DR. TIM BROTHERS

VERILOG PRIMER

Contents

List of Figures

List of Tables

Verilog Primer: Working with Number in Verilog

Numbering Systems

1. Decimal base 10 (represented with 0 to 9)
   (a) $327_{10} = (3 * 10^2) + (2 * 10^1) + (7 * 10^0)$
2. Binary base 2 (represented with 0 to 1)
   (a) $101_2 = (1 * 2^2) + (0 * 2^1) + (1 * 2^0) = 5_{10}$
3. Octal base 8 (represented with 0 to 7)
   (a) $735_8 = (7 * 8^2) + (3 * 8^1) + (5 * 8^0) = 477_{10}$
4. Hexadecimal (Hex) base 16 (represented with 0 to F)
   (a) $3B4_{16} = (3 * 16^2) + (11 * 16^1) + (4 * 16^0) = 948_{10}$

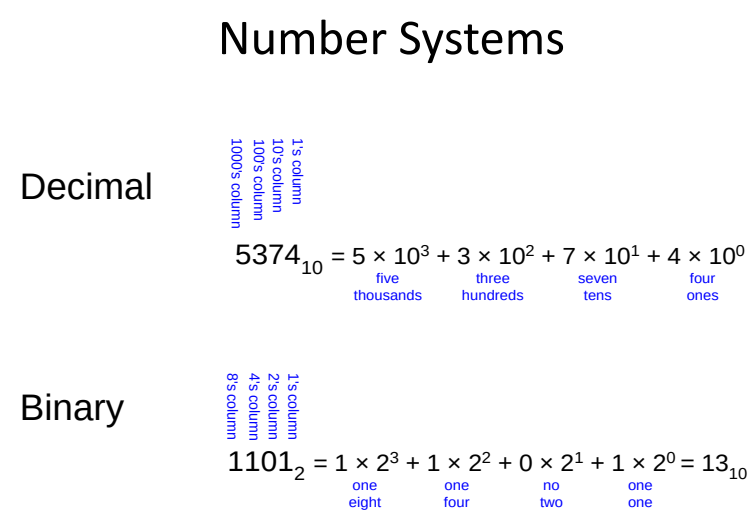


Figure 1: Comparison between different numbering systems

Number-Base Conversions

Convert 29_{10} to Binary, Octal, and Hex:

For binary, keep dividing the integer part by 2. The remainder values tell us the binary equivalent is:
$\boxed{29_{10} = 11101_2}$

1. $\frac{29}{2} = 14$ with a remainder of 1
2. $\frac{14}{2} = 7$ with a remainder of 0
3. $\frac{7}{2} = 3$ with a remainder of 1
4. $\frac{3}{2} = 1$ with a remainder of 1
5. $\frac{1}{2} = 0$ with a remainder of 1 <= Most Significant Bit (MSB)

Octal and Hex

Once you have the binary equivalent, octal and hex conversions are very easy. **Convert to binary first, then to octal or hex**

Octal

Organize the binary result into groups of 3 bits (pad with 0 as needed): $29_{10} = 11101_2 = 011\_101_2 = 35_8$

Hex (i.e., hexadecimal)

Organize the binary result into groups of 4 bits (pad with 0 as needed): $29_{10} = 11101_2 = 0001\_1101_2 = 1D_{16}$

What about Fractions

Convert 0.6875_{10} to Binary:

For binary, keep **MULTIPLYING** the integer part by 2. The integer values tell us the binary equivalent is:
$\boxed{0.6875_{10} = 0.1011_2}$

1. $0.6875 * 2 = 1.3750$ Integer value is 1 <= MSB
2. $0.375 * 2 = 0.75$ Integer value is 0
3. $0.75 * 2 = 1.5$ Integer value is 1
4. $0.5 * 2 = 1.0$ Integer value is 1
5. $0.0 * 2 = 0$ Integer value is 0

