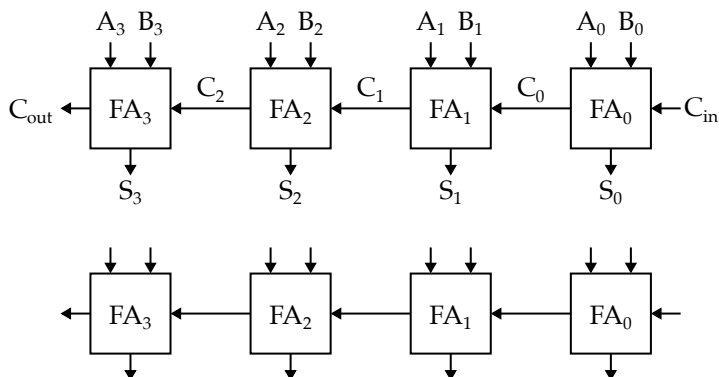


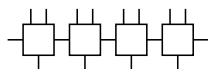
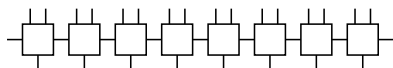
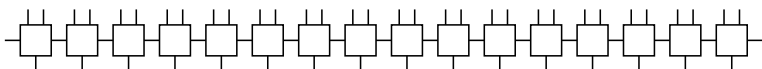
5.2. Ripple-Carry Adders

- Ripple-carry adder chains full adders together*



Path	Propagation Delay
$A_0 \rightarrow C_{out}$	
$B_0 \rightarrow C_{out}$	
$A_0 \rightarrow S_3$	
$B_0 \rightarrow S_3$	

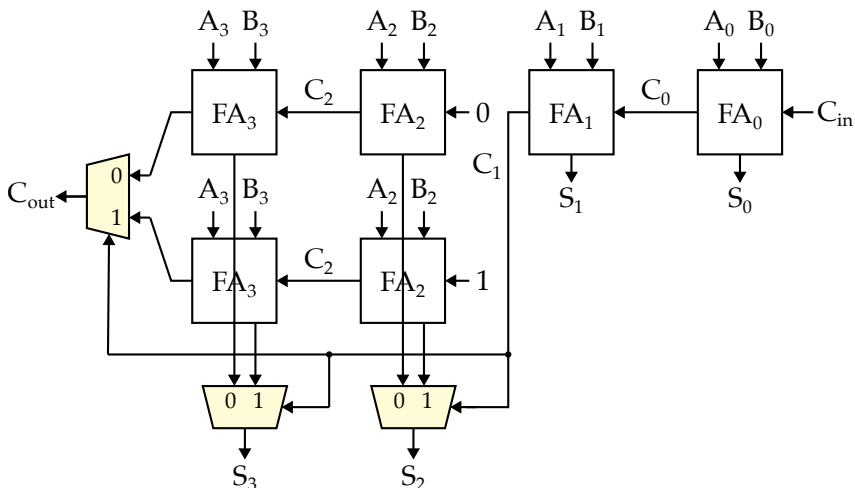
Total Area: _____

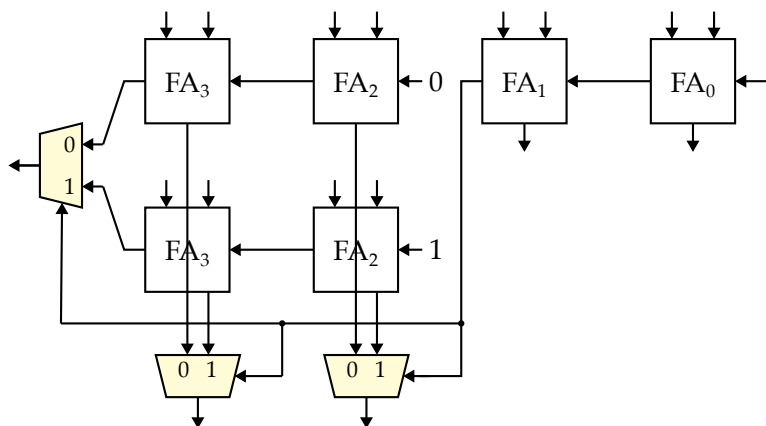
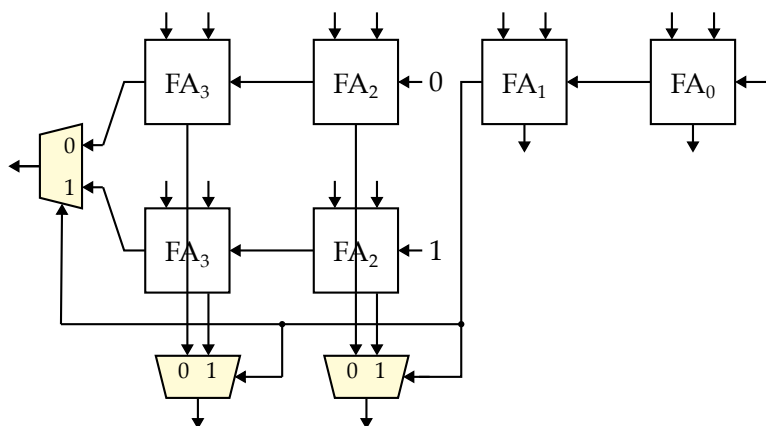
Consider 4-bit, 8-bit, 16-bit ripple carry adders $b = 4$  $b = 8$  $b = 16$ 

