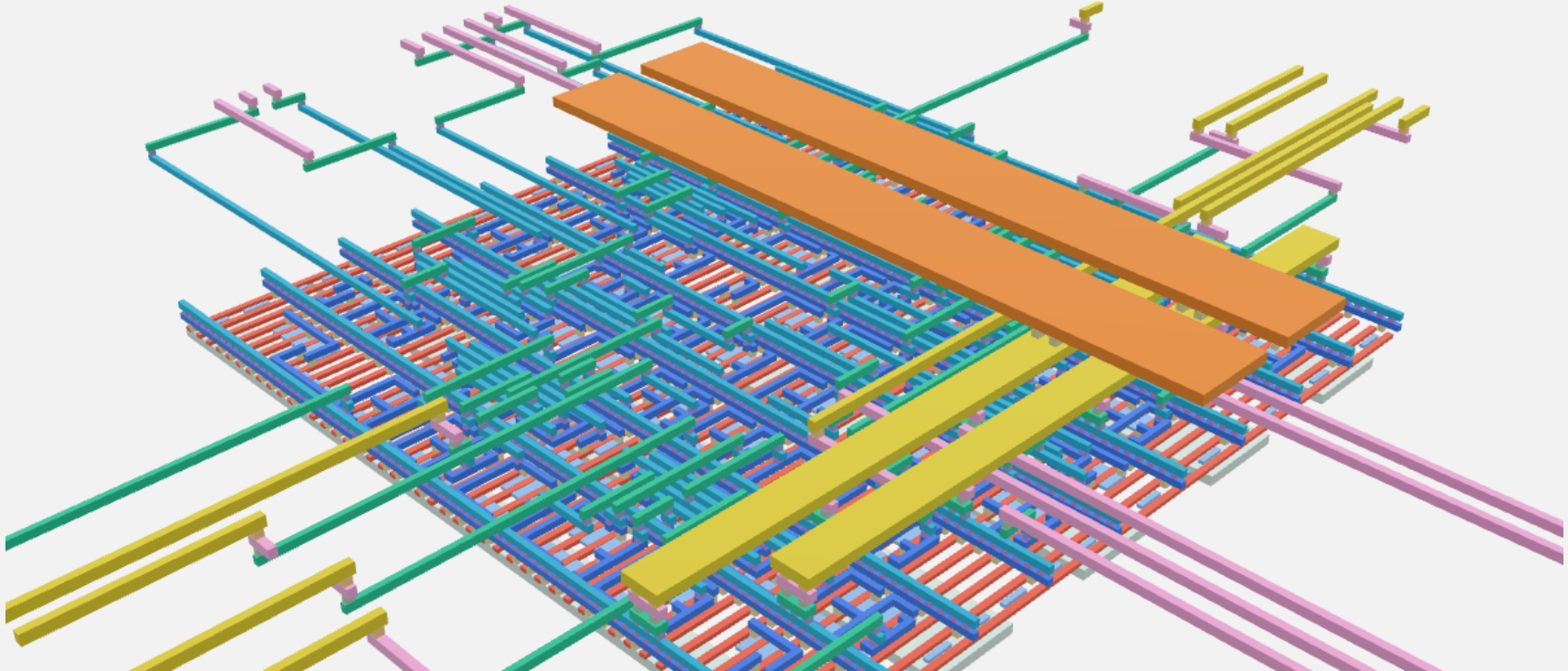
# 2.2
# CSE140 – Digital Design

2026 Summer Session 1

**Abarajithan G**



# Daily reminder:

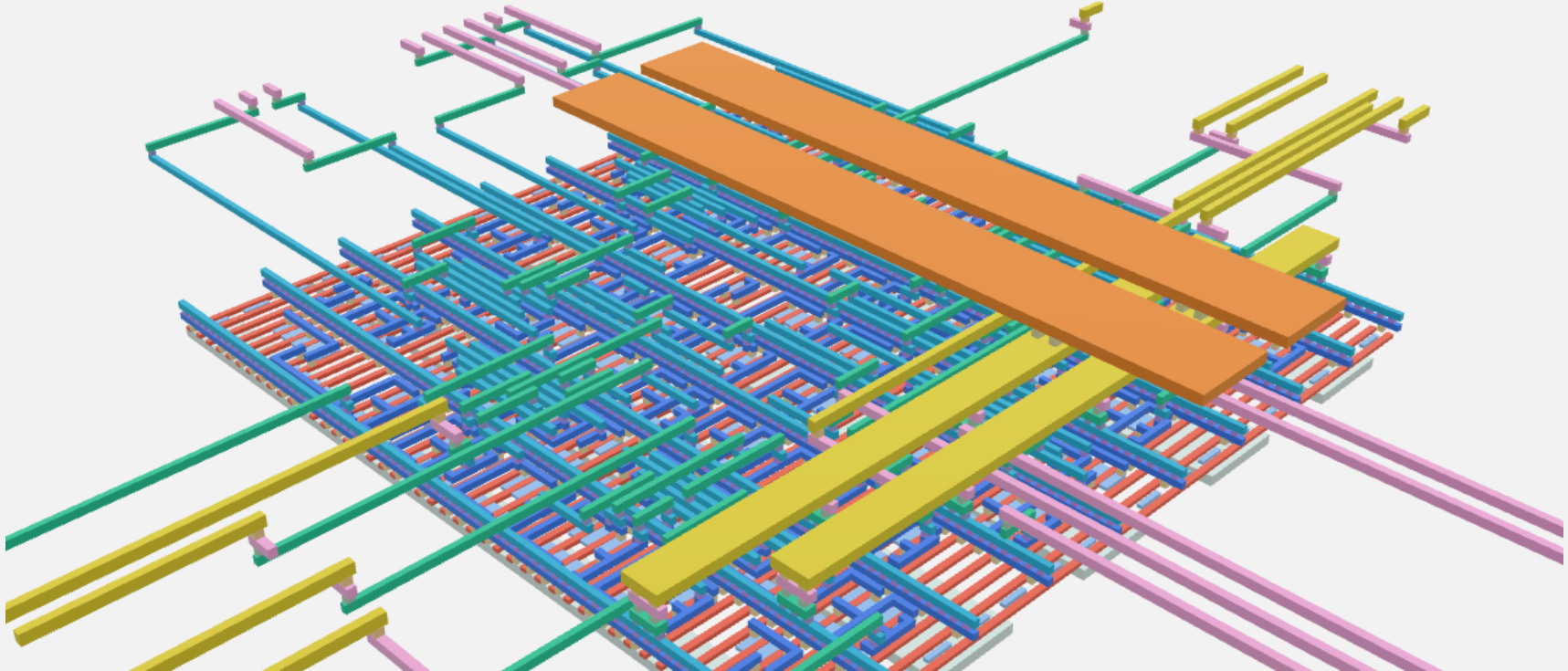
- Open FeedbackFruits
  Canvas → CSE140 → Assignments → Slides 2.2

