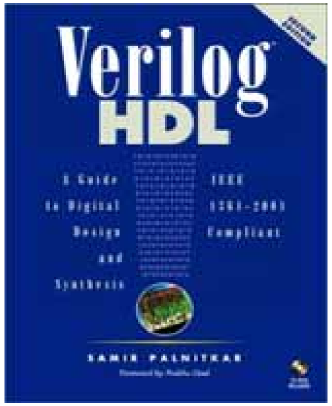
國立勤益科技大學
National Chin-Yi University of Technology

Introduction to VLSI Circuits and Systems
積體電路概論

Chapter 10

System Specifications Using Verilog HDL



Reference textbook: “*Verilog HDL- A Guide to Digital Design and Synthesis*”, *Samir Palnitkar*

賴秉樑
Dept. of Electronic Engineering
National Chin-Yi University of Technology
Fall 2007

Outline

- ❐ Introduction
- ❐ Language Structure
- ❐ Modeling Part 1
- ❐ Modeling Part 2

Hardware Description Language

- Better be standard than be proprietary
- Can describe a design at some levels of abstraction
- Can be interpreted at many level of abstraction
  - Cross functional, statistical behavioral, multi-cycles behavioral, RTL
- Can be used to document the complete system design tasks
  - Testing, simulation, ..., related activities
- User define types, functions and packages
- Comprehensive and easy to learn

Verilog vs. VHDL

- Simulation Revenue (1995 first three quarters)
  - Verilog: 51.84 millions (66.83: 1994)
  - VHDL: 26.79 millions (59.23: 1994)
- ASIC Synthesis Revenue
  - 97 millions on 1995 first three quarters
  - 79.6 millions on 1994 first three quarters

HDL in Japan

- Around 15% of digital CMOS designers using HDL
- In ASIC market, 10-20% uses HDL and Synthesis
  - Kanji Culture
- Two Years Behind USA
- A Verilog Stronghold

Design Methodologies

- Top-Down Design
  - Start with system specification
  - Decompose into subsystems, components, until indivisible
  - Realize the components
- Bottom-up Design
  - Start with available building blocks
  - Interconnect building blocks into subsystems, then system
  - Achieve a system with desired specification
- Meet in the middle
  - A combination of both

Verilog History

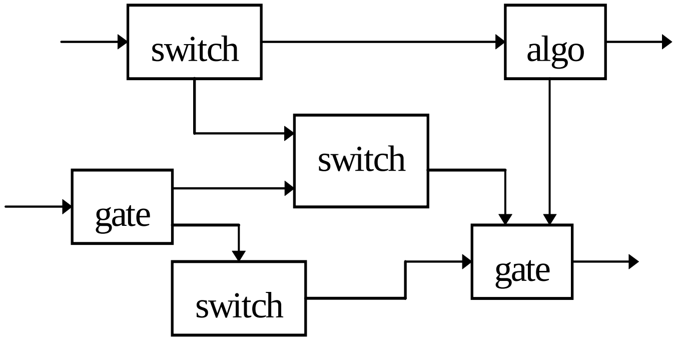
- Gateway Design Automation
  - Phil Moorbr in 1984 and 1985
- Verilog-XL, “XL algorithm”, 1986
  - Gate-level simulation
- Verilog logic synthesizer, Synopsis, 1988
  - Top-down design methodology
- Cadence Design Systems acquired Gateway
  - December 1989
  - a proprietary HDL

Verilog History

- Open Verilog International (OVI), 1991
  - Language Reference Manual (LRM)
- The IEEE 1364 working group, 1994
- Verilog become an IEEE standard (1364-1995)
  - December, 1995
- 2001, IEEE standard 1364-2001

What is Verilog HDL?

- Hardware description language
- Mixed level modeling
  - Behavioral
    - Algorithmic
    - Register transfer
  - Structural
    - Gate
    - Switch
- Single language for design and simulation
- Built-in primitives, logic function
- User-defined primitives
- Built-in data types
- High-level programming constructs



Basic unit (module)

- Module communication externally with input, output and bidirectional ports

- A module can be instantiated in another module

```
Module module_name(port_list);
    declarations:
        reg, wire, parameter, input, output, inout,
        function, task
    statements:
        initial block
        always block
        module instantiation
        gate instantiation
        UDP instantiation
        continuous assignment
endmodule
```

Behavioral Modeling

- Procedural blocks
  - **initial block:** executes only once
  - **always block:** executes in a loop
- Block execution is triggered based on user-specified conditional
  - **always** @ (**posedge** clk)
- All procedural blocks are automatically set at time 0 and execute concurrently
- **reg** is the main data type that is manipulated within a procedural block
  - It holds its value until assigned a new value

An Example

```
Module FA_SEQ(A, B, CIN, SUM, COUT);
    input A, B, CIN;
    output SUM, COUT;
    reg SUM, COUT;
    reg T1, T2, T3;

    always @ (A or B or CIN) /* at any time (A or B or CIN) changes */
        begin /* when (A or B or CIN) changes, the block executes again */
        // sequential execution
        SUM = (A ^ B) ^ CIN;
        T1 = A & CIN;
        T2 = B & CIN;
        T3 = A & B;
        COUT = (T1 | T2) | T3;
        // SUM, T1, T2, T3, COUT must be register data-type
        end
endmodule
```

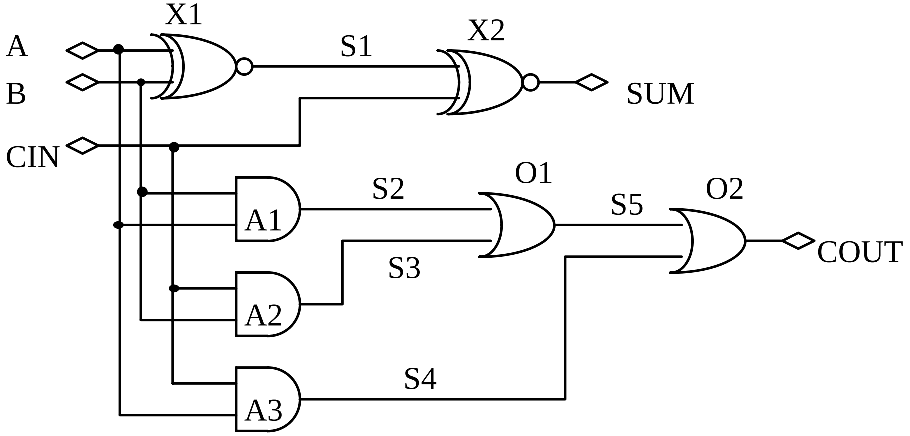
Structural Modeling

- ❒ Gate-level design
  - » Built-in set of gate primitives interconnected by nets

    and, nand, nor, ...
- ❒ Switch-level design
  - » Built-in switch-level primitives interconnected by nets nmos, ...
- ❒ Nets are continuously driven (like wires)

An Example

```
module FA_STR(A, B, CIN, SUM, COUT);
    input A, B, CIN;
    output SUM, COUT;
    wire S1, S2, S3, S4, S5;
  xor /* gate */
    X1(S1, A, B),
    X2(SUM, S1, CIN);
// X1, X2 are gate instantiation
  and
    A1(S2, A, B),
    A2(S3, B, CIN),
    A3(S4, A, CIN);
  or
    O1(S5, S2, S3),
    O2(COUT, S4, S5);
endmodule
```



Mixed Styles Modeling

- Structural and behavioral modelings can be freely mixed
- Values produced by procedural blocks can drive gates and switches
- Values from gates and switches can in turn be used with procedural blocks

An Example

```
module FA_MIX(A, B, CIN, SUM, COUT);
    input A, B, CIN;
    output SUM, COUT;
    reg COUT;
    reg T1, T2, T3;

    wire S1;

    xor X1(S1, A, B); // gate instantiation

    always @ (A or B or CIN) // Always block
    begin
        T1 = A & CIN;
        T2 = B & CIN;
        T3 = A & B;
        COUT = (T1 | T2 | T3);
    end
    assign SUM = S1 ^ CIN; // Continuous assignment
endmodule
```

Description Styles

- Structural styles
  - Gate level
  - Structural Hierarchy
- Data Flow
- Behavioral styles
  - Register Transfer Level
- Mixed

Gate Level Description Style

- Supports the following Verilog "gate type"

  and, **nand**, **or**, **nor**,

  xor, **xnor**, **not**

- The output or bidirectional terminals always come first in the terminal list, followed by the input terminals

  For example:

  nand N1 (Out1, In1, In2, In3, In4); // 4-input NAND

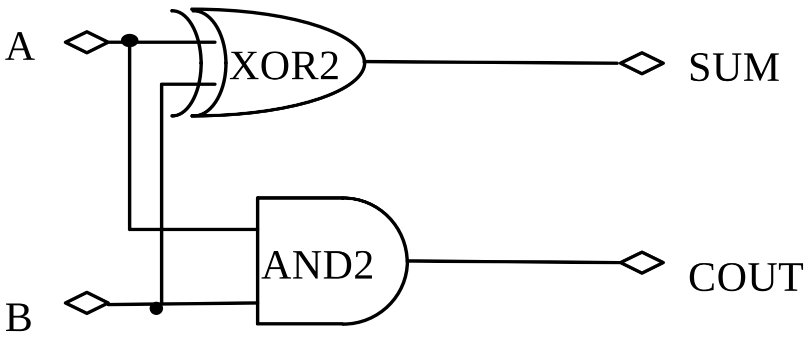
  xor X1 (Out2, In5, In6); // 2-input XOR

- In the lowest level of modeling and so the real benefits of using a high level synthesis tool are not been exploited

Gate Level Description Style

- Example 1: Half Adder

```
module HA (SUM, COUT, A, B);
    input A, B;
    output SUM, COUT;
xor XOR2 (SUM, A, B);
and AND2 (COUT, A, B);
endmodule
```



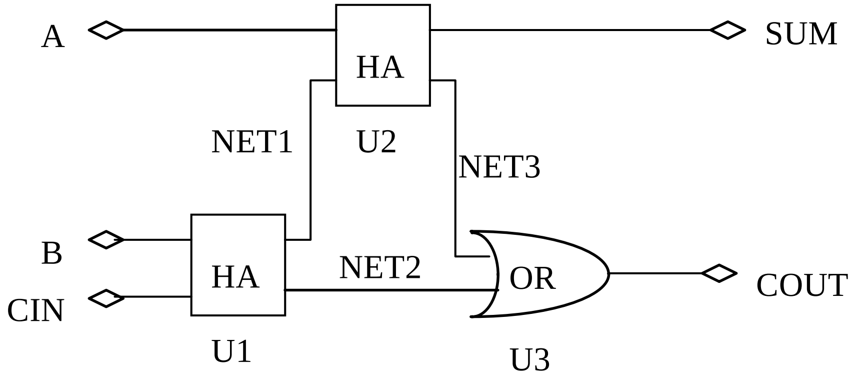
Structural Hierarchy Description Style

- Direct instantiation and connection of models from a separate calling model to form structural hierarchy in a design
  - Gate level
- A module may be declared anywhere in a design relative to where it is called
- Signals in the higher “calling” model are connected to signals in the lower “called” model by
  - named association
  - positional association

Structural Hierarchy Description Style

- Example 2: Full Adder

```
module FULL_ADD(A, B, CIN, SUM, COUT);
    input A, B;
    output SUM, COUT;
   wire NET1, NET2, NET3;
   HA U1 (NET1, NET2, B, CIN),
      U2 (SUM, NET3, A, NET1);
   OR2 U3 (COUT, NET2, NET3);
endmodule
```



Structural Hierarchy Description Style

- Example 3: Decoder

```
module NAND3 (A, B, C, Y);
     input A, B, C;
     output Y;
   assign Y = !(A & B & C);
endmodule
module INV (A, Y);
     input A;
     output Y;
   assign Y = !A;
endmodule
```

```
module DECODER2X4 (Y, A, B, ENABLE);
     input A, B, ENABLE;
     output [3:0] Y;
   // N0 and N1 use positional associations
   NAND3  N0 (ABAR, BBAR, ENABLE, Y[0]);
          N1 (ABAR, B, ENABLE, Y[1]);
   // N2 and N3 use named associations
          N2 (.C(ENABLE),.Y(Y[2]),.B(BBAR),.A(A)),
          N3 (.B(B),.Y(Y[3]),.A(A),.C(ENABLE));
   INV   I0 (A,ABAR), // positional
         I1 (.Y(BBAR),.A(B)); // named
endmodule
```

Data Flow Description Style

- Model combinational logic only
- Continuous signal assignments of type “wire” defined using the “assign”statement
- Continuous assignment statements are concurrent
- Right-hand side can be any function expression
- Left-hand side (target) can be a:
  - part-select
  - bit-select
  - concatenation of both
- Specified drive strengths and delays in an expression have no meaning to synthesis and are ignored

Data Flow Description Style

- Statements can take two forms:
  - Explicit
  - Specified in the declaration of the target wire

- Explicit

```
// declarations first
input [3:0] A, B;
input CIN;
output [3:0] SUM;
output COUT;
// theassignment
assign {COUT, SUM} = A + B + CIN;
```

- Specified in a wire declaration

```
// declaration and assignment combined
wire [3:0] parity = A ^ B;
```

Data Flow Description Style

- Example 4: Half Adder

```
module HALF_ADD(SUM, COUT, A, B);
      input A, B;
      output SUM, COUT;
    assign SUM = (A ^ B);
    assign COUT = (A & B);
endmodule
```

RTL Description Style

- Using the “always” block
- Can represent combinational logic, synchronous logic, or both
- Statements within a “always” block are sequential; their order of appearance is important
- Style of representation is very similar to C

- Example 5: Full Adder

```
module FULL_ADD (SUM, COUT, A, B, CIN);
     input A, B, CIN;
     output COUT, SUM;
     reg COUT, SUM;
   always @ (A or B or CIN)
   begin
      integer T1, T2, T3;
     SUM = (A ^ B) ^ CIN;
     T1 = A & CIN;
     T2 = B & CIN;
     T3 = A & B;
     COUT = (T1 | T2) | T3;
   end
endmodule
```

Mixed Description Style

- Example 6: Full Adder

```
module FA_MIX (SUM, COUT, A, B, CIN);
    input A, B, CIN;
    output SUM, COUT;
    reg COUT;
    wire S1;
    XOR X1 (S1, A, B);
    always @ (A or B or CIN)
    begin
        integer T1, T2, T3;
        T1 = A & B;
        T2 = A & CIN;
        T3 = B & CIN;
        COUT = ( T1 | T2) | T3;
    end
    assign SUM = S1 ^ CIN;
endmodule
```

Simulation

- One language for design, stimulus, control, saving responses, and verification
  - Stimulus and control
    - Use **initial** procedural block
  - Saving responses
    - save on change
    - stored data
  - Verification
    - automatic compares with expected responses

A Test Bench

```
module TOP;
   reg PA, PB, PCI;
   wire PCO, PSUM;
   // Instantiate module under test;
   FA_SEQ F1 (PA, PB, PCI, PSUM, PCO);
   // Positional.

   initial
      begin: ONLY_ONCE //block label
        reg [2:0] PAL; //3 bit vector
        for (PAL=0;PAL<8;PAL=PAL+1)
        begin
             {PA, PB, PCI} = PAL; //PA: MSB, PCI:LSM,
                 #5 $display (“PA,PB,PCI=%b%b%b”, PA, PB,
                     PCI, “ :::PCO,PSUM=%b%b”, PCO,PSUM);
        end
        $stop;
      end
endmodule
```

System-defined task

statements executed sequentially

Delay 5ns after previous statement executed

Another Example

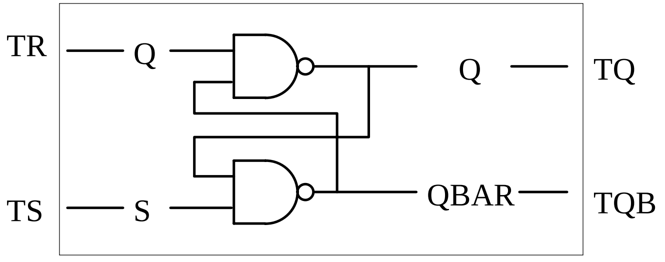
```
module RS_FF (Q, QBAR, R, S);
   output Q, QBAR;
   input R, S;

   nand #1 (Q, R, QBAR);
   nand #1 (QBAR, S, Q);
endmodule
module test;
   reg TS, TR;

   //Instantiate module under test;
   RS_FF INST_A (.Q(TQ), .QBAR(TQB), .S(TS), .R(TR));
   // Using name association.

   // Apply stimulus
   initial begin
      TR = 0; TS = 0;
      #5 TS = 1;
      #5 TS = 0; TR = 1;
```