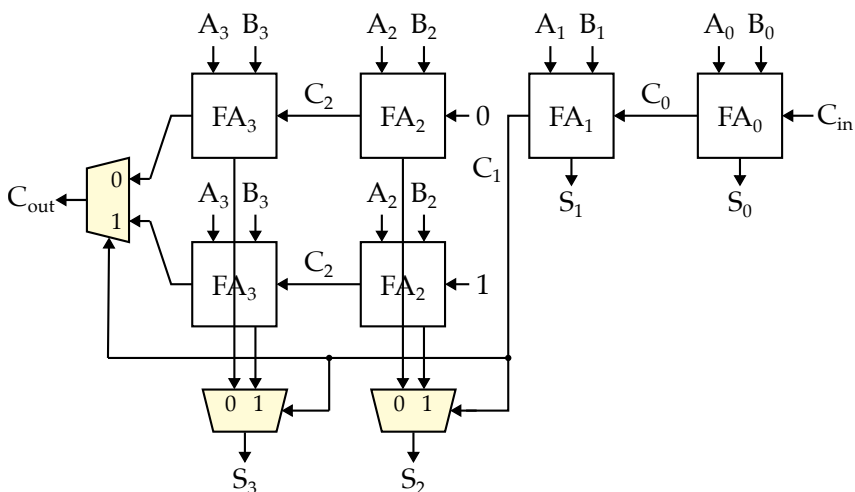
Bitwidth	Delay	Area
4		
8		
16		
b		

5.3. Carry-Select Adders

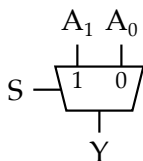
- The critical path of a ripple-carry adder is limited by the propagation delay of the carry chain
- Faster adders use additional hardware to reduce the propagation delay of the carry chain
- *Carry-select adders* break ripple carry adder into blocks; speculatively compute two versions of the result, one assumes carry in is zero, one assumes carry is one, then quickly choose correct result
 - *Block-Ripple Phase*: all blocks include two small ripple carry adders which speculatively compute in parallel (only need one ripple carry adder in first block)
 - *Select Phase*: once the actual carry in for each block is known, select the correct output for that block







Mux Path-Dependent Constant Delay Model



Mux Path	Propagation Delay
$S \rightarrow Y$	8τ
$A_0 \rightarrow Y$	7τ
$A_1 \rightarrow Y$	7τ

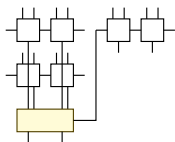
Mux Area: 11α

Path	Propagation Delay
$A_0 \rightarrow C_{out}$	
$B_0 \rightarrow C_{out}$	
$A_0 \rightarrow S_3$	
$B_0 \rightarrow S_3$	
$A_3 \rightarrow S_3$	
$B_3 \rightarrow S_3$	

Total Area: _____

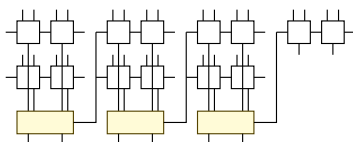
Consider 4-bit, 8-bit, 16-bit carry-select adders

$b = 4, k = 2$

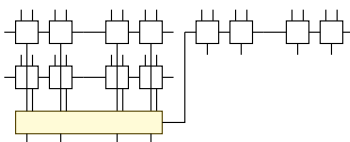


Derive equations for delay and area as a function of b on your own!