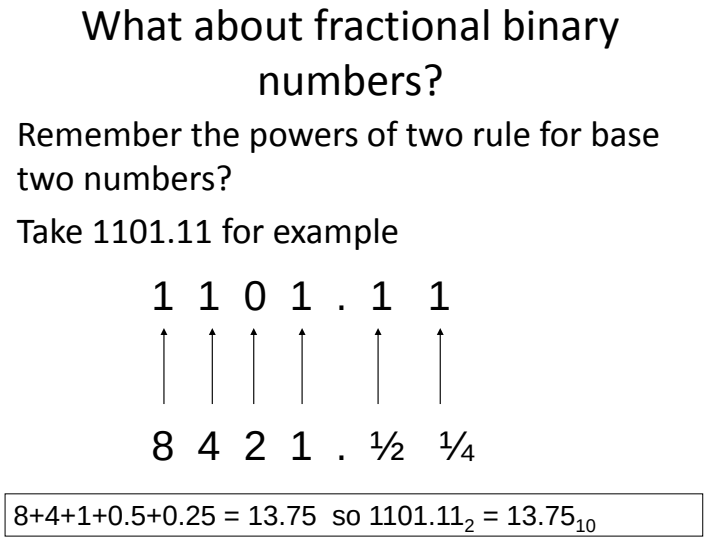


Figure 2: Convert a Binary Fraction to Decimal

Numbers and Verilog

Verilog Sentax: $8'hA3$

A number followed by an apostrophe denotes the number of bits used to hold the value of that number.
A letter denotes the base used to represent the number (where h, o, b, and d are the most commonly used variants)
The digits or letters express the number in the designated base

So $8'hA3$ is an 8-bit number, expressed in hex, with a value of $A3$. The binary equivalent of this would be 1010_0011, where the underscore between the 4-bit groups is just for readability.

What is the binary equivalent of $6'o72$? How about $8'o72$? How about $16'hFFFF$?

Note: hex is used so often for so many languages, there is another abbreviation you will see: ox $8'hA3$ may also be seen as $0xA3$. This is NOT Verilog syntax, however. The number of bits is only implied when using the $0x$ format.

Signed Numbers

There are several ways to encode signed numbers:

1. Signed Magnitude
   (a) Used for Floating Point Numbers
2. 1's Complement
3. 2's Complement

(a) Most Common way to store signed numbers

(b) Convert a number to 2's Compliment

i. Invert ones and zeros

ii. Add 1

Example: $-5_{10} = 0101_2$
Invert ones and zeros: 1010_2
Add 1: $1010_2 + 0001_2 = 1011_2$

Table 1 lists the 2's complement numbers for 3 bits. Note the max value is +3 and the minimum is -4. In general for n bits and 2's compliment:

$$\text{max value is } 2^{n-1} - 1 \text{ min value is } -2^{n-1}$$

Decimal	2's Comp
$+3$	011
$+2$	010
$+1$	001
0	000
-1	111
-2	110
-3	101
-4	100

Table 1: Table of 2's compliment numbers.

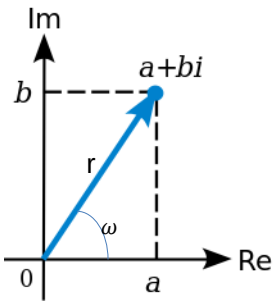
Complex Numbers

Complex numbers are summarized in Figure 3.

Addition, Subtraction, and Multiplication

Addition is summarized in Figure 4. Subtraction is summarized on Figures 5 and 6. One method for binary multiplication is shown in 7. There are many methods for computing multiplication but division is very hard. Division should be avoided if at all possible in FPGA programming. The one exception is right shifts and left shifts.

