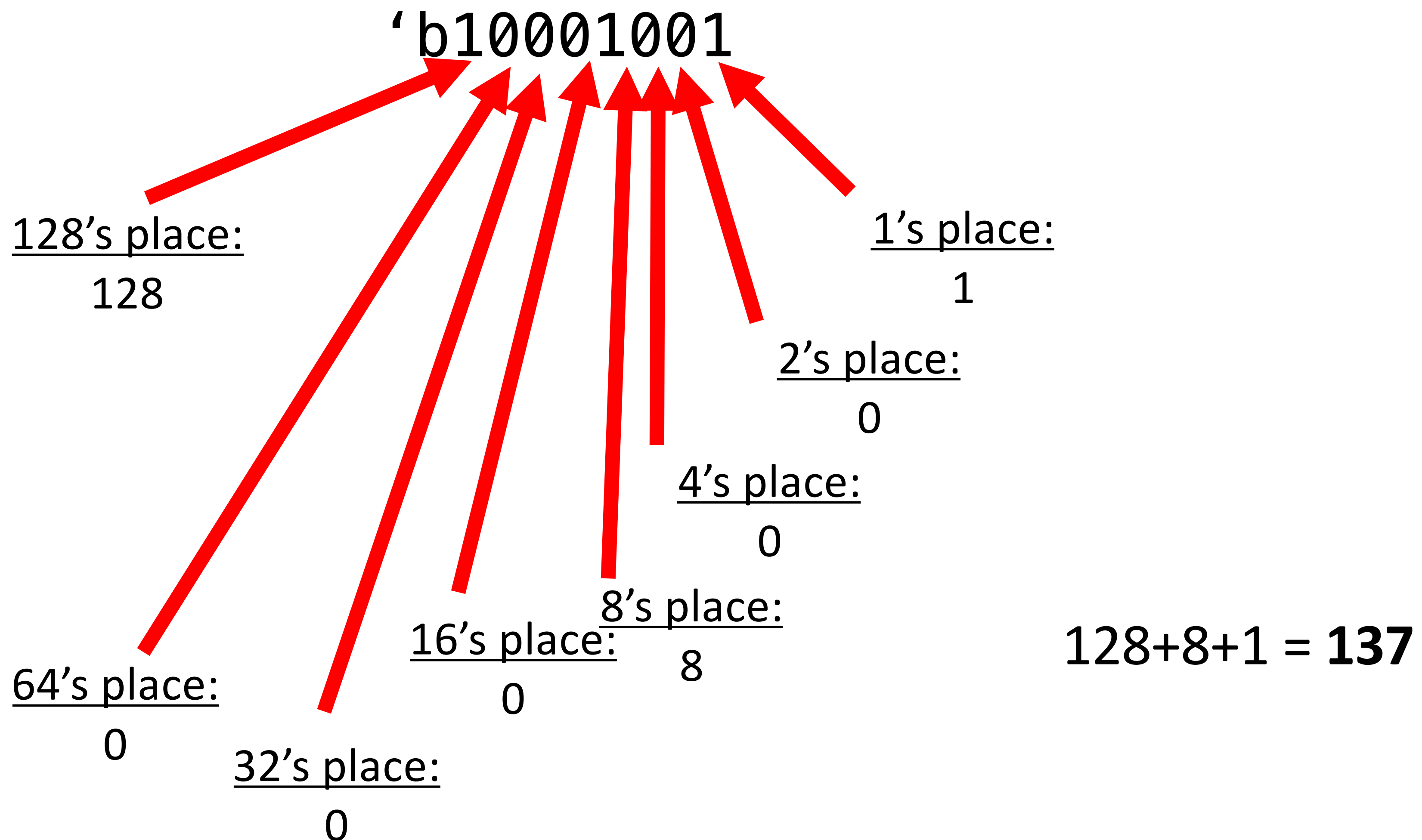
Pipelining, Math

Administrative

- Pset 08 due minutes ago.
- Pset 09 out tonight…it is just questions to prepare for final project….you're welcome.
- Lab 04A due tomorrow at 5pm.
- Lab 04B out soon after class.
- No Office hours until Tuesday (Tuesday is a Sunday in terms of lab staffing)

How to Represent Numbers

- Simplest approach is to just read the binary number in regular base 2 (just like in our friend base 10!)



128+8+1 = **137**

Most arithmetic works out well too!

- Add/Subtract:

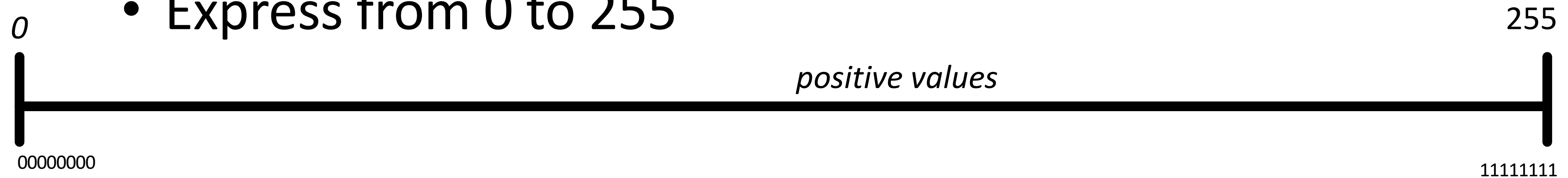
```
   b10001001      (137)
+  b00000101        (5)
-------------------------
   b10001110      (142)
```

- Multiply/Divide:

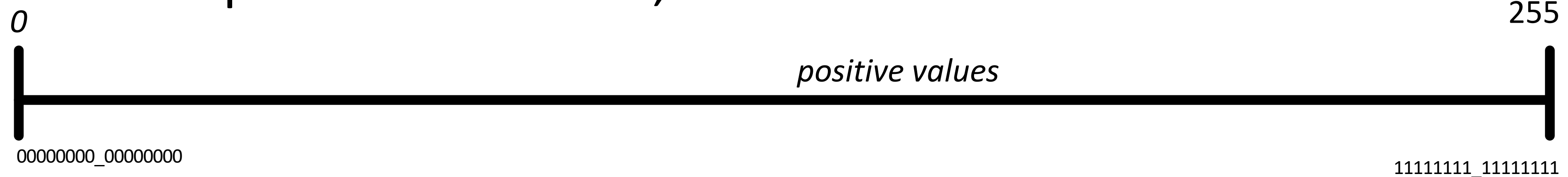
```
     ‘b00000101          (5)
   * ‘b00000110          (6)
-----------------------------
    ‘b00000000           (0)
   ‘b000001010          (10)
 +’b0000010100          (20)
-----------------------------
  ‘b000011110           (30)
```

Unsigned Values:

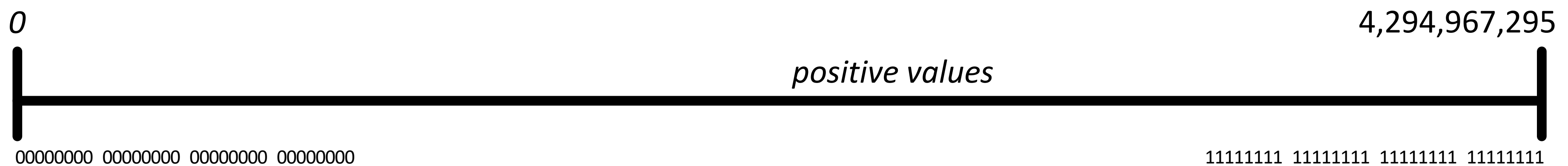
- 1 byte (8 bits): 2^8 values: 256 numbers to rep
  - Express from 0 to 255



- 2 bytes (16 bits): 2^{16} values: 65,536 numbers
  - Express from 0 to 65,535

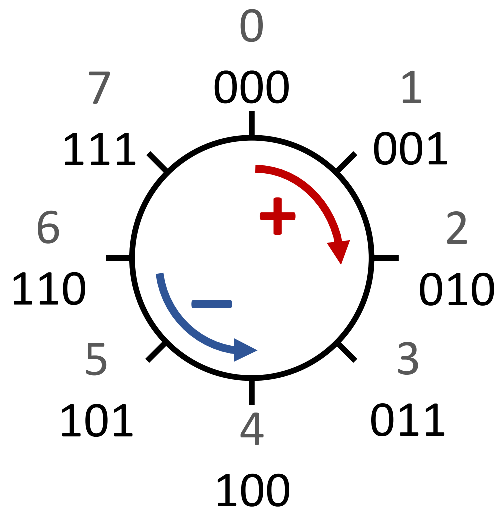


- 4 bytes (32 bits): 2^{32} values: 4,294,967,296 nums
  - Express from 0 to 4,294,967,295

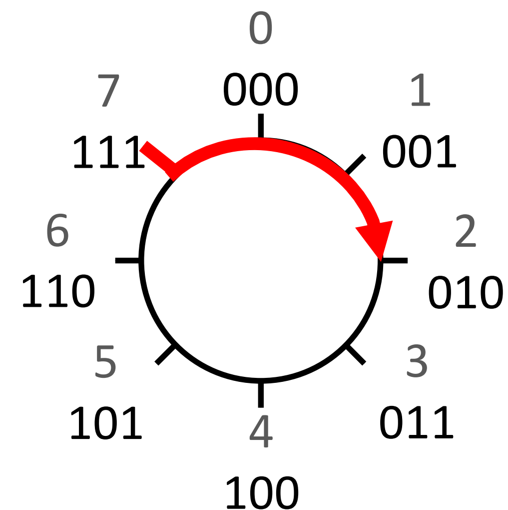


Inherent Modularity

- If we use a fixed number of bits, addition and other operations may produce results outside the range that the output can represent (up to 1 extra bit for addition)
  - This is known as an overflow
- Common approach: Ignore the extra bit
  - Gives rise to modular arithmetic: With N-bit numbers, equivalent to following all operations with mod 2^N
  - Visually, numbers "wrap around":



Example: (7 + 3) mod 2^3 ?