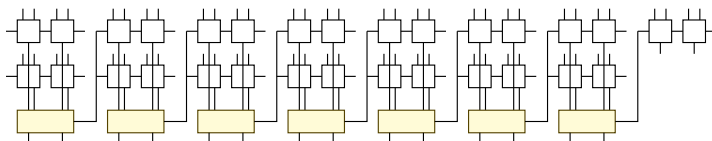
$b = 8, k = 2$



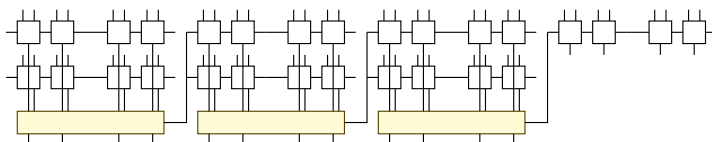
$b = 8x, k = 4$



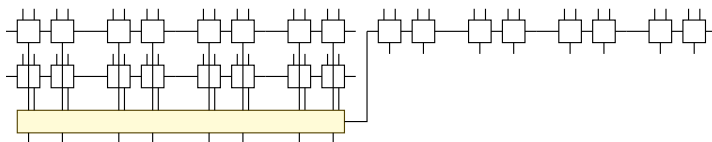
$b = 16, k = 2$



$b = 16, k = 4$

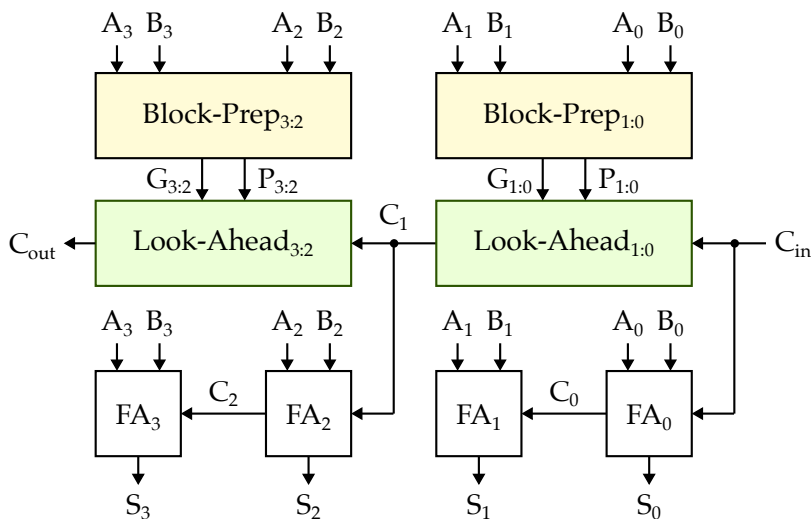


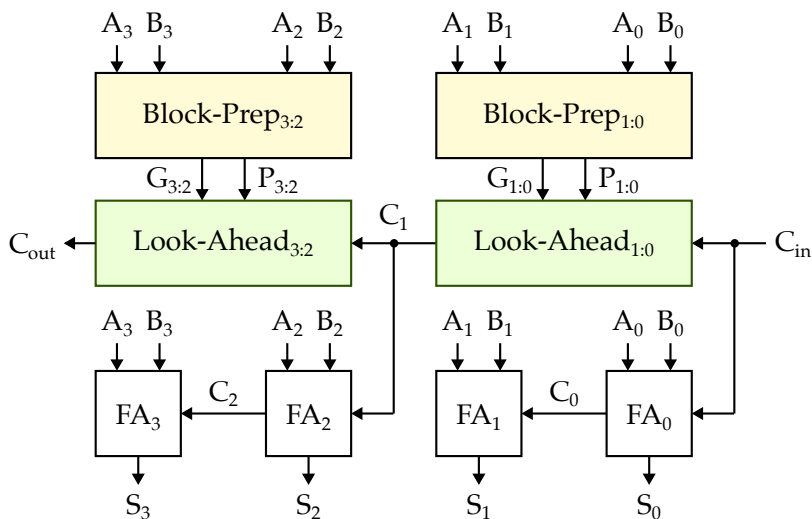
$b = 16, k = 8$

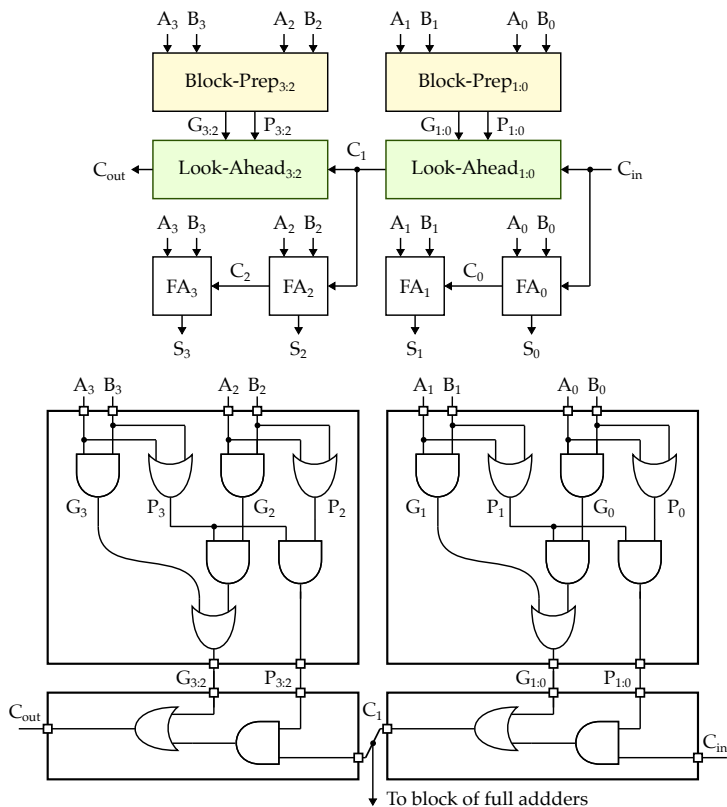


5.4. Carry Look-Ahead Adders

- Carry-look-ahead adders take a very different approach from carry-select adders but with a common goal: reduce the propagation delay of the carry chain
- *Carry-look-ahead adders* break ripple carry adder into blocks; precompute some information which then enables very quickly computing the carry in for each block
 - *Block-Prep Phase*: all blocks precompute a *propagate* and *generate* signal for each block in parallel
 - *Look-Ahead Phase*: carry chain uses precomputed propagate and generate signals to quickly compute the carry in for each block
 - *Block-Ripple Phase*: each block computes the sum in parallel using small ripple-carry adders







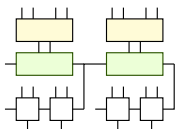
Gate	t_{pd}	Area
NOT	1τ	1α
AND2	3τ	3α
OR2	4τ	4α
XOR2	7τ	7α

Total Area: _____

Path	Propagation Delay
$A_1 \rightarrow C_{out}$	
$B_1 \rightarrow C_{out}$	
$A_1 \rightarrow S_3$	
$B_1 \rightarrow S_3$	

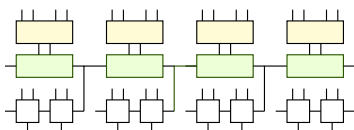
Consider 4-bit, 8-bit, 16-bit carry look-ahead adders

$b = 4, k = 2$

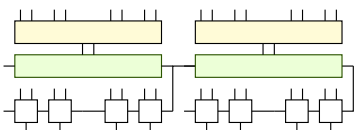


Derive equations for delay and area as a function of b on your own!