1 time unit (ns) delay

not defined

implicitly defined, one

Connect TQ to Q



```
      #5 TS = 1; TR = 0;
      #5 TS = 0;
      #5 TR = 1;
      #5 $stop;
    end

    // Display output;
    initial
      $monitor ("At time %t," $time, "TR=%b, TS=%b, TQ=%b, TQB=%b",
                    TR, TS, TQ, TQB);
endmodule
```

System-defined task

Outline

- ❒ Introduction
- ❒ Language Structure
- ❒ Modeling Part 1
- ❒ Modeling Part 2

Language Elements

Basics

- Free format
- Case sensitive
- white space (blank, tab, newline) can be used freely
- Identifiers: sequence of letters, $ and _(underscore). First has to be a letter or an _

  Symbol
  symbol
  _R2
  R12_3$

- Escaped identifiers: starts with a \ (backslash) and end with white space

  \7400
  \.*.$
  .{*}
  \~Q

- Keywords: Cannot be used as identifiers

  E.g. **initial**, **assign**, **module**

Basics (Contd)

- Comments: Two forms

```
/* First form: cam
     extend over
          many lines */
// Second form: ends at the end of this line
```

- $SystemTask / $SystemFunction

  $time

  $monitor

- Compiler-directiuve: directive remains in effect through the rest of compilation.

```
// Text substitution

’define MAX_BUS_SIZE  32
……
reg[’MAX_BUS_SIZE-1:0] ADDRESS;
```

Integer Numbers

- Integers: Decimal, hexadecimal, octal, binary
- Simple decimal form:

```
32              decimal 32
-15             decimal -15
```

  - » Signed integers
  - » Negative numbers are in two’s complement form
- Base format form:

[<*size*>] ’<*base*><*value*>

```
’hAF (h, A, F are case insensitive)            // 8-bit hex
’o721                                // 9-bit octal
5’O37                                // 5-bit octal
4’D2                                 // 4-bit decimal
4’B1x02                              // 4-bit binary
7’hx (x is case insensitive)         // 7-bit x (x enteded)
4’hz                                 // 4-bit z (z extended)
```

- Unsigned integers
- Padding:

  ```
  10’b10     // padded with 0’s
  10’bx10    // padded with x’s
  ```
- ? can replace z in a number: used to enhance readability where z is a high impedence

- _(underscore) can be used anywhere to enhance readability, except as the first charater

- Example:

  ```
  8’d-6      // illegal
  -8’d6      // -6 held in 8 bits
  ```

Real Numbers

- Decimal notation

```
10.5
1.41421
0.01
```

- Scientific notation

```
23_5.1e2 (e is case insensitive)   // 23510.0
3.6E2                              // 360.0
5E-4                               // 0.0005
```

- Must have at least one digit on either side of decimal

- Stored and manipulated in double precision (usually 64 bits)

Strings

- "Sequence of characters"
- \n, \t, \\, \", %%

  \n = newline
  \t = tab
  \\ = backslash
  \" = quote mark (")
  %% = % sign

Value Set

- 0: logic-0 / FALSE
- 1: logic-1 / TRUE
- x: unknown / don't care, can be 0, 1, or z
- z: high-impedence

Data Types

- Nets
  - Connects between structural elements
  - Value is that of its drivers such as a continuous assignment or a gate
  - If no drivers connected to net, default value is z

```
wire [MSB:LSB] DAT; // vector wire
wire RDY, START;
wire [2:0] ADDR;
```

- Registers
  - Represent abstract data storage elements
  - Manipulated within procedural blocks
  - Value is saved from one assignment to the next
  - Default value is x

```
reg A, B, C;        // 1-bit scalars
reg [-3:3] STRANGE;
reg [0:7] QBUS;
```

Net Types

- **wire, tri:** standard net
- **wor, trior:** wired-or net
- **wand, triand:** wired-and net
- **trireg:** capacitive (if all drivers at z, previous value is retained)
- **tri1:** pull up (if no driver, 1)
- **tri0:** pull down (if no driver, 0)
- **supply0:** ground
- **supply1:** power
- A net that is not declared defaults to a 1-bit wire

```
wire reset;
wor [7:0] DBUS;
supply0 GND;
wire [3:2] CLA, PLA;
tri [0:15] ABUS;
tri0 [-3:3] GND_BUS;
```

Register Types

- **reg:** any size, unsigned
- **integer:** 32-bit signed (2's complement)
- **time:** 64-bit unsigned
- **real, realtime:** 64-bit real number, defaults to an initial value of 0
- Examples:

```
reg CNT;
reg [31:0] WORD;

integer A, B, C;        // 32-bit
integer HIST[2:6];

real [3:2] f, d;
realtime CURR_TIME;

time mark [0:15];
```

Parameters

- Constants
- Can be modified at compilation time (using **defparam** statement or in the module instance)
- Examples:

  parameter LINE_LENGTH = 132, X_S = 16’bx;

  parameter BIT =1, BYTE = 8, PI = 3.14;

  parameter MID = (BYTE + BIT) / 2;

  parameter TEST_FILE=“/home/bhasker/TEST/testfile”;
- Common uses: to specify delays and widths

Memories

- Array of integers
- No multiple dimensions

  reg [0:3] MY_MEM [0:63]; // 64 s-bit registers

- Entire memory cannot be assigned a value in a single assignment

  reg [1:3] A; // 3-bit register, A=000

  reg B [1:3]; // 3 1-bit registers, {B[1], B[2], B[3]}=000

- Can load memory by using a system mask:

  $readmem<base>(“<filename>”,<memory_name>,<start>,<finish>);

  <base> is b or h

Gate-Level Modeling

Primitive Gates

- Built-in
- **and, nand, or, nor, xor, xnor**
  - » First terminal is output, followed by inputs

    wire a1 (out1, in1, in2);

    nand a2 (out2, in21, in22, in23, in24);

- **buf, not**
  - » One or more outputs first, followed by one input

    not N1 (no1, no2, no3, no4, nin);

    buf B1 (bo, bin);

- **bufif0, bufif1, notif0, notif1:** tree-state drivers
  - » Output terminal first, then input, then control

    bufif1 BF1 (outb, inb, ctrlb);

Primitive Gates

- **pullup, pulldown**
  - Put 1 or 0 on all terminals

    pullup PUP (pu1, pu2, pu3);
- Instance names are optional

  not (XBAR, X);

Delays

- Signal propagation delay from any gate input to the gate output
- Up tp three values per output: rise, fall, turn-off delay

```
not N1 (XBAR, X);                  // Zero delay
nand #(6) (out, in1, in2);         // All delays = 6
and #(3,5) (out, in1, in2, in3);   /* rise delay = 3, fall delay = 5,
                                      to_x_or_z = min(3,5) */
```

- Each delay can be written in *min:typ:max* form as well

```
nand #(2:3:4, 4:3:4) (out, in1, in2);
```

- Can also use a specify block to specify delays

Time Unit and Precision

- Compiler directive: ’**timescale**

  ’timescale *time_unit / time_precision*;

- 1, 10, 100 /s, ms, us, ns, ps, fs

```
’timescale 1ns / 100ps
module AND_FUNC (Z, A, B);
        output Z;
        input A, B;
        and # (5.22, 6.17) A1 (Z, A, B);
endmodule
/* Delays are in ns. Delays are rounded to one-tenth of a ns (100ps). Therefore,
   5.22 becomes 5.2ns, 6.17 becomes 6.2ns and 8.59 becomes 8.6ns */
// If the following timescale directive is used:
’timescale 10ns / 1ns
// Then 5.22 becomes 52ns, 6.17 becomes 62ns, 8.59 becomes 86ns
```

Delay Scaling

- The ’**timescale** directive specified the units and precision for all delay values in all modules that follows this directive until another ’**timescale** directive or reset is found

```
’timescale 1ns / 100ps
module AND_FUNC (Z, A, B);
        output Z;
        input A, B;
        and # (5.22, 6.17) A1 (Z, A, B);
endmodule
’timescale 10ns / 1ns
module TB;
        reg PUT_A, PUT_B;
        initial begin
                PUT_A=0; PUT_B=0;
                #5.21 PUT_B=1;
                #10.6 PUT_A=1;
                #15 PUT_B=0;
                #20 $finish;
        end
        AND_FUNC AF1 (GET_0, PUT_A, PUT_B);
endmodule
```

- Simulation always take place in smallest time precision, which is 100ps in the above example
- In module TB, delays are scales to 100ps. For example, 1 is applied to PUT_B at 520 hundred ps.

Getting Simulation Time

- System function, $**time**: returns the simulation time as an integer value scaled time unit specified.

```
’timescale 10ns / 1ns
module TB;
        ......
        initial
           $monitor (“PUT_A=%d PUT_B=%d”, PUT_A, PUT_B,
                       “GET_O=%d”, GET_O, “at time %t”, $time);
endmodule

PUT_A=0 PUT_B=0 GET_O=0 at time 0
PUT_A=0 PUT_B=1 GET_O=0 at time 5
PUT_A=0 PUT_B=0 GET_O=0 at time 11
......
/* $time calue is scaled to the time unit and then rounded */
```

- System function, $**realtime**: returns the simulation time as a real number scaled time unit.

```
’timescale 10ns / 1ns
module TB;
        ......
        initial
           $monitor (“PUT_A=%d PUT_B=%d”, PUT_A, PUT_B,
                       “GET_O=%d”, GET_O, “at time %t”, $realtime);
endmodule
```

```
......
PUT_A=0 PUT_B=1 GET_O=0 at time 5.2
PUT_A=0 PUT_B=0 GET_O=0 at time 10.6
```

Array of Instances

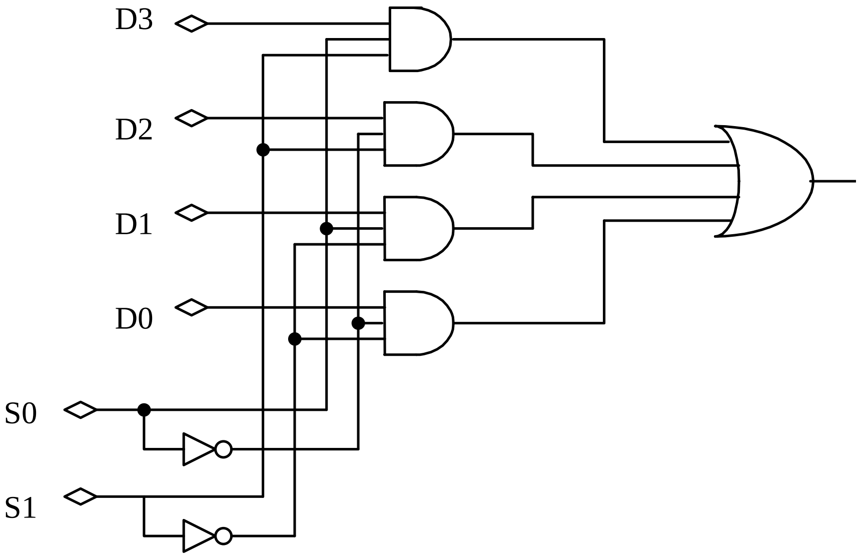
- An array of instances can be specified using the range specification

```
wire [3:0] OUT, INA, INB;
......
nand NG [3:0] (OUT, INA, INB);

// This is the same as:
nand    NG3 (OUT[3], INA[3], INB[3]),
        NG2 (OUT[2], INA[2], INB[2]),
        NG1 (OUT[1], INA[1], INB[1]),
        NG2 (OUT[0], INA[0], INB[0]);
```

Array of Instances

- 4x1 multiplexer circuit

```
module MUX4x1 (Z, D0. D1, D2, D3, S0, S1);
    output Z;
    input D0, D1, D2, D3, S0, S1;

    and (T0, D0, S0BAR, S1BAR),
          (T1, D1, S0BAR, S1),
          (T2, D2, S0, S1BAR),
          (T3, D3, S0, S1);

    not (S0BAR, S0),
         (S1BAR, S1);

    or (Z, T0, T1, T2, T3);
endmodule
```



A Clock Generator

```
module CLOCK (CLK);
        output CLK;

        initial begin
            START = 1;
            #3 START = 0;
        end

        nor #5 (CLK, START, CLK);
endmodule

// Generate a clock with on-off width of 5
// Not synthesizable
// For waveform only
```

Implicit nets

- If a net is not declared, it is assumed to be a single bit wire
- Compiler directive to override:

  ’default_nettype *net_type*

  // Example:

  ’default_nettype wand

- Directive occurs outside of module. Stays in effect until next same directive is reached or reset.

Printing Values

- **$display** system task: Prints the specified list of arguments to output and adds a newline automatically.

```
$display | $displayb | $displayh | $displayo
    (arg1, arg2, ……, argN);
// Example:
$display (“Simulation time is %t”, $time);
$display ($time, “R=%b, S=%b, Q=%b, QB=%b”, R, S, Q, QB);
```

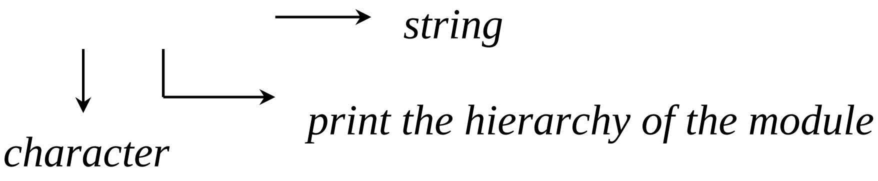
- **$monitor** system task: Task us called whenever a change occurs in argument list.

```
initial
    $ monitor (“At %d, d=%d, clk=%d”, $time, d, clk, “and q is %b”, q);
```

  - Only one monitor can be active at any time

```
$ monitoroff   // Disable all monitors;
$ monitoron    // Enable all monitors;
```

- Formatting chars.: %h, %d, %o, %b, %c, %m, %s



string

print the hierarchy of the module

character

User-Defined Primitives

Basics

- ❒ Arguments the set of predefined gate primitives.
- ❒ Use exactly the same way as gate primitives.
- ❒ May be a combinational UDP or a sequential UDP.
- ❒ A sequential UDP may model both level-sensitive and edge-sensitive behavior.
- ❒ Behavior is described as a table.
- ❒ Acceleratable.
- ❒ Each UDP has one output: 0, 1, x (z is not allowed).
- ❒ If input is z, it becomes an x.
- ❒ In a sequential UDP, output has the same value as the internal state.