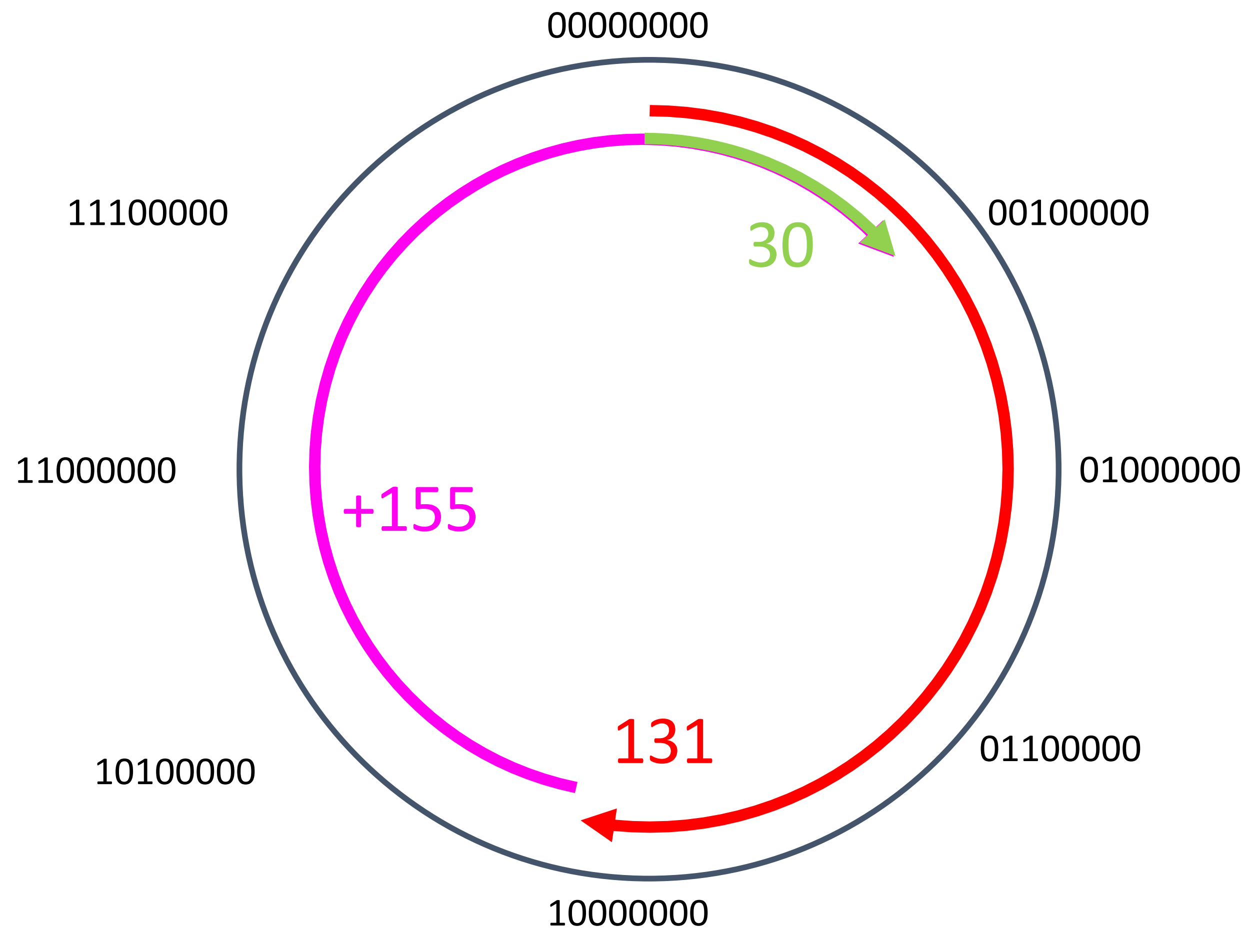


Happens with more bits too (8 bits)

- What happens if you add 131 to 155 with 8 bit?

```
   10000011   (131)
+  10011011   (155)
  ------------------
  100011110   (286)
   00011110   (30)
```

overflow



What About Negatives?

- Our Number Schemes so far only allow representation of positive numbers (and zero).
- What about negatives? How can we do this in an efficient manner?

One Solution: “Sign Bit”

- If most-significant-bit (msb) is 0, interpret like a negative sign:
  - If 0, lower bits are from a positive number
  - If 1, lower bits are from a negative number
- To get the negative of the number, flip the msb:

‘b00010001 == +(16+1) == 17

‘b10010001 == −(16+1) == −17

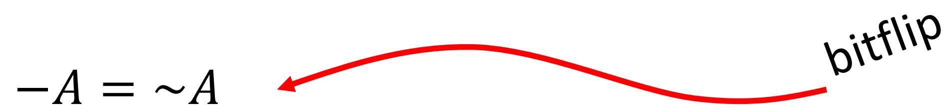
‘b00000000 == 0
‘b10000000 == −0

- Major problem(s)?

Another Solution: “One’s Complement”

- If most-significant-bit (msb) is 0, interpret like an unsigned value.
- If msb is 1, then number is negative, else positive.
- To get the negative of the number flip all the bits:

$$-A = \sim A$$




```
‘b00010001 == +(16+1) == 17

‘b11101110 == bitflip of 17== -17

‘b00000000 == 0
‘b11111111 == -0
```

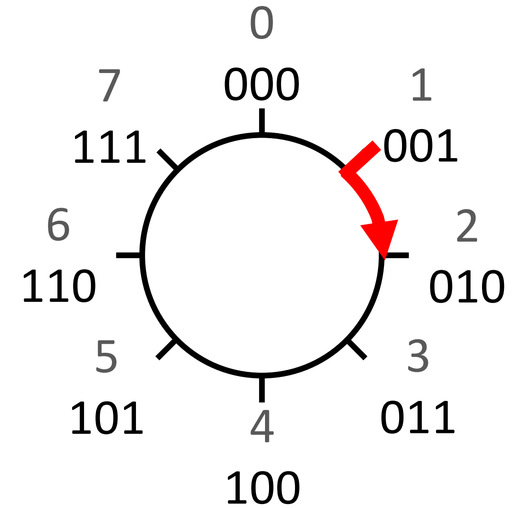
- Major problem(s)?

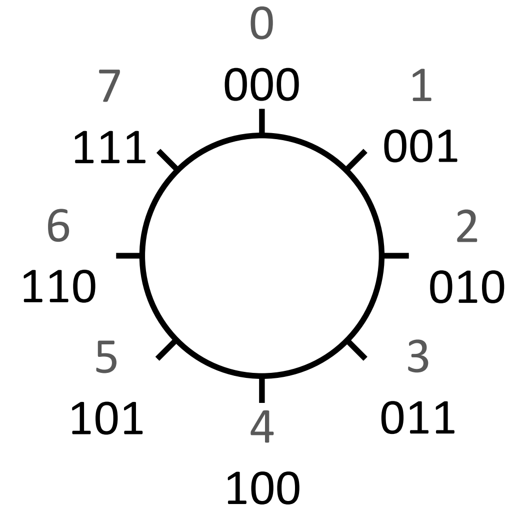
Inherent Modularity to the Rescue

- Return to our 3-bit* number system:

- If I want to add 1, I just add 1 and move clockwise by 1 unit

- If I want to subtract 1, is there a number I could add to get the same result? If so, that number could be seen as “-1” right?

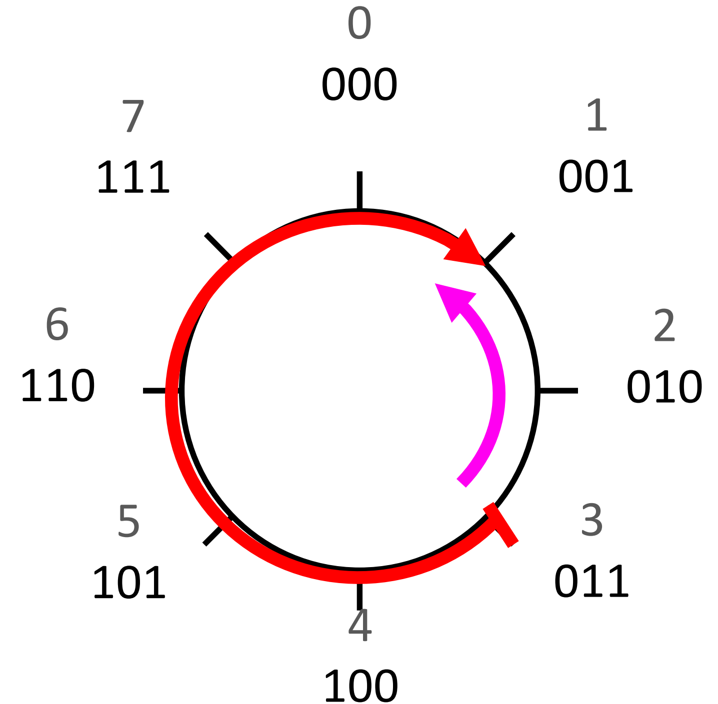




*3 bits here since easy to think about and draw, but could do with any number of bits

A Negative Number

- If I start at ”3” aka 011, what could I add to get to 1?
- To go back 2, I can add:
  - $2^3 - 2$ = 6
- (3+6)%8 = 1.
- Or: “-010” = 110



Negating a Number

- The negative of a number can be expressed as:

$$\text{“}-A\text{”} = 8 - A$$

- Or written a different way:

$$\text{“}-A\text{”} = 1 + \text{`b111} - A$$

- ‘b111 *minus* any 3 bit value will be the same as the bitflip of that value (~A)

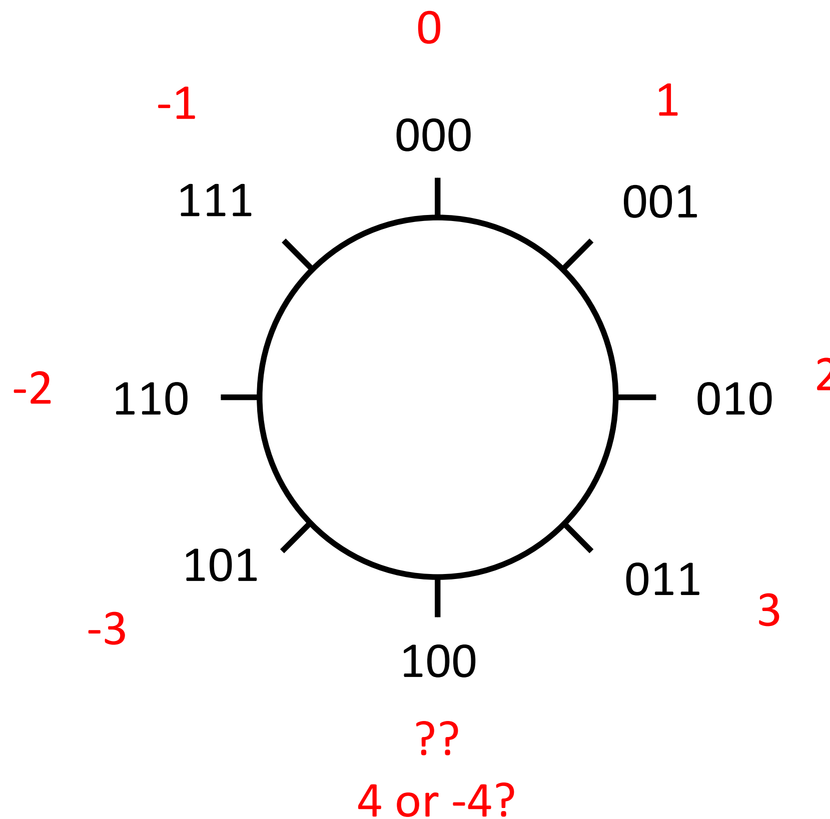
$$\text{“}-A\text{”} = 1 + (\text{`b111} - A)$$

- So the negative of any value must be:

$$-A = 1 + {\sim}A$$

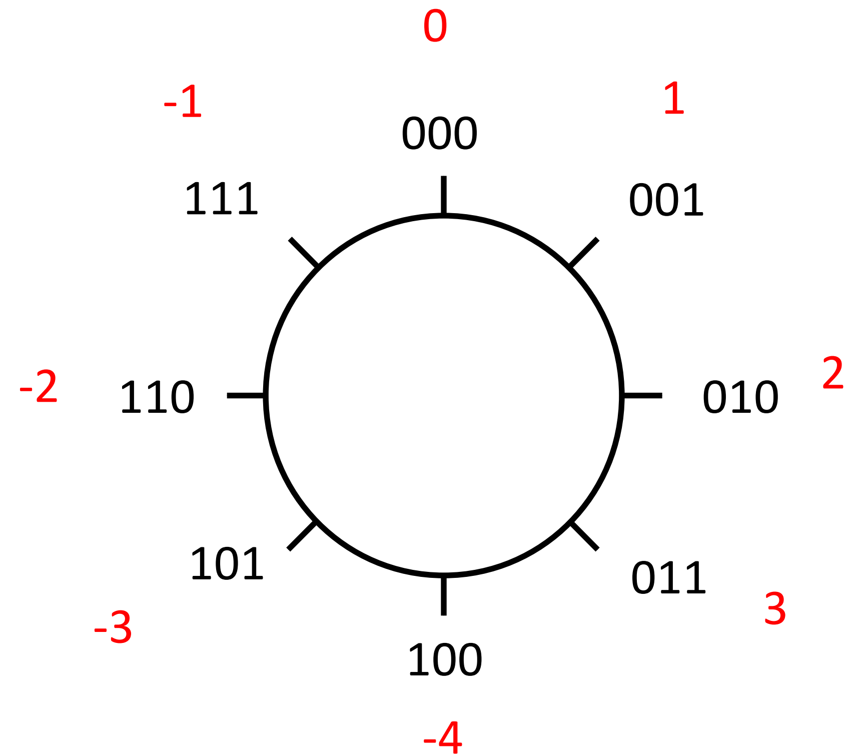
The Solution: 2's Complement

- For 000 to 111 what numbers do we get in this scheme?