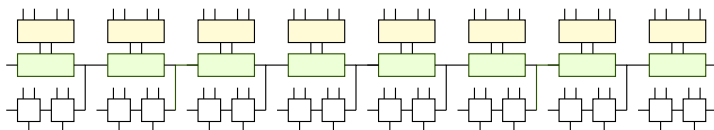
$b = 8, k = 2$



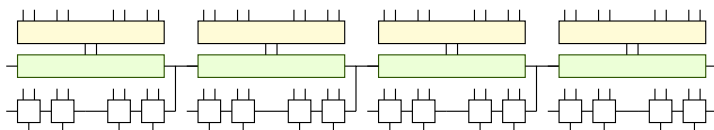
$b = 8, k = 4$



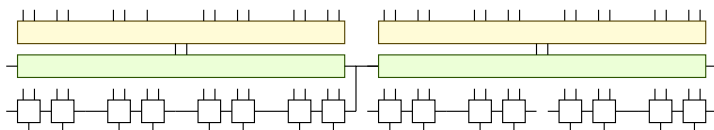
$b = 16, k = 2$



$b = 16, k = 4$

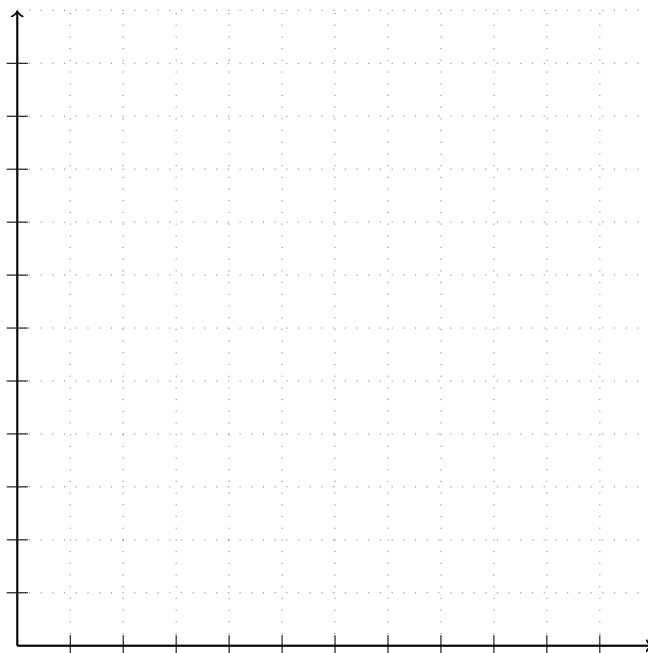


$b = 16, k = 8$



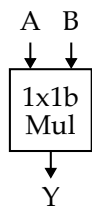
5.5. Comparing Adders

Adder Implementation	Delay	Area
Ripple-Carry		
Carry-Select		
Carry-Look-Ahead		

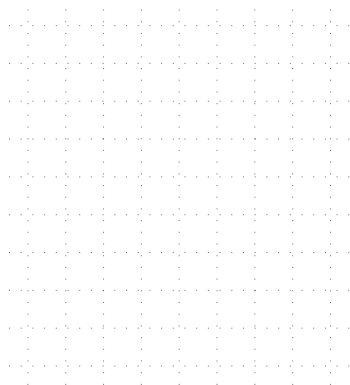


6. Multipliers

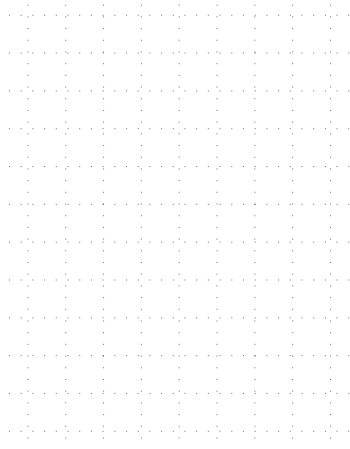
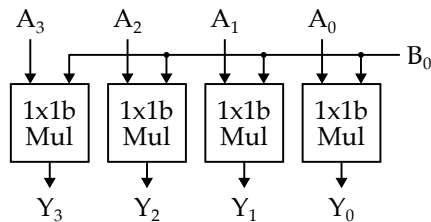
6.1. 1x1b Multiplier