- Attendance is mandatory

- Paper exams follow assignments. Don’t use AI

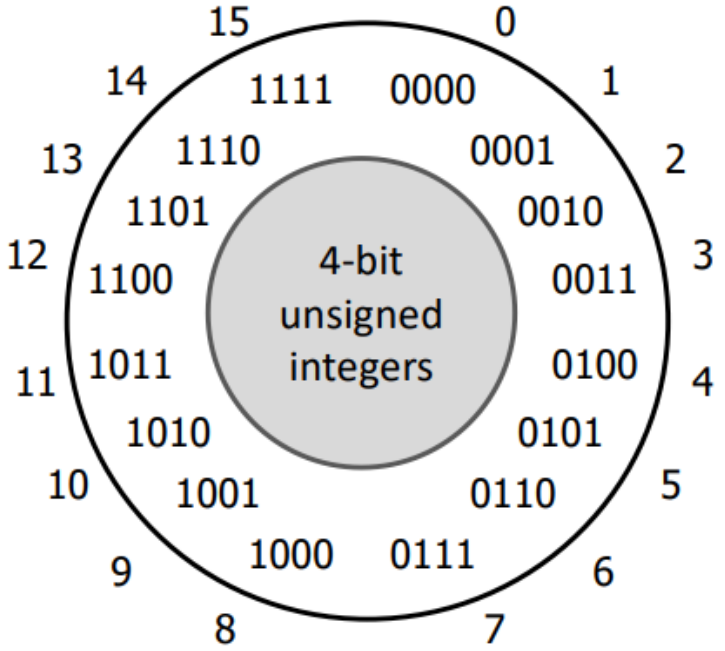
- SystemVerilog Basics

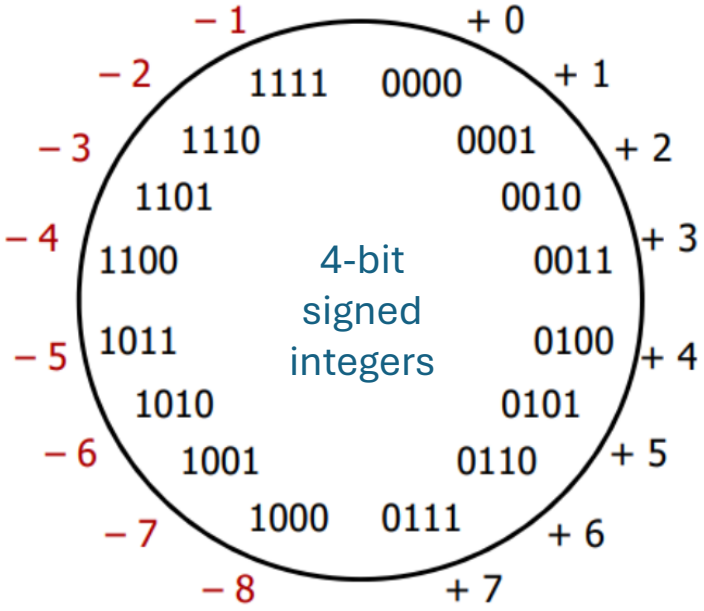
# Recap &
# Number Representation



# Recap

- If you have B bits
  - Unsigned range [smallest, largest] = **[0, $2^B$-1]**
  - Signed (2's complement)
    - Largest number: **$2^{B-1}$-1**
    - Smallest number: **-$2^{B-1}$**
- -X in two's complement: **~X+1**
- Clamping/clipping: check if the input is within the range of output. If not, pass the maximum/minimum
- Number line becomes a circle – overflow!



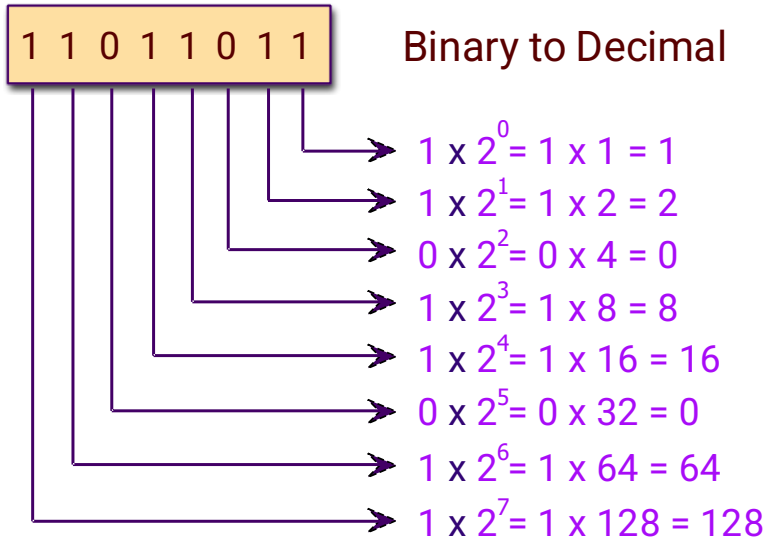



# Number Representation

- Decimal to binary: repeatedly divide by two, keeping track of remainders
- Binary to Decimal: Multiply ith digit (from LSB – least significant bit) by 2^i and add

| | | |
|---|---|---|
| 2 | 172 | |
| 2 | 86 | 0 |
| 2 | 43 | 0 |
| 2 | 21 | 1 |
| 2 | 10 | 1 |
| 2 | 5 | 0 |
| 2 | 2 | 1 |
| | 1 | 0 |

**172 = 10101100**




1 + 2 + 8 + 16 + 64 + 128 = 219

$(11011011)_2 = (219)_{10}$

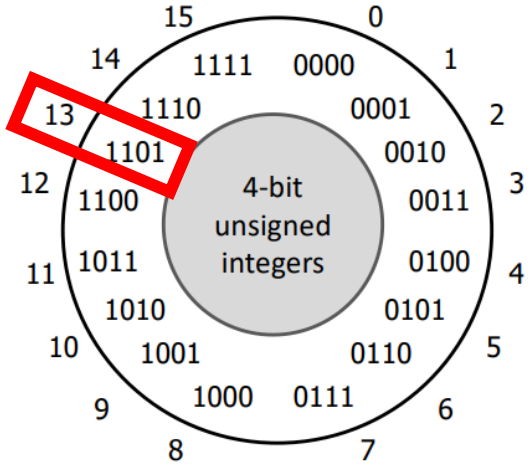
© w3resource.com

# Number Representation

- Decimal:

| 1 | 0 | 2 | 5 |
|---|---|---|---|
| x | x | x | x |
| $10^3$ | $10^2$ | $10^1$ | $10^0$ |

1000 + 0 + 20 +5 = 1025




- Unsigned Binary:

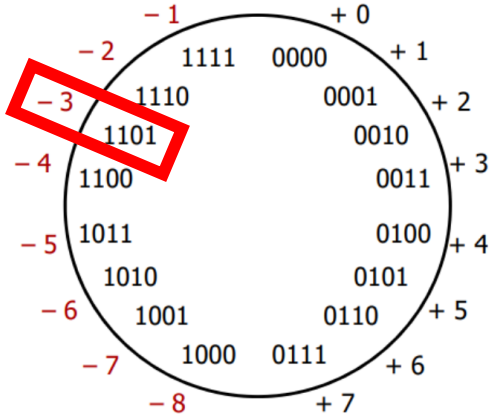
| 1 | 1 | 0 | 1 |
|---|---|---|---|
| x | x | x | x |
| $2^3$ | $2^2$ | $2^1$ | $2^0$ |

8 +4 +0 +1 = 13

- Signed Binary (Two's complement):

| 1 | 1 | 0 | 1 |
|---|---|---|---|
| x | x | x | x |
| $-2^3$ | $2^2$ | $2^1$ | $2^0$ |

(-8) +4 +0 +1 = -3



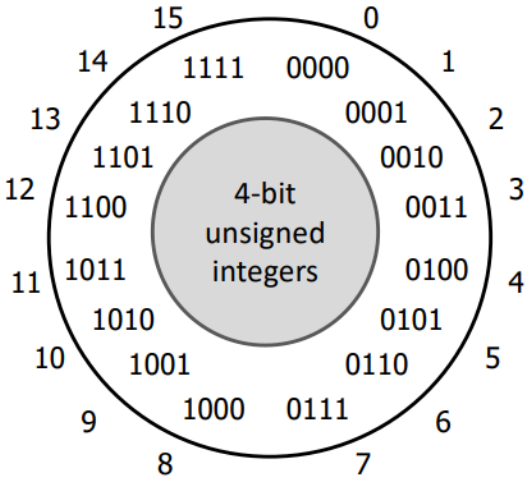
# Bit growth during Multiplication

- When multiplying two numbers (X,Y) with bitwidths ($B_x$,$B_y$), bits needed to correctly represent the result is **$B_x+B_y$**
- Proofs (for your curiosity):
  - Unsigned (Z = X.Y)
    - Largest operands: $X = 2^{Bx}-1$, $Y = 2^{By}-1$
    - Worst case, largest result: $(2^{Bx}-1) \times (2^{By}-1)$
    - This is less than $2^{Bx} \times 2^{By} = 2^{Bx+By}$
    - Therefore, $B_x + B_y$ bits are enough to represent X.Y
    - At home, try proving $(B_x + B_y -1)$ bits are not enough
  - Signed (Z = X.Y)
    - Most negative operands: $X = -2^{Bx-1}$, $Y = -2^{By-1}$
    - Worst case, largest positive result = $(-2^{Bx-1})(-2^{By-1}) = 2^{Bx+By-2}$
    - A signed W-bit number can represent at most: $2^{W-1} -1$
    - When W= $(B_x + B_y)$, largest result: $(2^{Bx+By-1} -1) >= 2^{Bx+By-2}$ enough to represent Z
    - When W= $(B_x + B_y-1)$, largest result: $2^{Bx+By-2} -1 < 2^{Bx+By-2}$ therefore $B_x + B_y$ bits needed