Combinational UDP

```
primitive MUX1BIT (Z, A, B, SEL);
          output Z;
          input A, B, SEL;

          table
            //       A         B         SEL     :         Z
                     0         ?         1       :         0    ;
                     1         ?         1       :         1    ;
                     ?         0         0       :         0    ;
                     ?         1         0       :         1    ;
                     0         0         x       :         0    ;
                     1         1         x       :         1    ;
          endtable
endprimitive
```

Don’t care

- Any combination that is not specified is an x.
- Output port must be the first port.
- “?” represents iteration over “0”, “1”, or “x” logic values

Sequential UDP (Level)

- Level-sensitive sequential UDP example:

```
primitive LATCH (Q, CLK, D);
    output Q;
    reg Q;
    input CLK, D;

    table
      // CLK   Data      State        Output(next state)
         1      1    :  ?  :    1          ;
         1      0    :  ?  :    0          ;
         0      ?    :  ?  :    -          ;
    endtable
Endprimitive
```

- “?” means don’t-care

- “-” means no change in output

Sequential UDP (Edge)

- Edge-sensitive sequential UDP example:

```
primitive D_EDGE_FF (Q, CLK, D);
    output Q;
    reg Q;
    input CLK, D;

    table
      // CLK   Data       State       Output(next state)
         (01)   0    :    ?    :    0           ;
         (01)   1    :    ?    :    1           ;
         (0x)   1    :    1    :    1           ;
         (0x)   0    :    0    :    0           ;
      // Ignore negative edge of clock;
         (?0)   ?    :    ?    :    -           ;
      // Ignore data change on steady clock;
         ?      (??) :    ?    :    -           ;
    endtable
endprimitive
```

Continuous Assignments

Basics

- Models behavior of combinational logic
- Dataflow style
- Assign a value to a net
- Examples:

```
wire [3:0] Z, PRESET, CLEAR;
assign Z = PRESET & CLEAR;

wire COUT, CIN;
wire [3:0] SUM, A, B;
assign {COUT, SUM} = A+ B + CIN;
assign  MUX = (S == 0) ? A: ’bz,
        MUX = (S == 1) ? B: ’bz,
        MUX = (S == 2) ? C: ’bz,
        MUX = (S == 3) ? D: ’bz;
assign Z = ~(A | B) & ( C | D);
```

- Expression on right-hand side is evaluated whenever any operand changes

Net Declaration

- Can have an assignment in the net declaration

```
wire [3:0] A = 4’b0;
assign PRESET = ’b1;

wire #10 A_GT_B = A > B;
```

- Only one assignment to a net using net declaration

- Multiple assignements to a net is done using continuous assignments

Delay

- Delay between assignment of right-hand side to left-hand side

```
wire #10 A = B && C;        // Continuous delay
```

- Net delay

```
wire #10 A;
// Any change to A is delayed 10 time units before it takes effect
```

- If delay is in a net declaration assignment, then delay is not net delay

```
wire #10 A = B + C;
// 10 time units id it part of the continuous assignment and not net delay
```

- If value changes before it has a chance to propagate, latest value change will be applied
  - Inertial delay

Operands

1. Numbered operands
2. Functional call operands
   - A functional call can be used as an operand within an expression

```
wire [7:0] A;
// PARITY is a function described elsewhere
assign PAR_OUT = PARITY(A);
```

3. Bit selects

```
input [3:0] A, B, C;
output [3:0] SUM;
assign SUM[0] = (A[0] ^ B[0] ^ C[0]);
```

4. Part selects
5. Memory addressing

```
reg [7:0] RegFile [0:10];       // 11 b-bit registers
reg [7:0] A;
 RegFile[3] = A;                // A assigned to 3rd register in RegFile
```

Operators

1. Arithmetic
2. Relational
3. Equality
4. Logical
5. Bit-wise
6. Reduction
7. Shitf
8. Conditional
9. Concatenation

Arithmetic Operators

+	(plus)
-	(minus)
*	(multiply)
/	(divide)
%	(modulus)

- Integer division will truncate
- % gives the remainder with the sign of the first operand
- If any bit of operand is **x** or **z**, the result is **x**
- *reg* data type holds an unsigned value, while *integer* data type holds a signed value

```
reg [0:7] A;
integer B;
A = -4’d6;          // reg A has value unsigned 10
B = -4’d6;          // integer B has value signed -6
A-2                 // result is 8
B-2                 // result is -8
```

Relational Operators

>	(greater than)
<	(less than)
>=	(greater than or equal to)
<=	(less than or equal to)

- If there are x or z in operand, the result is x
- If unequal bit lengths, smaller operand is zero-filled on most significant side (I.e., on left)

Equality Operators

==	(logical equality, result may be unknown)
!=	(logical inequality, result may be unknown)
===	(case equality, x and z also compared)
!==	(case inequality, x and z also compared)

A = ’b11x0;
B = ’b11x0;

A == B is known.
A === B is true

- Unknown is same as false in synthesis
- Compare bit by bit, zero-filling on most significant side

Logical Operators

&&	(logical and)
||	(logical or)

```
A = ’b0110;     // non-zero
B = ’b0100;     // non-zero
```

A || B is 1.

A && B is also 1

- Non-zero value is treated as 1
- If result is ambiguous, set it to x

Bit-wise Operators

~	(unary negation)
&	(binary and)
|	(binary or)
^	(binary exclusive-or)
~^,^~	(binary exclusive-nor)

A = ’b0110;

B = ’b0100;

A | B is 0110

A & B is 0100

- If operand sizes are unequal, smaller is zero-filed in the most significant bit side

Reduction Operators

&	(unary and)
~&	(unary nand)
|	(unary or)
~|	(unary nor)
^	(unary xor)
~^	(unary xnor)

A = ’b0110;
B = ’b0100;

| B is 1
& B is 0

- Bitwise operation on a single operand to produce 1-bit result

Shift Operators

<<	(left shift)
>>	(right shift)

```
reg [0:7] D;

D = 4’b0111;
```

D >> 2 has the value 0001

- Logic shift, fill vacant bits with 0
- If right operand is an x or a z, result is x
- Right operand is always an unsigned number

Conditional Operators

expr1 ? expr2:expr3

wire [0:2] GRADE = SCORE > 60 ? PASS:FAIL;

- If *expr1* is an x or a z, *expr2* and *expr3* are combined bit by bit (all x's except 0 with 0 = 0, 1 with 1 = 1)

Concatenation and Replication

```
wire [7:0] DBUS;
wire [11:0] ABUS;

assign ABUS[7:4] = {DBUS[0], DBUS[1], DBUS[2], DBUS30]};
assign ABUS = {DBUS[3:0], DBUS[7:4]};
assign DBUS[7:4] = {2{4'B1011}};   // 1011_1011
assign ABUS[7:4] = {{4{DBUS[7]}}, DBUS};   // sign extension
```

Outline

- ❒ Introduction
- ❒ Language Structure
- ❒ Modeling Part 1
- ❒ Modeling Part 2

Behavioral Modeling

Procedure Blocks

- Use procedural blocks to describe the operation of the circuit
- Two procedural blocks:
  - *always* block: executes repetitively
  - *initial* block: executes once
- Concurrent procedural blocks
- All execute concurrently
- All activated at time 0

An Example

```
module TWO_PHASE;
    reg PH_A, PH_B;

    initial
    begin
        PH_A = ’b0;
        PH_B = ’b1;
    end

    always
    begin
        #10 PHA_A = ~PH_A;
    end

    always
    begin
        #15 PHA_B = ~PH_B;
    end
endmodule
```

Another Example

```
reg [0:5] instr_reg;
reg [3:0] accum;
wire execute_cycle;

always @(execute_cycle)
    begin
        case (instr_reg{0:1])
            2’b00 : STORE (accum,instr_reg[2:5]);
            2’b01 : LOAD (accum,instr_reg[2:5]);
            2’b10 : JUMP (instr_reg[2:5]);
            2’b11 :;
        endcase
    end
 // STORE, LOAD, and JUMP are user defined tasks
```

The *always* Block

Can model:

- combinational logic
- sequential logic

Syntax:

```
// Single statement
always @ (event expression)
    statement

// Sequential statement
always @ (event expression)
    begin
        sequential statements
    end
```

The *always* Block

- “event expression” specified a set of events based on which statements within the always block are executed sequentially
- The type of logic synthesized is based on what is specified in the “event expression”
- Four forms of event expressions are supported
  - An OR of several identifiers (comb/seq logic)
  - The rising edge clock (register inference)
  - The falling edge clock (register inference)
  - Asynchronous reset (register inference)

Event Expressions

1. An OR of several identifiers

   » Combinational or synchronous logic may be represented by a set of sequential statements

```
always @ (id1 or id2 or id3 or ... or idn)
   begin
      sequential_statements
   end
```

- A synchronous block may appear inside an always block (representing synchronous logic) in two forms:

```
always @ (posedge clock_name)
   begin
      sequential_statements
   end
always @ (negedge clock_name)
   begin
      sequential_statements
   end
```

- Sequential statements not within a sequential block represents combinational logic

Example:

```
module Comb (A, B, C, Y);
   input A, B, C;
   output Y;
   reg Y;

   always @ (A or B or C)
     begin
       Y = A ^ B ^ C;
     end
endmodule
```

Event Expressions

2. The rising edge clock (register inference)

- » The event expression denotes the rising edge of a clock
- » The behavior within block represents synchronous logic triggered on the rising edge of clock

```
always @ (posedge CLK)
   begin
      Q = D;
   end
```

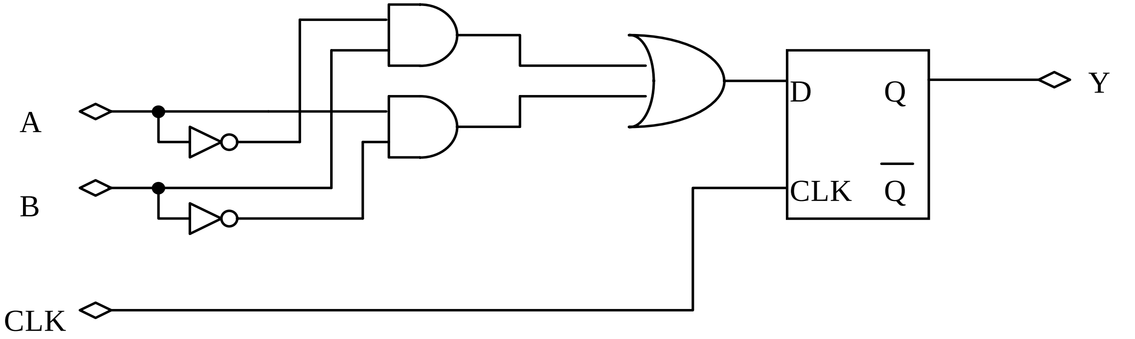
3. The falling edge clock (register inference)

- » The event expression denotes the falling edge of a clock
- » The behavior within block represents synchronous logic triggered on the rising edge of clock

```
always @ (negedge CLK)
   begin
      Next_state = Current_state;
   end
```

Register Inference

```
module SEQ (CLK, A, B, Y);
   input CLK, A, B;
   output Y;
   reg Y;

   always @ (posedge CLK)
     begin
       Y = A + B;
     end
endmodule
```



Event Expressions

4. Asynchronous reset (register inference)

- Asynchronous resets in addition to register inferences (2 & 3 above)

```
always @ (negedge reset1or posedge CLK or posedge reset2)
    begin
        if (!reset1)
            begin
                /* sequential_statements
                asynchronous input triggered by the false condition of reset1 to the registers */
            end
        else
            begin
                /* sequential_statement
                Optional for sequential statements. Could well have “else-if” clauses */
                    if (!reset2)
                        begin
                            /* sequential_statements: Asynchronous inputs triggered by reset2 */
                        end
                    else
                        begin
                            // sequential_statements: register inference statements.
                        end
            end
    end
```

Register Inference

- The language constructs “registers” is synthesized as a hardware register (flip-flop) if the register is assigned a value in:
  - A sequential block
  - An “always” block that has an event expression denoting a rising or falling clock edge
- It is illegal to assign a register value on both rising and falling edges of a clock

Sequential Statements and the Always Block

- Only “registers” and “integers” may be assigned values in sequential statements
- If an output port of the module is to be assigned a value in an always block it generally must be declared to be of a register type as well
- Possible sequential statements within an always block are
  - procedure assignment
  - synchronous block
  - if statement
  - case statement
  - for-loop statement
  - repeat loop statement
  - block statement
  - task enabling
- An “always” block is concurrent and so may appear in any order within a module body with other continuous assignments (module instantiations or other always block)
- Data is passed out of an always block using register variables

Procedural Assignments

- The assignment statements that can be used inside an always blocks
- Similar to continuous assignment statements except that the target must be a register or integer type
- The following forms are allowed as a target of a procedural assignment:

  register variables

  bit-select of register variables

  part-select of register variables

  concatenations of above

  Integers

- Examples:

```
A = B << 2;
CTRL = (opcode == 2'b01);
sum = a + b;
```

Procedural Assignments

- Assignments are always processed bit-wise
- If the bit-width of an expression on the right-hand side is larger than that of the target, the extra bits are discarded
- If an expression's bit-width is less than that of the target, zeros are added at the left, or signed bits is extended for signed numbers
  - Note that integers always hold signed numbers, while wires and registers always hold unsigned numbers

Blocking v.s. Non-Blocking

- Blocking (=)
  - Assignment are blocked, i.e., they must be executed before subsequent statements are executed

```
always @ (posedge clock)
    begin
        B = A;
        C = B;
        D = C;
    end
```

  - 1 flip-flop (Data in is A, data out is D)

- Non-blocking (<=)
  - Assignment are not blocked, i.e., can be scheduled to occur without blocking the procedural flow

```
always @ (posedge clock)
    begin
        B <= A;
        C <= B;
        D <= C;
    end
```

  - 3 pipelined flip-flops (A to B to C to D)

Blocking v.s. Non-Blocking

- Blocking procedural assignment:
  - Assignment is executed before any of the following ones are executed.
  - Only applies to its own sequential block.

```
reg_a = 10;
```

- Non-blocking procedural assignment.
  - The procedural flow is not blocked

```
# 2
    reg_a <= LOAD;
    reg_b <= STORE;
```

  - Evaluate right-hand side and schedules the assignment.
  - At the end of the current loop, assignment to the left-hand side is made

Examples

- Blocking assignment

```
initial
begin
    CLR = #5 0;
    CLR = #4 1;
    CLR = #10 1;
end
// CLR is assigned at time 5, and then at 9 and then at 19
```

delay between *RHS* and *LHS*

- Non-blocking assignment

```
initial
begin
    CLR <= #5 1;
    CLR <= #4 0;
    CLR <= #10 1;
end
// CLR is assigned 0 at time 4, 1 at time 5 and 1 at time 10
```

  - Value is indetermined if multiple values are assigned at the same time

Procedural Continuous Assignment

- Allow expression to be driven continuously into integers or nets
- *assign* and *deassign* procedural statements: for integers
- *force* and *release* procedural statements: for nets

Assign and Deassign

- An *assign* procedural statement overrides all procedural assignments to a register
- The *deassign* procedural statement ends the continuous assignment to a register
- Value remains until assigned again
- If assign applied to an alreasy assigned register, it is deassigned first before making the new procedural continuous assignment

```
module DFF (D, CLR, CLK, PRESET, Q);
    input D, CLR, CLK, PRESET;
    output Q;
    reg Q;
    always @ (CLR or PRESET)
        if (!CLR)
            assign Q = 0;  // D has no effect on Q
        else if (!PRESET)
            assign  Q = 1;  // D has no effect on Q
        else
            deassign Q;
    always @ (posedge CLK)
        Q = D;
endmodule;
```

Force and Release

- Similar to assign-deassign, except that it can be applied to nets as well as registers
- *force* procedural statement on a net overrides all drivers of the net, until a *release* is executed on the net

```
......
or #1 (PRT, STD, DXZ);

initial
begin
    force PRT = DXZ & STD;
    #5                  // Wait for 5 time units.
    release PRT;
    $finish;
end
```

High-Level Constructs

- if statement
- loop statement (*forever*, *repeat*, *while*, *for*)
- *case* statement

If Statement

```
if (expression)
    statements
{ else if (expression)
    statements          }
[ else
    statements          ]
```

- Example:

```
if (total < 60)
begin
    GRADE = C;
    TOTAL_C = TOTAL_C + 1;
end
else if (SUM < 75)
    GRADE = B;
else
    GRADE = A;
```

Case Statement

```
case (case_expression)
      case_item_expression {, case_item_expression} : statements
      ......
      ......
      [ default: statements                    ]
endcase
```

- *case_expression* is evaluated first
- *case_item_expressions* are evaluated and compared in the order given
- *case_expressions* or *case_item_expressions* need not be constant expressions
- .
- x and z values are compared as well

An Example

```
wire [3:0] A, B;
        reg [3:0] Z;

        parameter   ADD_INSTR = 2’b10, SUB_INSTR = 2’b11;
                    MULT_INSTR = 2’b01, DIV_INSTR = 2’b00;

        case (OP_CODE),
            ADD_INSTR:        Z = A + B;
            SUB_INSTR:        Z = A - B;
            MULT_INSTR:       Z = A * B;
            DIV_INSTR:        Z = A / B;
        endcase
```

Don’t-Care in Case

- *casez* considers z values (in case-expression and in case_intem_expressions) as don’t-cares
- *casex* statement consider *x* and *z* values as don’t-cares

```
casez (MASK);
     4’b1???:          DBUS[4] = 0;
     4’b01??:          DBUS[3] = 0;
     4’b001?:          DBUS[2] = 0;
     4’b0001:          DBUS[1] = 0;
endcase
```

- A z value is same as a ? in a literal to mean a don’t-care

Sequential Statements in a Block

- "case" statement
  - Specifies a multi-way branch based on the value of an expression.
  - Is sequential and therefore may be nested to any level.
  - $parallel ensures not priority encoded (less logic)

```
module EX_CASE (A, B, F, D, H, N);
            input A, B;
            input [3:0] F;
            inout H, N;
            output [1:0] D;
            reg H, N;
            reg [1:0] D;
       always @(A or B or F or H or N)
            begin
                 case (F) // $parallel
                      0, 4, 8, 9: D = {H, A};
                      5: N = N & B;
                      7: H = B;
                      default: begin
                            D = {H, A}; N = N & A; H = B;
                      end
                 endcase
            end
endmodule
```