Interesting…

- If we make 100 into -4, the system of numbers becomes consistent and easily extensible to more bits.



- With this model we can come up with some rules/observations…

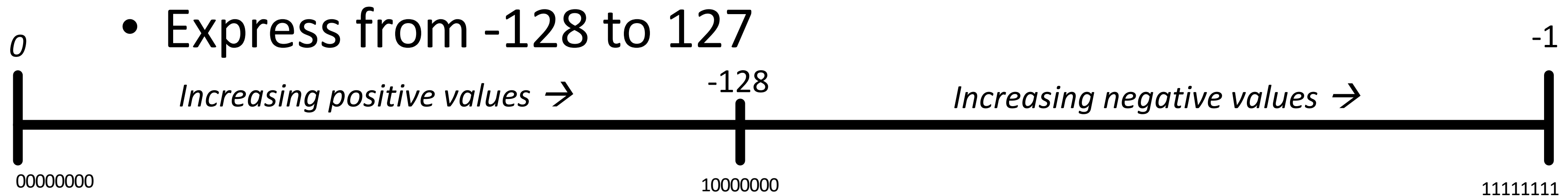
Two's Complement (Signed) Ints

- For an n bit signed int, we represent from:
  - Min: -2^{n-1}
  - Max: $2^{n-1} - 1$
  - Zero is always *all* zeros
- The negative of a number A is always $-A = 1 + \sim A$
- A number is positive if the msb is 0:
  - If so, just add up non-zero digits by weight as you do for unsigned
- A number is negative if msb is 1:
  - If so add weight of msb, then for all bits below that subtract off the weight of any non-zero digits:

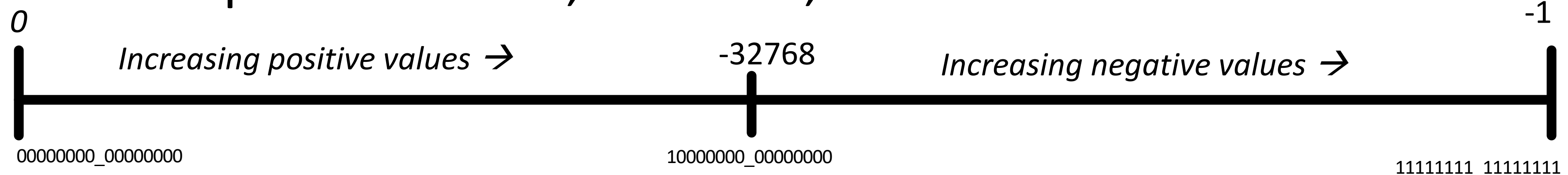
*msb = most significant bit

Signed Ints (ints):

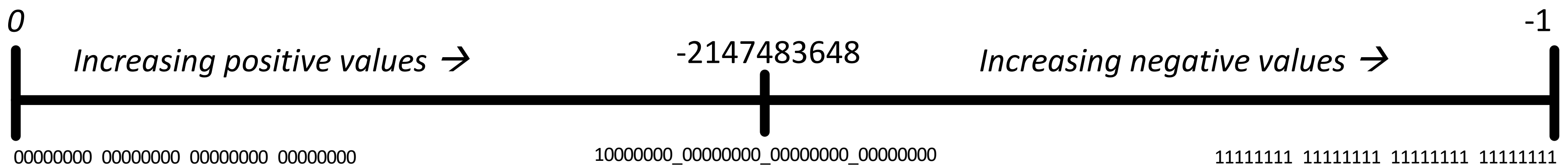
- 1 byte (8 bits): 2^8 values: 256 numbers to rep
  - Express from -128 to 127



- 2 bytes (16 bits): 2^{16} values: 65,536 numbers
  - Express from -32,768 to 32,767



- 4 bytes (32 bits): 2^{32} values: 4,294,967,296 nums
  - Express from -2,147,483,648 to 2,147,483,647



Math Operations Still Work

- Two's Complement is pretty nice because you can still do all your regular math operations pretty easily
- Also No double-zero!
- Pretty much all modern digital systems use two's complement math to represent signed integers

Signed Arithmetic in Verilog

Just add “signed” modifier to your variable declaration.

```
logic [15:0] a; // Unsigned
logic signed [16:0] signed_a; //signed
```

Using Signed Arithmetic in Verilog

ALL OF THE FOLLOWING ARE TREATED AS ***UNSIGNED*** IN VERILOG!!!

- *Any* operation on two operands, unless **both operands are signed**
- Based numbers (e.g. 12’d10), unless the explicit “s” modifier is used)
- Bit-select results $a[5]$
- Part-select results $a[4:2]$
- Concatenations

```
logic [15:0] a; // Unsigned
logic signed [15:0] b;
logic signed [16:0] signed_a;
logic signed [31:0] a_mult_b;

assign signed_a = a;//Convert to signed
assign a_mult_b = signed_a * b
```

Example of multiplying signed by unsigned

http://billauer.co.il/blog/2012/10/signed-arithmetics-verilog/

For example, consider these two test bench examples:

```
module test_one;
  logic signed [3:0] x;
  logic [3:0] y;
  logic signed [8:0] z;
  initial begin
    x = −2;
    y=3;
    z = x*y;
    $display(x, y, z);
    $finish;
  end
endmodule
```

Result:

-2 3 42

```
module test_two;
  logic signed [3:0] x;
  logic signed [3:0] y;
  logic signed [8:0] z;
  initial begin
    x = −2;
    y=3;
    z = x*y;
    $display(x, y, z);
    $finish;
  end
endmodule
```

Result:

-2 3 -6

Not really synthesizable here ($finish, $display, etc)…but shows what Verilog is thinking

Sign extension

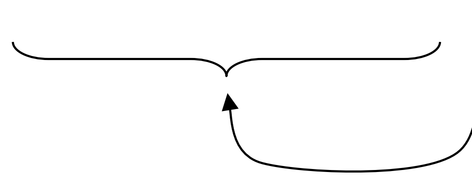
Consider the 8-bit 2's complement representation of:

```
42 = 00101010        -5 = ~00000101 + 1
                         =  11111010 + 1
                         =  11111011
```

What is their 16-bit 2's complement representation?

```
42 = 0000000000101010

-5 = 1111111111111011
```



Extend the MSB (aka the "sign bit") into the higher-order bit positions

Using Signed Arithmetic in Verilog

"**<<<**" and "**>>>**" tokens result in arithmetic (signed) left and right shifts: multiple by 2 and divide by 2.

Right shifts will maintain the sign by filling in with sign bit values during shift

```
logic signed [3:0] x;
logic signed [3:0] value = 4'b1000; // -8
x = value >> 2 // results in 0010 or 2
x = value >>> 2 // results in 1110 or -2

logic [3:0] value = 4'b1000; // -8
x = value >> 2 // results in 0010 or 2
x = value >>> 2 // results in 0010 or -2 (is unsigned…extends with 0's)
```

Few Other Things

- When specifying numbers/constants you cand put a **s** in front to specify it as signed.

```
logic signed [7:0] x;
initial begin
  x = -'d5;
  $display("%d %8b", x,x); //prints: -5 11111011
  x = -'sd5;
  $display("%d %8b", x,x); //prints: -5 11111011
  x = 'd5;
  $display("%d %8b", x,x); //prints: 5 00000101
  x = 'sd5;
  $display("%d %8b", x,x); //prints: 5 00000101
  x = 'd234;
  $display("%d %8b", x,x); //prints: -22 11101010
  x = 'sd128;
  $display("%d %8b", x,x); //prints: -128 10000000
  #100;
  $finish;
end
```

```
logic [7:0] x;
initial begin
  x = -'d5;
  $display("%d %8b", x,x); //prints: 251 11111011
  x = -'sd5;
  $display("%d %8b", x,x); //prints: 251 11111011
  x = 'd5;
  $display("%d %8b", x,x); //prints: 5 00000101
  x = 'sd5;
  $display("%d %8b", x,x); //prints: 5 00000101
  x = 'd234;
  $display("%d %8b", x,x); //prints: 234 11101010
  x = 'sd128;
  $display("%d %8b", x,x); //prints: 128 10000000
  #100;
  $finish;
end
```

Need to make a thing signed?

- Either use $signed
- Or declared signed types to route through:

```
logic signed [3:0] x = 4'b1110; // -2 also -4'
logic [3:0] y = 4'b1100; //12 unsigned, (-4 signed)
logic signed [4:0] z
assign z = x*$signed(y);//interpret y as signed
//results in z having 5’b11000 in it (-8)
//OR:
logic signed [3:0] y_signed;
assign y_signed = y;
assign z = x*y_signed; //multiplication of two signed things is signed
//results in z having 5’b11000 in it (-8)
```

Signed Numbers Guideline

- Once you start using signed Verilog, just make everything you're using is signed. If you do that, you should be ok.
- Make sure everything upstream of a calculation has been done in only a signed environment (held in signed logics and used with signed logics.
- Signed/Unsigned bugs are some of the hardest to find so be cautious
- When in doubt also use $signed

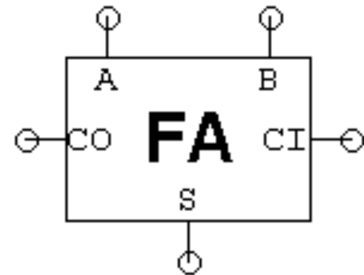
The Complexity of Math Operations

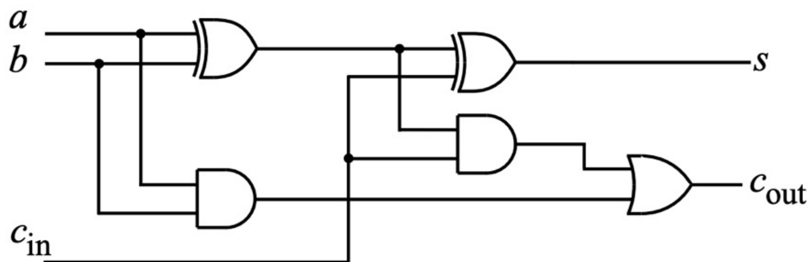
Let’s look at some basic math circuits:

+, -, *, /

"Full Adder" building block

The "half adder" circuit has only the A and B inputs (no carry) Full adders handle carry bits





A	B	CI	S	CO
0	0	0	0	0
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
1	1	0	0	1
1	1	1	1	1

Adder: a circuit that does addition

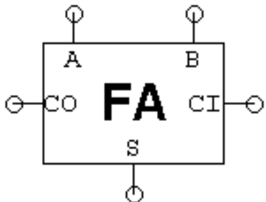
Here's an example of binary addition as one might do it by "hand":

Carries from previous column

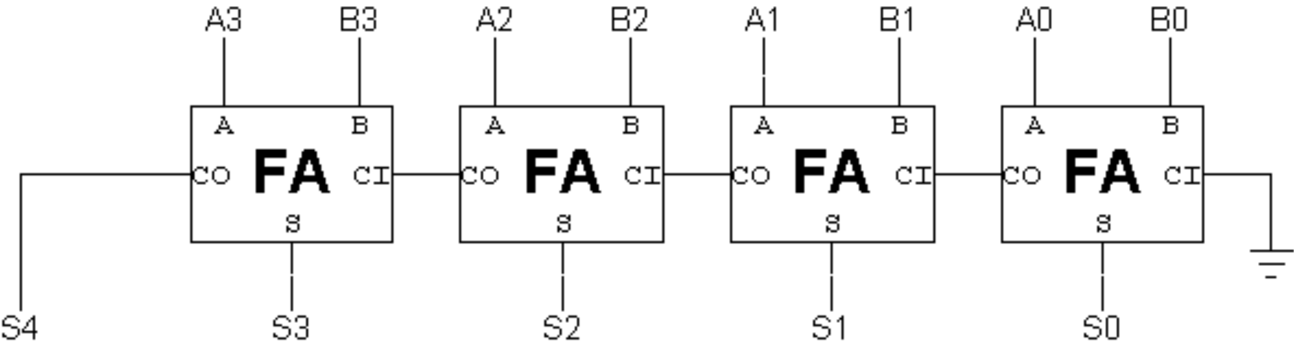
```
 1 1 0 1
   1101
+  0101
  -----
  10010
```

Adding two N-bit numbers produces an (N+1)-bit result

If we build a circuit that implements one column:



we can quickly build a circuit to add two 4-bit numbers…

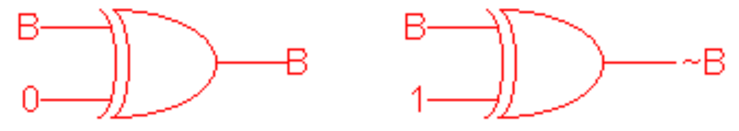


"Ripple-carry adder"

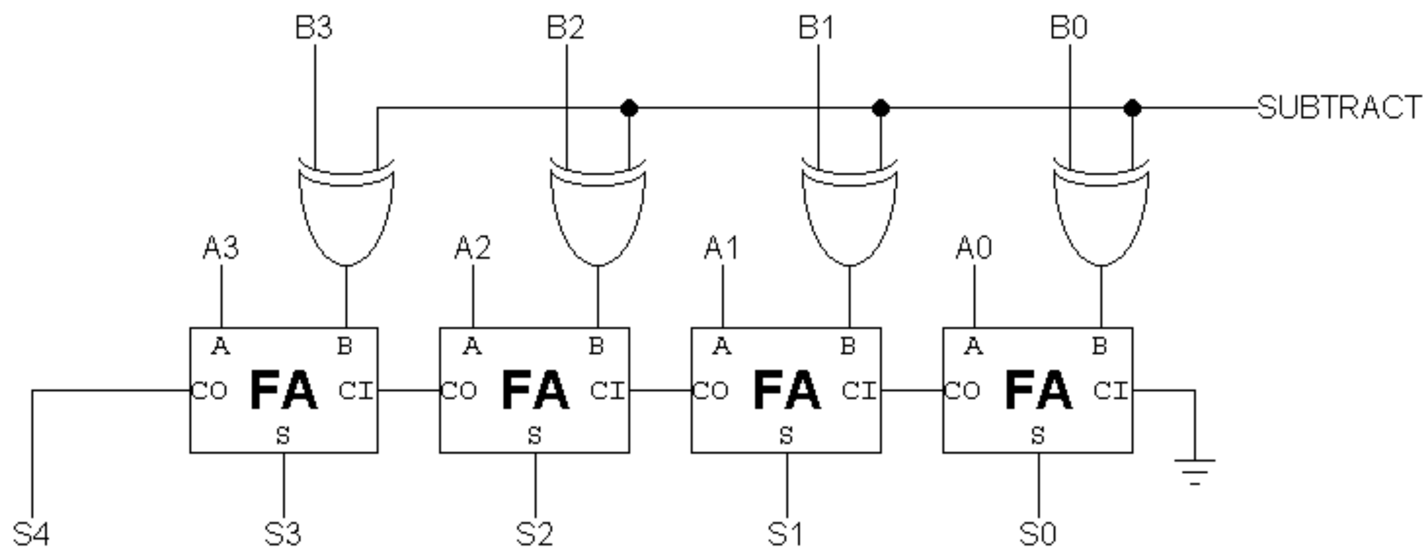
Subtraction: A-B = A + (-B)

Using 2's complement representation: –B = ~B + 1

~ = bit-wise complement



So let's build an arithmetic unit that does both addition and subtraction. Operation selected by control input:

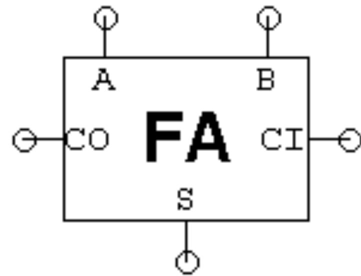


When SUBTRACT is 1:

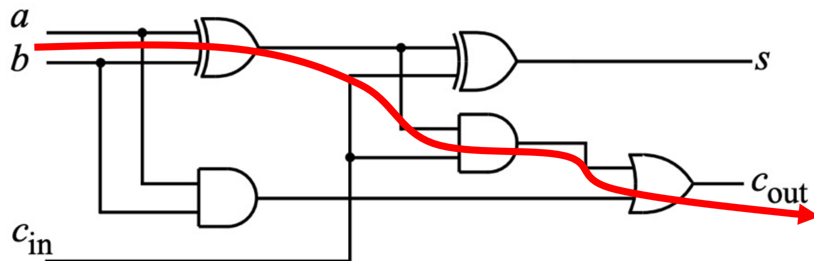
- *Invert each bit*
- *Start with a Carry of 1 (same as adding 1)*

“Full Adder” building block

The “half adder” circuit has only the A and B inputs



A	B	CI	S	CO
0	0	0	0	0
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
1	1	0	0	1
1	1	1	1	1



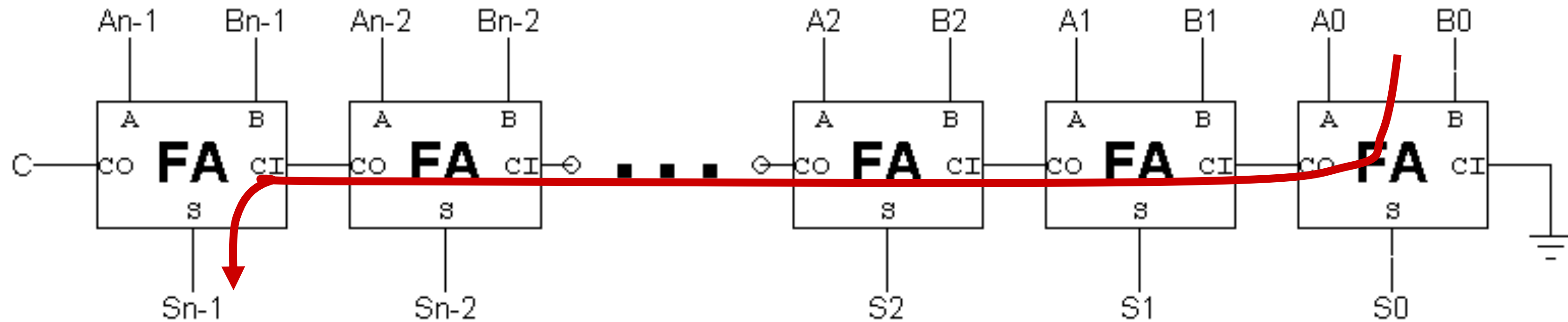
t_{pd} dictated by carry path!

(pset 01)

Can also rewrite the carry path as: $c_{out} = (a \,\&\, c_{in})|(b \,\&\, c_{in})\ |(a \,\&\, b)$

Speed: t_{PD} of Ripple-carry Adder

$C_O = AB + AC_I + BC_I$



Worst-case path: carry propagation from LSB to MSB, e.g., when adding 11…111 to 00…001.

$$t_{PD} = \underbrace{(N-1)*(t_{PD,OR} + t_{PD,AND})}_{\text{CI to CO}} + \underbrace{t_{PD,XOR}}_{CI_{N-1} \text{ to } S_{N-1}} \approx \Theta(N)$$

$$t_{adder} = (N-1)t_{carry} + t_{sum}$$

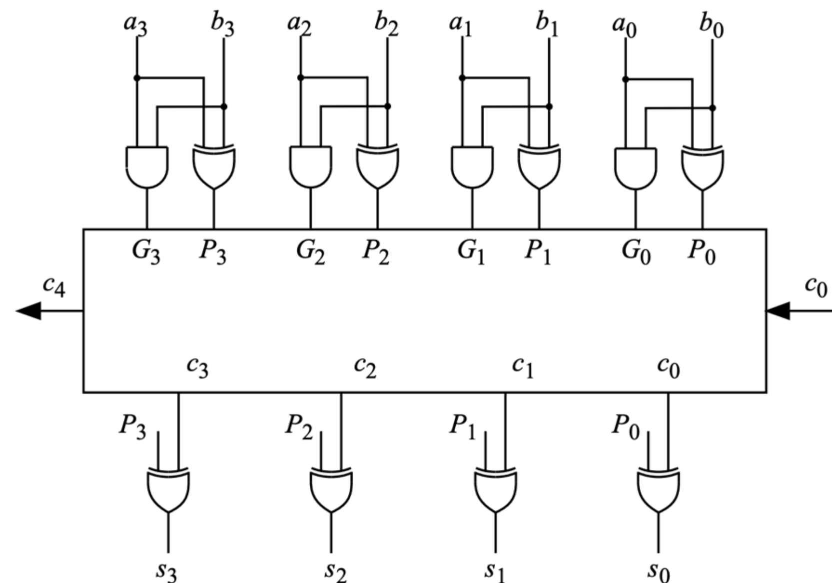
Θ(N) is read "order N" : means that the latency of our adder grows at worst in proportion to the number of bits in the operands.

