHz = 100.0; // 100 MHz
  parameter Mark = 45.0; // Mark length %
  // Mark time in ns
  parameter ClockHigh =
            (Mark*10)/ClockFreq_MHz;
  // Space time in ns
  parameter ClockLow =
            ((100 - Mark)*10)/ClockFreq_MHz;

  logic n_reset, clk, a, b, Ci, S, Co;
  logic fa_error;

  initial
    begin
      clk = '0;
      forever
      begin
        #ClockLow  clk = '1;
        #ClockHigh clk = '0;
      end
    end
```

```
initial
  begin
    a = '0; b = '0; Ci = '0;

    n_reset = '1;
    #5ns  n_reset = '0;
    #5ns  n_reset = '1;

    #10ns a = '1;
    #10ns a = '0; b = '1;
    #10ns a = '1;
    #10ns a = '0; b = '0; Ci= '1;
    #10ns a = '1;
    #10ns a = '0; b = '1;
    #10ns a = '1;
  end

syncfulladder f0 (.*);
// syncfulladder f0 (.*,
//                   .clk(mclk)); // If mclk is
//                                   the testbench clock


always @(Co,S)
  begin
    fa_error = {Co,S} != {a}+{b}+{Ci};
    $display("%t %b+%b+%b= Sum=%b Carry=%b \t Error=%b",
             $time, a, b, Ci, S, Co, fa_error);
  end

Endmodule
```

Format Specifiers and Special Characters

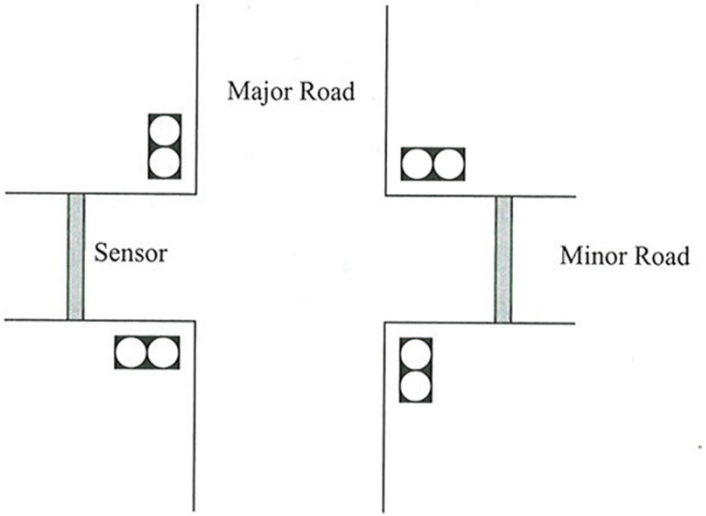
Table 5.7 Format Specifiers

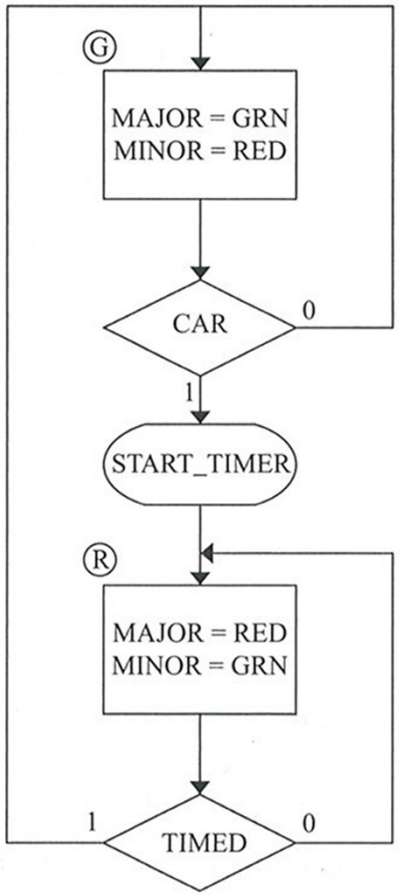
Specifier	Meaning
%h	Hexadecimal format
%d	Decimal format
%o	Octal format
%b	Binary format
%c	ASCII character format
%v	Net signal strength
%m	Hierarchical name of current scope
%s	String
%t	Time
%e	Real in exponential format
%f	Real in decimal format
%g	Real in exponential or decimal format

Table 5.8 Special Characters

Symbol	Meaning
\n	New line
\t	Tab
\\	\character
\"	" character
\xyz	Where xyz are octal digits—the character given by that octal code
%%	% character

State Machines in SystemVerilog: Traffic Controller





State Machines in SystemVerilog; implementation