Case Statement

- Full and Parallel

```
module FULL (c1, c2, c3, c4, a, y);
   input c1, c2, c3, c4;
   output y;
   input [1:0] a;
   output [6:0] y;
   reg [6:0] y;
always @ (posedge Pclk)
   begin
      if (Preset)
         begin
            y  <= #2 {6’b0, 1’b0};
         end
      else casex ({c1, c2, c3, c4}) // $full $parallel
            4’b1x0x:          y[0] <= #2 a[0];
            4’b1x1x:          y[1:0] <= #2 a;
            4’b01x0:          y <= #2 y + 7’b1;
            4’b0101:          y <= #2 {y[6:1]+6’b1, 1’b0};
            4’b1111:          y <= #2 {y[6:2]+5’b1, 2’b0};
         endcase
   end
```

Forever and Repeat

- *forever-loop* statement

  forever

      statements

  ```
  // should only be used with timing controls or with the disable statement
  forever
      #10 CLOCK = ~CLOCK;
  ```

- *repeat-loop* statement

  repeat (*loop-count*)

      statements

  ```
  repeat (FAC)
      FACSUM = FACSUM + FAC;
  repeat (shift_by)
      P_reg = P_reg << 1;
  ```

While and For

- *while-loop* statement

```
forever (condition)
    statements
while (BY > 0)
begin
    acc = acc << 1;
    BY = BY - 1;
end
```

- *for-loop* statement

```
for (initial_assignment; condition; step_assignment)
    statements
integer K;  // K must be of register type
for (k=0; K<Max_range; K=K+1)
begin
    if (ABUS[K] == 0) ABUS[K] = 1;
    else if (ABUS[K] == 1) ABUS[K] = 0;
    else $display (“ABUS[K] is an x or an z”);
end
```

“For” - “Loop” Statement

- Nested for-loops supported
- A synchronous logic block is not allowed within a for-loop statement
- Example (“for”- “loop”)

```
for (J=5; J<8; J=J+4)
    begin
        F = J + E;
        C = F;
    end
```

“Repeat” - “Loop” Statement

- Nested repeat-loops supported
- A synchronous logic block is not allowed within a repeat-loop statement
- Example (“repeat”- “loop”)

```
parameter size = 5;
repeat (size)
      begin
            H = A = B;
            I = H*3;
      end
```

Timing Controls

- Timing control over when procedural statements can occur
- Delay control
  - *# delay*
  - Timing duration from time initially encountered the statement to the time it executes
- Event control
  - Statement execution is delayed until the occurrence of some simulation event
  - *@ symbol*
  - Edge-triggered control
  - Level-sensitive control

Delay Control

- The procedural statement execution is delayed by the specified delay
- If delay expression is x or z, it is 0
- If delay expression is negative, use two's complement unsigned integer

```
#2 TX = RX - 5;

# STROBE          COMPARE = TX ^ MASK;

#(PERIOD/2)       CLOCK = ~CLOCK;
```

Edge-Triggered Event Control

```
@ (posedge CLOCK)                    CURR_STATE = NEXT_STATE;
@ (posedge RESET)  COUNT = 0;
@ (CTRL_A or CTRL_B)            DBUS = ’bz;
@ CLA   ZOO = FOO;
// Assign on any change of value in register CLA.
@ (posedge CLEAR or negedge RESET)          Q = 0;
```

- Negative edge: (1->x, z, or 0), (x or z -> 0)
- Positive edge: (0->x, z, or 0), (x or z -> 1)
- Events can be OR’ed as well to indicate “if any one of the events occur”

Levels-Sensitive Event Control

- Execution of a procedural statement is delayed until a condition becomes true
- wait statement:

  wait (*condition*)

  statement

- If condition is already true, the next statement is evaluated immediately.

  wait (SUM > 22) SUM = 0;

Intra-assignment Timing Control

- Timing control within an assignment
- Delay assigning right-hand side to left-hand side
- Right-hand side expression is evaluated before the delay

```
DONE = #5 A;
// is the same as
begin
    temp = A;
    #5 DONE = temp;
end

Q = @ (posedge CLK) D;
// is the same as
begin
    temp = D;
    @ (posedge CLK) Q = temp;
end
```

Example

- Negative edge-triggered D flip-flop with asynchronous preset:

```
module DFF (CLK, D, PRESET, Q, QBAR);
    input CLK, D, PRESET;
    output Q, QBAR;
    reg Q, QBAR;

    always wait (PRESET == 1)
    begin
        #3 Q = 1;
        #2 QBAR = 0;
        wait (PRESET == 0);
    end
    always @ (negedge CLK)
    begin
        if (PRESET != 1)
        begin
            #5 Q = D;
            #1 QBAR = ~Q;
        end
    end
endmodule
```

Block Statement

- Grouping of statements as one statement
- Two types:
  - *Sequential* block (*begin-end* block):

    Statements are executed sequentially in the given order
  - *Parallel* block (*fork-join* block):

    Statements in the block executed concurrently
- Blocks can be named optionally
  - Registers can be declared locally
  - Blocks can be referenced (*disable* statement)
  - Can uniquely identify registers

Sequential Block

```
begin
      [ : block_id { declarations } ]
      statements
end

// Waveform generation
begin
      #2 stream = 1;
      #7 stream = 0;
      #10 stream = 1;
      #14 stream = 0;
      #16 stream = 1;
      #21 stream = 0;
end
```

Parallel Block

- Control passes out of block after all statements finish

```
fork
     [ : block_id { declarations } ]
     statements
join

// Waveform generation
fork
     #2 stream = 1;
     #5 stream = 0;
     #3 stream = 1;
     #4 stream = 0;
     #2 stream = 1;
     #5 stream = 0;
join
```

Initial Statement

- Executes only once

```
initial
    statements
```

- Used for initialization and waveform generation

```
// Initialization:
reg [7:0] RAM[0:1023];
reg RIB_REG;

initial
begin: SEQ_BLK_A
    integer INX;

    RIB_REG = 0;
    for (INX=0; INX<SIZE;INX=INX+1)
        RAM[INX] = 0;
end
```

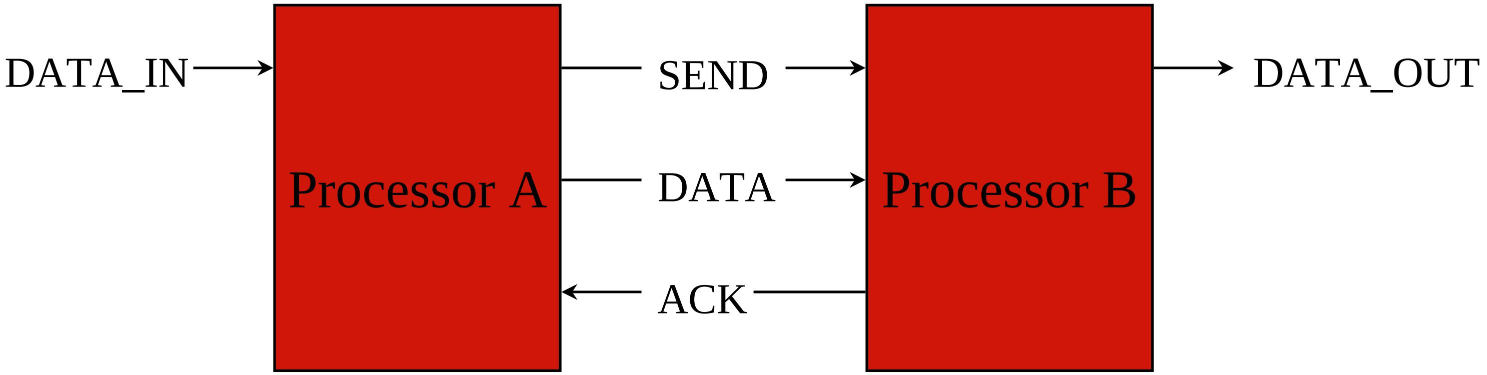
Always Statement

- Executes continuously; must be used with some form of timing control.

```
always
      statements

always
      CLK = ~CLK;
// Will loop indefinitely

always
      #5 CLK = ~CLK;
// Waveform on CLK of period 10
```

- Any number of *initial* and *always* statements may appear within a module
- *initial* and *always* statements all execute in parallel

Handshake Example



```
’timescale 1 ns/100ps
module HAND_SHAKE (DATA_IN, DATA_OUT);
    input [0:31] DATA_IN;
    output [0:31] DATA_OUT;
    reg SEND, ACK;
    reg [0:31] DATA;
    initial {ACK, SEND} = 0;
```

```
        always
        begin
            SEND = 1;
            DATA = DATA_IN;
            wait (ACK = 1);
            SEND = 0;
            #50;                            // Wait for 50 time units.
        end

        always
        begin
            #25;                            // Wait for 25 time units.
            DATA_OUT = DATA;
            ACK = 1;
            #25 ACK = 0;
            wait (SEND == 1)
        end
    endmodule
```

Other Topics

Tasks

- Procedure
- Can contain timing control
- Can call other functions and tasks
- 0 or more arguments
- Output or input arguments can be updated
- Task definition:

```
task task_id;
          [ declarations]
          statements
         endtask
```

- Cannot declare a new type within a task

Task Example

```
parameter MAXBITS = 8;

task REVERSE_BITS;
    input [MAXBITS-1 : 0] DIN;
    output [MAXBITS-1 : 0] DOUT;
    integer K;
    begin
       for (K=0;K<MAXBITS;K=K+1)
          DOUT [MAXBITS-K] = DIN[K];
    end
 endtask
```

Task Calling

- Task calling:

  task_id [(*expr1*, *expr2*, . . . , *exprN*)];

- List of arguments must match the order of arguments in task definition
- Arguments are passed by value
- Task can be called more than once concurrently
- Local variables are static; if concurrently called, same local variables are shared
- Calling statement for task REVERSE_BITS:

```
// Declarations:
reg [MAXBITS-1 : 0] REG_X, NEW_REG_X;

REVERSE_BITS (REG_X, NEW_REG_X);  // Calling task.
```

Tasks and Functions

- Provides the ability to call common procedures from different places within a description
- Enables large procedures to be broken into smaller ones making reading and debugging easier
- The I/O passed in a task enabling statement must match that of the I/O in the task declaration

Example: (Task)

```
task PROC
      input A, B;
      inout D;
  begin
      D = A + B
  end
endtask
```

Example: (Task enabling for the left task)

```
always @ (in1 or in2)
      if (in1)
            PROC (in1, Y, result);
      else if (in2)
            PROC (in2, Y, result);
```

Functions

- Executes in one simulation time unit.
- No delays.
- Cannot call another task.
- Must have at least one input.
- Returns a value that can be used in an expression.
- Function definition:

```
function [range] function_id;
            input_declarations
            other_declarations
            statements
 endtask
```

- If no range is specified, 1 bit is assumed.

Function Example

```
parameter MAXBITS = 8;

function [MAXBITS-1:0] REVERSE_BITS;
        input [MAXBITS-1 : 0] DIN;
        reg K;
        begin
              for (K=0;K<MAXBITS;K=K+1)
                  REVERSE_BITS [MAXBITS-K] = DIN[K];
        end
 endfunction
```

- Implicit declaration of a reg, same as function name
- Must include assignment to function name

Function Call

- Can be used in any expression
- Function call:

  function_id [(*expr1*, *expr2*, . . . , *exprN*)];

- Calling function REVERSE_BITS:

```
// Declarations:
reg [MAXBITS-1 : 0] REG_X, NEW_REG_X;

NEW_REG_X = REVERSE_BITS (REG_X);
```

Function Call

- Return a value (unlike task)
- Must have at least one input argument
- May not enable task, but task may enable other tasks and functions

Example: (function)

```
function [3:0] DEPTH
      input D, T;
      reg [3:0] R, p;
  begin
      if (D)
        p = p*2;
      else if (~T)
        R = *p;
      DEPTH = R*p;
  end
endfunction
```

Example: (calling function)

```
// RESULT is a 4-bit wire
assign RESULT = DEPTH (in1, in2);
```

Hierarchy

- Module definition:

  module *module_name* (*port_list*);

  declarations_and_statements

  endmodule

- Module instantiation statement:

  module_name instance_name (*port_associations*);

- Port associations can be positional or named; cannot be mixed

  local_net_name // positional

  Port_Name(*local_net_name*) // Named

- Ports can be: input, output, inout

- Port can be declared as a net or a reg; must have same size as port

Hierarchy

- Unconnected module inputs are driven to z state
- Unconnected module outputs are simply unused

```
DFF d1 (.Q(QS), .QBAR(), .DADA(D),
           .PRESET(), .CLOCK(CK));          // Named

DFF d2 (QS, , D, , CK);                     // Positional
// Output QBAR is not connected
// Input PRESET is open and hence set to calue z
```

Hierarchical Instantiation

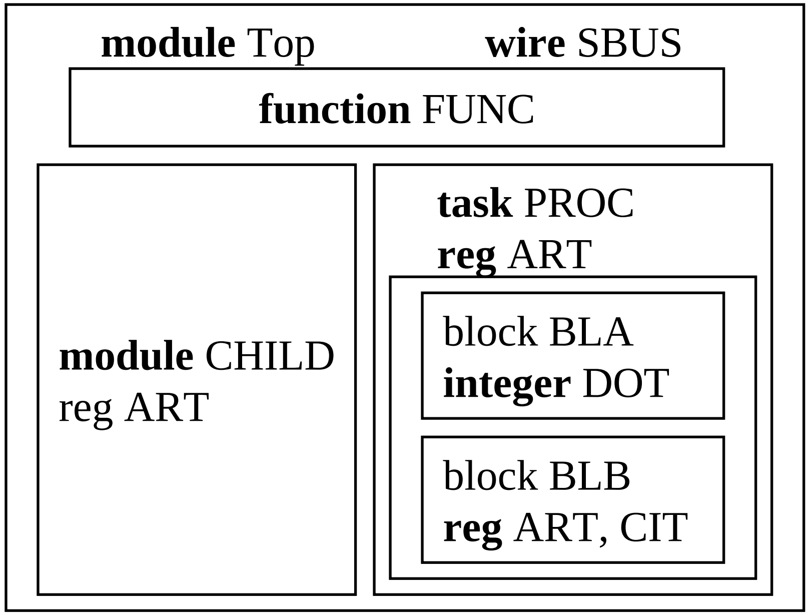
```
module sub_block1 (a, z);
input a;
output z;
wire a, z;
IV U1 (.A(a), .Z(z));
endmodule

module sub_block2 (a, z);
input a;
output z;
wire a, z;
IV U1 (.A(a), .Z(z));
endmodule
```

```
module top (din1, din2, dout1, dout2);
input din1;
input din2;
output dout1;
output dout2;

wire din1, din2, dout1, dout2;

sub_block1 U1(.a(din1), .z(dout1));;
sub_block2 U2(.a(din2), .z(dout2));;
endmodule
```

Hierarchical Path Name

- Every identifier has a unique hierarchical path name
- Period character is the separator
- New hierarchy is defined by: module instantiation, task definition, function definition, named block

TOP.CHILD.ART
TOP.PROC.ART
TOP.PROC.BLB.CIT
TOP.PROC.BLA.DOT
TOP.SBUS



Mixing Structure and Behavior

```
module module_name (port_list);
    Declarations:
        Net declarations.
        Reg declarations.
        Parameter declarations.

    Initial statements.
    Gate instantiation statements.
    Module instantiation statements.
    UDP instantiation statements.
    Always statements.
    Continuous assignment.
endmodule
```

Module Parameter Values

- Two ways
- Defparam statement:
  - Parameter value in any module instance can be changed by using hierarchical name.

```
defparam FA.n1.XOR_DELAY = 2,
         FA.n2.AND_DELAY = 3;
```

- Module instance parameter value assignment:
  - Specify the parameter value in the module instantiation.
  - Order of assignment is the same as order of declarations within module

```
HA # (2, 3) h1 (.A(p), .B(Q), .S(S1), .C(X1));
```

System Tasks

- Display System Tasks
  - $**display** and $**monitor** system tasks.
  - $**write** task: Write the specified argument values to output, but does not add a newline character (as opposed to $**display** which does)

    $**write** | $**writeh** | $**writeb** | $**writeo** (*arg1*, *arg2*, . . . , *argN*)

    - Different default bases

    $**write** (“simulation time is ”);

    $**write** (“%t\n”, $**time**);
  - $**strobe** task: Display run data at selected time but at end of current simulation time.

    always @ (posedge RST)

    $**strobe** (“the flip-flop value is %b at time %t”, Q, $**time**);

    /* After end of simulation time when RST has a positive edge, the $strobe task prints the values of Q and current simulation time. */
  - %h, %d, %o, %b, %c, %m, %s

System Tasks (cnt'd)

- File I/O system tasks
  - **$fopen**, **$fclose**
  - **$fdisplay**, **$fwrite, $fstrobe**, **$fmonitor**
  - **$readmemb**, **$readmemh**: Loads memory data from a file

```
reg [0:3] MEM_A [0:63];
$readmemb (“ones_and_zeros.vec”, MEM_A);
```

- Simulation control system tasks
  - **$finish**;   // make the simulation exit
  - **$stop**; // Causes simulation to suspend

- Simulation time system functions
  - **$time**   // 64-bit time value is returned
  - **$stime**// 32-bit unsigned integer time value is returned
  - **$realtime**   // return real number time

Disable Statement

- Can be used to terminate a task or a block before it finishes executing all its statements
- Used to model hardware interrupts and global resets

  disable task_id;

  disable block_id;

- Execution continues with the next statement

```
begin BLK_A
        // statement 1
        disable BLK_A;
        // statement 2
end
// statement 3
```

// Statement 2 is never executed. After disable statement is executed, statement 5 is executed

Value Change Dump File

- Contain information about value changes on specified variables in design

  $**dumpfile** (“uart.dump”);

  $**dumpvars** (level_num, CLK, CLR);

  $**dumpvars**; // Dumps all variables.