The Carry Path Becomes Limiting

- Solution is the Carry-Look-ahead Adder:

$$G_i = a_i.b_i$$
$$P_i = a_i \oplus b_i$$

$$c_1 = G_0 + P_0.c_0$$
$$c_2 = G_1 + P_1.G_0 + P_1.P_0.c_0$$
$$c_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.c_0$$
$$c_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.c_0$$



Can do some factoring/redesign and cut-down on t_{pd} of the carry path

https://www.ece.uvic.ca/~fayez/courses/ceng465/lab_465/project1/adders.pdf

Logic Slices Can Add/Subtract

- Can synthesize the addition of two 4 bit numbers with fast carry

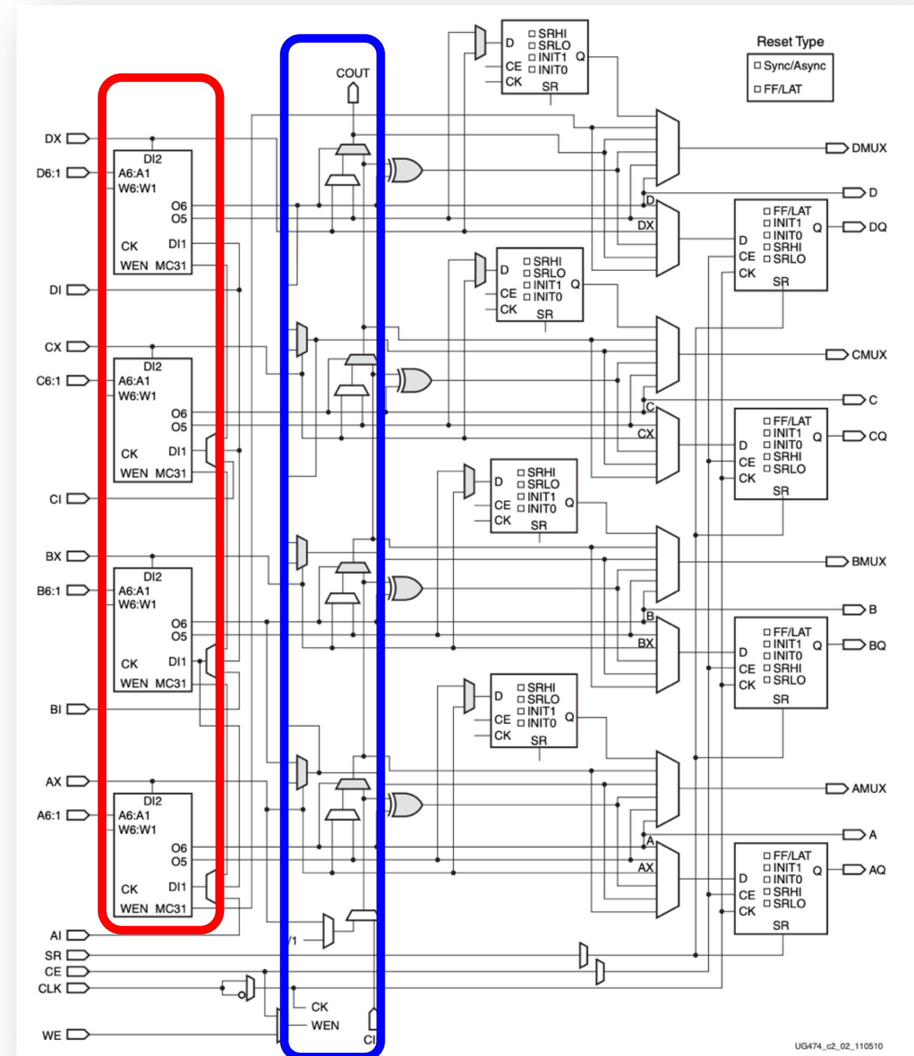
$$\begin{array}{r} A_3A_2A_1A_0 \\ +B_3B_2B_1B_0 \\ \hline \end{array}$$

A_3 B_3

A_2 B_2

A_1 B_1

A_0 B_0



Fast Carry-Chain

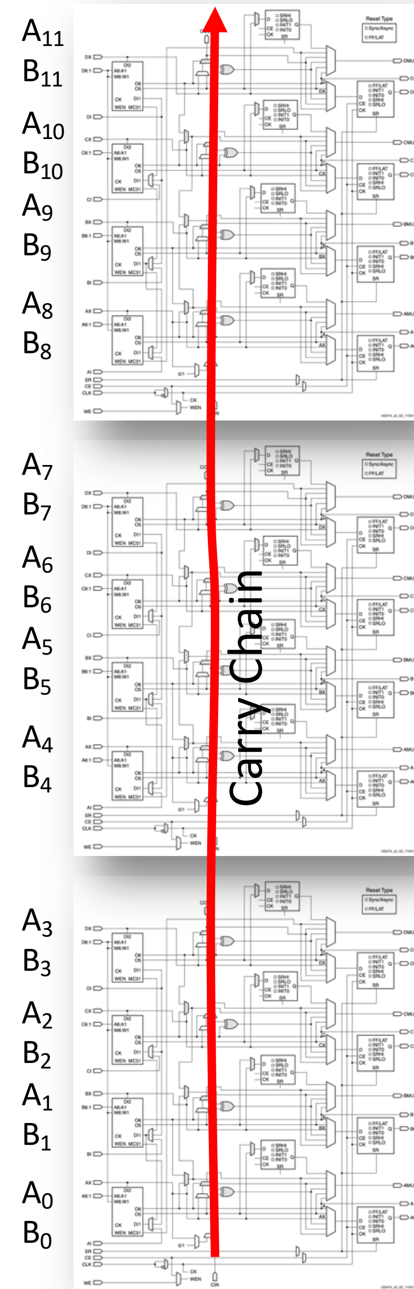
Series 7 Logic Slice

Add/Subtract on the FPGA

- + and – can be done combinationally very quickly:
  - 32 bit add can be done in a clock cycle (<10 ns)
  - Several smaller adds (A+B+C+D) can be done in clock cycle as well (10 ns)
- CLBs (the generic function generators, of which we have 15,000) are capable of being chained together to allow large adds.

Slices can stack to give more bits

$$
\begin{array}{r}
A_{11}A_{10}A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0 \\
+B_{11}B_{10}B_9B_8B_7B_6B_5B_4B_3B_2B_1B_0 \\
\hline
\end{array}
$$



+ or - in Verilog

- Generally + or – on its own will get synthesized using logic slices unless specified
- Very large additions or subtractions may start to take too long!

Multiplication on the FPGA

- Multiplication can be done on the FPGA on 2’s complement numbers
- Takes more time:
  - Depending on size of operands may/may not be doable in one clock cycle
- Where possible try to get away with bit shifts and adds.

Multiplications with shifts

- <<1 is multiply by 2
- >>1 is divide by 2
- Can do a lot with this if get creative

```

logic [7:0] x;
logic [7:0] y; //want this to be seven times X
assign y = (x<<2) + (x<<1) + x;

```

Generic Digital Multiplication

$$
\begin{array}{ccccccccl}
 & & & a_3 & a_2 & a_1 & a_0 & \leftarrow & \textit{Multiplicand} \\
 & & & b_3 & b_2 & b_1 & b_0 & \leftarrow & \textit{Multiplier} \\
\hline
 & & X & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & \rbrace & \\
 & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & & & \textit{Partial} \\
 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & & \textit{products} \\
a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & & & \\
\hline
 & & & \ldots & & a_1b_0+a_0b_1 & a_0b_0 & \leftarrow & \textit{Product}
\end{array}
$$

In base 2 multiplication these are all very simple calculations done with XOR

*Some really cool factoring can be done to make the overall propagation delay of a multiplier relatively short, though there's a lot of logic in it**

*Lecture on Multiplier architectures: https://inst.eecs.berkeley.edu/~eecs151/sp18/files/Lecture21.pdf

DSP Blocks

- Add-then-multiply is a common operation chain in many things, particularly Digital Signal Processing
- FPGA has dedicated hardware multiplier modules called DSP48 blocks on it
  - 240 of them on Nexys FPGA
  - Capable of single-cycle multiplies
- Can get inferred from using * in your Verilog that isn't a power of 2:
  - x*y, for example, will likely will result in DSP getting used
  - May take a full clock cycle so would need to budget tiing accordingly
- Can infer multiple for larger multiplies

DSP48 Slice (High Level)

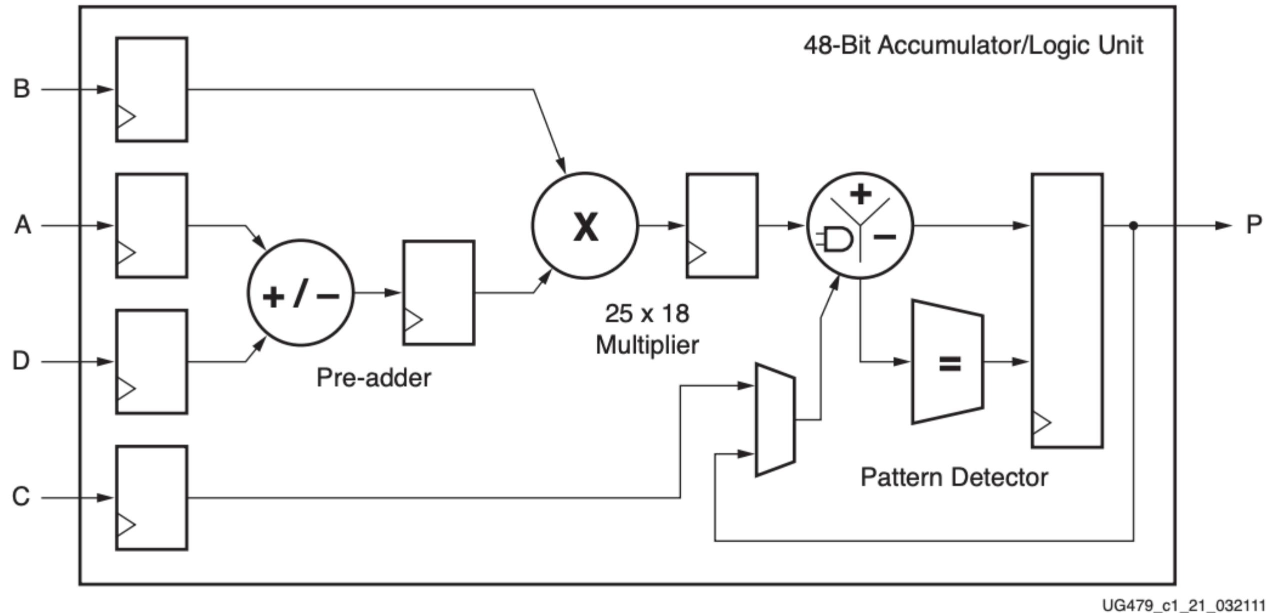


Figure 1-1: **Basic DSP48E1 Slice Functionality**

Much of the benefit/speed of this module comes from the hardwired internal routing, keeping it very fast. This device is not as generalized as a LUT/logic cell. It can only do a subset of math operations

https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

Division

- The outlier in the + - * / set…
- Division is a significantly harder math operation to do compared to multiplication
- Where possible try to avoid
- Try to divide by powers of 2 (use right shift)!