```
module traffic_2(output logic start_timer,
                              major_green,
                              minor_green,
                 input  logic clock,
                              n_reset,
                              timed,
                              car);


 enum {G, R} present_state, next_state;


 always_ff @(posedge clock, negedge n_reset)
  begin: SEQ
    if (~n_reset)
      present_state <= G;
    else
      present_state <= next_state;
  end: SEQ
```

```
always_comb
begin: COM
  start_timer = '0;
  minor_green = '0;
  major_green = '0;
  next_state = present_state;

  case (present_state)
    G: begin
       major_green = '1;
       if (car)
         begin
         start_timer = '1;
          next_state = R;
         end
       end
    R: begin
       minor_green = 1'b1;
       if (timed)
         next_state = G;
       end
  endcase
end: COM

endmodule
```

State Machines in SystemVerilog; impl. cont.

```
always_comb
begin: COM
  start_timer = '0;
  minor_green = '0;
  major_green = '0;
  next_state = present_state;

  case (present_state)
    G: begin
       major_green = '1;
       if (car && timed)
       // if (car == 1'b1 && timed == 1'b1)
         begin
         start_timer = '1;
          next_state = R;
         end
       end
    R: begin
       minor_green = 1'b1;
       if (timed)
         start_timer = '1;
         next_state = G;
       end
  endcase
end: COM
```

A Simple Microprocessor.

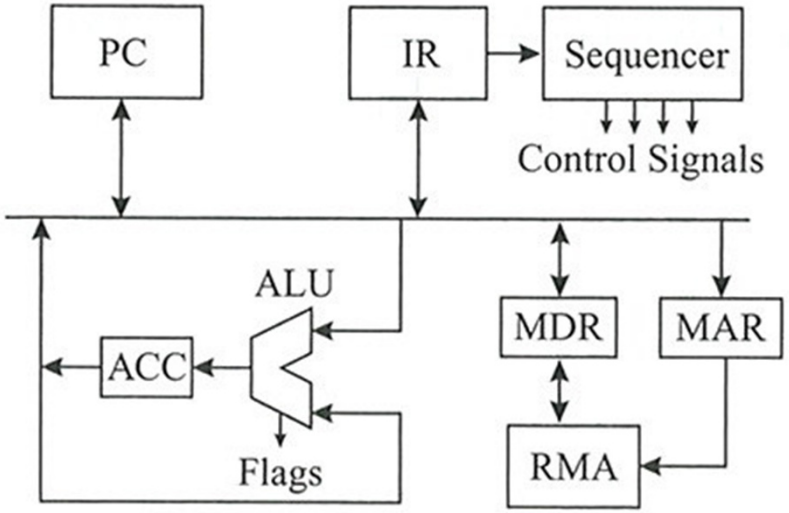
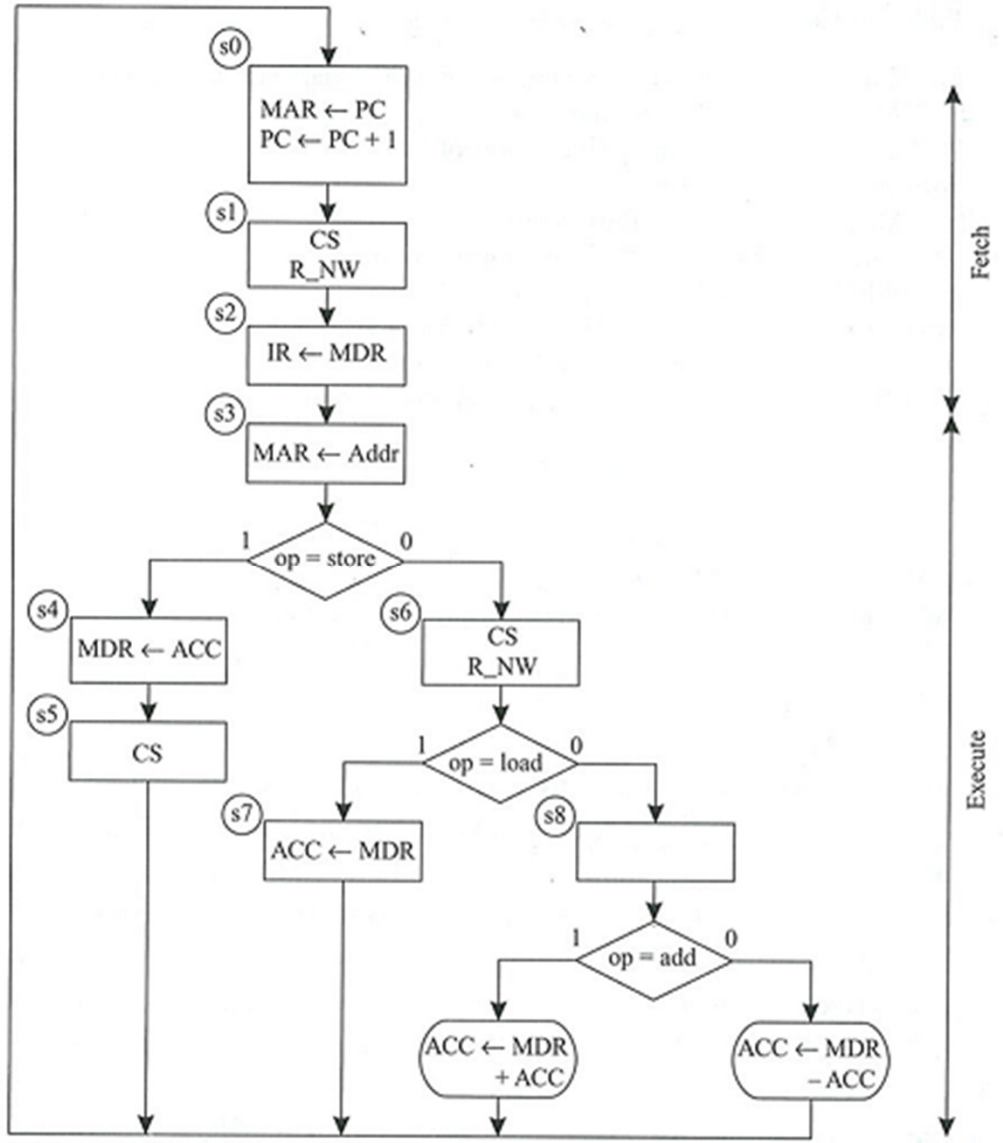


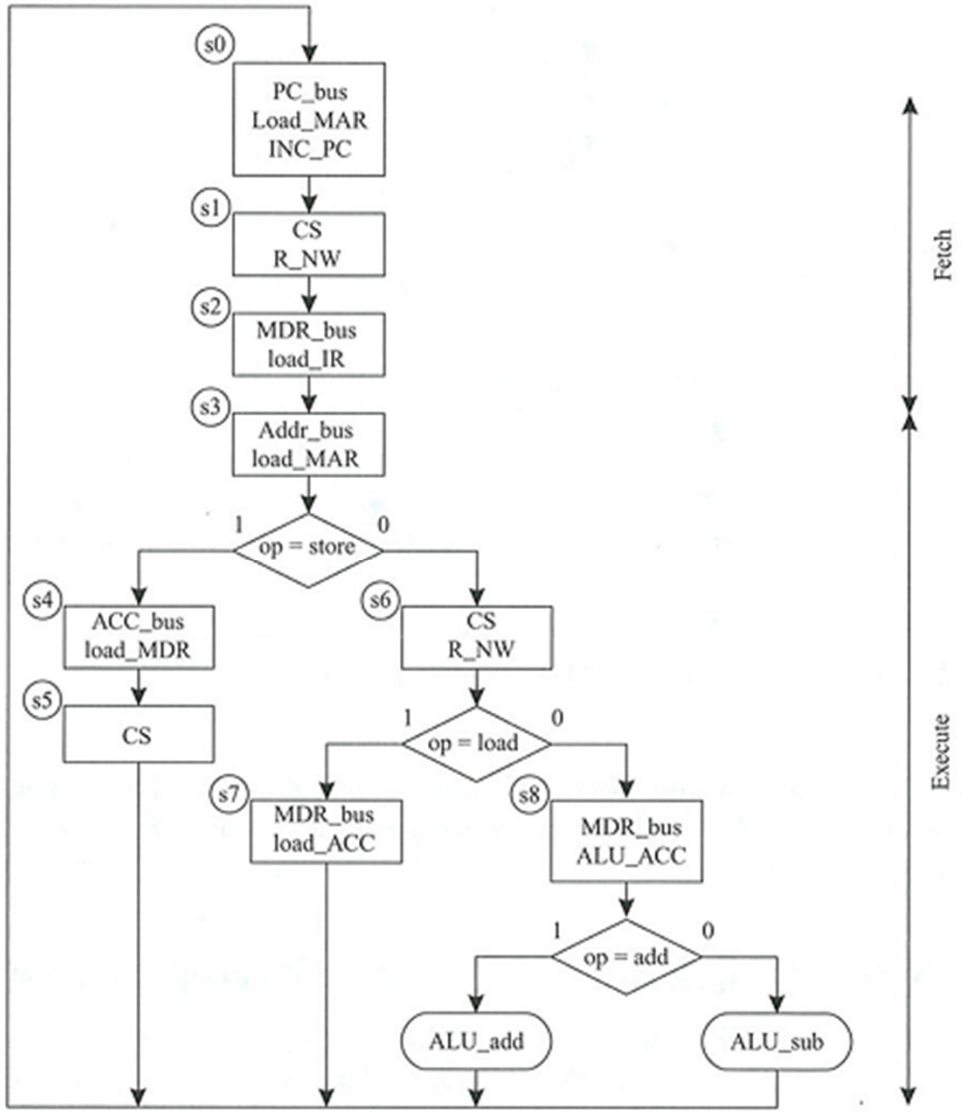
Table 7.1 Control Signals of a Microprocessor

ACC_bus	Drive bus with contents of ACC (enable three-state output)
load_ACC	Load ACC from bus
PC_bus	Drive bus with contents of PC
load_IR	Load IR from bus
load_MAR	Load MAR from bus
MDR_bus	Drive bus with contents of MDR
load_MDR	Load MDR from bus
