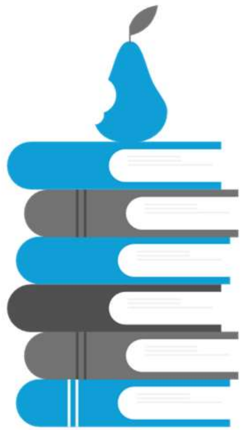
Verilog

EE370 Digital Electronics

References

1. (textbook) Verilog HDL: A guide to digital design and synthesis, Samir Palnitkar

1. (NPTEL lectures): Lecture series (lec-1 to lec-7) on Electronic Design and Automation by Prof. I Sengupta, IIT KGP https://www.youtube.com/playlist?list=PLBBCE226922E31394



Simulators/synthesis tools

- Simulation:
  - Icarus Verilog, Vivado

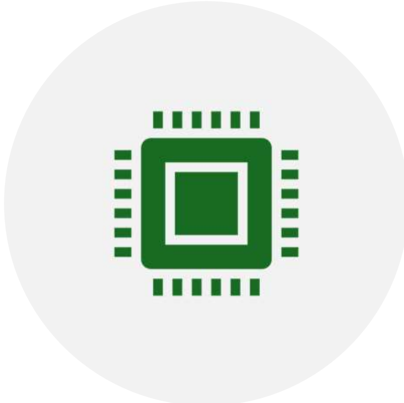
- Synthesis+ Post-synthesis sim
  - Yosys + Icarus Verilog, Vivado

Standardization using HDLs

- HDLs provide formats for representing the outputs of various design steps

- An HDL-based CAD tool transform a design from HDL input to HDL output which contains more hardware information

Synthesizable vs Non-synthesizable codes



Non-synthesizable:

Cannot be translated into a hardware. Useful for simulation and verification.

Synthesizable

Can be translated into a hardware

Behavioral Representation

- Specifies how a particular design should respond to a given set of inputs
- May be specified by
  - Boolean equations
  - Algorithms

Behavioral representation: 4-bit adder

```
module adder(a, b, s, cy);

        input [3:0]a,b;
        output reg [3:0]s;
        output reg cy;

        always @ (*)
        begin
                {cy, s} = a+b;
        end

endmodule
```

Behavioral representation: 4-bit adder

- A -bit adder is constructed by cascading n 1-bit adders
- A 1-bit adder has
  - Two operand inputs A and B
  - A carry input C
  - A carry output Cy
  - A sum output S

S=A.B’.C’+A’.B’.C+A’.B.C’+A.B.C
Cy=A.B+A.C+B.C

A behavioral description of Cy

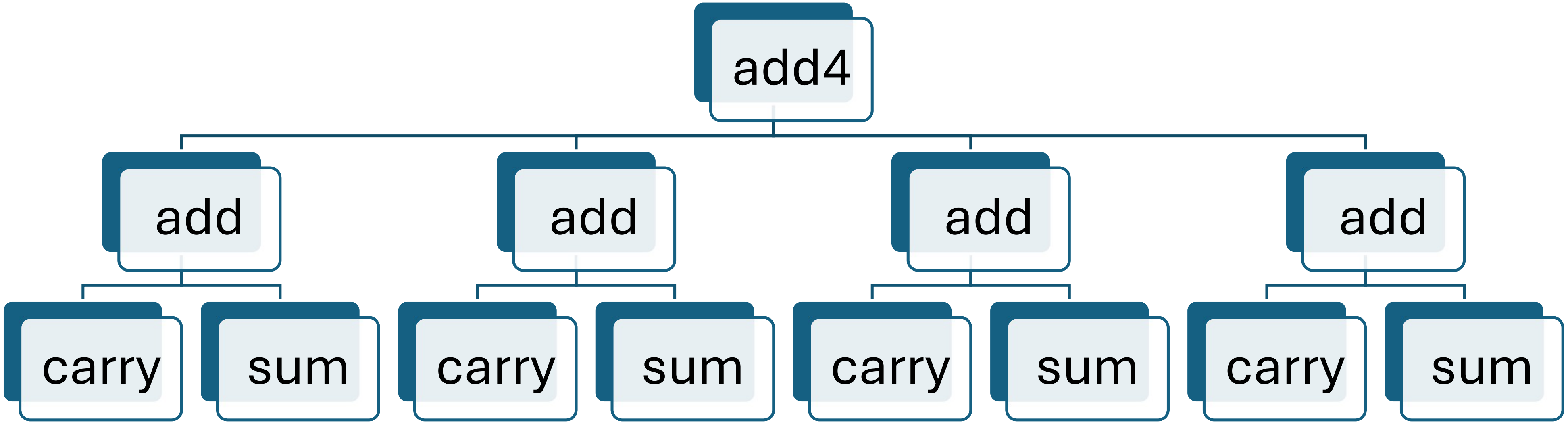
```
module carry (cy, a, b, c)
     input a, b, c;
     output cy;
     assign cy=(a&b)|(b&c)|(c&a);
endmodule
```