# Multiplication

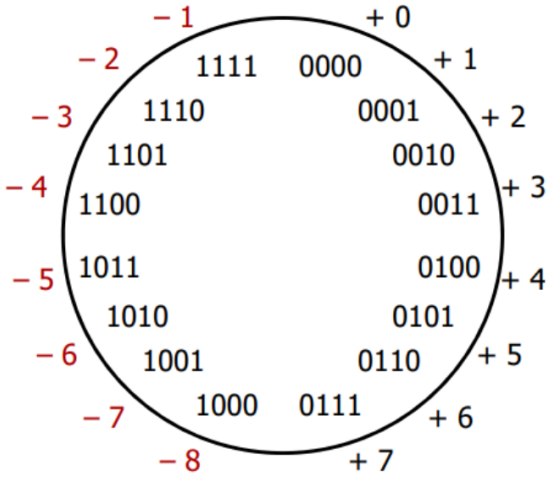
- Unsigned Binary:

```
      0011   : 3
×     0100   : 4
------------
      0000   : 2⁰ × 0 × 0011
     0000    : 2¹ × 0 × 0011
    0011     : 2² × 1 × 0011
+  0000      : 2³ × 0 × 0011
------------
  00001100   : 12
```




- Signed Binary
  (Two's complement):

```
      0010   : 2
×     1101   : -3
------------
  00000010   :  2⁰ × 1 × 0010
  00000000   :  2¹ × 0 × 0010
  00010      :  2² × 1 × 0010
- 00010      : -2³ × 1 × 0010
------------
  11111010   : -6
```



# Signed Multiplication

- Two things to remember:
  - Sign extend to $B_X + B_Y$
  - Subtract the term corresponding to sign bit
- Example by flipping the operands: XY = YX

```
      0010  : 2
×     1101  : -3
------------
   0000010  :  2^0 × 1 × 0010
   000000   :  2^1 × 0 × 0010
   00010    :  2^2 × 1 × 0010
-  0010     : -2^3 × 1 × 0010
------------
  11111010  : -6
```

```
      1101  : -3
×     0010  : 2
------------
  0000000   :  2^0 × 0 × 1101
  1111101   :  2^1 × 1 × 1101
  000000    :  2^2 × 0 × 1101
- 00000     : -2^3 × 0 × 1101
------------
  11111010  : -6
```

# SystemVerilog Literals

- **<width>'<s?><radix><value>**
- Width: 8, 32, anything
- S: signed or not. Typically, we don't use it
- Radix:
  - d: decimal
  - b: binary
  - h: hex
  - o: octal
- Value:
  - The actual value, can use _

```
module literals_tb;


  initial begin
    // Binary, hexadecimal, decimal
    $display("bin           = %b" , 8'b1010_1100);
    $display("hex           = %h" , 8'hAC       );
    $display("dec           = %0d", 8'd172      );


    // Unsigned vs signed decimal
    $display("unsigned dec = %0d", 8'd255      ); // 255
    $display("signed dec   = %0d", 8'sd255     ); // -1
    $display("common pattrn= %0d",-8'd1        ); // -1


    // Unsigned vs signed hexadecimal
    $display("unsigned hex = %0d", 8'hFF       ); // 255
    $display("signed hex   = %0d", 8'shFF      ); // -1


    $finish;
  end

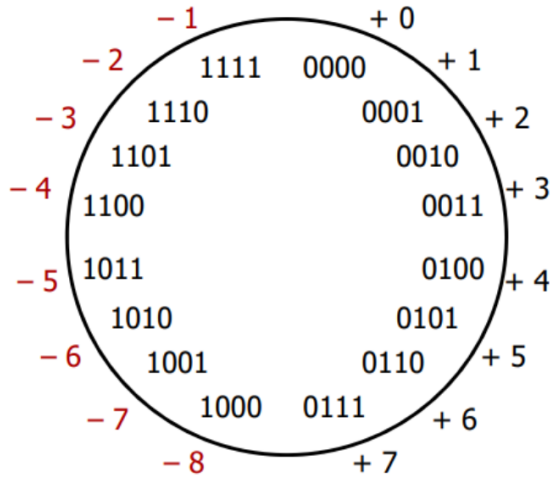

endmodule
```

# Fixed Point Representation

- How to represent fractions? – Needed for neural networks!
- Fixed-point: keep an implicit power-of-two associated with the data
- Fixed in hardware / software

$$y_{real} = (y/2^F)$$

- where F= fractional bits, y= binary, $y_{real}$ = real number (fraction)




- Example, a neural network has 4-bit weights (k, $F_K$=3), inputs (x, $F_X$=2)
  - k = 4'b1101 = -4'd3 ; $k_{real} = (-3)/2^3 = -3/8 = -0.375 = 1.101$
  - x = 4'b0010 = 4'd2 ; $x_{real} = (2)/2^2 = 2/4 = 0.5 = 00.10$

# Fixed Point Arithmetic

- Example:
  - $(B_K=4, F_K=3)$ k = 4'b1101 = -4'd3 ; $k_{real} = (-3)/2^3 = -3/8 = -0.375 = 1.011$
  - $(B_x=4, F_x=2)$ x = 4'b0010 = 4'd2 ; $x_{real} = (2)/2^2 = 2/4 = 0.5 = 00.10$
- Addition: z = k+x
  - $z_{real}=k_{real}+ x_{real} = k/2^{Fk}+x/2^{Fx} = k/2^3+x/2^2 = (\mathbf{k + x.2})/2^3 = (-3+2*2)/8 = 0.125$
  - First align the point to the operand with largest F, and then add. **$F_z = \max(F_k, F_x)$**
- Multiplication: z = kx
  - $z_{real}=k_{real}*x_{real} = (k/2^{Fk})(x/2^{Fx}) = xk/2^{Fk+Fx} = xk/2^{3+2} = \mathbf{xk}/2^5 = (-3*2)/2^5 = -6/32 = 111.11010$
  - Multiply directly, **$F_z = F_k+F_x$** , $B_z = B_k+B_X$
- **Note:** The hardware always operates on integers (add,mul), we move the implicit point accordingly

# Calculate  z = kx+y

```
(Bk=3,Fk=3) k = 101   : -3
(Bx=4,Fx=2) x = 0111  :  7
(By=3,Fy=3) y = 110   : -2
```

1. Calculate p = kx
    1. Find Bp (bits needed), sign extend an operand to that.
    2. Multiply. Remember the negative sign of the sign bit
    3. Find p as a decimal whole number. Find Fp
2. Calculate z = p+y
    1. Align the other operand to largest F
    2. Find Bz (bits needed)
    3. Sign extend and add. Find z as binary (without radix), Bz, Fz.

# Calculate z = kx+y

```
(Bk=3,Fk=3) k = 101    : -3
(Bx=4,Fx=2) x = 0111   :  7
(By=3,Fy=3) y = 110    : -2

Sign-extend k to 7 bits: 1111101: -3

       101   : -3
×     0111   : 7
-------------
   1111101   :  2^0 × 1 × 1111101
   1111010   :  2^1 × 1 × 1111101
   1110100   :  2^2 × 1 × 1111101
+  0000000   :  2^3 × 0 × 1111101
-------------
   1101011   : -21

(Bp=3+4=7, Fp=3+2=5) p=kx = -21
```