- If you can’t avoid we must do it. (lab04)

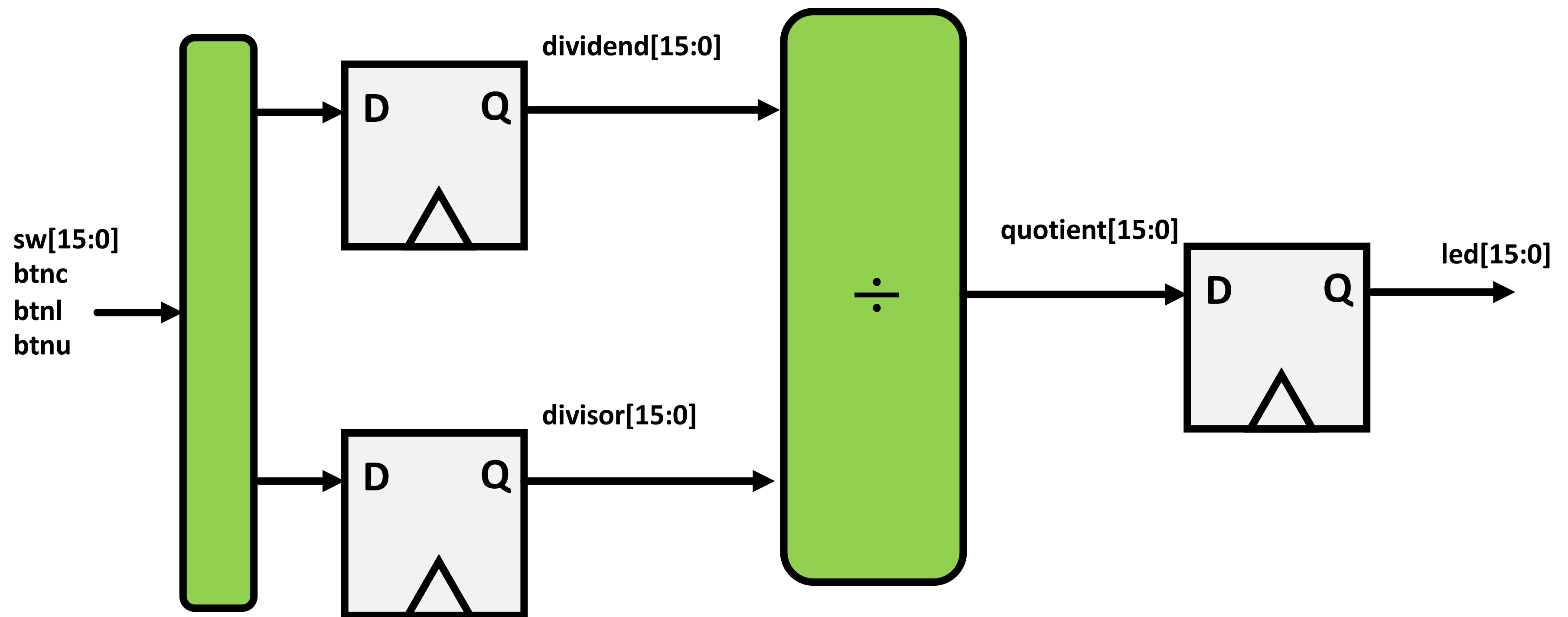
One “Bad” Attempt at Division

- In last lecture looked at *what* this actually builds
- We can ask Vivado to synthesize division logic for us, and it actually will do it.
- This code constrains the act of division to having to exist between two clock edges.:

```
module top_level(
      input wire clk_100mhz, //clock @ 100 mhz
      input wire [15:0] sw, //switches
      input wire btnc, //btnc (used for reset)
      input wire btnu, //btnc (used for reset)
      input wire btnl, //btnc (used for reset)
      output logic [15:0] led //just here for the funs
      );
  logic old_btnl;
  logic old_btnu;
  logic old_btnc;
  logic [15:0] quotient;
  logic [15:0] dividend;
  logic [15:0] divisor;
  assign led = quotient;
  always_ff @(posedge clk_100mhz)begin
    old_btnl <= btnl;
    old_btnu <= btnu;
    old_btnc <= btnc;
  end

  always_ff @(posedge clk_100mhz)begin
    if (btnu & ~old_btnu)begin
      quotient<= dividend/divisor; //divide
    end
    if (btnc & ~old_btnc)begin
      dividend <= sw; //divide //load dividend
    end
    if (btnl & ~old_btnl)begin
      divisor <= sw; //divide //load divisor
    end
  end
endmodule
```

Circuit Built:



Build the Stupid Divider

Violates timing!

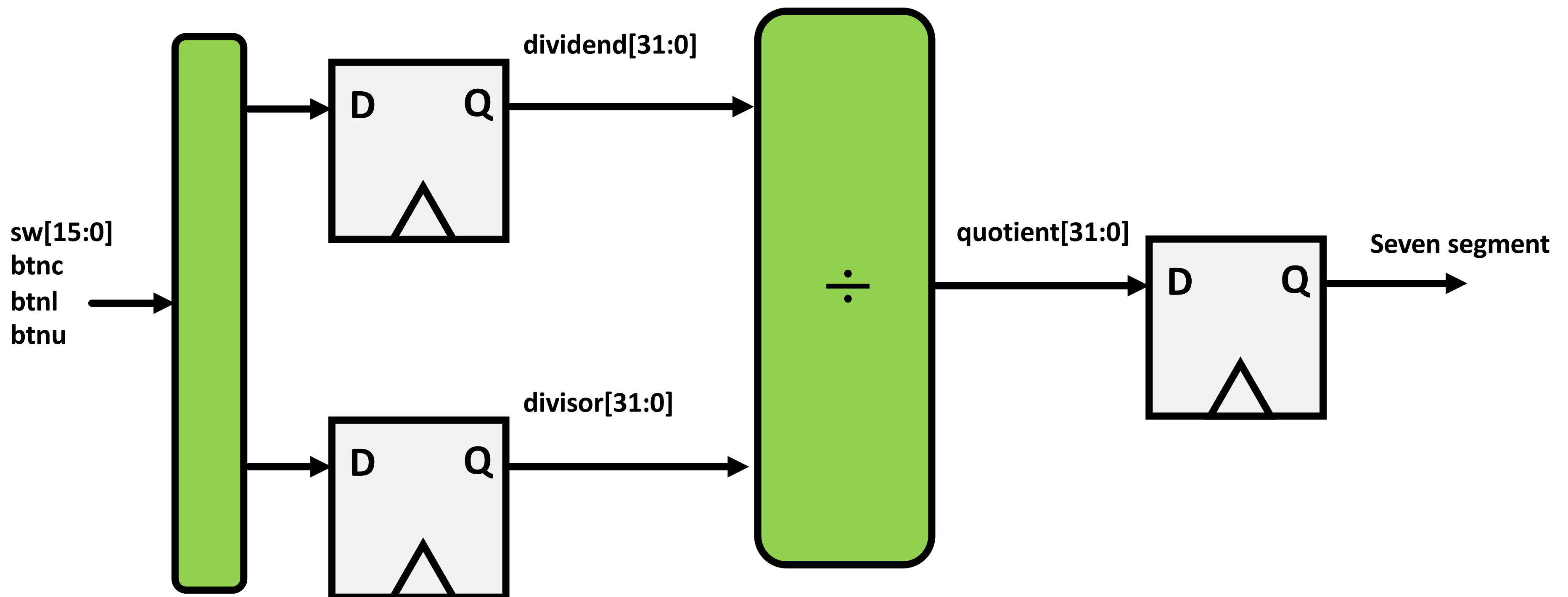
```
Phase 10 Post Router Timing
INFO: [Route 35-57] Estimated Timing Summary | WNS=-22.720| TNS=-140.635| WHS=0.344  | THS=0.000  |

WARNING: [Route 35-328] Router estimated timing not met.
Resolution: For a complete and accurate timing signoff, report_timing_summary must be run after
route_design. Alternatively, route_design can be run with the -timing_summary option to enable a
complete timing signoff at the end of route_design.
```

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	100	0	0	15850	0.63
SLICEL	89	0			
SLICEM	11	0			
LUT as Logic	274	0	0	63400	0.43
using O5 output only	0				
using O6 output only	274				
using O5 and O6	0				
LUT as Memory	0	0	0	19000	0.00
LUT as Distributed RAM	0	0			
LUT as Shift Register	0	0			
Slice Registers	55	0	0	126800	0.04
Register driven from within the Slice	16				
Register driven from outside the Slice	39				
LUT in front of the register is unused	26				
LUT in front of the register is used	13				
Unique Control Sets	4		0	15850	0.03

Now Do same Thing With 32 bits:

```
if (btnu & ~old_btnu) begin
  quotient <= dividend/divisor;
end
```



**See lecture code for full implementation and build. (divider0)*

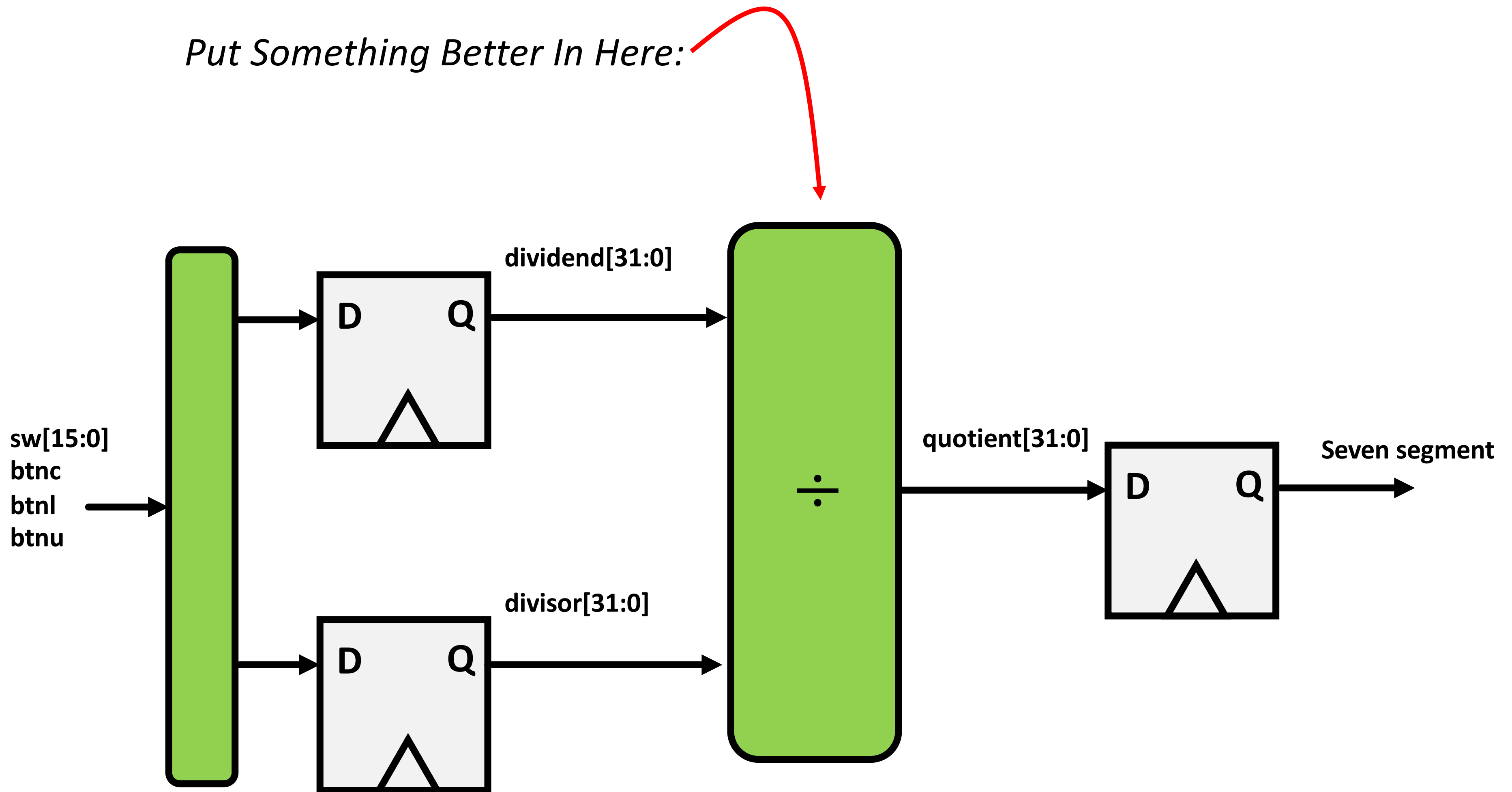
Build the Stupider Divider

```
Phase 10 Post Router Timing
INFO: [Route 35-57] Estimated Timing Summary | WNS=-68.215| TNS=-950.764| WHS=0.193  | THS=0.000  |

WARNING: [Route 35-328] Router estimated timing not met.
Resolution: For a complete and accurate timing signoff, report_timing_summary must be run after
route_design. Alternatively, route_design can be run with the -timing_summary option to enable a complete
timing signoff at the end of route_design.
Phase 10 Post Router Timing | Checksum: 29c6a1ed3
```