Structural representation

- Specifies how components are interconnected.
- In general, the description is a list of modules and their interconnects
  - called netlist
  - can be specified at various levels

Structural representation

- At the structural level, the levels of abstraction are
  - the module level
  - the gate level
  - the circuit level
- In each level more detail is revealed about the implementation


add4
add
add
add
add
carry
sum
carry
sum
carry
sum
carry
sum

Structural representation of a 4-bit adder

```
module add4(s, cy4, cy_in, x, y);
      input [3:0] x, y;
      input cy_in;
      output [3:0]s;
      output cy4;
      wire [2:0]cy_out;
            add B0 (cy_out[0], s[0], x[0], y[0], cy_in);
            add B1 (cy_out[1], s[1], x[1], y[1], cy_out[0]);
            add B2 (cy_out[2], s[2], x[2], y[2], cy_out[1]);
            add B3 (cy4, s[3], x[3], y[3], cy_out[2]);
endmodule
```

```
module add (cy_out, sum, a, b, cy_in);
      input a, b, cy_in;
      output sum, cy_out;
            sum s1 (sum, a, b, cy_in);
            carry c1 (cy_out, a, b, cy_in);
endmodule

module carry (cy_out, a, b, cy_in);
      input a, b, cy_in;
      output cy_out;
      wire t1, t2, t3;
            and g1 (t1, a, b);
            and g2 (t2, a, c);
            and g3 (t3, b, c);
            or g4 (cy_out, t1, t2, t3);
endmodule
```

Primitive gates

- and, nand
- or, nor
- xor, xnor
- buf, not
- bufif1, notif1
- bufif0, notif0

Value set

Value	Conditions in Hardware
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown value
z	High impedance, floating value

Primitive gates example

```
wire out, in1, in2, in3;

//basic gate instantiations
and a1(out, in1, in2);
nand na1(out, in1, in2);
or or1(out, in1, in2);
xnor nx1(out, in1, in2);

and na1_3inp(out, in1, in2, in3);
```

Commenting in Verilog

- Comments are inserted in the code for readability and documentation
- A good and much needed habit!
- Comments cannot be nested

```
a=b+c; // This is a one-line comment

/* This is a multiple
 line comment. */

/* This is /* an illegal */ comment.*/
```

Datatypes in Verilog

- net (wire)
- reg
- integer
- real
- time
- strings
- parameters

Nets in Verilog –(I)

- Most commonly used: wire

```
output  wire a;    // output port declared as wire
wire b, c;         // two internal wires
wire d = 1'b0;     // internal wire initialized to 0
                  at declaration
```

- Other net types: wand, wor, tri, triand, trior, tri0, tri1, trireg, supply0, supply1

Nets in Verilog –(II)

wand: wired and

wor: wired or

```
wand out1;
wor out2;

buf b1(out1, 1’b0);
buf b2(out1, 1’b1); //out1 final value 1’b0

buf b3(out2, 1’b0);
buf b4 (out2, 1’b1);   //out2 final value 1’b1
```

Nets in Verilog –(III)

tri: similar syntax and function as wire. Used to denote nets that have multiple drivers.

```
module mux (out, a, b, sel);
     input a, b, sel;
     output tri out;

     bufif0 b1(out, a, sel); //drives z if sel=1
     bufif1 b2(out, b, sel); //drives z if sel=0
endmodule
```

If there is a logic contention, tri net will get `x` value.

Registers (reg)

- Can be used to implement a combinational circuit or a sequential circuit
- Can assign value inside an always/initial block

Vectors

Nets or reg can be declared as a vectors

```
wire a;              //single bit wire, default
wire [7:0] bus;      //8-bit bus
wire [31:0] busA, busB, busC; //3 buses, each 32-bit

reg clock;           //single bit register
reg [0:4]addr;       //5-bit register
```

Vector: Single element that is n-bit wide
Array: Multiple elements that are 1-bit wide

Integer

- Default width is typically 32 bits
- Stored as signed quantities

```
integer counter;
initial
    counter = -1;
```

Real numbers

- Can specify in decimal notation or in scientific notation
- Not synthesizable, can be used in verification

```
real a, b;
integer i;
initial
begin
     a = 4e10;
     b = 2.13;
     i = b; //i=2
end
```

Time

- Can be used to store simulation time
- Not synthesizable, can be used in verification

```
time sim_time;

always @ (posedge(flag))
     sim_time = $time;
```