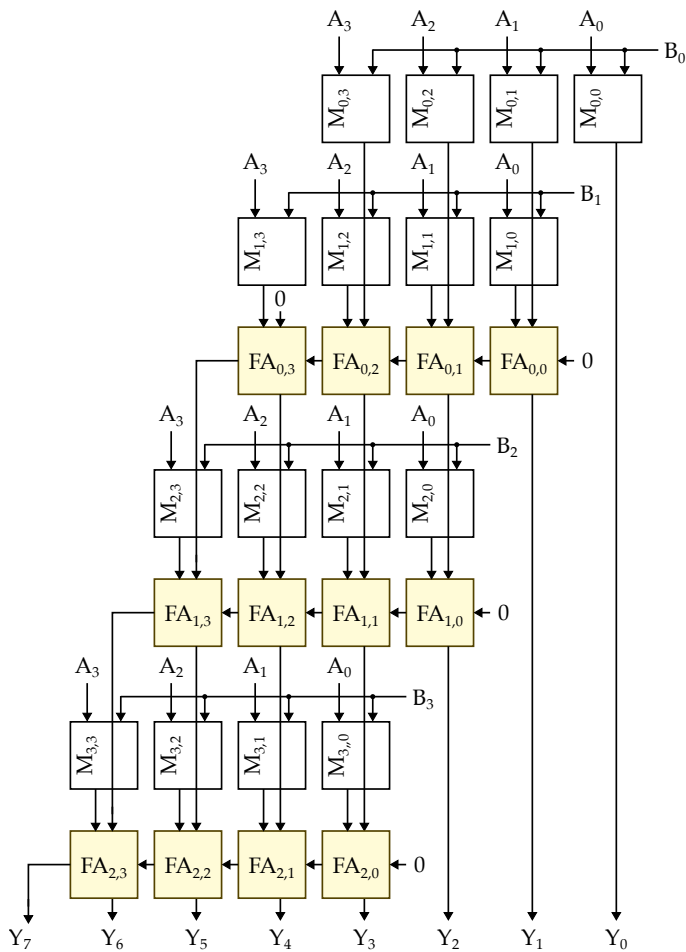
A	B	Y
0	0	
0	1	
1	0	
1	1	



6.2. 1x4b Multiplier



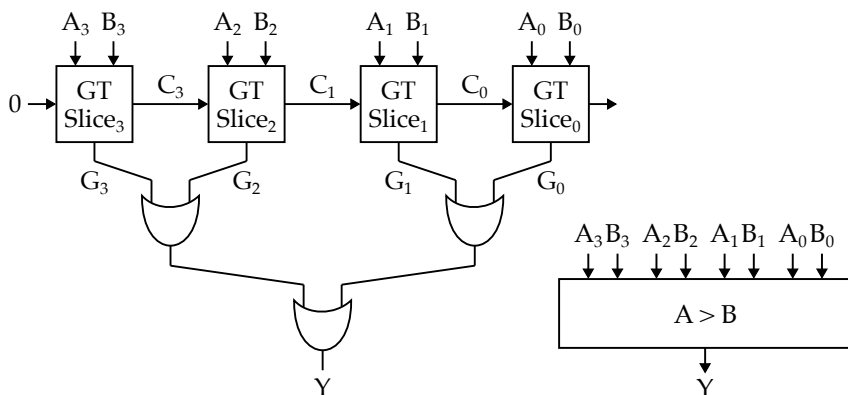
6.3. 4x4b Multiplier



7. Verilog Modeling

- *Gate-level modeling* explicitly instantiates gates
- *Register-transfer-level modeling* (RTL) captures how data moves through hardware at a higher level of abstraction
 - Boolean equation modeling is a kind of RTL modeling
 - Simple RTL modeling: use more abstract multi-bit operators
 - Complex RTL modeling: use `if/case` statements in `always` blocks

7.1. Greater Than Comparator



```

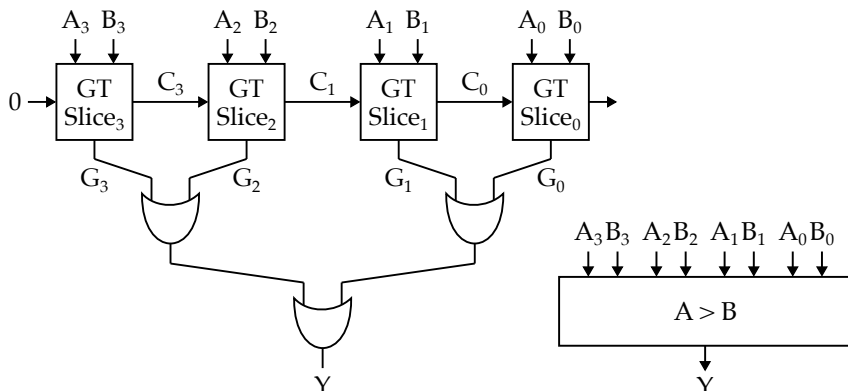
1 module GTComparator_1b_GL
2 (
3     input wire cin,
4     input wire a,
5     input wire b,
6     output wire cout
7     output wire x,
8 );
9
10 wire b_n, w0;
11 not( b_n, b );
12 and( w0, b_n, cin );
13 and( x, a, w0 );
14
15 wire w1;
16 xnor( w1, a, b );
17 and( cout, cin, w1 );
18
19 endmodule

```

```

1 module GTComparator_4b_GL
2 (
3     input wire [3:0] a,
4     input wire [3:0] b,
5     output wire      y
6 );
7 wire [3:0] x;
8 wire [3:0] c;
9
10 GTComparator_1b_GL cmp3
11 (
12     .cin (1'b0),
13     .a   (a[3]),
14     .b   (b[3]),
15     .cout (c[3]),
16     .x    (x[3])
17 );
18
19 ...
20
21 endmodule

```



```

1 module GTComparator_1b_RTL
2 (
3     input wire cin,
4     input wire a,
5     input wire b,
6     output wire cout,
7     output wire x
8 );
9
10 assign cout = cin & ~( a ^ b );
11 assign x = cin & a & ~b;
12
13 endmodule