  $**dumpoff** ;

  $**dumpon**;

  $**dumpall**;

  $**dumplimit** (file_size);

  $**dumpflush**;

Compiler Directives

- Always begins with a character ’
- Definition holds accross multiple files for one compilation

```
’default_nettype wand          // specifies net type for implicit net
//
’define WORD 16                // Create a macro for text substitution
’undef WORD                    // Undefines a previously defined text macro.
’ifdef SUN
. . .
[’else
. . .
’endif ]                       // Conditional compilation
’include “../../rx.h”          // Inserted the contents of the specified file.
’resetall                      // All compiler directives are reset to default.
’timescale 1 ns / 10 ps        // 1ns is the time unit and delays are rounded to 10ps
```

Nested Macros

```
module NESTED_MACROS (A, B, C, D, Y);
input A, B, C, D;
output Y;

reg Y;

’define ONE 1’b1
’define ZERO 1’b0
’define TEST (A == ’ONE)

always @ (A or B or C or D)
if (’TEST && (C != D))
Y = 1;
else
Y = 0;

endmodule
```

Specify Block

- Used to assign delay to these paths
- Used to describe paths between a source and a destination
- Used to perform timing checks.

```
specify
    spec_param_declarations          // Declares parameters
    // for use only within the specify block.
    path_declarations
    system_timing_checks
endspecify
```

Specify Block

```
specify
    specparam tCLK_Q = (4:5:6);

// Path delays:
(CLOCK*> Q) = tCLK_Q;
(DATA*> Q) = 12;
(CLEAR, PRESET*> Q) = (4:5:3);

/* Can specify pulse width to be rejected and a range for which to generate an x */

    specparam PATHPULSE$ = (1,2);
// Reject limit = 1, error limit = 2.
    specparam PATHPULSE$DATA$Q = 6;
endspecify
```

Outline

- ❐ Introduction
- ❐ Language Structure
- ❐ Modeling Part 1
- ❐ Modeling Part 2

Modeling for Synthesis

Modeling Structures

- Net-list
  - structural description
- Primitives
- Continuous assignment
  - Data flow specification
  - Verilog operators
- Procedural blocks
  - always and initial blocks
    - Allow timing control and concurrency
  - C-like procedural statements
- Task and function

Continuous Assignments

- Model combinational logic
- Operands + operators
- Drive values to a net
  - assign out = a & b;
  - assign eq = (a==b);
  - wire #10 inv = ~in;
  - wire [7:0] c = a + b;
- Avoid logic loops
  - assign a = a + b;
  - asynchronous design

Operator Precedence

[ ]	bit-select or part select
( )	parenthesis
!,~	logic and bit-wise negation
&, |, ~&, ~|, ^, ~^, ^~	reduction operators
+,-	unary arithmetic
{ }	concatenation
*,/,%	arithmetic
+,-	arithmetic
>>,<<	shift
>, >+, <, <=	relational
==, !=	logical equality
&	bit-wise AND
^, ^~, ~^	bit-wise XOR and XNOR
|	bit-wise OR
&&	logical AND
||	logical or
?:	conditional

RTL Model

- Describe the system at a higher level of abstraction
- Specify a set of concurrently active procedural blocks
- Procedural blocks
  - *initial* blocks
    - For test-fixtures to generate test vectors
  - *always* blocks
    - Can be combinational circuits
    - Can infer latches or flip-flops
  - Procedural blocks have the following components
    - Procedural assignment statements
    - timing controls
    - high-level programming language constructs

RTL Statements

- Procedural and RTL assignments
  - reg and integer
  - out = a+ b;
- begin ... end block statements
  - group statements
- if ... else statements
- case statements
- for loops
- while loops
- forever loops
- disabled statements
  - Disable a named block

Combinational Always Blocks

- A complete sensitivity list (inputs)

```
always @ (a or b or c)
    f = a&~c | b&c;
```

- Simulation results

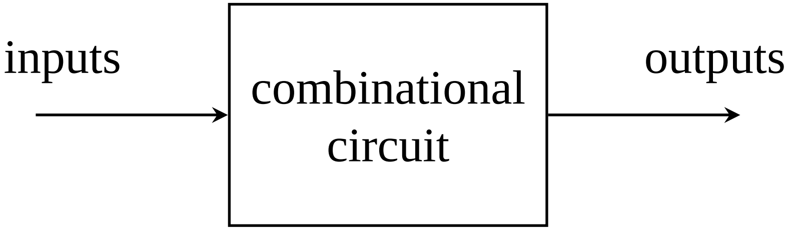
```
always @ (a or b)           // conditions are ignored by synthesizer
    f = a&~c | b&c;
```

- Parentheses

```
always @ (a or b or c or d)
    z = a+b+c+d;            // z = (a+b) + (c+d);
```

Combinational Circuit Design

- Outputs are functions of inputs



- Examples
  - Mux
  - Decoder
  - Priority encoder
  - Adder

Multiplexor

- Net-list (gate-level)

```
module mux2_1 (out, a, b, sel);
      output out;
       input a, b, sel;
       not (sel_, sel);
       and (a1, a, sel_);
       and (b1, b, sel);
       or (out, a1, b1);
endmodule
```

- Continuous assignment

```
module mux2_1 (out, a, b, sel);
      output out;
      input a, b, sel;
      assign out = (a&~sel) | (b&sel);
endmodule
```

- RTL modeling

```
module mux2_1 (out, a, b, sel);
      output out;
      input a, b, sel;
      always @ (a or b or sel)
          if (sel)
              out = b;
          else
              out = a;
endmodule
```

Multiplexor

- 4-to-1 multiplexor

```
module mux4_1 (out, in0, in1, in2, in3, sel);
     output out;
     input in0, in1, in2, in3;
     input [1:0] sel;

     assign out =   (sel == 2’b00) ? in0 :
                    (sel == 2’b01) ? in1 :
                    (sel == 2’b10) ? in2 :
                    (sel == 2’b11) ? in3 :
                      1’bx;
endmodule
```

- 4-to-1 multiplexor

```
module mux4_1 (out, in, sel);
     output out;
     input [3:0] in;
     input [1:0] sel;
     reg out;

     always @ (sel or in) begin
         case (sel)
             2’d0: out = in[0];
             2’d1: out = in[1];
             2’d2: out = in[2];
             2’d3: out = in[3];
             default: out = 1’bx;
         endcase
     end
endmodule
```

Decoder

- 3-8 decoder with an enable control

```
module decoder (o, enb_, sel);
    output [7:0] o;
    input enb_;
    input [2:0] sel;
    reg [7:0] o;
    always @ (enb_ or sel)
        if (enb_)
           o = 8’b1111_1111;
        else
           case (sel)
           3’b000: o = 8’b1111_1110;
           3’b001: o = 8’b1111_1101;
           3’b010: o = 8’b1111_1011;
           3’b011: o = 8’b1111_0111;
           3’b100: o = 8’b1110_1111;
           3’b101: o = 8’b1101_1111;
           3’b110: o = 8’b1011_1111;
           3’b111: o = 8’b0111_1111;
           default: o = 8’bx;
           endcase
    endmodule
```

Priority Encoder

```
always @ (d0 or d1 or d2 or d3)
        if (d3 == 1)
             {x, y, v} = 3'b111;
        else if (d2 == 1)
             {x, y, v} = 3'b101;
        else if (d1 == 1)
             {x, y, v} = 3'b011;
        else if (d0 == 1)
             {x, y, v} = 3'b001;
        else
             {x, y, v} = 3'bxx0;
```

inputs				outputs		
d0	d1	d2	d3	x	y	v
0	0	0	0	x	x	0
1	0	0	0	0	0	1
x	1	0	0	0	1	1
x	x	1	0	1	0	1
x	x	x	1	1	1	1

Parity Checker

```
module parity_checker (data, parity);
    input [0:7] data;
    output parity;
    reg parity;
    always @ (data)
     begin: check_parity
       reg partial;
       integer n;
       partial = data[0];
       for (n=1; n<=7; n=n+1)
        begin
           partial = partial^data[n];
        end
       parity <= partial;
     end
endmodule
```

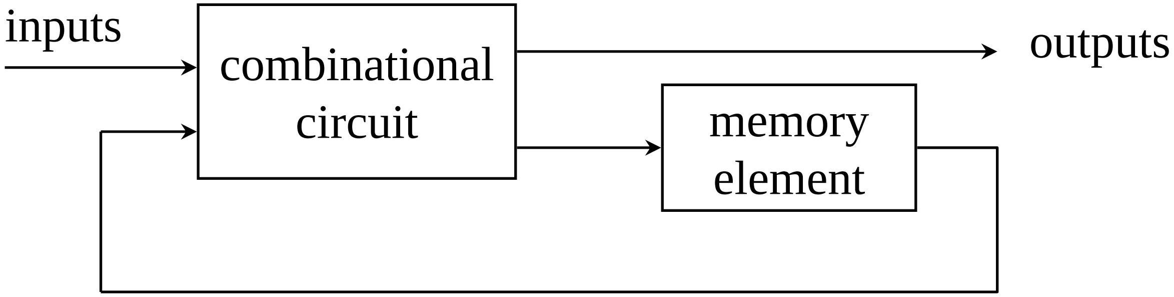
Adder

- RTL modeling

```
module adder (c, s, a, b);
    output c;
    output [7:0] s;
    input [7:0] a, b;

    assign {c, s} = a + b;
endmodule
```

- Logic synthesis
  - Carry Look-Ahead (CLA) adder for speed optimization
  - Ripple adder for area optimization

Sequential Circuit Design



- A feedback path
- The state of the sequential circuits
- The state transition
  - » Synchronous circuits
  - » Asynchronous circuits

Register Inference

- Inference of Positive Edge Triggered Flip-flops

```
module ff1 (data, clk, q);
    input [3:0] data;
    input clk;
    output [3:0] q;
    reg [3:0] q;
       always (posedge clk)
           q = data;
endmodule
```

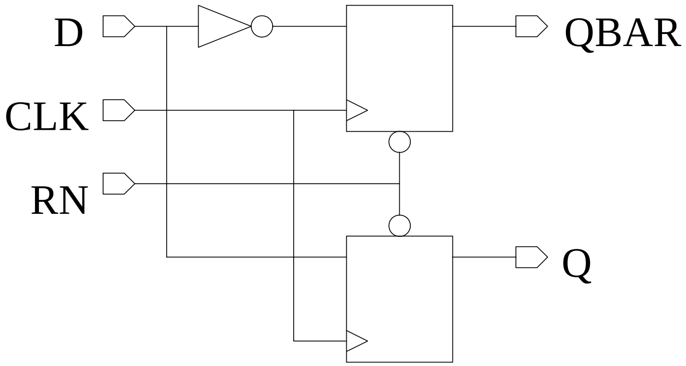
- Inference of Positive Edge Triggered Flip-flops with active low reset

```
module inf_ff (clk, reset, a, b, q);
    input clk, reset;
    input [3:0] a, b;
    output [3:0] q;
    reg [3:0] q;
       always (posedge clk or negedge reset)
           if (!reset) q = 0;                        // asynchronous input first
           else q = a&b;
endmodule
```

Variable Within Always

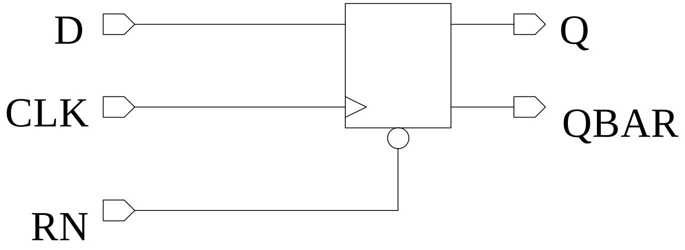
- Redundant register variables can cause extra logics

```
module BAD_DESIGN (D, CLK, RN, Q, QBAR);
      input D, CLK, RN;
      output Q, QBAR;
      reg Q, QBAR;

      always (negedge RN or posedge CLK)
      begin
            if (! RN) begin
                  Q = 1'b0;
                  QBAR = 1'b1;
              end
            else begin
                  Q = D;
                  QBAR = ~D;
              end

endmodule
```



Variable Within Always

```
module GOOD_DESIGN (D, CLK, RN, Q, QBAR);
      input D, CLK, RN;
      output Q, QBAR;
      reg Q, QBAR;

      always (negedge RN or posedge CLK)
      begin
            if (! RN)
                  Q = 1’b0;
            else
                  Q = D;
              end
      assign ABAR = ~Q;

endmodule
```



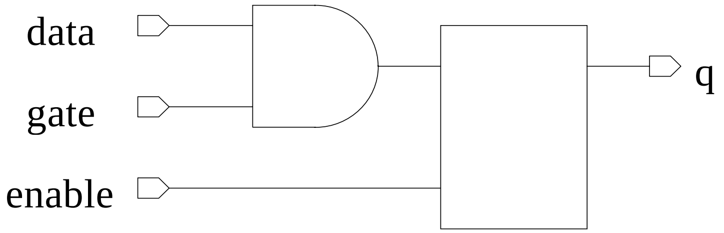
D Latches

- D latch

```
always @ (enable or data)
  if (enable)
        q = data;
```

- D latch with gated asynchronous data

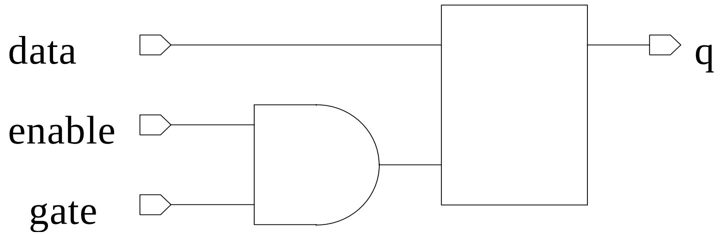
```
always @ (enable or data or gate)
  if (enable)
        q = data&gate;
```



Latches

- D latch with gated “enable”

```
always @ (enable or data or gate)
  if (enable&gate)
      q = data;
```



- D latch with asynchronous reset

```
always @ (reset or data)
  if (reset)
      q = 1’b0;
  else if (enable)
      q = data;
```

Latch Inference

- What if g is false?

```
module latch4 (d, en, g, k);
    input [3:0] d;
    input en, g;
    output [3:0] k;
    reg [3:0] k;
      always (d or en or g)
          if (g)
              k = d & {en, en, en, en};
endmodule
```

- y and z are not fully specified

```
always @ (control or a or b or c) begin
    if (control > 2)
      begin
          x = a;
          y = b;
          z = c;
      end
    else x = b;
end
```

Latch Inference

- What if bcd is 5?

```
wire [3:0] bcd;
...
case (bcd)
    4’d0: begin zero = 1’b1; {one, two} = 2’b0; end
    4’d1: begin {zero, two} = 2’b0; one = 1’b1; end
    4’d2: begin {zero, one} = 2’b0; two = 1’b1; end
Endcase
```

- Using default to prevent latch inference

```
wire [3:0] bcd;
...
case (bcd)
    4’d0: begin zero = 1’b1; {one, two} = 2’b0; end
    4’d1: begin {zero, two} = 2’b0; one = 1’b1; end
    4’d1: begin {zero, one} = 2’b0; two = 1’b1; end
    default: {zero, one, two} = 3’bxxx;
endcase
```

Latch Inference

- Why latches are inferred?