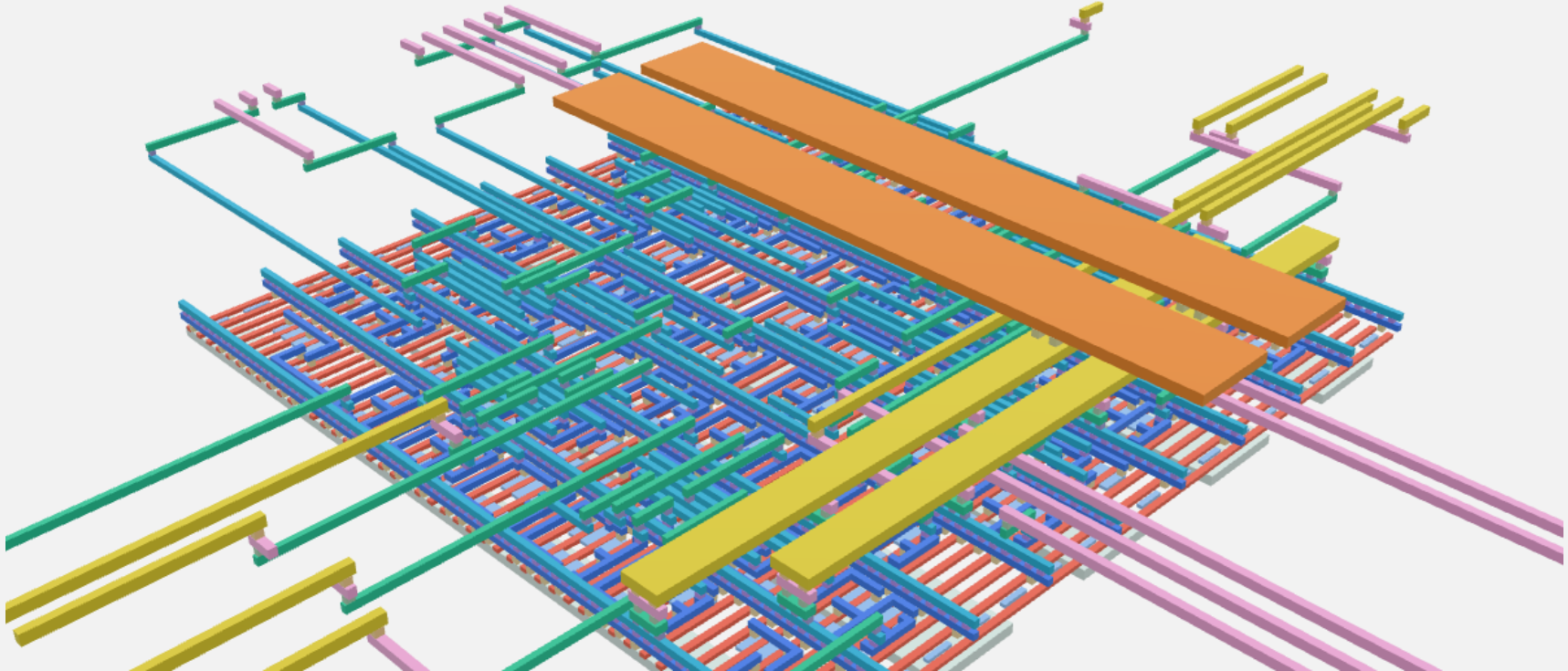
```
(Bp=7,Fp=5) p = 1101011 : -21
(By=3,Fy=3) y = 110     : -2

Align y to F=5: y*2^2 = 11000
Sign extend to 8 bits and add

    11101011   : -21
+   11111000   :  -8
---------------
    11100011   : -29


(Bz=8,Fz=5) z = 111.00011 : -29/32
                          = -0.90625
```

# Combinational Circuits

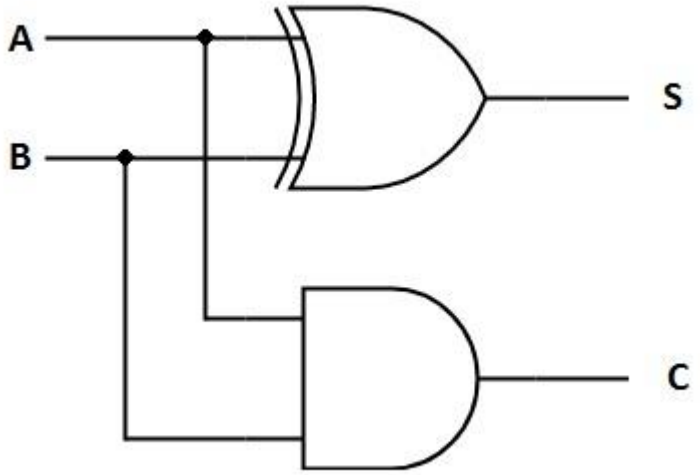


# Half Adder

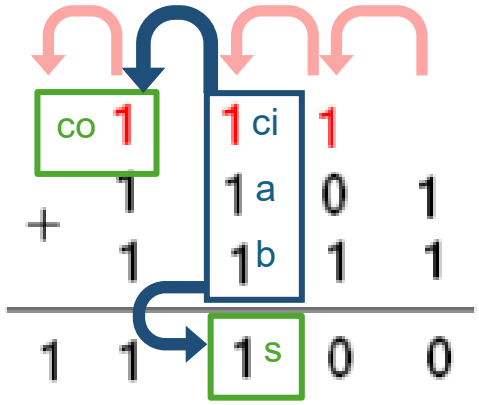
- Adding two 1-bit numbers
- Generates a carry-out
- To add two N-bit numbers, we should be able to chain many 1-bit adders – can't!

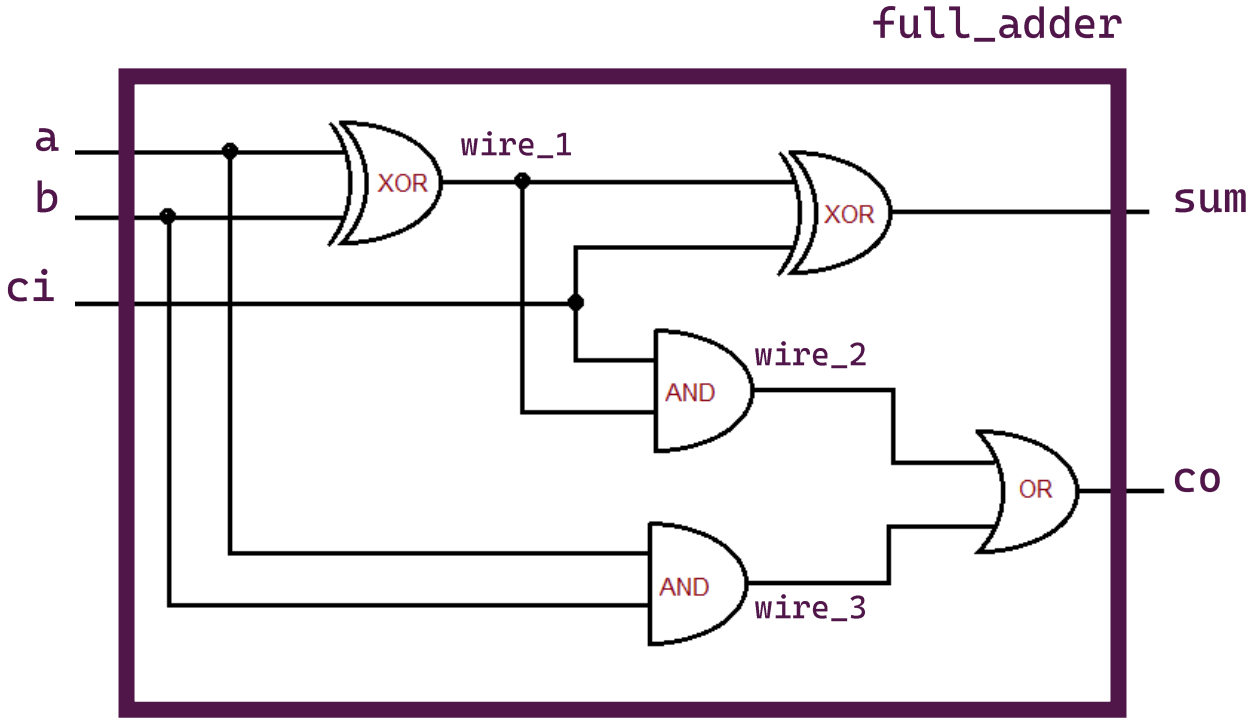
| A | B | Sum | COut |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

XOR AND



# Full Adder




```
`timescale 1ns/1ps

                                  Module name
module full_adder (
  input  logic a, b, ci,
  output logic sum, co
);
  logic wire_1, wire_2, wire_3;

  always_comb begin
    wire_1 = a ^ b;              // bitwise XOR
    wire_2 = wire_1 & ci;        // bitwise AND
    wire_3 = a & b;              // bitwise AND


    co     = wire_2 | wire_3;    // bitwise OR
    sum    = wire_1 ^ ci  ;      // bitwise XOR
  end
```

Input, output ports

Combinational circuit

```
endmodule
```

# Testbench: Full Adder

```
`timescale 1ns/1ps
```
Time units

```
module tb_full_adder;
```
No need inputs / outputs.
Software, not hardware

```
logic a=0, ci=0, b=0, sum, co;
```
Variable declaration, with and without initial values

```
full_adder dut (.*);
```
Module instantiation: (dot-star way, connects to signals with same names and types)

```
initial begin // simulation starts
0 $dumpfile(`FST_PATH); $dumpvars;
```
Dump waveforms

```
1 #3 a = 1; b = 0; ci = 0;
2 #1 a = 0; b = 0; ci = 1;
```
delay/wait and drive input pins

```
3 #2 a = 1; b = 1; ci = 0;
```
Concatenation