Figure 3: Complex Numbers



Polar: $r\ e^{i\omega}$ where r = $\sqrt{a^2 + b^2}$
Polar: $r\ angle(\omega)$

Rect: $a + ib\ or\ a + jb$

Figure 4: Binary Addition

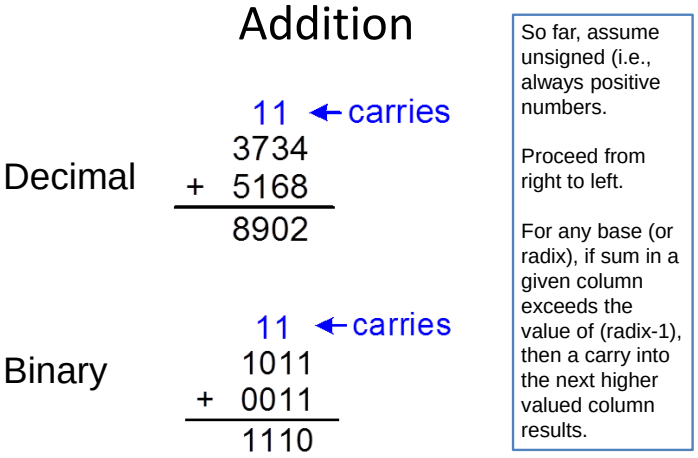


Figure 5: Binary Subtraction

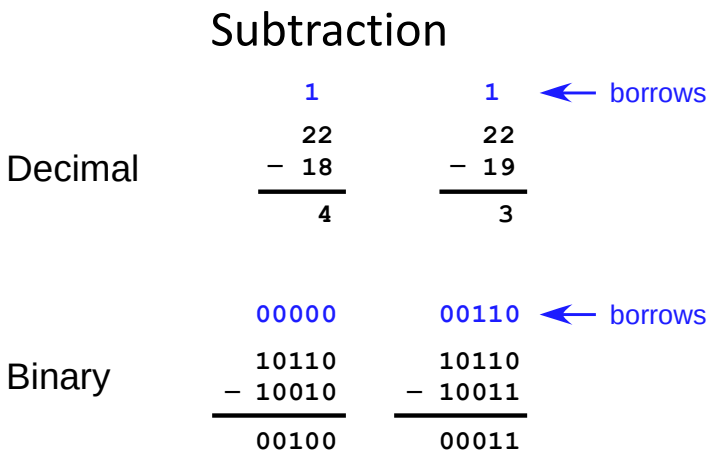


Figure 6: Binary Subtraction Expanded

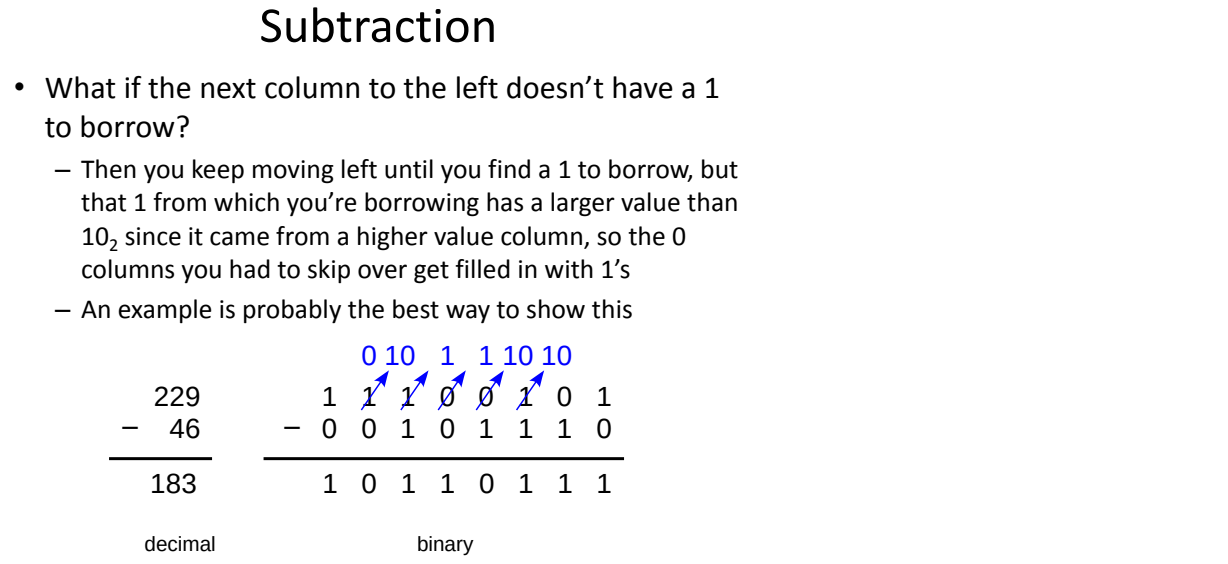
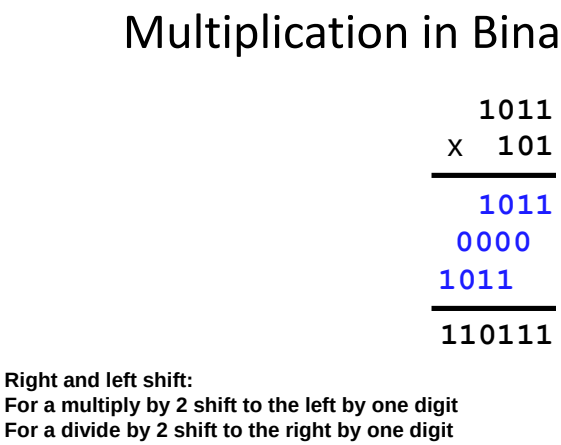