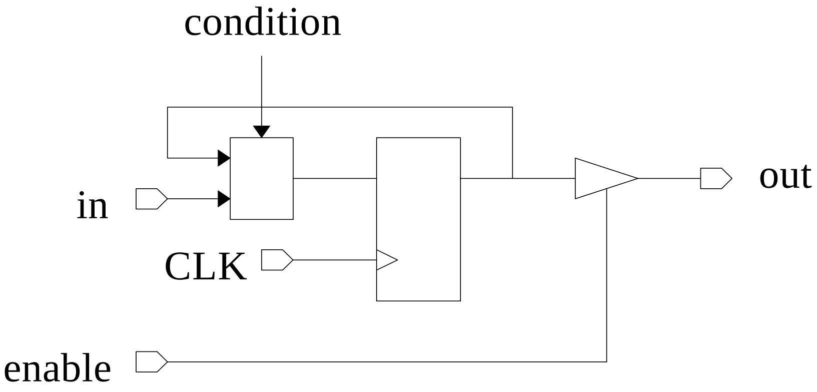
```
wire [3:0] bcd;
...
case (bcd)
    4’d0: zero = 1’b1;
    4’d1: one = 1’b1;
    4’d1: two = 1’b1;
    default: {zero, one, two} = 3’xxx;
endcase
```

- Using full_case directive to prevent latch inference

```
wire [3:0] bcd;
...
case (bcd) //synopsys full_case
    4’d0: begin zero = 1’b1; {one, two} = 2’b0; end
    4’d1: begin {zero, two} = 2’b0; one = 1’b1; end
    4’d1: begin {zero, one} = 2’b0; two = 1’b1; end
endcase
```

Registered Three-State

```
always (posedge CLK)
begin
      if (enable)
            out = (~condition) ? in : out;
      else
            out = 1'bz;
       end
```



Inefficient Description

```
module count ( clock, reset, and_bits, or_bits, xor_bits);
input clock, reset;
output and_bits, or_bits, xor_bits;
reg and_bits, or_bits, xor_bits;
reg [2:0] count;
    always @ (posedge clock) begin
      if (reset)
        count = 0;
      else
        begin
            count = count + 1;
            and_bits = &count;
            or_bits = |count;
            xor_bits = ^count;
        end
    end
Endmodule
```

- Six inferred registers

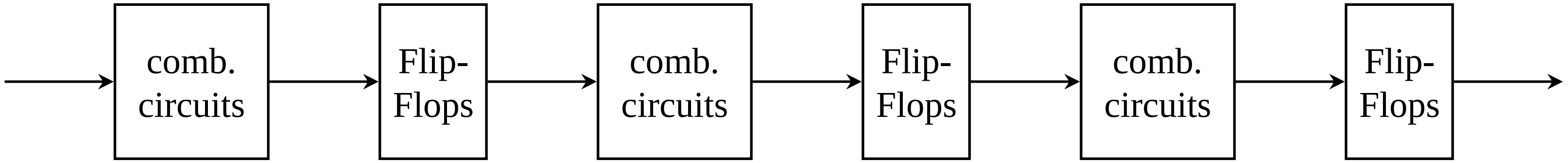
Efficient Description

- Separate combinational and sequential circuits

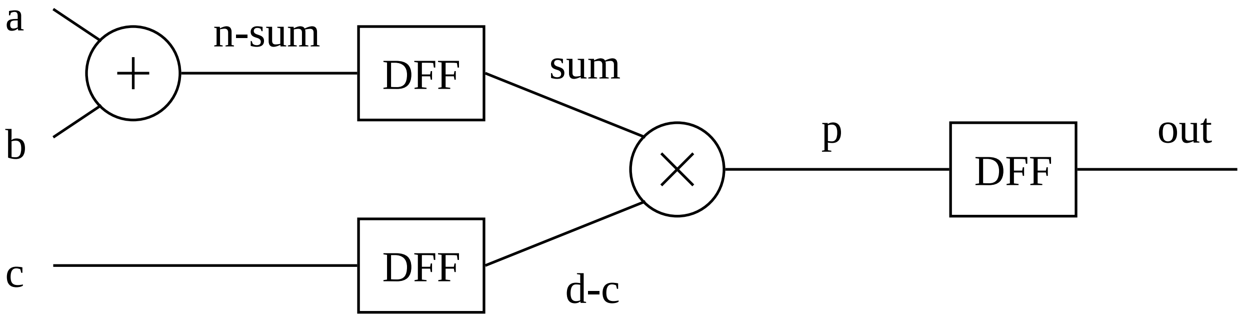
```
module count ( clock, reset, and_bits, or_bits, xor_bits);
input clock, reset;
output and_bits, or_bits, xor_bits;
reg and_bits, or_bits, xor_bits;
reg [2:0] count;
    always @ (posedge clock) begin
       if (reset)
          count = 0;
       else
          count = count + 1;
    end
    always @ (count) begin            // combinational circuits
          and_bits = &count;
          or_bits = |count;
          xor_bits = ^count;
    end
endmodule
```

- Three registers are inferred

Pipelines



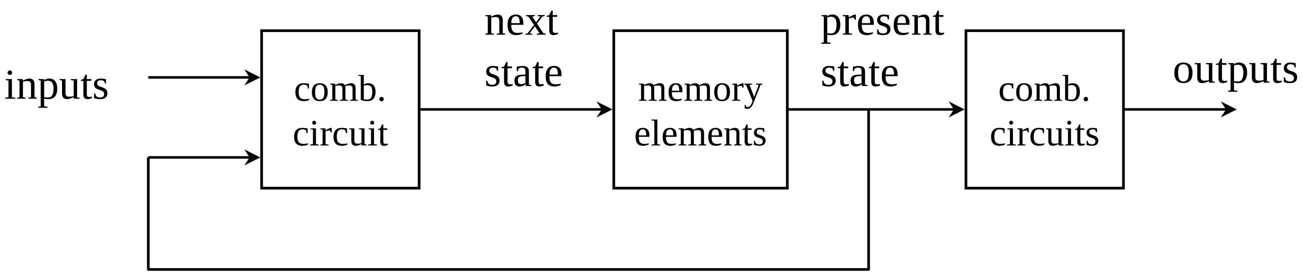
- An example



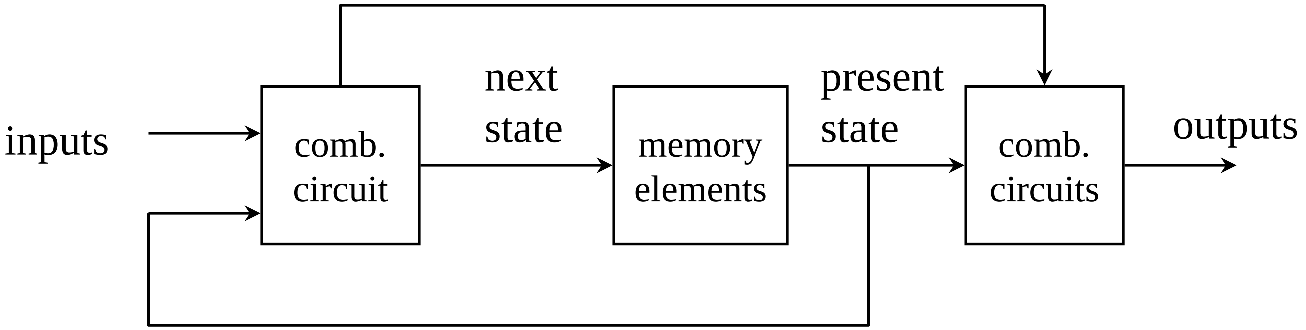
```
assign n_sum = a + b;
assign p = sum*d_c;
// plus D flip-flops
```

Finite State Machine

- Moore mode1



- Mealy model

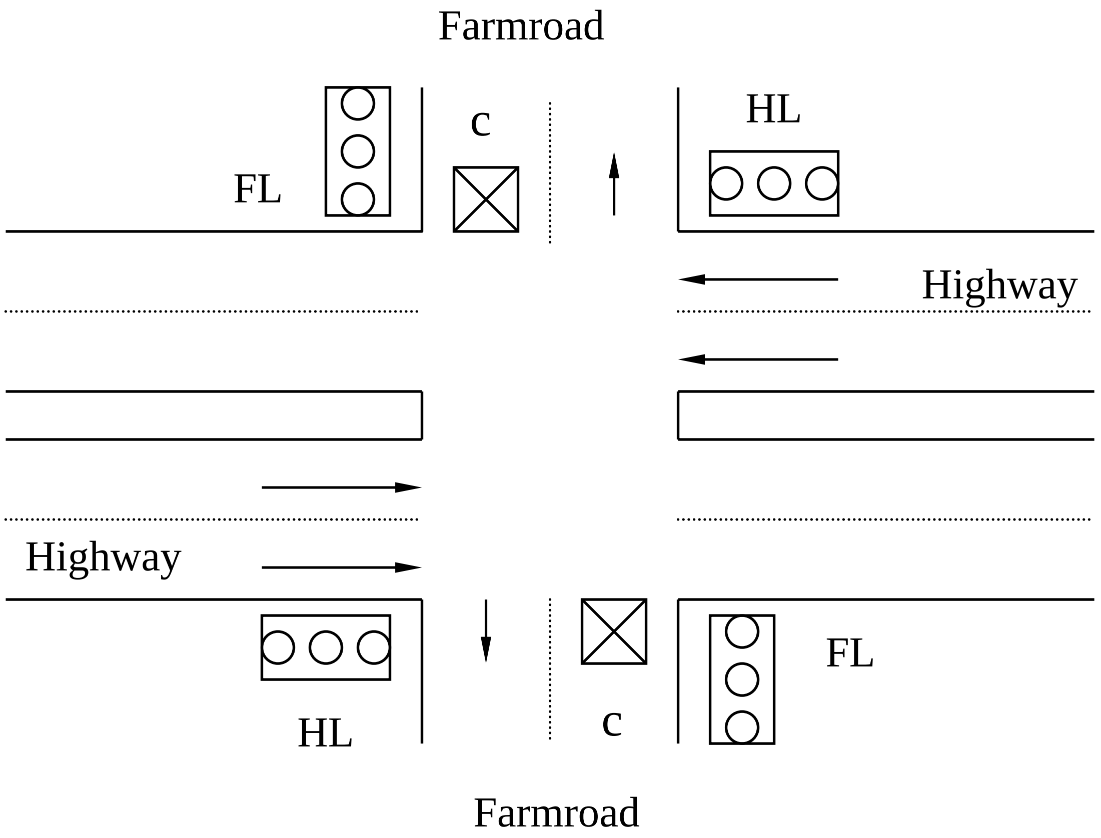


Mealy Machine Example

```
module mealy (in1, in2, clk, reset, out);
input in1, in2, clk, reset;
output out;
reg present_state, next_state, out;
      // state flip-flops
      always @ (posedge clk or negedge reset)
          if (!reset)
              present_state = 0;
          else
              present_state = next_state;
       // combinational circuits
      always @ (in1 or in2 or present_state)
          case (present_state)
              0: begin
                    next_state = 1;
                    out = 1'b0;
              end
              1: begin
                    next_state = 0;
                    out = 1'b1;
              end
          endcase
endmodule
```

A FSM Example

- Traffic Light Controller



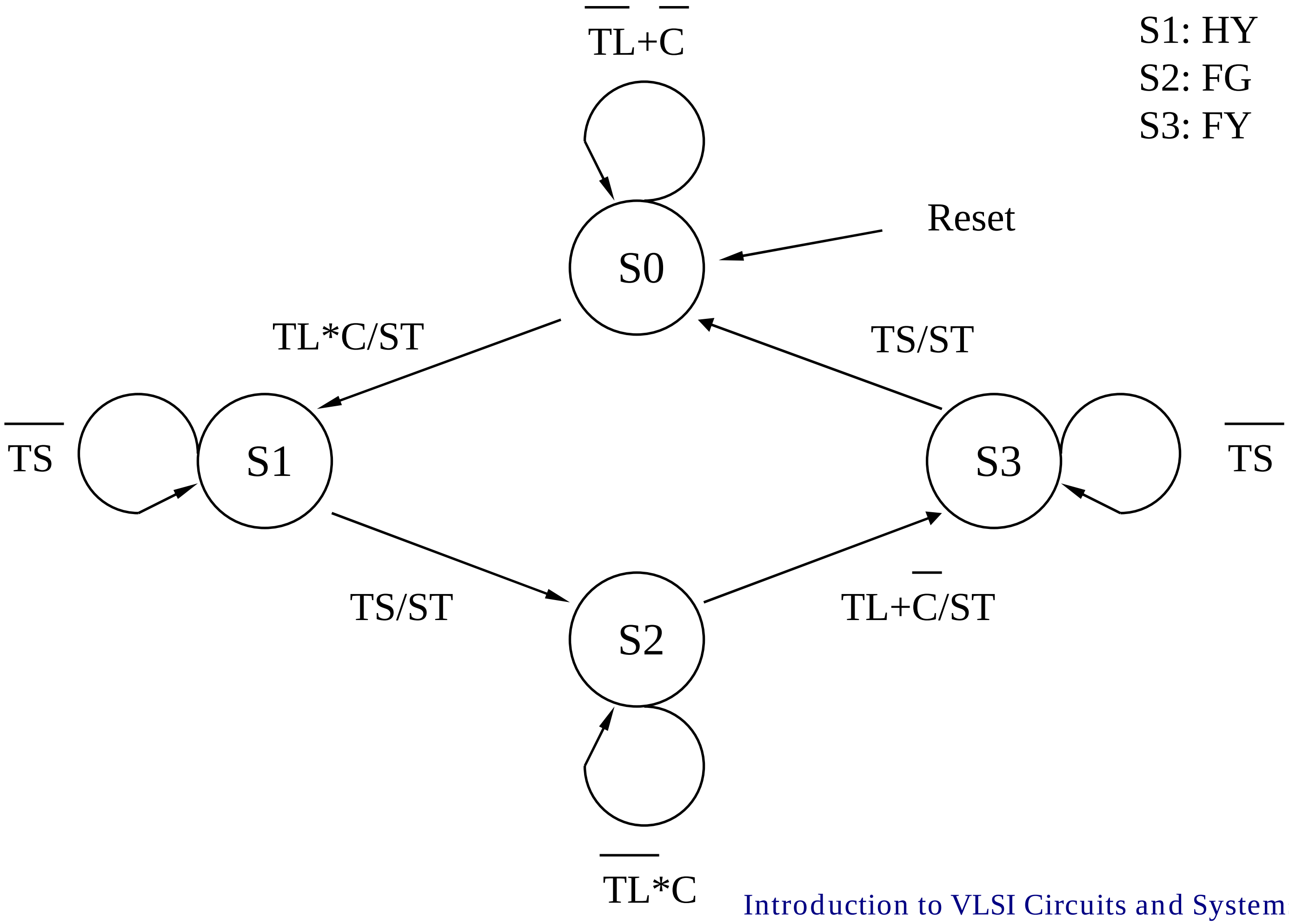
Specification

❒ Input Signal	Description
Reset	place control in initial state
C	detects vehicle on farmroad in either direction
TS	short timer interval has expired
TL	long timer interval has expired

❒ Output Signal	Description
HG, HY, HR	assert green, yellow, red highway lights
FG, FY, FR	assert green, yellow, red farmroad lights
ST	commence timing a long or short interval

❒ State	Description
S0	highway green (farmroad red)
S1	highway yellow (farmroad red)
S2	farmroad green (highway red)
S3	farmroad yellow (highway red)