ALU_ACC	Load ACC with result from ALU
INC_PC	Increment PC and save the result in PC
Addr_bus	Drive bus with operand part of instruction held in IR
CS	Chip select Use contents of MAR to set up memory address
R_NW	Read, not write When false, contents of MDR are stored in memory
ALU_add	Perform an add operation in the ALU
ALU_sub	Perform a subtract operation in the ALU

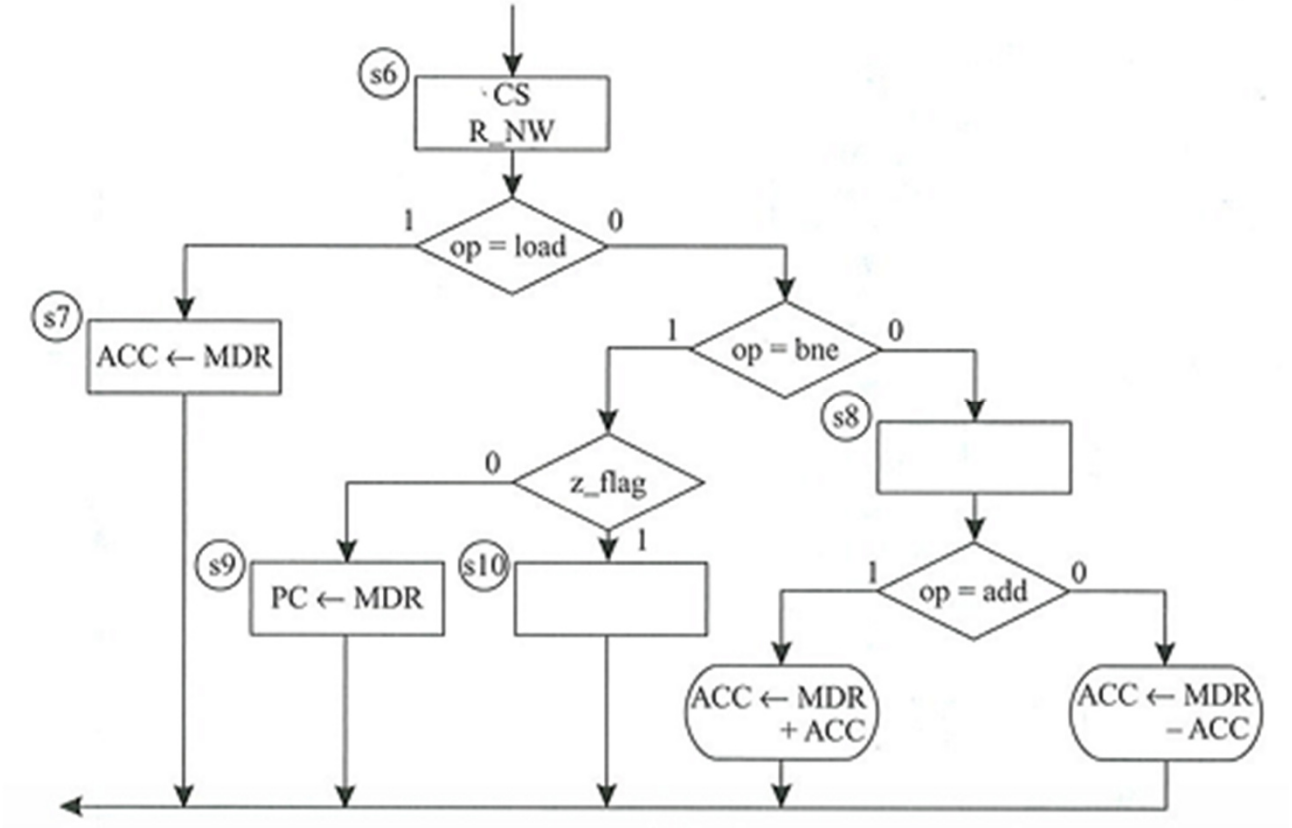
Miroprocessor ASM chart; Register Transfer Operations



Miroprocessor ASM chart; Control Signals View



Microprocessor ASM chart; including branching



Simple Microprocessor in SystemVerilog; CPU definition package