Figure 7: Binary Multiplication



Right and left shift:
For a multiply by 2 shift to the left by one digit
For a divide by 2 shift to the right by one digit

Verilog Primer: Sythesizable Code

Verilog Structure

There are two ways to compile code

1. Simulation
   (a) Simulation does not go into the Chip.
   (b) Simulation is used to test a design.
2. Synthesis
   (a) The code that actually runs on the chip.
   (b) Objective of this class is to write code that can be Synthesized.

There are several rules that must be followed to ensure code is synthesizable. The major stumbling points are listed below.

1. Do not generate latches
   (a) Latches are generated when all possible paths are not covered
      i. For every "if" or "case" define all signals for all possibilities of the input conditions. The best way to do this is to:
         A. Any signal assigned in an "if" condition must be assigned in all branches of the "if else" structure. It also must be assigned in an "else" condition.

```
always@(A or B or C)
        begin: example_block
                // If Statement
                if (A == 1'b1) begin
                        output_1 = 1'b1;
                        output_2 = 1'b0;
                        output_3 = 1'b0;
                        end
                else if (B == 1'b1) begin
                        output_1 = 1'b0;
```

```
                output_2 = 1'b1;
                output_3 = 1'bo;
                end
        else if (C == 1'b1) begin
                output_1 = 1'bo;
                output_2 = 1'bo;
                output_3 = 1'b1;
                end
        else begin
                // Default Conditions
                output_1 = 1'bo;
                output_2 = 1'bo;
                output_3 = 1'bo;
                end

end
```

B. Any signal assigned in a "case" must be assigned in all branches of the "case" structure. It also must be assigned in a "default" condition.

```
always@(A or B or C) begin: example_block
        // Case Statement
        case (A & B & C)
                3'b100: begin
                        output_1 = 1'b1;
                        output_2 = 1'bo;
                        output_3 = 1'bo;
                        end
                3'bo10: begin
                        output_1 = 1'bo;
                        output_2 = 1'b1;
                        output_3 = 1'bo;
                        end
                3'boo1: begin
                        output_1 = 1'bo;
                        output_2 = 1'bo;
                        output_3 = 1'b1;
                        end
                default: begin
                        // Default Conditions
                        output_1 = 1'bo;
                        output_2 = 1'bo;
                        output_3 = 1'bo;
                        end
```

```
        endcase
```

ii. Another way of doing this is to assign the default conditions before the "if" or "case" structure.

A. If statement example:

```
always@(A or B or C)
        begin: example_block
                // Default Conditions
                output_1 = 1'b0;
                output_2 = 1'b0;
                output_3 = 1'b0;

                // If Statement
                if (A == 1'b1)
                        output_1 = 1'b1;
                else if (B == 1'b1)
                        output_2 = 1'b1;
                else if (C == 1'b1)
                        output_3 = 1'b1;
        end
```

B. Case statement example

```
always@(A or B or C) begin: example_block
        // Default Conditions
        output_1 = 1'b0;
        output_2 = 1'b0;
        output_3 = 1'b0;

        // Case Statement
        case (A & B & C)
                3'b100: output_1 = 1'b1;
                3'b010: output_2 = 1'b1;
                3'b001: output_3 = 1'b1;
                default: begin
                        // Default Conditions
                        output_1 = 1'b0;
                        output_2 = 1'b0;
                        output_3 = 1'b0;
                        end
                endcase
        end
```

2. For combinational logic include all signals that are read inside the block in the sensitivity list. The sensitivity list is the list of signals

following the "always" keyword. The sensitivity list triggers the action inside the block. It must include all signals that are compared inside an "if" or "case" statement. It must also include all signals on the right hand side of the assignment operator.