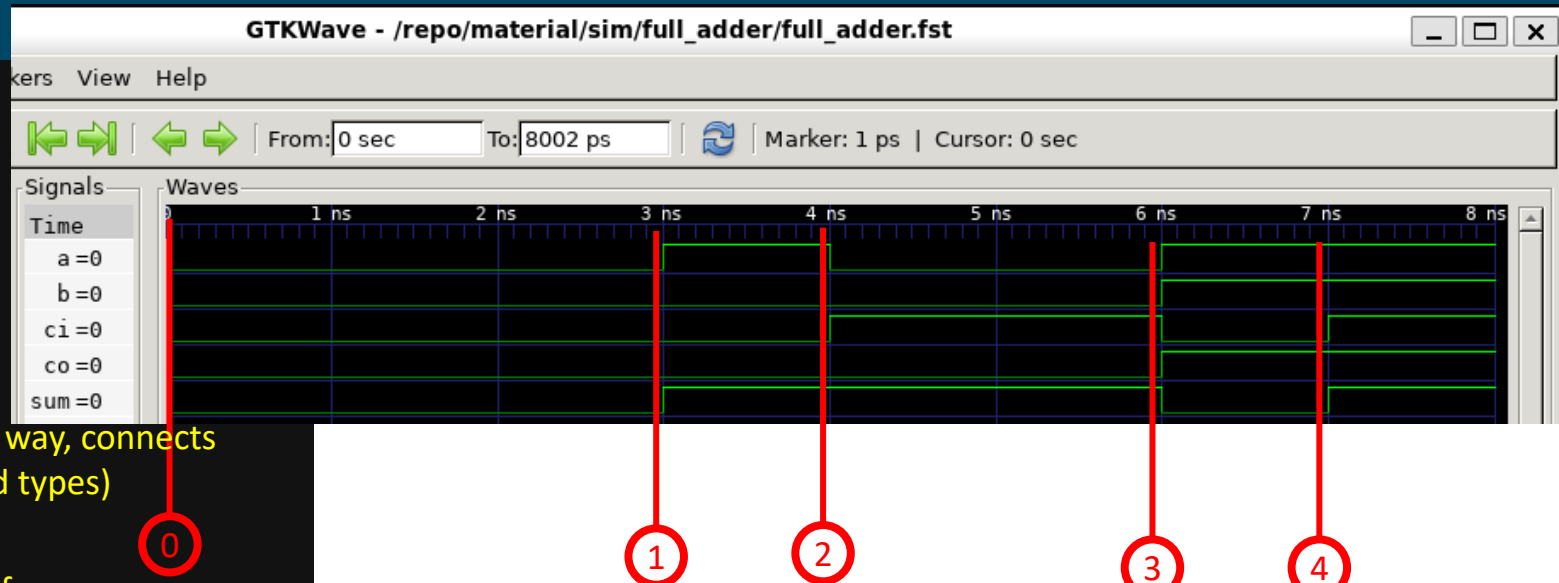
```
#1ps assert ({co,sum} == a+b+ci)
      $display ("OK");
      else $error("Not OK");
```
Assertion (condition)
pass statement;
fail statement;

```
4 #1 a = 1; b = 1; ci = 1;
```
Accessing deeper signals

Assertion – only fail

```
#1ps assert (dut.wire_1 == 0) // 1 (xor) 1 = 0
      else $error("False. wire_1:%d", dut.wire_1);
$finish(); // simulation ends
end
endmodule
```
String formatting with integer argument

Stop simulation without errors

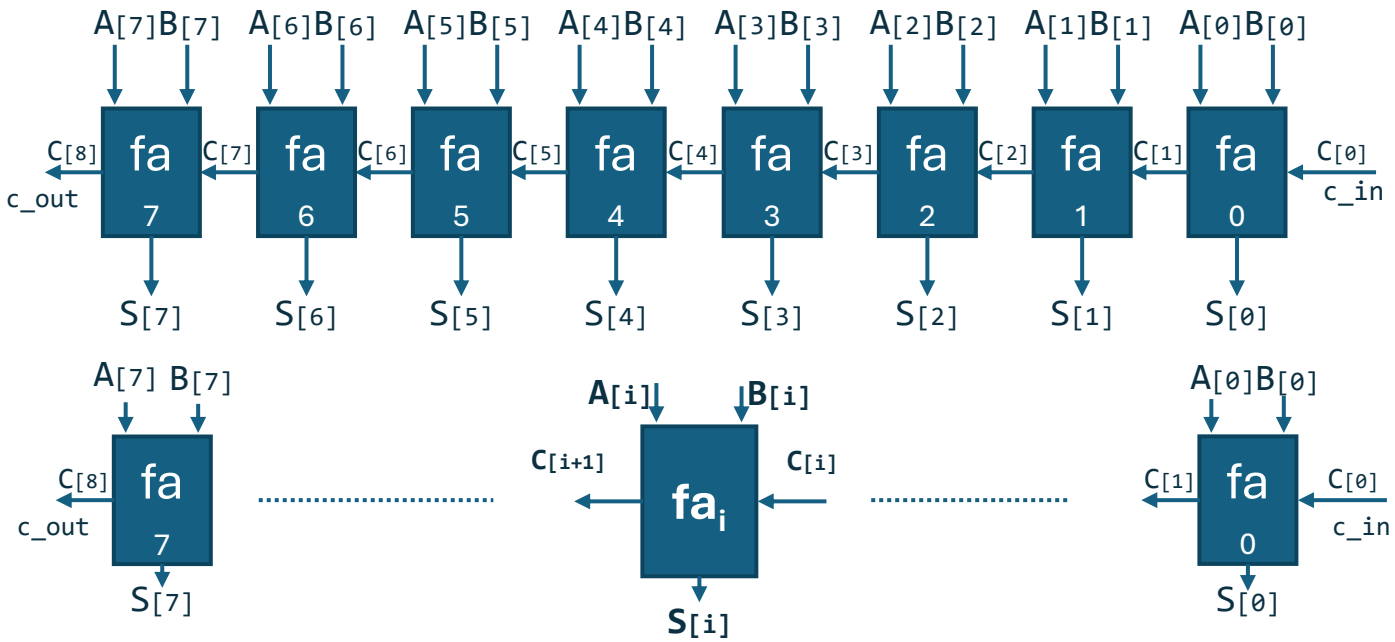



| | A | B | Cin | Sum | Cout |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 1 | 0 |
| | 0 | 1 | 0 | 1 | 0 |
| | 0 | 1 | 1 | 0 | 1 |
| | 1 | 0 | 0 | 1 | 0 |
| | 1 | 0 | 1 | 0 | 1 |
| 3 | 1 | 1 | 0 | 0 | 1 |
| 4 | 1 | 1 | 1 | 1 | 1 |

```
# inside container (after make enter)

make sim gtkwave DESIGN=full_adder
```

# Ripple Carry Adder




| Descending range | Ascending range |
|---|---|
| N-bit vector (wire) declared as:<br>`logic [N-1 : 0] wire_1;` | N-bit vector (wire) declared as:<br>`logic [0 : N-1] wire_1;` |

```
`timescale 1ns/1ps

module n_adder #(
  parameter N=8
)(
  input  logic [N-1:0] A, B,
  input  logic ci,
  output logic [N-1:0] S,
  output logic co
);
  logic [N:0] C;

  always_comb C[0] = ci;

  genvar i;
  for (i=0; i<N; i=i+1) begin:add
    // full_adder fa (A[i],B[i],C[i],C[i+1],S[i]);
    full_adder fa (
      .a    (A[i  ]),
      .b    (B[i  ]),
      .ci   (C[i  ]),
      .co   (C[i+1]),
      .sum  (S[i  ])
    );
  end

  always_comb co = C[N];

endmodule
```

Parameterized module

N-bit, descending wire vectors

(N+1)-bit vector

Referencing certain bits

Generating instances

Module Instantiation (named)

Chaining

# Testbench: Ripple Carry Adder

```
# inside container (after make enter)

make sim gtkwave DESIGN=n_adder
```

```
`timescale 1ns/1ps

module tb_n_adder;

  localparam N = 8;

  logic [N-1:0] A, B, S;
  logic ci, co;
  int status;

  n_adder #(.N(N)) dut (.*);

  initial begin
    $dumpfile(`FST_PATH);
    $dumpvars;
```

Localparam

Parameterized module instantiation

```
    A = 8'd5; B = 8'd10; ci = 0;
    #1 assert (S == 8'd15) else $error("Fail");
```

Delay to allow simulator to send value into DUT

```
    #1 A = 8'd30;  B = -8'd10; ci = 0;
    #1 A = 8'd5;   B = 8'd10;  ci = 1;
    #1 A = 8'd127; B = 8'd1;   ci = 0;
```