```
package cpu_defs;

  parameter WORD_W = 8;
  parameter OP_W = 3;

  enum logic[2:0] {LOAD=3'b000,
                   STORE=3'b001,
                   ADD=3'b010,
                   SUB=3'b011,
                   BNE=3'b100} opcodes;

endpackage
```

CPU module interfaces

```
import cpu_defs::*;

interface CPU_bus (input logic clock, n_reset,
                   inout wire [WORD_W-1:0] sysbus);

logic ACC_bus, load_ACC, PC_bus, load_PC,
      load_IR,  oad_MAR, MDR_bus, load_MDR, ALU_ACC,
      ALU_add,  ALU_sub, INC_PC, Addr_bus, CS, R_NW, z_flag;

logic [OP_W-1:0] op;

modport IR_port(input clock, n_reset, Addr_bus, load_IR,
                inout sysbus,
                output op);

modport RAM_port (input clock, n_reset, MDR_bus, load_MDR,
                  load_MAR, CS, R_NW,
                  inout sysbus);

modport ALU_port (input clock, n_reset, ACC_bus,
                  load_ACC, ALU_ACC,
                  ALU_add, ALU_sub,
                  inout sysbus,
                  output z_flag);

modport PC_port (input clock, n_reset, PC_bus,
                 load_PC, INC_PC,
                 inout sysbus);

modport seq_port (input clock, n_reset, z_flag,
                  input op,
                  output ACC_bus, load_ACC,
                  PC_bus, load_PC, load_IR,
                  load_MAR, MDR_bus,
                  load_MDR, ALU_ACC,
                  ALU_add, ALU_sub,
                  INC_PC, Addr_bus, CS,
                  R_NW);

endinterface
```

CPU modules ALU and PC

```
import cpu_defs::*;

module ALU (CPU_bus.ALU_port bus);

logic [WORD_W-1:0] acc;

assign bus.sysbus =  bus.ACC_bus ? acc : 'z;
assign bus.z_flag =  acc == 0 ? '1 : '0;

always_ff @(posedge bus.clock, negedge bus.n_reset)
  begin
  if (!bus.n_reset )
    acc <= 0;
  else
    if (bus.load_ACC)
      if (bus.ALU_ACC)
        begin
        if (bus.ALU_add)
          acc <= acc + bus.sysbus;
        else if (bus.ALU_sub)
          acc <= acc - bus.sysbus;
        end
      else
        acc <= bus.sysbus;
  end
endmodule
```

```
import cpu_defs::*;

module PC (CPU_bus.PC_port bus);

logic [WORD_W-OP_W-1:0] count;

assign bus.sysbus = bus.PC_bus ?
       {{OP_W{1'b0}},count} : 'z;

always_ff @(posedge bus.clock, negedge bus.n_reset)
  begin
  if (!bus.n_reset)
    count <= 0;
  else
    if (bus.load_PC)
      if (bus.INC_PC)
        count <= count + 1;
      else
        count <= bus.sysbus;
  end
endmodule
```

Top level CPU module

```
import cpu_defs::*;

module CPU (input logic clock, n_reset,
                    inout wire [WORD_W-1:0] sysbus);

  CPU_bus bus (.*);

  sequencer s1  (.*);

  IR i1  (.*);

  PC p1 (.*);

  ALU a1 (.*);

  RAM r1 (.*);

endmodule
```

Execution Order

From the pseudo-code and this description, it can be seen that the list of active events is one of the lists that has been created during some previous simulation cycle, together with any (active) events that are generated during the current cycle.

Events may be processed from the active event list in any order (or to think of it another way, events can be added to the event lists in any order). This is the fundamental cause of non-determinism in SystemVerilog.[1] We can be sure of only two things:

1. Statements between **`begin`** and **`end`** will be executed in the order stated.
2. Nonblocking assignments will be performed after blocking assignments.

---

1. VHDL experts may be looking for *delta delays*. There is no such thing in SystemVerilog. The existence of the delta delay means that a VHDL simulation is deterministic and repeatable between simulators. The absence of delta delays in SystemVerilog means that simulations may not be deterministic and may not be repeatable between different simulators.

Races Examples:

Example 1:

```
assign p = q;

initial
begin
  q = 1;
  #1 q= 0;
  $display(p);
end
```

Example 3:

```
always @(posedge clock)
  b = c;

always @(posedge clock)
  a = b;
```

Example 2:

```
module Race_example2;

  // timeunit 1ns;
     // 100ps default in Modelsim
  // timeprecision 10ps;

  logic clock;

  initial clock = '0;

  always
  begin
    // #10ps clock = ~ clock;
    #10 clock = ~ clock;
  end

// initial
//   begin
//    clock = '0;
//    forever
//      begin
//      #10ps clock = '1;
//      #10ps clock = '0;
//      end
//    end

endmodule
```