(a) Example:

```
always@(A or B or C) begin: example_block
        // Default Conditions
        output_1 = A;

        // If Statement
        if (B)
                output_1 = C;
end
```

(b) For a register block the sensitivity list should include the clock preceded by the posedge or negedge keyword. It is possible to include a reset in the sensitivity list but it is not good design practice.

```
always@(posedge clk) begin: example_block
        if (reset)
                example_reg <= 1'b0;
        else
                example_reg <= example_sig;
end
```

3. Delays are not sythesizable. Using a delay such as #5 will result in a simulated delay, but does not synthesize.

4. Loops of undefined lengths are not sythesizable. Using a "while" loop or a "for" loop of a variable size can be useful in simulation, but does not synthesize. A "for" loop of a known size will replicate hardware for every iteration of the "for" loop.

5. Initial Blocks are sometimes sythesizable. General rule of thumb is to only use initial blocks in testbeds.

6. Forever Blocks are not synthesizable. I personally do use use forever blocks. But they can be useful in testbeds.

7. Clocks cannot be generated in synthesizable code. Clocks can be generated in a testbed for simulation, but for an actual design the clock must be provided from an off-chip source.

8. Only assign a variable in ONE block.

9. A Module must have at least one input and one output to be sythesized.

   (a) A Module with no inputs or outputs can be used in simulation only.

Verilog Primer: Verilog Building Blocks

Signals

Verilog variable types:

1. Nets
   (a) wire
   (b) reg
   (c) lots of other unsued nets such as wor and tri.
2. parameter
   (a) Very similar to a constant in programming
   (b) Very powerful in HDL programming.
   (c) Used to make code adaptable without hardcoding values
3. integer
   (a) general purpose variables for use in loops.
   (b) by default they are signed and produce 2's compliment results

Rules for reg and wire declaration:

1. Rule 1
   (a) A Net must be declared as a reg if it is on the left hand side of a signal assignment made inside an always, initial, or forever block.
2. Rule 2
   (a) A Net must be declared as a wire if it is on the left hand side of an "assign" statement.
3. Rule 3
   (a) When a Net is used to to instantiate a module it is a wire unless Rulr 1 applies.

Verilog is based on triggering events. Blocks are driven by events. The following are the main building blocks of the code structure.

1. assign statements

   (a) Simple statements are accomplished using the assign statement.

```
assign output = enable ? (A & B) : 1'bo;
```

2. Module Instantiation

   (a) Heigharcy can be acheived by instantiating multiple modules into the design. There are two ways to instantiate modules. By name and by possition. Here is an example of module instantiation by name

```
ClockSync_cont #(
    .bitWidthIn(bitWidth)
  )
  DUT
  (
        //inputs
    .clk(clk),
    .real_in(real_data_in),
    .imag_in(imag_data_in),

        //outputs
    .phase_out(data_out),
   .freq_error_out(freq_error_out)
);
```

   Here is an example of module instantiation by possition

```
ClockSync_cont #(bitWidth) DUT (
        clk,real_data_in,imag_data_in,
        data_out,freq_error_out);
```

   Modules can also be instantiaged using a generate block

```
module generate_example();
        parameter clock_cycle = 20;

        reg clk = 0;
        reg read = 1;
        reg write = 1;
        reg [31:0] data_in  = 0;
        reg [3:0] address = 0;
        wire [31:0] data_out;
```

```
        wire valid_out;

        genvar i;

        always @ (posedge clk)
        begin
                data_in <= data_in+1'b1;
                address <= address +1'b1;
        end


        generate
          for (i=0; i < 4; i=i+1) begin : MEM
             memory U (clk, read, write,
                    data_in[(i*8)+7:(i*8)],
                    address,data_out[(i*8)+7:(i*8)]);
          end
        endgenerate

        always #(clock_cycle / 2) clk = !clk; // Clock generator
endmodule

// Lower module that will be connected muliple times
module memory (
        input wire clk,
        input wire read,
        input wire write,
        input wire [7:0] data_in,
        input wire [3:0] address,
        output wire [7:0] data_out);

        reg [7:0] mem_reg [0:15];

        always @ (posedge clk)
        begin
                if(write) mem_reg[address] <= data_in;
        end

        assign data_out = mem_reg[address];
        assign valid_out = read;
endmodule
```

3. always block
   (a) Most used block

   (b) Triggered by a sensitivity list

4. initial block

   (a) **Sometimes** synthesizable.