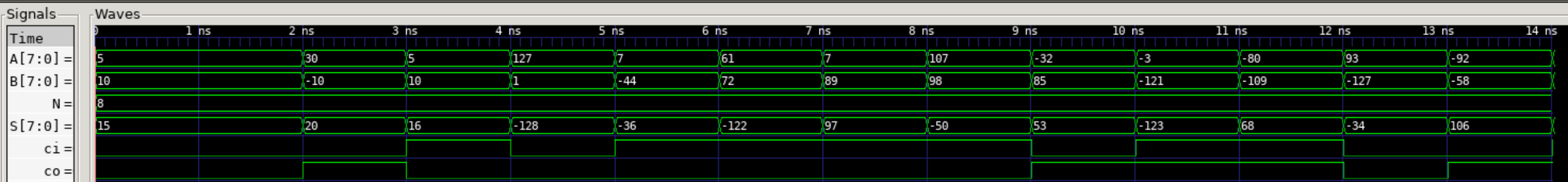
repeat loop

```
    repeat(10) begin
      #1
      status = std::randomize(ci);
      status = std::randomize(A) with { A inside {[0:255]}; };
      status = std::randomize(B) with { B inside {[0:255]}; };
      #1ps
      assert ({co,S} == {1'b0, A} + {1'b0, B} + ci)
        else $error("%d+%d+%d != {%d,%d}", A,B,ci,co,S);
    end
```

Randomize with constraints

```
    $finish;
  end
endmodule
```

- Radix
- View internal signals
- Arrange signals

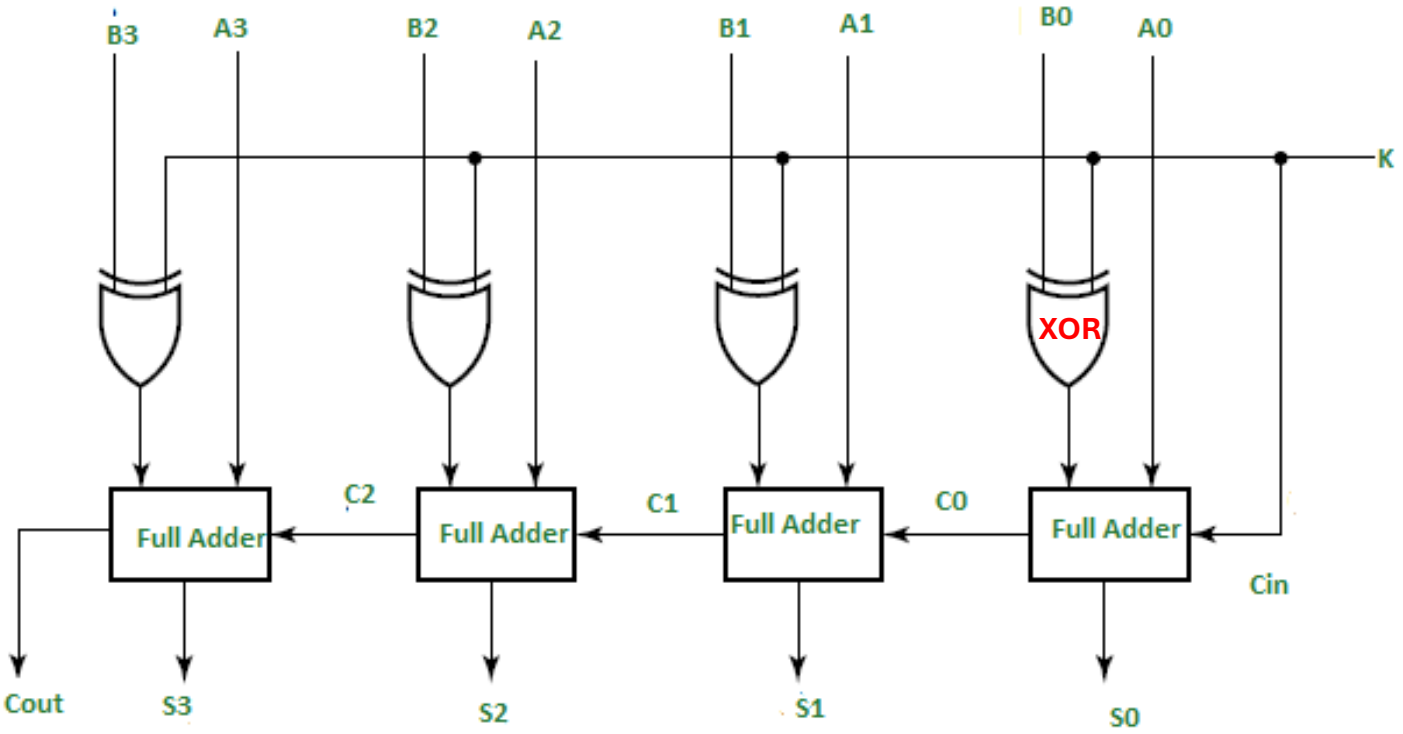


# Adder / Subtractor

$$A-B = A + (-B)$$

$$= A + \underbrace{(\sim B + 1)}$$

(-B) in 2's complement




Truth Table of XOR gate

| i | K | $B_0$ | $Y_0$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 2 | 1 | 0 | 1 |
| 3 | 1 | 1 | 0 |

When K=0,
$Y_0=B_0$
$\{C_o, S\} = A+B+0 = A+B$

When K=1,
$Y_0=(B_0)'$
$\{C_o, S\} = A+(\sim B)+1 = A-B$

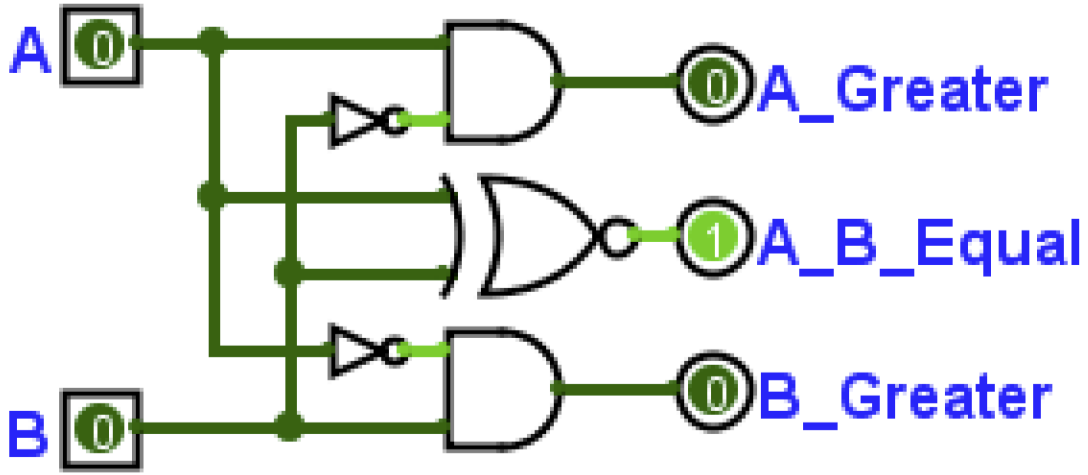
```
`timescale 1ns/1ps
module n_add_sub #(
  parameter N = 8
)(
  input  logic [N-1:0] A, B,
  input  logic         sub,
  output logic [N-1:0] S,
  output logic         co
);
  logic [N:0]   C;
  logic [N-1:0] Y;

  always_comb C[0] = sub;

  genvar i;
  for (i=0; i < N; i=i+1) begin:add
    always_comb Y[i] = B[i] ^ sub;
    full_adder fa (
      .a   (A[i  ]),
      .b   (Y[i]),
      .ci  (C[i  ]),
      .co  (C[i+1]),
      .sum (S[i  ]));
  end
  always_comb co = C[N];
endmodule
```

# One-bit comparator

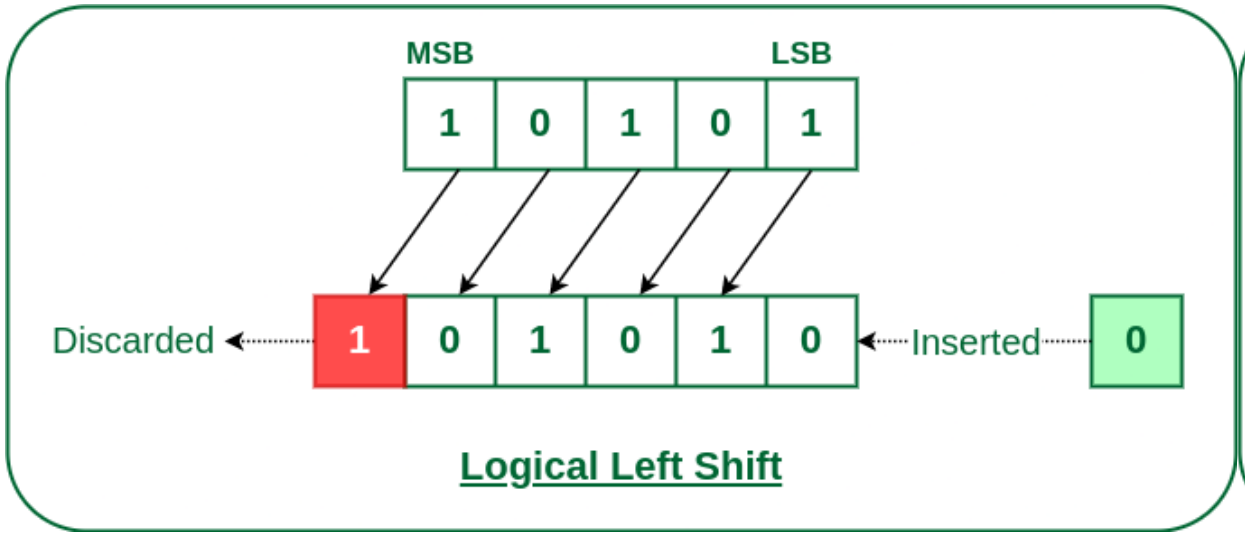
| Inputs | | Outputs | | |
|---|---|---|---|---|
| $B$ | $A$ | $A > B$ | $A = B$ | $A < B$ |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |

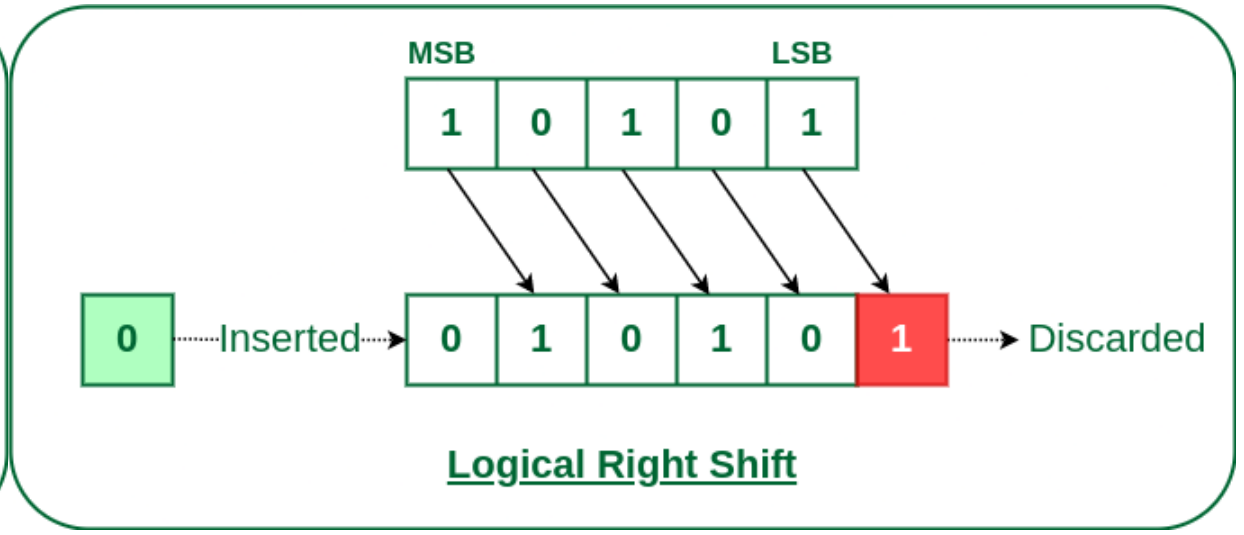



- Multiple bits? – Write the truth table, derive the circuit

# Logical Shifting

- Discard bits
- Insert Zeros



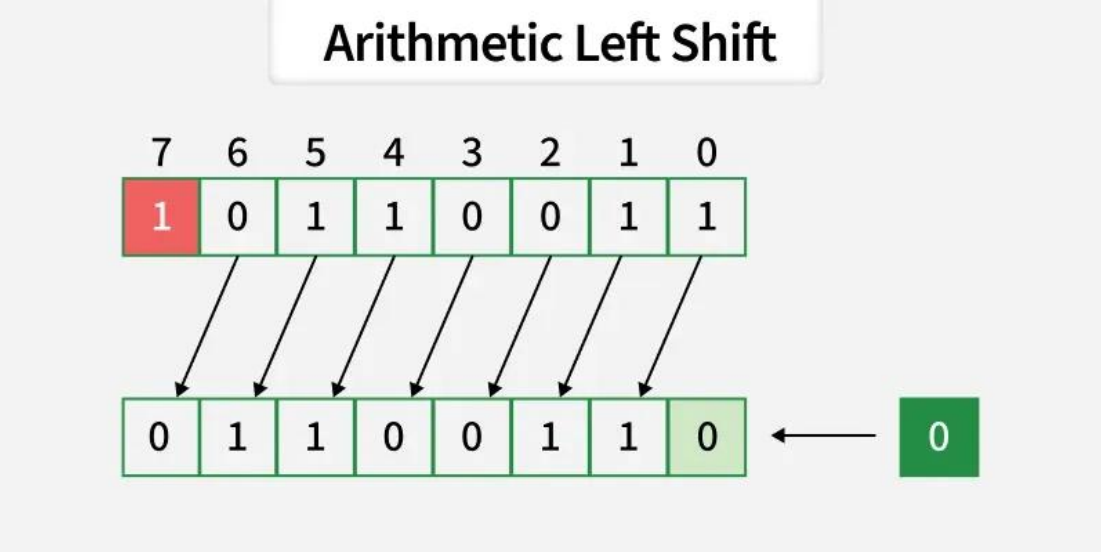

```
y = x << s
```

```
y = x >> s
```

# Arithmetic Shifting

**Left:**

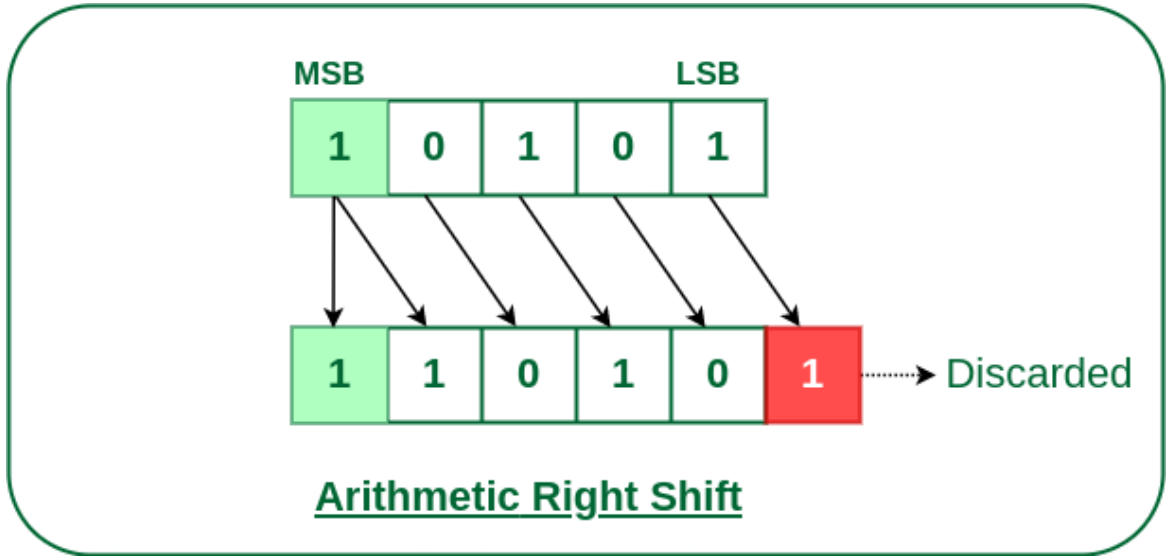
- Discard bits
- Insert Zeros




```
y = x <<< s
```

**Right:**

- Discard bits
- Insert **sign bit,** if x is signed




```
y = $signed(x) >>> s
```

# Fixed-point conversion

- Example: Convert (B,F)=x:(8,4) → y:(5,2)

  x = 00001110 = 14

  y_real = x_real

  y/2^2 = x/2^4

  y = x/2^2 = 14/4 = 3.5

- But y has to be an integer. What to do?
  - Truncate : 3.5 → 3, y = 00.11
  - Round up : 3.5 → 4, y = 001.00 (B=5,F=2)
- Always rounding down/up accumulates error – Problem!
- Banker's rounding!