```

```

1 module GTComparator_4b_RTL
2 (
3     input wire [3:0] a,
4     input wire [3:0] b,
5     output wire      y
6 );
7     wire [3:0] x;
8     wire [3:0] c;
9
10    GTComparator_1b_RTL cmp3
11    (
12        .cin  (1'b0),
13        .a    (a[3]),
14        .b    (b[3]),
15        .cout (d[3]),
16        .x    (x[3])
17    );
18
19    ...
20
21 endmodule

```

- Simple RTL modeling provides a larger set of operators that can operate on multi-bit signals
- Use logic for RTL modeling
- Even though we are raising the level of abstraction, **must always keep in mind the hardware we are modeling!**

```

1 module GTComparator_4b_RTL
2 (
3     input  logic [3:0] a,
4     input  logic [3:0] b,
5     output logic       y
6 );
7
8     assign y = a > b;
9
10 endmodule

```

Reduction Operators

&a	reduce via AND
~&a	reduce via NAND
a	reduce via OR
~ a	reduce via NOR
^a	reduce via XOR
~^a	reduce via XNOR

Relational Operators

a == b	equal
a != b	not equal
a > b	greater than
a >= b	greater than or equals
a < b	less than
a <= b	less than or equals

Shift Operators

a >> b	shift right
a << b	shift left

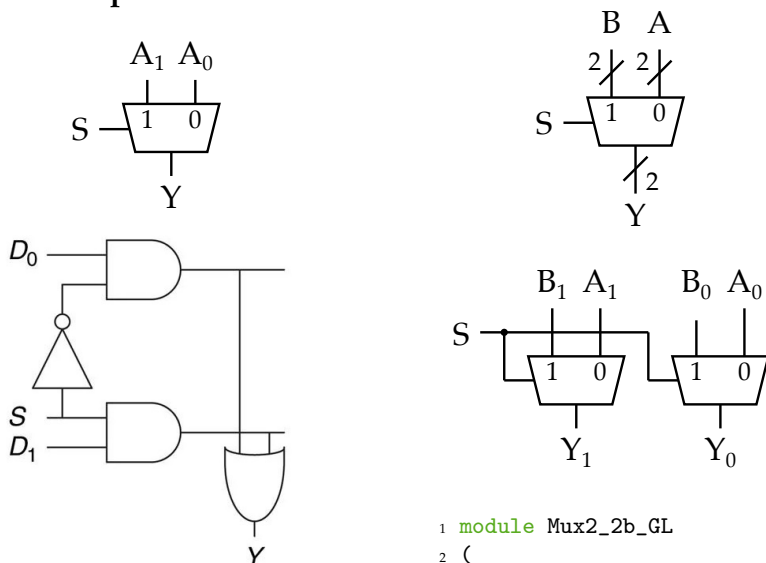
Arithmetic Operators

a + b	addition
a - b	subtraction
a * b	multiplication

Other Operators

a?b:c	ternary
{a,b}	concatenate
{N{a}}	replicate N times

7.2. Multiplexor



```

1 module Mux2_1b_GL
2 (
3     input wire a,
4     input wire b,
5     input wire sel,
6     output wire y,
7 );
8     wire sel_b;
9     not( sel_b, sel );
10
11     wire w0, w1;
12     and( w0, a, sel );
13     and( w1, sel_b, b );
14     or( y, w0, w1);
15
16 endmodule

```

```

1 module Mux2_2b_GL
2 (
3     input wire [1:0] a,
4     input wire [1:0] b,
5     input wire      sel,
6     output wire [1:0] y,
7 );
8
9     Mux2_1b_GL mux0
10    (
11        .a  (a[0]),
12        .b  (b[0]),
13        .sel (sel),
14        .y  (y[0])
15    );
16
17    ...
18
19 endmodule

```

```

1 module Mux2_1b_RTL
2 (
3   input logic a,
4   input logic b,
5   input logic sel,
6   output logic y,
7 );
8   assign y = (a & sel)
9             | (b & ~sel);
10 endmodule

```

```

1 module Mux2_2b_RTL
2 (
3   input logic [1:0] a,
4   input logic [1:0] b,
5   output logic      y
6 );
7
8   assign y = sel ? b : a;
9
10 endmodule

```

- always_comb blocks can raise level of abstraction even further, but **must always keep in mind the hardware we are modeling!**

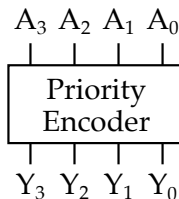
```

1 module Mux2_2b_RTL
2 (
3   input logic [1:0] a,
4   input logic [1:0] b,
5   input logic      sel,
6   output logic [1:0] y
7 );
8
9   always_comb begin
10     case ( sel )
11       1'd0 : y = a;
12       1'd1 : y = b;
13       default : y = 'x;
14     endcase
15   end
16
17 endmodule