State Transition Diagram



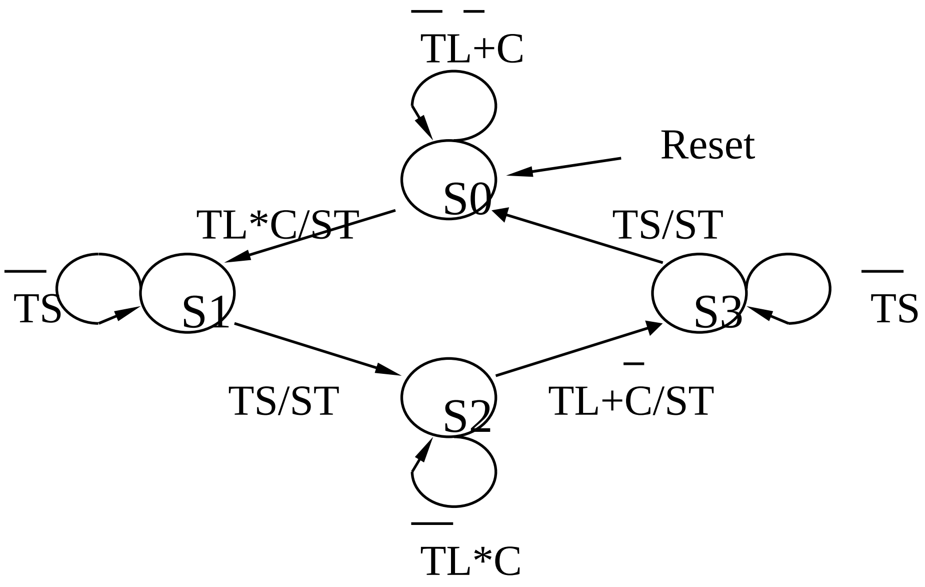
S0: HG
S1: HY
S2: FG
S3: FY

Verilog Description

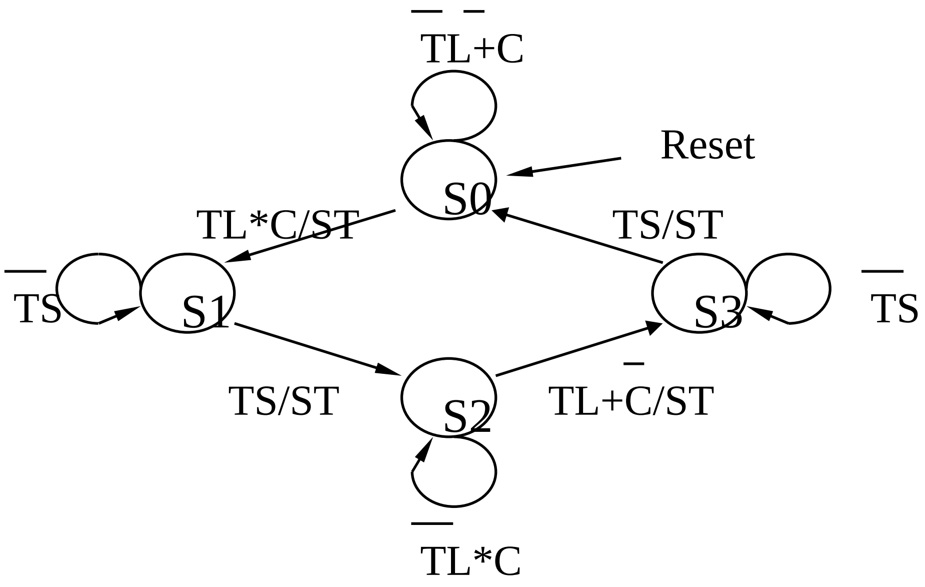
```
module traffic_light (HG, HY, HR, FG, FY, FR, ST_o, tl, ts, clk, reset, c);
output HG, HY, HR, FG, FY, FR, ST_o;
input tl, ts, clk, reset, c;
reg ST_o, ST;
reg [0:1] state, next_state;
parameter EVEN=0, ODD=1;
parameter S0=2b’00, S1=2b’01, S2=2b’10, S3=2b’11;
assign HG = (state==S0);
assign HY = (state==S1);
assign HR = (state==S2) || (state==S3);
assign FG = (state==S2);
assign FY = (state==S3);
assign FR = (state==S0) || (state==S1);
```

```
//flip-flops
always @(posedge clk or posedge reset)
     if (reset)  // an asynchronous reset
       begin
         state = S0;
         ST_o = 0;
       end
     else
       begin
         state = next_state;
         ST_o = ST;
     end
```

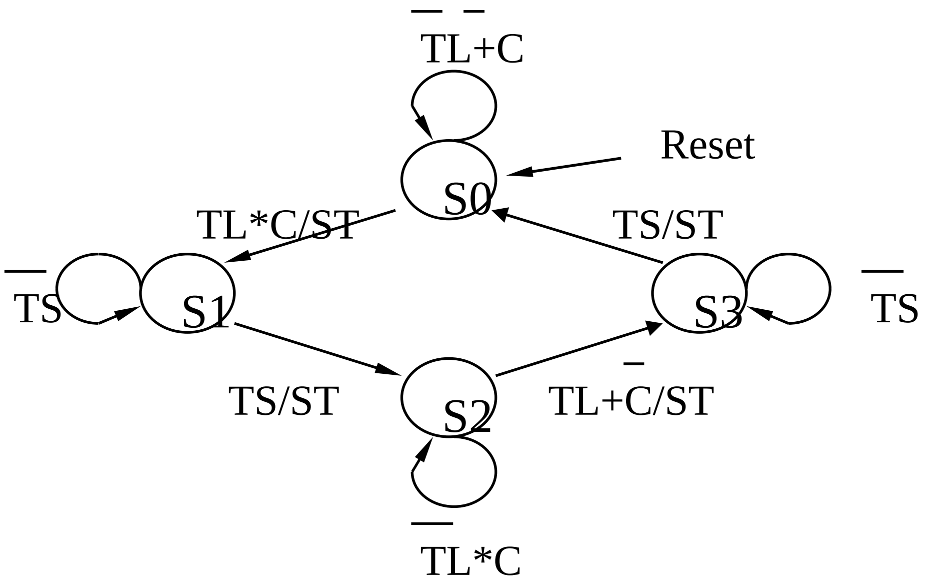
```
always @(state or c or tl or ts)
     case (state)              // state transition
     S0:
             if (tl & c)
               begin
                 next_state = S1;
                 ST = 1;
               end
             else
               begin
                 next_state = S0;
                 ST = 0;
               end
```



```
S1:
          if (ts) begin
               next_state = S2;
               ST = 1;
            end
          else begin
               next_state = S1;
               ST = 0;
            end
S2:
          if (tl | !c) begin
               next_state = S3;
               ST = 1;
            end
          else begin
               next_state = S2;
               ST = 0;
            end
```



```
        S3:
                if (ts)
                  begin
                    next_state = S0;
                    ST = 1;
                  end
                else
                  begin
                    next_state = S3;
                    ST = 0;
                  end
        endcase
endmodule
```



FSM 1

```
module FSM1(CLK, OP1, OP2);
   input CLK;
   output OP1, OP2;
   reg OP1, OP2;
   parameter S0=0, S1=1, S2=2;
   reg [1:0] STATE;
always @(posedge CLK)
   begin
      case (STATE)
       S0: begin
             OP1 = 1;
             OP2 = 1;
             STATE=S1;
          end
       S1: begin
             OP1 = 0;
             STATE=S2;
          end
       S2: begin
             OP2 = 0;
             STATE=S0;
          end
       default:
             STATE=S0;
     endcase
   end
endmodule
```

FSM 2

```
module FSM2(CONTROL, CLK, RESET, Y);
  input control, CLK, RESET;
  output [0:2] Y;
  reg [0:2] Y;
  parameter S0=0, S1=1, S2=2, S3=3;
  reg [1:0] STATE;
always @(posedge CLK)
  begin
    if (RESET) begin
          Y=0;
          STATE=S0;
        end
      else
          case (STATE)
               S0: begin
                   Y = 1;
                   STATE=S1;
                 end
               S1: begin
                   Y = 2;
                   if (CONTROL == 1)
                              STATE=S2;
                   else
                              STATE=S3;
                 end
               S2: begin
                   Y=3;
                   STATE=S3;
                 end
               S3: begin
                   Y=4;
                   STATE=S0;
               default:
                   STATE=S0;
          endcase
  end
endmodule
```

FSM 3

```
module FSM3(CONTROL, CLK, RESET, Y);
  input control, CLK, RESET;
  output [0:2] Y;
  reg [0:2] Y;
  parameter S0=0, S1=1, S2=2, S3=3;
  reg [1:0] STATE;
always @(posedge RESET or posedge CLK)
  begin
    if (RESET)
         STATE=S0;
     else
         case (STATE)
              S0: STATE=S1;
              S1: if (CONTROL == 1)
                    STATE=S2;
                  else
                    STATE=S3;
              S2: STATE=S3;
              S3: STATE=S0;
              default: STATE=S0;
         endcase
    end
 always @(STATE)
     case (STATE)
              S0: Y=1;
              S1: Y=2;
              S2: Y=3
              S3: Y=4;
              default: Y=0;
     endcase
 end

endmodule
```

FSM 4

```
module FSM4(CONTROL, CLK, RESET, Y);
  input control, CLK, RESET;
  output [0:2] Y;
  reg [0:2] Y;
  parameter S0=0, S1=1, S2=2, S3=3;
  reg [1:0] PRESENT_STATE, NEXT_STATE;
always @(PRESENT_STATE or CONTROL)
  begin
         Y=0;
         NEXT_STATE=S0;
         case (PRESENT_STATE)
              S0: begin
                    Y=1;
                    NEXT_STATE=S1;
                  end
              S1: begin
                    if (CONTROL == 1)
                             NEXT_STATE=S2;
                  else
                             NEXT_STATE=S3;
```

```
              S2: begin
                    Y=3;
                    NEXT_STATE=S3;
                  end
              S3: begin
                    Y=4;
                    STATE=S0;
         endcase
   end
 always @(posedge RESET or posedge
     CLK)
     begin
       if (RESET)
              PRESENT_STATE=S0;
       else
              PRESENT_STATE =
                             NEXT_STATE;
 end
endmodule
```

Z Handling

- To compare with “z” (High Impedance) use only equality operators
- Using “==” z is always evaluated FALSE
- Using “!=” z is always evaluated TRUE

Example: (z handling using “==” operator)

```
if (A ==’bz)
    B = 0;
else
    B = 1;
// B is always 1 (comparison evaluated FALSE)
```

Example: (z handling using “!=” operator)

```
if (A != 4’hz)
    B = 4’h0;
else
    B = 4’h1;
// B is always 0 in 4-bit hex (comparison evaluated TRUE)
```

Z Handling

- Tri-state inference
  - Tri-state buffer may be inferred from the high impedance value z.
  - When the high impedance value z is assigned to a variable, the output of the tri-state buffer is disabled.

Example: (Tri-state inference using z handling)

```
module tri_state(a, b);
   parameter N = 15;
   input [N:0] a;
   input enable;
   output [N:0] b;

   assign b = (enable) ? a : 16’bz;
endmodule;
```

X Handling

X (unknown) handling

- Similar to value “z”
- Using “==” x is always evaluated FALSE
- Using “!=” x is always evaluated TRUE

Example: (x handling using “==” operator)

```
if (A ==’bx)
    B = 0;
else
    B = 1;
// B is always 1 (comparison evaluated FALSE)
```

Example: (x handling using “!=” operator)

```
if (A != 4’hx)
    B = 4’h0;
else
    B = 4’h1;
// B is always 0 in 4-bit hex (comparison evaluated TRUE)
```

Wired-AND

- Resolves the shorting of an output wire receiving multiple assignments to an AND gate
  - » Declaring “wand”
- Example

```
module mywand (a, b, c);
    input a, b;
    output c;

    wand c;

    assign c = a;
    assign c = b;
 endmodule;
```

Wired-OR

- Resolves the shorting of an output wire receiving multiple assignments to an OR gate
  - Declaring “wor”
- Example

```
module mywor (a, b, c);
     input a, b;
     output c;

     wor c;

     assign c = a;
     assign c = b;
 endmodule;
```

Tri-State

- Shorting of an output wire receiving multiple assignments
  - Declaring “tri”
- Example

```
module mytri (a, b, c);
     input a, b, control;
     output c;

     tri c;

     assign c = (control ? a : ’bz);
     assign c = (control ? ’bz : b);
 endmodule;
```

Resolution Functions

- Example

```
module T (A, B, C, D, E, Z);
     input A, B, C, D, E;
     output Z;

     // wor BAT;
     // wand BAT;
     // tri BAT;
     wire BAT;

     assign BAT = A & B;
     assign BAT = C | D;
     assign Z = BAT | E;
 endmodule;
```

Resolution Functions

- Case 1: If BAT is a wire
  - Warning: Unable to determine wired-logic type for multiple-driver net ‘BAT’
  - Information: Assuming multiple-driver net ‘BAT’ is a wired-AND
  - Connects the drivers of BAT using an AND gate
- Case 2: If BAT is a tri
  - Warning: In design ‘T’, there is 1 three-state bus with non-three state drivers.
  - Connects the drivers of BAT using an AND gate
- Case 3: If BAT is a wand
  - Information: In design ‘T’, there is 1 wired-AND net
  - Connects the drivers of BAT using an AND gate
- Case 4: If BAT is a wor
  - Information: In design ‘T’, there is 1 wired-OR net
  - Connects the drivers of BAT using an OR gate

Resolution Functions

- Example

```
module T (A, B, C, D, E, F, G, Z);
     input A, B, C, D, E, F, G;
     output Z;

     // wor BAT;
     // wand BAT;
     tri BAT;
     // wire BAT;

     assign BAT = (F ? A&B : ’bz);
     assign BAT = (G ? C | D : ’bz);
     assign Z = BAT | E;
 endmodule;
```

Resolution Functions

- Case 5: If BAT is a tri
  - BAT is shorted after tri-state gates
- Case 6: If BAT is a wire/wand/wor
  - BAT is shorted after tri-state gates

Resolution Functions

- Example

```
module T (A, B, C, D, E, F, G, Z);
     input A, B, C, D, E, F, G;
     output Z;
     // wor BAT;
     wand BAT;
     // tri BAT;
     // wire BAT;

     assign BAT = (F ? A&B : ’bz);
     assign BAT = C | D;
     assign Z = BAT | E;
 endmodule;
```

- Case 7: BAT is a wand/wor/tri/wire
  - Warning: In design ‘T’, there is 1 three-state but with non three-state drivers
  - BAT drivers are shorted

Resolution Functions

- Example

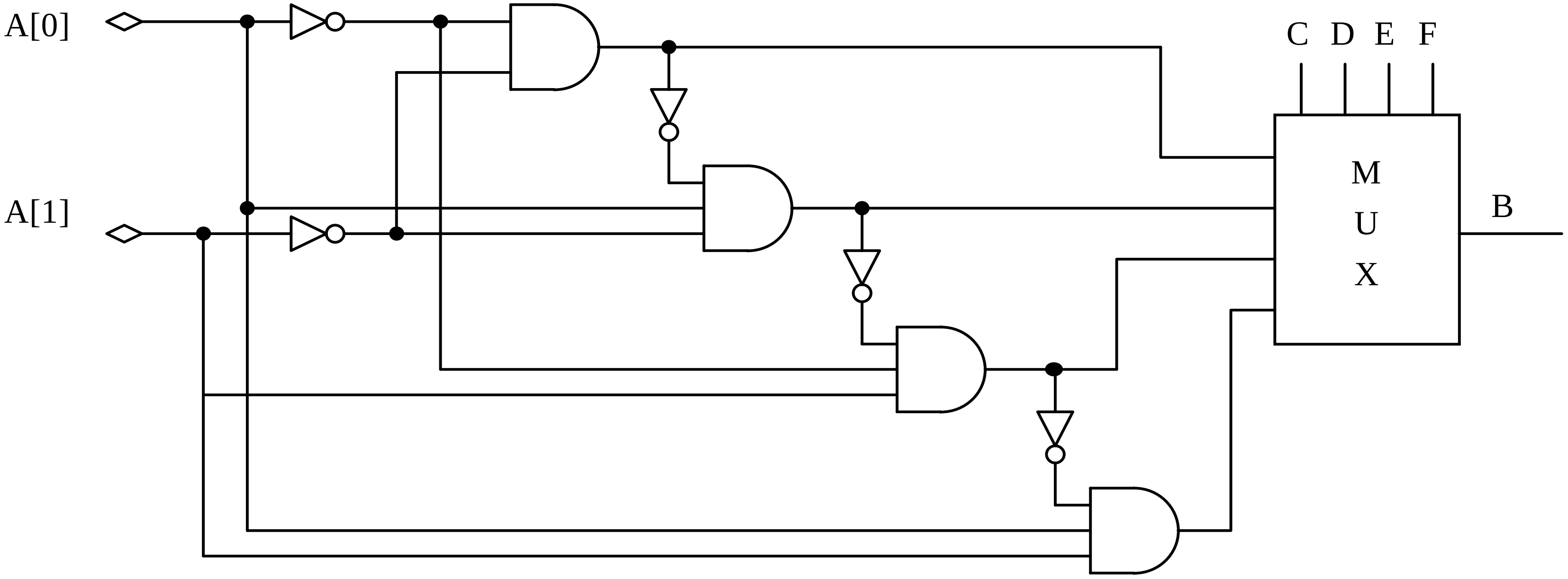
```
module T (A, B, C, D, E, F, G, H, J, Z);
    input A, B, C, D, E, F, G, H, J;
    output Z;
    wor BAT;
    // wand BAT;
    // tri BAT;
    // wire BAT;

    assign BAT = (F ? A&B : ’bz);
    assign BAT = C | D;
    assign BAT = H ^ J;
    assign Z = BAT | E;
 endmodule;
```