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	303	0	0	15850	1.91
SLICEL	209	0			
SLICEM	94	0			
LUT as Logic	946	0	0	63400	1.49
using O5 output only	0				
using O6 output only	924				
using O5 and O6	22				
LUT as Memory	0	0	0	19000	0.00
LUT as Distributed RAM	0	0			
LUT as Shift Register	0	0			
Slice Registers	132	0	0	126800	0.10
Register driven from within the Slice	67				
Register driven from outside the Slice	65				
LUT in front of the register is unused	30				
LUT in front of the register is used	35				
Unique Control Sets	7		0	15850	0.04

A Better Divider?



**See lecture code for full implementation and build. (divider0)*

A Divider (#1)

```python
def divider (dividend, divisor):
  count = 0
  if dividend <=0:
    return (0,divisor)
  if divisor==0:
    return -1
  while dividend>=divisor:
    dividend -= divisor
    count += 1
  return (count, dividend)
```

- This is a Verilog FSM example of the division algorithm above which will run an unknown number of times given a set of inputs
- This is how the functionality of a while loop could be developed in your modules
- Will not handle negative, or 0 or other things
- Giver you the 32 bit one in lab04A

```verilog
module divider (input wire clk_in,
                input wire rst_in,
                input wire[15:0] dividend_in,
                input wire[15:0] divisor_in,
                input wire data_valid_in,
                output logic[15:0] quotient_out,
                output logic[15:0] remainder_out,
                output logic data_valid_out,
                output logic error_out,
                output logic busy_out);
  localparam RESTING = 0;
  localparam DIVIDING = 1;
  logic [15:0] quotient, dividend, divisor;
  logic state;
  always_ff @(posedge clk_in)begin
    if (rst_in)begin
      quotient <= 16'b0;
      dividend <= 16'b0;
      divisor <= 16'b0;
      remainder_out <= 16'b0;
      busy_out <= 1'b0;
      error_out <= 1'b0;
      state <= RESTING;
      data_valid_out <= 1'b0;
    end else begin
      case (state)
        RESTING: begin
          if (data_valid_in)begin
            state <= DIVIDING;
            quotient <= 16'b0;
            dividend <= dividend_in;
            divisor <= divisor_in;
            busy_out <= 1'b1;
            error_out <= 1'b0;
          end
          data_valid_out <= 1'b0;
        end
        DIVIDING: begin
          if (dividend<=0)begin
            state <= RESTING; //similar to return statement
            remainder_out <= divisor;
            quotient_out <= 0;
            busy_out <= 1'b0; //tell outside world i'm done
            error_out <= 1'b0;
            data_valid_out <= 1'b1; //good stuff!
          end else if (divisor==0)begin
            state <= RESTING;
            remainder_out <= 0;
            quotient_out <= 0;
            busy_out <= 1'b0; //tell outside world i'm done
            error_out <= 1'b1; //ERROR
            data_valid_out <= 1'b1; //valid ERROR
          end else if (dividend < divisor) begin
            state <= RESTING;
            remainder_out <= dividend;
            quotient_out <= quotient;
            busy_out <= 1'b0;
            error_out <= 1'b0;
            data_valid_out <= 1'b1; //good stuff!
          end else begin
            //state staying in.
            quotient <= quotient + 1'b1;
            dividend <= dividend-divisor;
          end
        end
      endcase
    end
  end
endmodule
```

Build divider1 og

```
Phase 10 Post Router Timing
INFO: [Route 35-57] Estimated Timing Summary | WNS=4.592  | TNS=0.000  | WHS=0.125  | THS=0.000  |

INFO: [Route 35-327] The final timing numbers are based on the router estimated timing analysis. For
a complete and accurate timing signoff, please run report_timing_summary.
Phase 10 Post Router Timing | Checksum: 12383e828
```

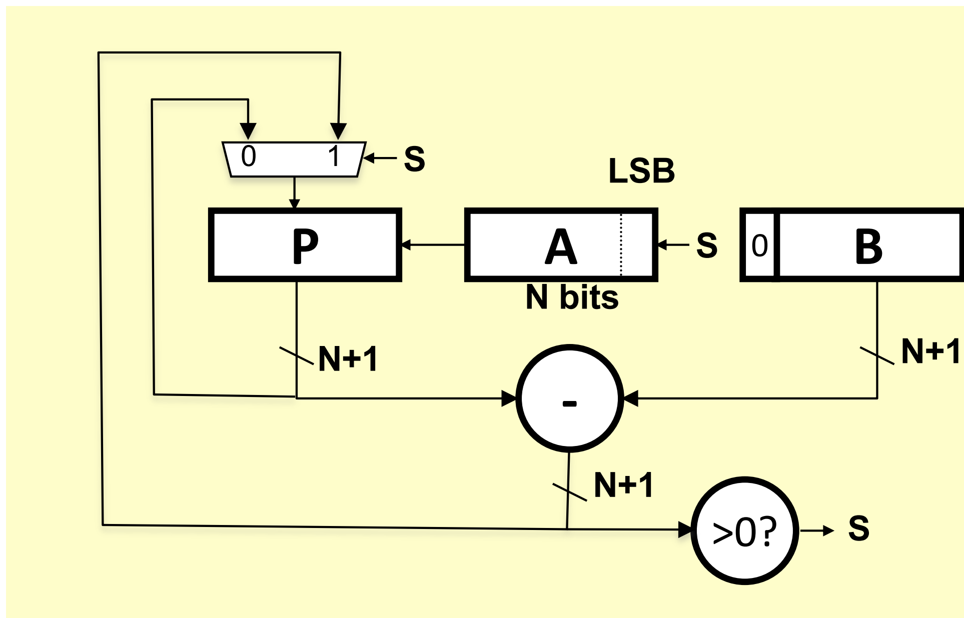
Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	65	0	0	15850	0.41
SLICEL	45	0			
SLICEM	20	0			
LUT as Logic	139	0	0	63400	0.22
using O5 output only	0				
using O6 output only	106				
using O5 and O6	33				
LUT as Memory	0	0	0	19000	0.00
LUT as Distributed RAM	0	0			
LUT as Shift Register	0	0			
Slice Registers	207	0	0	126800	0.16
Register driven from within the Slice	100				
Register driven from outside the Slice	107				
LUT in front of the register is unused	54				
LUT in front of the register is used	53				
Unique Control Sets	9		0	15850	0.06

A Better Algorithm?

- This can't be how we actually do division in real-life right?
- No there are actual algorithms that are base-2 friendly that we can use instead.
- Further more, there are algorithms that operate in a fixed number of cycles.

Divider (Fixed # of Steps)

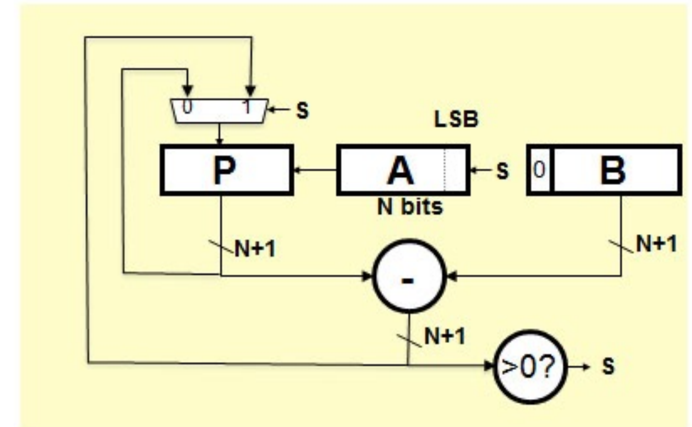
Assume the Dividend (**A**) and the divisor (**B**) have **N** bits. If we only want to invest in a single N-bit adder, we can build a sequential circuit that processes a single subtraction at a time and ***then cycle the circuit N times***. This circuit works on unsigned operands; for signed operands one can remember the signs, make operands positive, then correct sign of result.




```
Init: P←0, load A and B
Repeat N times {
    shift P,A left one bit
    temp = P-B
    if (temp >= 0){
     P←temp
     A_LSB←1
    }else{
         A_LSB←0
    }
}
Done: Q in A, R in P
```

Divider

P	A	P-B	7/3 0111/11 B=0011
0000	0111		Initial value
0000	1110		Shift
0000		-3	Subtract
0000	1110		Restore, set $A_{lsb} = 0$
0001	1100		Shift
0001		-2	Subtract
0001	1100		Restore, set $A_{lsb} = 0$
0011	1000		Shift
0011		0	Subtract
0000	1001		Subtact, set $A_{lsb} = 1$
0001	0010		Shift
0001		-2	Subtract
0001	0010		Restore, set $A_{lsb} = 0$
R	Q		




```
Init: P←0, load A and B
Repeat N times {
    shift P,A left one bit
    temp = P-B
    if (temp >= 0){
     P←temp
     A_LSB←1
    }else{
         A_LSB←0
    }
}
Done: Q in A, R in P
```

divider2

- This is an FSM implementation of the “smarter” algorithm just shown:
- Latency:
  - 32 clock cycles
- Throughput:
  - 1/32 clock cycles
- This is “blocking” implementation, meaning that when it is running it cannot accept new inputs.
- Even with some sort of FIFO, this will never process more than 1 division per 32 cycles.
- Simulate to verify it works.

```
module divider2 #(parameter WIDTH = 32) (input wire clk_in,
                    input wire rst_in,
                    input wire[WIDTH-1:0] dividend_in,
                    input wire[WIDTH-1:0] divisor_in,
                    input wire data_valid_in,
                    output logic[WIDTH-1:0] quotient_out,
                    output logic[WIDTH-1:0] remainder_out,
                    output logic data_valid_out,
                    output logic error_out,
                    output logic busy_out);
  logic [WIDTH-1:0] quotient, dividend;
  logic [WIDTH-1:0] divisor;
  logic [5:0] count;
  logic [31:0] p;
  enum {RESTING, DIVIDING} state;
  always_ff @(posedge clk_in)begin
    if (rst_in)begin
      quotient <= 0;
      dividend <= 0;
      divisor <= 0;
      remainder_out <= 0;
      busy_out <= 1'b0;
      error_out <= 1'b0;
      state <= RESTING;
      data_valid_out <= 1'b0;
      count <= 0;
    end else begin
      case (state)
        RESTING: begin
          if (data_valid_in)begin
            state <= DIVIDING;
            quotient <= 0;
            dividend <= dividend_in;
            divisor <= divisor_in;
            busy_out <= 1'b1;
            error_out <= 1'b0;
            count <= 31;//load all up
            p <= 0;
          end
          data_valid_out <= 1'b0;
        end
        DIVIDING: begin
          if (count==0)begin
            state <= RESTING;
            if ({p[30:0],dividend[31]}>=divisor[31:0])begin
              remainder_out <= {p[30:0],dividend[31]} - divisor[31:0];
              quotient_out <= {dividend[30:0],1'b1};
            end else begin
              remainder_out <= {p[30:0],dividend[31]};
              quotient_out <= {dividend[30:0],1'b0};
            end
            busy_out <= 1'b0; //tell outside world i'm done
            error_out <= 1'b0;
            data_valid_out <= 1'b1; //good stuff!
          end else begin
            if ({p[30:0],dividend[31]}>=divisor[31:0])begin
              p <= {p[30:0],dividend[31]} - divisor[31:0];
              dividend <= {dividend[30:0],1'b1};
            end else begin
              p <= {p[30:0],dividend[31]};
              dividend <= {dividend[30:0],1'b0};
            end
            count <= count-1;
          end
        end
      endcase
    end
  end
endmodule
```