   (b) Mostly used for TestBed

   (c) Some tools use this for register initialization

5. forever block

   (a) Not Synthesizable

   (b) I have never used a forever loop

If statements, Case statements, and Loops

1. if else block

   (a) Creates cascaded logic

   (b) always cover all cases for the if/else if/ else condition.

2. Case

   (a) generates a mux

   (b) good use of resorces

3. casex

   (a) case with don't cares. Very useful

   (b) allows both high impedance (Z) and unknown (X) values to be treated as "don't care" conditions

4. casez

   (a) case with don't cares. Not as useful as casexl

   (b) allows both high impedance (Z) values to be treated as "don't care" conditions

5. while

   (a) not synthesizable

6. for loop

   (a) creates copies of the same logic over and over again

   (b) if it is implemented correctly it can by synthesizable

```
always @ (opcode or a or b or c)
        casex(opcode)
        4'b1zzx : begin // Don't care  2:0 bits
                out = a;
                $display("@%0dns 4'b1zzx is selected , opcode %b",$time,opcode);
                end
        4'b01?? : begin // bit 1:0 is don't care
                out = b;
                $display("@%0dns 4'b01?? is selected , opcode %b",$time,opcode);
                end
        4'b001? : begin // bit 0 is don't care
                out = c;
                $display("@%0dns 4'b001? is selected , opcode %b",$time,opcode);
                end
        default : begin
                $display("@%0dns default is selected , opcode %b",$time,opcode);
                end
        endcase
```

Concatenation and Replication

Concantenation is used to put multiple signals together.

```
assign output = {a, b[3:0], c, 4'b1001};
```

Replication is used to repeat a signal.

```
assign output1 = {3{a}};                //these are the same
assign output2 = {a, a, a};      //these are the same
```

Verilog Primer: State Machines

Finite-State Machine

Moore FSM from
`By Cepheus - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=1234893`
is shown in Figure 8.

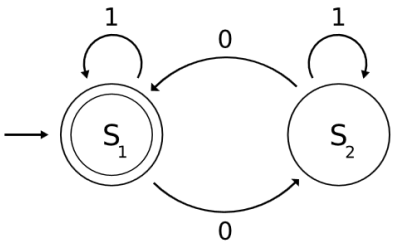


Figure 8: Moore FSM

Mealy FSM from:
`By OmniGraffle., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=11376306`
is shown in Figure 9.

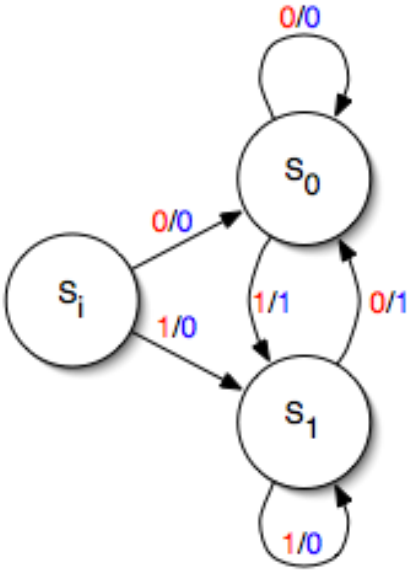


Figure 9: Mealy FSM

Algorithmic State Machine (ASM)

State block, conditional block, and decision block (Figure 10) for ASM from:

`By FarkasEngineering - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=6997929`