- Case 8: BAT is a wand/wor/tri/wire
  - » 2nd and 3rd drivers are AND’ed. Could be a bug

If-Else Statement

```
if (A[0] == 0 && A[1] == 0)
    B = C;
else if (A[0] == 1 && A[1] == 0)
    B = D;
else if (A[0] == 0 && A[1] == 1)
    B = E;
else
    B = F;
```

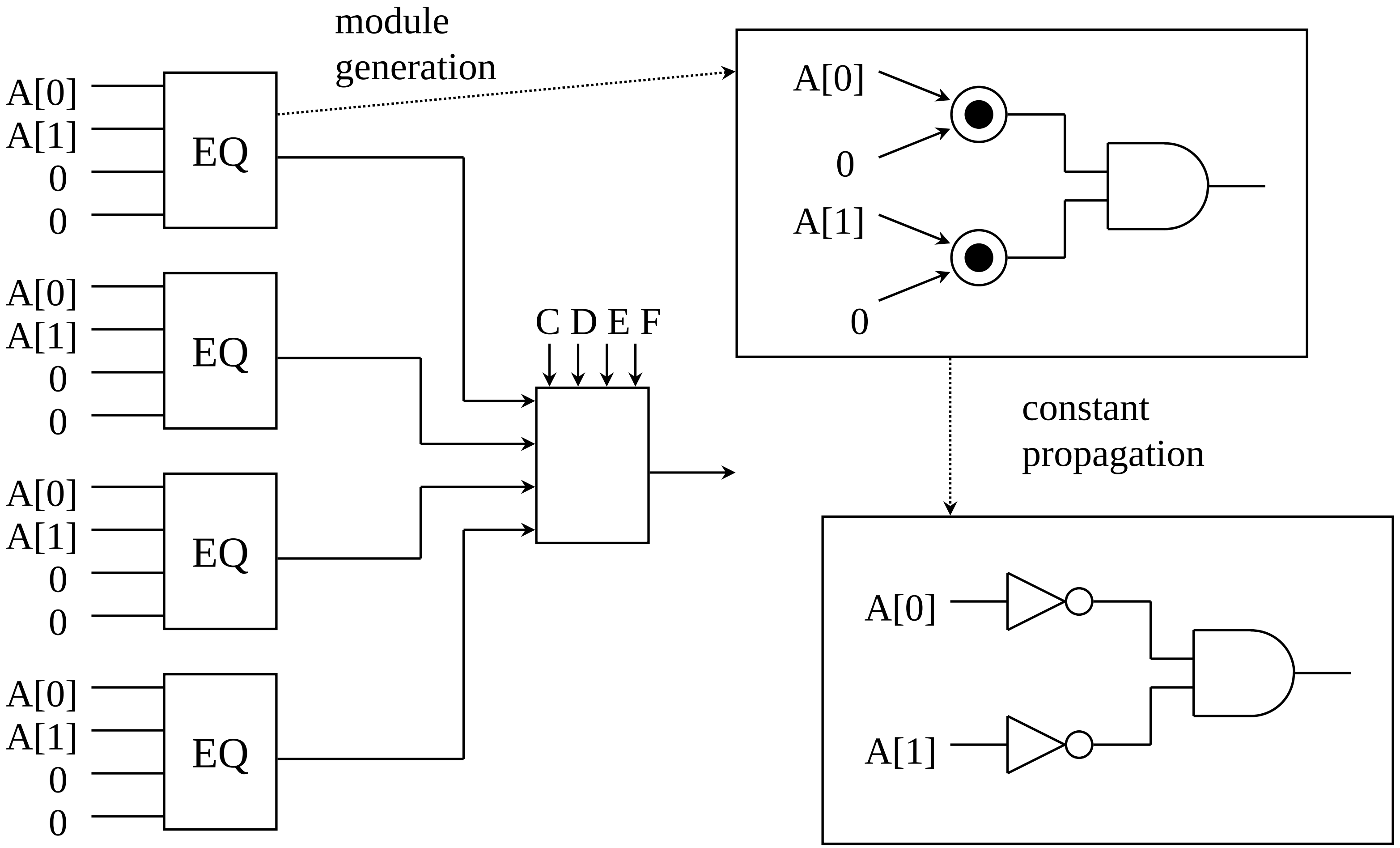
If-Else Hardware Implementation



Case Statement

```
case (A)
    2’b00: B = C;
    2’b01: B = D;
    2’b10: B = E;
    2’b11: B = F;
endcase
```

Case Hardware Implementation



Modeling for Optimization

- Sharing common subexpressions
  - Data path
  - Common block of logic
  - Sharing between synchronous and combinational sections
- Adding Structure
  - Sharing resource explicitly
  - Using parentheses
  - Detailing the logic structure
- Using design knowledge
  - Bit-width calculation
  - Constant propagation
- Understanding hardware implication
  - Hardware implication over software efficiency

Common Subexpression

- Don’t repeatedly calculate the same operation
  - Use temporary assignment
- Original

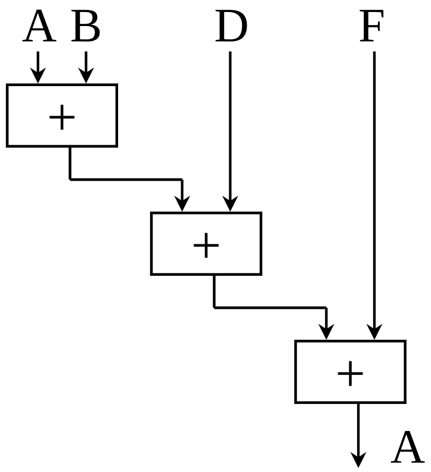
```
A = B + C + D;
D = F + C + B;
```

- Modified

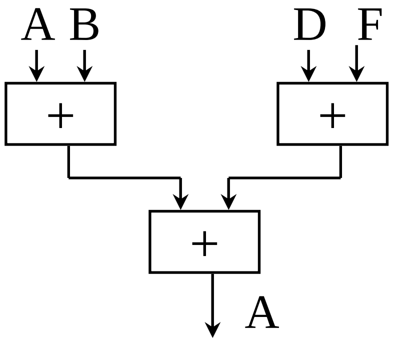
```
E = B + C;
A = E + D;
D = F + E;
```

Using Parenthesis

A = B + C + D + F;



A = (B + C) + (D + F);



Constant Propagation

- Original style

```
if (RST == ’b0) begin
    A = 11 - 2;
    B = A + 1
    D = 12 - B;
    C = 2 * D;
  end
else
  C = B;
```

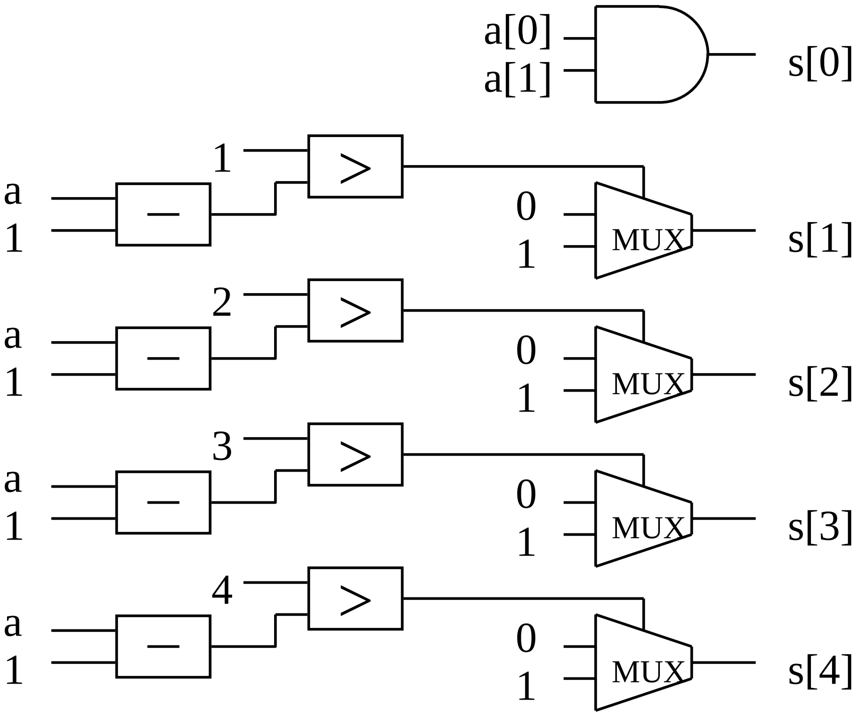
- Better style

```
if (RST == ’b0)
  C = 4;
else
  C = 10;
```

Control of Logic Network

- The most straightforward coding is not always the best choice for actual hardware implementation

```
/* Verilog Version */
module worst_ver (a, s);
      input [4:0] a;
      output [4:0] s;
      reg [4:0] s; integer 1;
  always @ (a) begin
      if (a == 0) s = 5'b11110;
      else begin
          s[0] = a[0] && a[1];
          for (I=1; I <= 4; I=I+1)
             if (I > a-1'b1)
                s[I] = 1'b1;
             else
                s[I] = 1'b0;
      end
  end
endmodule
```

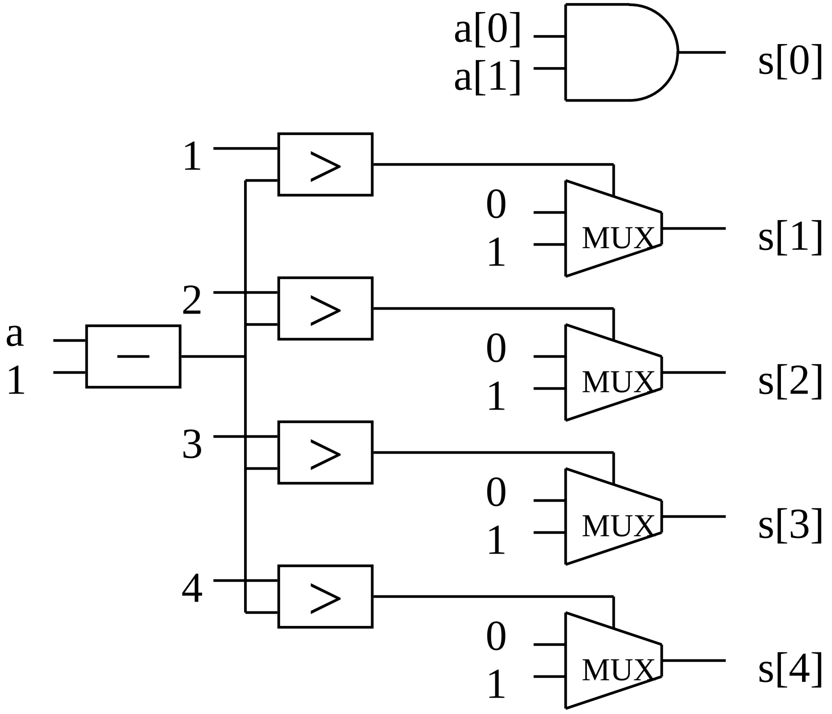


Generic logic schematic

Control of Logic Network

- Adding structure information by introduction of temporary variables

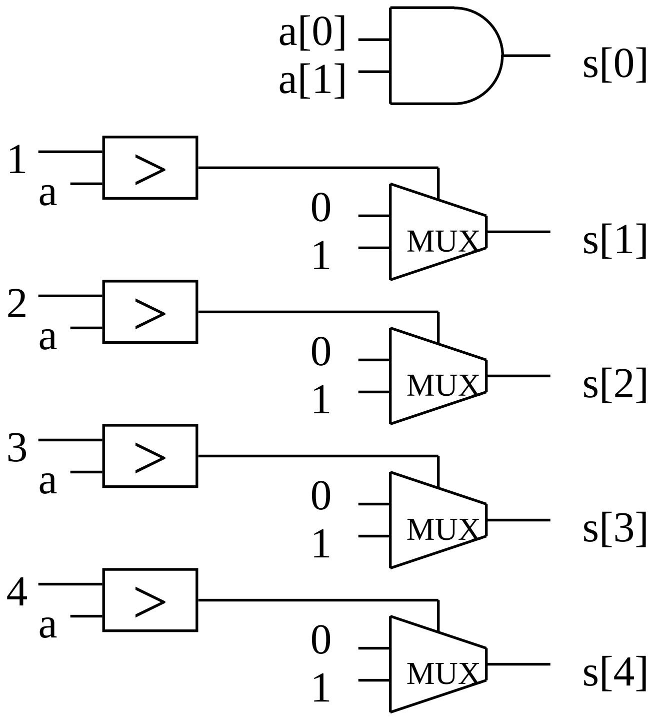
```
/* Verilog Version */
module worse_ver (a, s);
     input [4:0] a;
     output [4:0] s;
     reg [4:0] s, tmp;
     integer 1;
  always @ (a) begin
     if (a == 0) s = 5'b11110;
     else begin
         s[0] = a[0] && a[1];
         tmp = a-1;
         for (I=1; I <= 4; I=I+1)
            if (I > tmp) s[I] = 1'b1;
            else s[I] = 1'b0;
     end
  end
endmodule
```



Control of Logic Network

- Move constants to one side

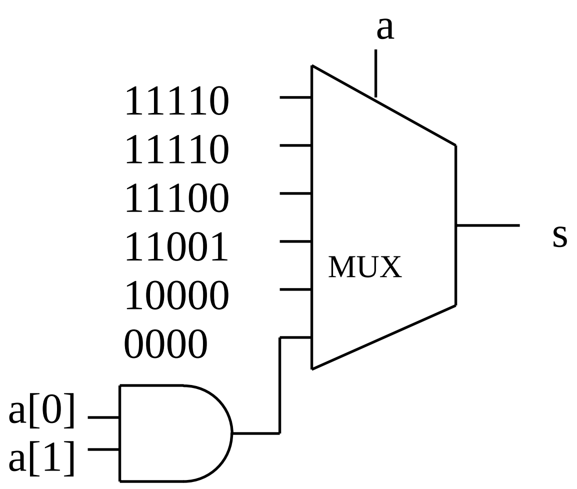
```
/* Verilog Version */
module good_ver (a, s);
      input [4:0] a;
      output [4:0] s;
      reg [4:0] s;
      integer 1;
  always @ (a) begin
      if (a == 0) s = 5’b11110;
      else begin
          s[0] = a[0] && a[1];
          for (I=1; I <= 4; I=I+1)
             if (I+1 > a)
                 s[I] = 1’b1;
             else
                 s[I] = 1’b0;
      end
  end
endmodule
```



Control of Logic Network

- Think more as a logic specification than a function specification

```
/* Verilog Version */
module better_ver (a, s);
      input [4:0] a;
      output [4:0] s;
      reg [4:0] s;
      integer 1;
  always @ (a) begin
      case (a)
          5'b00000: s = 5'b11110;
          5'b00001: s = 5'b11110;
          5'b00010: s = 5'b11100;
          5'b00011: s = 5'b11001;
          5'b00100: s = 5'b10000;
          default: s={4'b0000,a[0]&&a[1]};
      endcase
  end
endmodule
```



Generic logic schematic

Hardware Implication

- Smart comparison

```
if (CTRL >= 3'b100)
if (CTRL[2] == 1'b1)
```

- Multiplication vs. shifting

```
A * 4
A << 2
```

- Unnecessary operation

```
if ((cnt+1) == 120)
if (cnt == 119)
```

Complex Operation

- Multiplication, division and remainder by variables or constants (not power of 2)
- Addition and subtraction by variables
- Shifting by non-computable amount
- Non-computable array indexing
- Comparison with variables
- Preferred actions:
  - Avoiding
  - Simplifying
  - Sharing