System Tasks

- $display
- $monitor
- $finish

System Tasks – (I)

$display: Display values of variables/strings/expressions

// To display current simulation time

```
$display($time)
```

//Display value of port_id in binary

```
reg [4:0] port_id;
$display("ID of the port is %b", port_id);
```

System Tasks – (I)

$monitor: Display values of variables/strings/expressions everytime any change occurs in the variables being monitored.

```
initial
begin
     $monitor ($time, value of signals clock = %b
               reset=%b”, clock, reset);
end
```

System Tasks – (I)

$finish: Terminates the simulation

```
initial
begin
     clock =0;
     reset =1;
     #100 $finish;
end
```

Operators: Arithmetic

a + b	a plus b
a - b	a minus b
a * b	a multiplied by b (or a times b)
a / b	a divided by b
a % b	a modulo b
a ** b	a to the power of b

Operators: Arithmetic

```
a=4'b0011; b=4'b0100;  //declared as reg vectors
d=6; e=4;              // declared as integers

a*b  // 4'b1100
d/e  // Evaluates to 1. Truncates fractional part.
d%e  // Evaluated to 2
a+b  // 4'b0111
b-a  // 4'b0001
```

Operators: Logical Operators

- logical and (&&), logical or (||), logical not (!)
- Evaluates to 1-bit value
  - 0: false
  - 1: true
  - x: ambiguous
- If an operand is nonzero, it is equivalent to logical 1 (true)

```
A=3; B=0;
A && B   // Evaluates to 0
A || B   // Evaluates to 1
!A       // Evaluates to 0
!B       // Evaluates to 1

A=2'b0x; B=2'b01;
A && B // Evaluates to x

(A==2) && (B==1) // What's the result?
```

Operators: Relational

```
a = 4; b = 3
x = 4’b1010; y = 4’b1101; z = 4’b1xxx

a <= b    // Evaluates to 0
a > b     // Evaluates to 1
y > x     // Evaluates to 1
y < z     // Evaluates to x
```

Operators: Equality

```
a = 4; b = 3
x = 4’b1010; y = 4’b1101;
z = 4’b1xxz; m = 4’b1xxz; n= 4’b1xxx;

a == b   // Evaluates to 0
x != y   // Evaluates to 1
x == z   // Evaluates to x

z === m  // Evaluates to 1 (all bits match, incl x,z)
z === n  // Evaluates to 0
m !== n  // Evaluates to 1
```

Operators: Equality

```
a = 4; b = 3
x = 4'b1010; y = 4'b1101;
z = 4'b1xxz; m = 4'b1xxz; n= 4'b1xxx;

a == b    // Evaluates to 0
x != y    // Evaluates to 1
x == z    // Evaluates to x
```

logical equality

```
z === m   // Evaluates to 1 (all bits match, incl x,z)
z === n   // Evaluates to 0
m !== n   // Evaluates to 1
```

case equality

Operators: Bitwise operations

```
x = 4'b1010; y = 4'b1101;
z = 4'b10x1;

~x          // 4'b0101
x & y       // 4'b1000
x | y       // 4'b1111
x ^ y       // 4'b0111
x ~^ y      // 4'b1000

x & z       // 4'b10x0
y & z       // 4'b1001
```

Operators: Bitwise operations

```
x = 4'b1010; y = 4'b1101;
z = 4'b10x1;

~x          // 4'b0101
x & y       // 4'b1000
x | y       // 4'b1111
x ^ y       // 4'b0111
x ~^ y      // 4'b1000

x & z       // 4'b10x0
y & z       // 4'b1001
```

```
a = 4'b1010; b=4'b0000

a | b     //4'b1010
a || b    // (1 || 0)=1
```

Operators: Reduction operations

Performs bit-wise operation on a single vector operand and yields a 1-bit result.

```
x = 4'b1010;

&x   // 1'b0
|x   // 1'b1
^x   // 1'b0

~&x  // 1'b1
~|x  // 1'b0
~^x  // 1'b1
```

Operators: Shift operations

Shift a vector operand to right/left by specified number of bits.

```
x = 4'b1100;

y = x >> 1;   // 4'b0110
y = x << 1;   // 4'b1000
y = x << 2;   // 4'b0000
```

Useful for modelling shift and add algo, multiplication by 2, division by 2 etc.

Operators: Concatenation

Append multiple operators.

```
a = 1’b1; b = 2’b00; c = 2’b10; d = 3’b110;

y = {b,c};                        // 4’b0010
y = {a,b,c,d,3’b001};             // 11’b10010110001
y = {a,b[0],c[1]};      // 3’b101
```

Operators: Conditional

```
assign out = control ? in1 : in0;

assign out = (A==3) ? (control?x:y) : (control?m:n);
```