# Banker's rounding / Round to Nearest Even

- When remainder=0.5, round to the nearest even.
  - 1.5 →2, 2.5→2, (-1.5)→(-2), (-2.5)→(-2)
- **Algorithm**:
  - Calculate the SHIFT (ΔF)
  - Arithmetic shift X to get quotient
  - Shift quotient back and subtract to get remainder
  - Find value of half: 1 << (SHIFT-1)
  - If remainder > half, out=quot+1
  - Else:
    - If remainder is half and quotient is odd, out=quot+1
    - Else, out = quot (truncated / round down)

```
(B,F)=x:(8,4) → y:(5,2)
X = 00001110 = 14

SHIFT = Fx-Fy = 2
14 >>> 2 = 14//4 = 3 = quot
14-(3<<<2) = 14-12 = 2 = rem
1 <<< (2-1) = 2 = half
(rem=2) > (half=2) ? No

rem=half(=2), quot(=3) odd
out = 4
```

# Activity:

**(B,F)=x:(8,5) → y:(5,2) x = 00100100 = 36**

- When remainder=0.5, round to the nearest even.
  - 1.5 →2, 2.5→2, (-1.5)→(-2), (-2.5)→(-2)
- **Algorithm**:
  - Calculate the SHIFT (ΔF)
  - Arithmetic shift X to get quotient
  - Shift quotient back and subtract to get remainder
  - Find value of half: 1 << (SHIFT-1)
  - If remainder > half, out=quot+1
  - Else:
    - If remainder is half and quotient is odd, out=quot+1
    - Else, out = quot (truncated / round down)

# Activity:

**(B,F)=x:(8,5) → y:(5,2) x = 00100100 = 36**

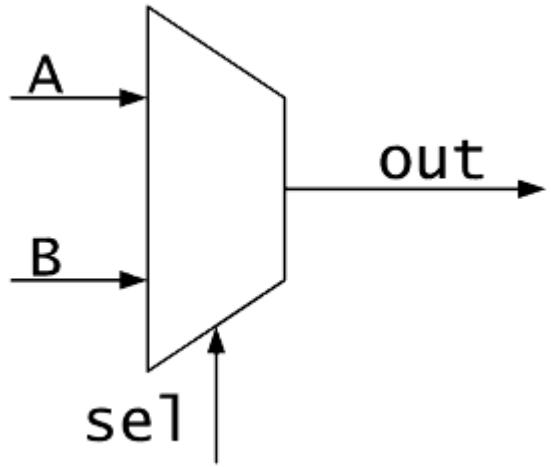
- When remainder=0.5, round to the nearest even.
  - 1.5 →2, 2.5→2, (-1.5)→(-2), (-2.5)→(-2)
- **Algorithm**:
  - Calculate the SHIFT (ΔF)
  - Arithmetic shift X to get quotient
  - Shift quotient back and subtract to get remainder
  - Find value of half: 1 << (SHIFT-1)
  - If remainder > half, out=quot+1
  - Else:
    - If remainder is half and quotient is odd, out=quot+1
    - Else, out = quot (truncated / round down)

```
SHIFT = Fx-Fy = 3
36 >>> 3 = 36/8 = 4 = quot
36 - (4 << 3)= 4 = rem
1 <<< (2) = 4 = half
(rem=4) > (half=4) ? No
rem=half(=4), quot(4) even
out = 4 (round down)
```

# Multiplexer

- 3 inputs
- Based on select, choose A or B
- Write the truth table (optional)
- Draw K-map
- Derive the expression for output

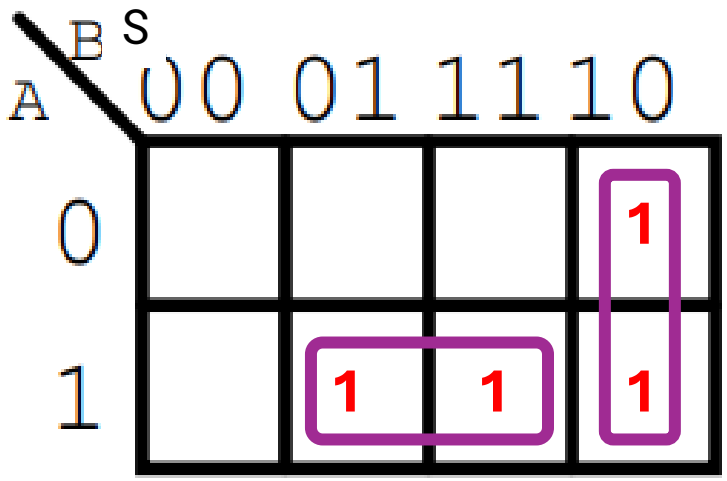
| S | Out |
|---|---|
| 0 | B |
| 1 | A |




DONE?

# Multiplexer

| A | B | S | Out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |




Out = AS + BS’

# Multiplexer in SV

- Many ways, similar results

1. If-else
2. Ternary operator
3. Unique case
4. Array indexing using a variable

```
module mux #( parameter TYPE = 0, W= 8)(
  input  logic [W-1:0] a, b,
  input  logic         sel,
  output logic [W-1:0] out
);

  if (TYPE == 0) begin : if_else

    always_comb
      if (sel) out = a;
      else     out = b;

  end else if (TYPE == 1) begin : ternary

    always_comb out = sel ? a : b;

  end else if (TYPE == 2) begin : unique_case

    always_comb
      unique case (sel)
        1'b0: out = b;
        1'b1: out = a;
      endcase

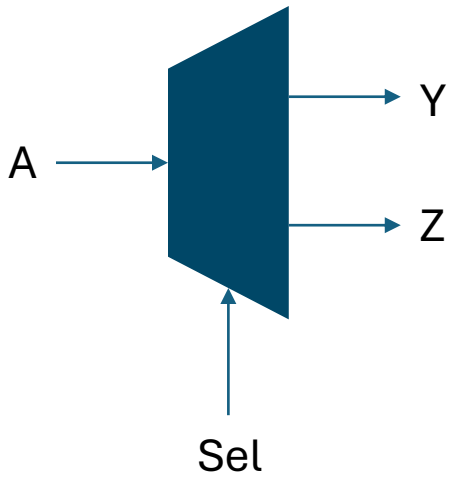
  end else begin : array_index

    logic [1:0][W-1:0] arr;

    always_comb begin
      arr = {a, b};      // arr[0]=b, arr[1]=a
      out = arr[sel];
    end

  end
endmodule
```

# Demultiplexer




| Inputs | | Outputs | |
|---|---|---|---|
| A | Sel | Y | Z |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

**Y = A . S**
**Z = A . S'**

```
`timescale 1ns/1ps


module demux #(
  parameter W     = 3,
  parameter W_SEL = 2
)(
  input  logic [W-1:0]                 in,
  input  logic [W_SEL-1:0]             sel,
  output logic [2**W_SEL-1:0][W-1:0] out
);


  always_comb begin
    out      = '0;
    out[sel] = in;
  end

                    Inside an always block,
                    Last assignment wins! – Verilog feature
endmodule

                    out=0 : general case
                    Override that by out[sel] = in;
```

# SV Functions

- Functions help to re-use logic
- They synthesize to combinational logic
- Can use in both design & testbench
- Return value:
  - Either assign to the function name
  - Or do return x;
- Automatic – function lifetime (vs static). We usually use automatic.

```
module sv_function #(
  parameter W = 8, F = W
)(
  input  logic [W-1:0] in,
  output logic [W-1:0] out
);
  localparam WA = 3*W + F + 2;

  function automatic logic [W-1:0]
           scurve (input logic [W-1:0] x);
    logic [WA-1:0] u;
    u = WA'(x);
    scurve = W'((3*u*u*(2**F) - 2*u*u*u) >> (2*F));
  endfunction


  always_comb out = scurve(in);
endmodule
```

# Look-up Tables

- Often complex functions will require too many logic cells.
- LUT is an alternative.
- Just memorize the truth table!
- Same function, but precomputed in compile time (elaboration), becomes a LUT.
- Note: the tool might decide to implement it differently

```
module look_up_table #(
  parameter W = 4,
  parameter F = W
)(
  input  logic [W-1:0] in,
  output logic [W-1:0] out
);
  localparam WA = 3*W + F + 2;

  function automatic logic [W-1:0]
            scurve (input logic [W-1:0] x);
    logic [WA-1:0] u;
    u = WA'(x);
    scurve = W'((3*u*u*(2**F) - 2*u*u*u) / 2**(2*F));
  endfunction

  logic [2**W-1:0][W-1:0] lut;
  for (genvar i = 0; i < 2**W; i++) begin : fill
    always_comb lut[i] = scurve(W'(i));
  end

  always_comb out = lut[in];
endmodule
```

# Daily Feedback

- One thing I liked about today’s class:
- One thing that could be improved:
- One place where I felt lost (if any):