```

- In this course, an always_comb block is only allowed to have a single case statement
 - Must have a default statement
 - default statement must assign 'x to all signals written in the case
- No other statements are allowed in an always_comb
 - No blocking assignment
 - No if/else statements

7.3. Priority Encoder



```

1 module PriorityEncoder_4b_RTL
2 (
3     input  logic [3:0] a,
4     output logic [3:0] y
5 );
6
7     always_comb begin
8         case ( a )
9
10            4'b0000 : y = 4'b0000;
11            4'b0001 : y = 4'b0001;
12            4'b0010 : y = 4'b0010;
13            4'b0011 : y = 4'b0001;
14
15            4'b0100 : y = 4'b0100;
16            4'b0101 : y = 4'b0001;
17            4'b0110 : y = 4'b0010;
18            4'b0111 : y = 4'b0001;
19
20            4'b1000 : y = 4'b1000;
21            4'b1001 : y = 4'b0001;
22            4'b1010 : y = 4'b0010;
23            4'b1011 : y = 4'b0001;
24
25            4'b1100 : y = 4'b0100;
26            4'b1101 : y = 4'b0001;
27            4'b1110 : y = 4'b0010;
28            4'b1111 : y = 4'b0001;
29
30            default : y = 'x;
31        endcase
32    end
33
34 endmodule

```

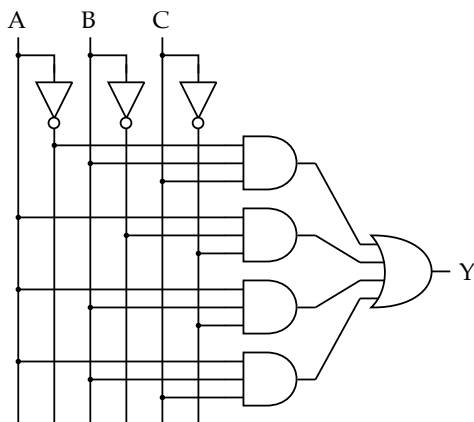
```

1 module PriorityEncoder_4b_RTL
2 (
3     input  logic [3:0] a,
4     output logic [3:0] y
5 );
6
7     always_comb begin
8         casez ( a )
9            4'b0000 : y = 4'b0000;
10           4'b0001 : y = 4'b0001;
11           4'b001? : y = 4'b0010;
12           4'b01?? : y = 4'b0100;
13           4'b1??? : y = 4'b1000;
14           default : y = 'x;
15        endcase
16    end
17
18 endmodule

```

7.4. Truth Tables

A	B	C	Y	
0	0	0	0	m_0
0	0	1	0	m_1
0	1	0	0	m_2
0	1	1	1	m_3
1	0	0	1	m_4
1	0	1	0	m_5
1	1	0	1	m_6
1	1	1	1	m_7



```

1 module TruthTable1
2 (
3   input wire a,
4   input wire b,
5   input wire c,
6   output wire y
7 );
8
9   always_comb begin
10     case ( { a, b, c } )
11       3'b000 : y = 1'b0;
12       3'b001 : y = 1'b0;
13       3'b010 : y = 1'b0;
14       3'b011 : y = 1'b1;
15       3'b100 : y = 1'b1;
16       3'b101 : y = 1'b0;
17       3'b110 : y = 1'b1;
18       3'b111 : y = 1'b1;
19     default : y = 'x;
20   endcase
21 end
22
23 endmodule

```

```

1 module TruthTable2
2 (
3     input wire a,
4     input wire b,
5     output wire y,
6     output wire z
7 );
8
9     task automatic set_outputs
10    (
11        input logic y_,
12        input logic z_
13    );
14        y = y_;
15        z = z_;
16    endtask
17
18    always_comb begin
19        case ( { a, b } )
20            //
21            { 1'b0, 1'b0 } : set_outputs( y, z );
22            { 1'b0, 1'b1 } : set_outputs( y, z );
23            { 1'b1, 1'b0 } : set_outputs( y, z );
24            { 1'b1, 1'b1 } : set_outputs( y, z );
25            default : y = 'x;
26        endcase
27    end
28
29 endmodule

```

	A	B	Y	Z
0	0	0	1	0
1	0	1	1	1
2	1	0	0	0
3	1	1	0	1