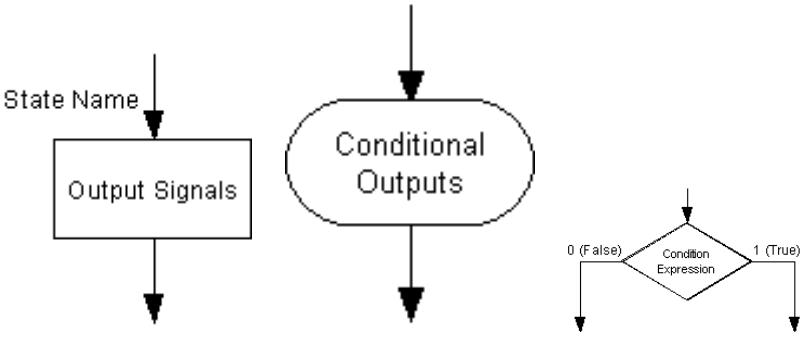


Figure 10: State, Conditional, and Decision Box of ASM

Watch the following videos

Watch the following videos:

`https://youtu.be/C-BXzVBoJ3U` : Algorithmic State Machine

`https://youtu.be/5EcYI0NfvXg` : Watching from time 0 to 21 minutes is required. The rest of the video is optional.

Verilog Primer: Verilog Operators

Verilog Operators

The operators for Verilog are shown in Figure 11.

Here are some examples of different ways to write a Comparator.

```
module Comparator_Ex(A_in, B_in, A_eq_B_out, A_lt_B_out, A_gt_b_out);
        input [5:0] A_in, B_in;
        output A_eq_B_out, A_lt_B_out, A_gt_b_out;

        reg A_eq_B_out;
        reg A_lt_B_out;
        wire A_gt_b_out;

        always @ (A_in or B_in)
        begin
                A_eq_B_out = 0;
                if (A_in == B_in)
                        A_eq_B_out = 1;
        end

        always @ (A_in or B_in)
        begin
                if (A_in < B_in)
                        A_lt_B_out = 1;
                else
                        A_lt_B_out = 0;
        end

        //(condition) ? true : false;
        assign A_gt_b_out = (A_in > B_in) ? 1 : 0;
endmodule
```

The block diagram of the hardware that is created from this is a fully parallel comparator. What happens when we use if else blocks?

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	case equality	equality
!=	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

Figure 11: Verilog Operators

```
module Comparator_Ex2(A_in, B_in, A_eq_B_out, A_lt_B_out, A_gt_b_out);
        input [5:0] A_in, B_in;
        output A_eq_B_out, A_lt_B_out, A_gt_b_out;

        reg A_eq_B_out;
        reg A_lt_B_out;
        reg A_gt_b_out;

        always @ (A_in or B_in)
        begin
                //default conditions
                A_eq_B_out = 0;
                A_lt_B_out = 0;
                A_gt_b_out = 0;

                //compare
                if (A_in == B_in)
                        A_eq_B_out = 1;
                else if (A_in < B_in)
                        A_lt_B_out = 1;
                else    //the only remaining possibility
                        //is greater than.
                        A_gt_b_out = 1;
        end
endmodule
```

Important things to remember. Only assign a variable inside one block (exculding initial conditions). For instance only store data into A_eq_B_out inside a single always block.

Also inside an always block use only blocking or non-blocking assignments. Do not mix blocking and non-blocking in the same always module. The rule of thumb is to use non-blocking for synchronous logic and blocking for combinational logic.

Lets look at a more complex compare.

```
module Comparator_Ex3(A_in, B_in, A_eq_B_out, A_lt_B_out,
                A_lt_eq_B_out, A_gt_b_out, A_gt_eq_b_out);
        input [5:0] A_in, B_in;
        output reg A_eq_B_out, A_lt_B_out, A_lt_eq_B_out, A_gt_b_out, A_gt_eq_b_out;

        always @ (A_in or B_in)
        begin
                //default conditions
                A_eq_B_out = 0;
                A_lt_B_out = 0;
```

```
                A_lt_eq_B_out = 0;
                A_gt_b_out = 0;
                A_gt_eq_b_out = 0;

                //compare
                case ({A_in == B_in, A_in < B_in, A_in > B_in,
                A_in <= B_in, A_in >= B_in})
                        5'b10011: begin
                                A_gt_b_out = 1;
                                A_lt_eq_B_out = 1;
                                A_gt_eq_b_out = 1;
                                end
                        5'b01010: begin
                                A_lt_B_out = 1;
                                A_lt_eq_B_out = 1;
                                end
                        5'b00101: begin
                                A_gt_B_out = 1;
                                A_gt_eq_B_out = 1;
                                end
                        default: begin
                                A_eq_B_out = 0;
                                A_lt_B_out = 0;
                                A_lt_eq_B_out = 0;
                                A_gt_b_out = 0;
                                A_gt_eq_b_out = 0;
                                end
                endcase
        end
endmodule
```

Another ways to do this same thing.

```
module Comparator_Ex4(A_in, B_in, A_eq_B_out, A_lt_B_out, A_lt_eq_B_out,
                A_gt_b_out, A_gt_eq_b_out);
        input [5:0] A_in, B_in;
        output A_eq_B_out, A_lt_B_out, A_lt_eq_B_out, A_gt_b_out, A_gt_eq_b_out;

        reg A_eq_B_out, A_lt_B_out, A_gt_b_out;
        wire A_lt_eq_B_out, A_gt_eq_b_out;


        always @ (A_in or B_in)
        begin
                //default conditions
```

```
                A_eq_B_out = 0;
                A_lt_B_out = 0;
                A_gt_b_out = 0;

                //compare
                case ({A_in == B_in, A_in < B_in})
                        2'b10: A_eq_B_out = 1;
                        2'b01: A_lt_B_out = 1;
                        default: A_gt_b_out = 1;
                endcase
        end

        assign A_lt_eq_B_out = A_eq_B_out | A_lt_B_out;
        assign A_gt_eq_B_out = A_eq_B_out | A_gt_b_out;
endmodule
```

The block diagram for Compare_Ex4 is a compare block followed by two or gates.

Another ways to do this same thing.

```
module Comparator_Ex5(A_in, B_in, A_eq_B_out, A_lt_B_out,
                A_lt_eq_B_out, A_gt_b_out, A_gt_eq_b_out);
        input [5:0] A_in, B_in;
        output wire A_eq_B_out, A_lt_B_out, A_lt_eq_B_out, A_gt_b_out, A_gt_eq_b_out;

        assign A_eq_B_out = A_in == B_in;
        assign A_lt_B_out = A_in < B_in;
        assign A_gt_b_out = A_in > B_in
        assign A_lt_eq_B_out = A_eq_B_out | A_lt_B_out;
        assign A_gt_eq_B_out = A_eq_B_out | A_gt_b_out;
endmodule
```

Working with signed numbers.

```
module Sign (A, B, Y1, Y2, Y3);
        input [2:0] A, B;
        output [3:0] Y1, Y2, Y3;
        reg signed [3:0] Y1, Y2, Y3;

        always @(A or B)
        begin
                Y1=+A/-B;
                Y2=-A+-B;
                Y3=A*-B;
        end
endmodule
```

Logical operations.

```
module Logical (A, B, C, D, E, F, Y);
        input [2:0] A, B, C, D, E, F;
        output Y;
        reg Y;

        always @(A or B or C or D or E or F)
        begin
                if ((A==B) && ((C>D) || !(E<F)))
                        Y=1;
                else
                        Y=0;
        end
endmodule
```

Bit wise operators

```
module Bitwise (A, B, Y);

        input [6:0] A;
        input [5:0] B;
        output [6:0] Y;
        reg [6:0] Y;

        always @(A or B)
        begin
                Y(0)=A(0)&B(0); //binary AND
                Y(1)=A(1)|B(1); //binary OR
                Y(2)=!(A(2)&B(2)); //negated AND
                Y(3)=!(A(3)|B(3)); //negated OR
                Y(4)=A(4)^B(4); //binary XOR
                Y(5)=A(5)~^B(5); //binary XNOR
                Y(6)=!A(6); //unary negation
        end
endmodule
```

Reduction

```
module Reduction (A, Y1, Y2, Y3, Y4, Y5, Y6);

        input [3:0] A;
        output Y1, Y2, Y3, Y4, Y5, Y6;
        reg Y1, Y2, Y3, Y4, Y5, Y6;

        always @(A)
        begin
```

```
            Y1=&A; //reduction AND
            Y2=|A; //reduction OR
            Y3=~&A; //reduction NAND
            Y4=~|A; //reduction NOR
            Y5=^A; //reduction XOR
            Y6=~^A; //reduction XNOR
        end
endmodule
```