Build divider2:

```
Phase 10 Post Router Timing
INFO: [Route 35-57] Estimated Timing Summary | WNS=5.270  | TNS=0.000  | WHS=0.149  | THS=0.000  |

INFO: [Route 35-327] The final timing numbers are based on the router estimated timing analysis.
For a complete and accurate timing signoff, please run report_timing_summary.
Phase 10 Post Router Timing | Checksum: 114e803ee
```

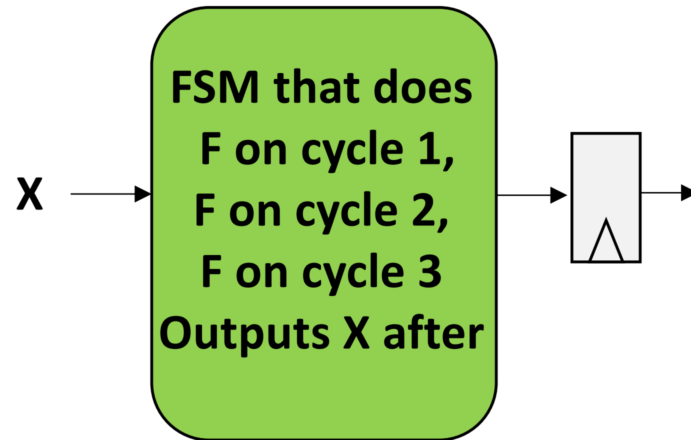
Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	53	0	0	15850	0.33
SLICEL	41	0			
SLICEM	12	0			
LUT as Logic	130	0	0	63400	0.21
using O5 output only	0				
using O6 output only	72				
using O5 and O6	58				
LUT as Memory	0	0	0	19000	0.00
LUT as Distributed RAM	0	0			
LUT as Shift Register	0	0			
Slice Registers	212	0	0	126800	0.17
Register driven from within the Slice	110				
Register driven from outside the Slice	102				
LUT in front of the register is unused	45				
LUT in front of the register is used	57				
Unique Control Sets	10		0	15850	0.06

Can We Make it Better?

- We have an algorithm that takes a fixed amount of cycles per divide (32 in our case)
- Because of this we know exactly how many calculations we need to do.
- This allows us to set up a fully-pipelined

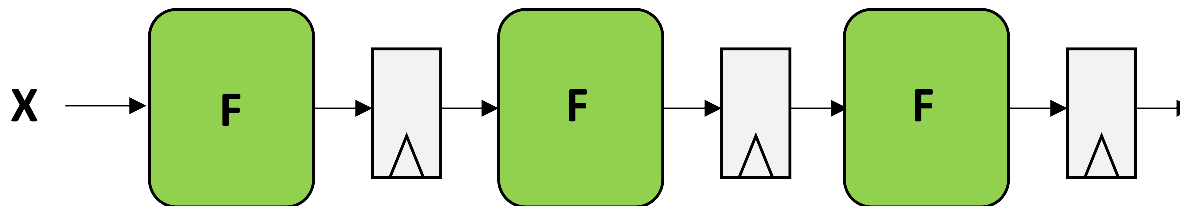
What?

- Currently we're doing this:



Latency: $3*T_{clk}$
Throughput: 1/ ($3*T_{clk}$)
MIGHT use less logic,flops

- What if we instead did this ("unwrap the loop"):



Latency: $3*T_{clk}$
Throughput: 1/ T_{clk}
Uses more logic,flops

divider3

- Fully pipelined 32 step division
- Each step is carried out and results placed in registers which are used by next step in pipeline
- Latency still 32 cycles
- Throughput is now 1/1 cycle
  - Assembly line! Stage 0 can always have something to do
- Simulate it (it works)
- Now build…

```
module divider3 #(parameter WIDTH = 32) (input wire clk_in,
                                input wire rst_in,
                                input wire[WIDTH-1:0] dividend_in,
                                input wire[WIDTH-1:0] divisor_in,
                                input wire data_valid_in,
                                output logic[WIDTH-1:0] quotient_out,
                                output logic[WIDTH-1:0] remainder_out,
                                output logic data_valid_out,
                                output logic error_out,
                                output logic busy_out);

  logic [31:0] p[31:0]; //32 stages
  logic [31:0] dividend [31:0];
  logic [31:0] divisor [31:0];
  logic data_valid [31:0];

  assign data_valid_out = data_valid[31];
  assign quotient_out = dividend[31];
  assign remainder_out = p[31];

  always_ff @(posedge clk_in)begin
    data_valid[0] <= data_valid_in;
    if (data_valid_in)begin
      divisor[0] <= divisor_in;
      if ({31'b0,dividend_in[31]}>=divisor_in[31:0])begin
        p[0] <= {31'b0,dividend_in[31]} - divisor_in[31:0];
        dividend[0] <= {dividend_in[30:0],1'b1};
      end else begin
        p[0] <= {31'b0,dividend_in[31]};
        dividend[0] <= {dividend_in[30:0],1'b0};
      end
    end
    for (int i=1; i<32; i=i+1)begin
      data_valid[i] <= data_valid[i-1];
      if ({p[i-1][30:0],dividend[i-1][31]}>=divisor[i-1][31:0])begin
        p[i] <= {p[i-1][30:0],dividend[i-1][31]} - divisor[i-1][31:0];
        dividend[i] <= {dividend[i-1][30:0],1'b1};
      end else begin
        p[i] <= {p[i-1][30:0],dividend[i-1][31]};
        dividend[i] <= {dividend[i-1][30:0],1'b0};
      end
      divisor[i] <= divisor[i-1];
    end
  end
endmodule
```

Build divider3

```
Phase 10 Post Router Timing
INFO: [Route 35-57] Estimated Timing Summary | WNS=3.208  | TNS=0.000  | WHS=0.039  | THS=0.000  |

INFO: [Route 35-327] The final timing numbers are based on the router estimated timing analysis. For
a complete and accurate timing signoff, please run report_timing_summary.
Phase 10 Post Router Timing | Checksum: af8d8e91
```

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	461	0	0	15850	2.91
SLICEL	292	0			
SLICEM	169	0			
LUT as Logic	1509	0	0	63400	2.38
using O5 output only	0				
using O6 output only	1004				
using O5 and O6	505				
LUT as Memory	58	0	0	19000	0.31
LUT as Distributed RAM	0	0			
LUT as Shift Register	58	0			
using O5 output only	13				
using O6 output only	45				
using O5 and O6	0				
Slice Registers	1647	0	0	126800	1.30
Register driven from within the Slice	1084				
Register driven from outside the Slice	563				
LUT in front of the register is unused	77				
LUT in front of the register is used	486				
Unique Control Sets	7		0	15850	0.04

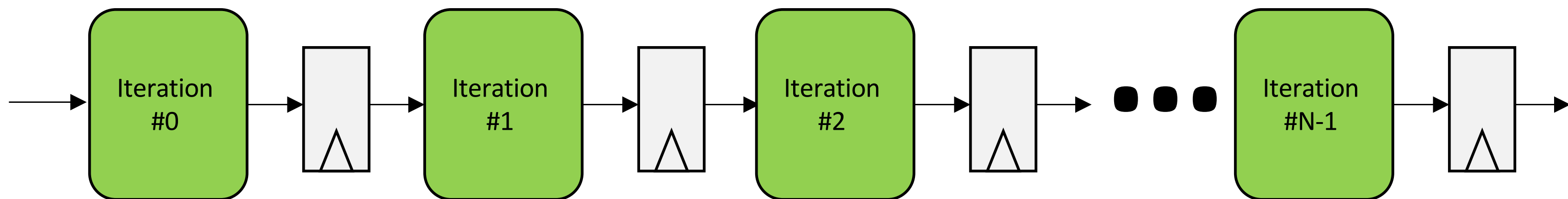
Resource usage went way up

Do it Better?

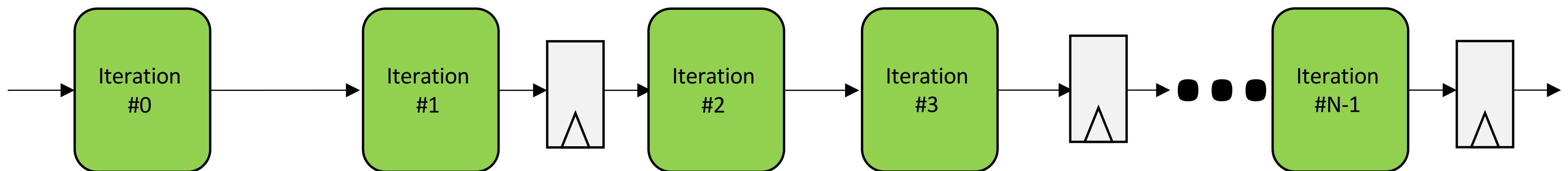
- Can I get my result faster?

```
Init: P←0, load A and B
Repeat N times {
    shift P,A left one bit
    temp = P-B
    if (temp >= 0){
      P←temp
      A_LSB←1
    }else{
        A_LSB←0
    }
}
Done: Q in A, R in P
```

$L = N*t_{clk}$, $T = 1/t_{clk}$



$L = 0.5*N*t_{clk}$, $T = 1/t_{clk}$ And maybe use fewer registers!!!:



divider4

- Improved Pipeline
- Shove two stages of our algorithm between each register pair.
- Therefore this should allow the same throughput of division but a halving of latency!
- In theory anyways.
- Simulate it (it works)
- Now build…

```
module divider4 #(parameter WIDTH = 32) (input wire clk_in,
                        input wire rst_in,
                        input wire[WIDTH-1:0] dividend_in,
                        input wire[WIDTH-1:0] divisor_in,
                        input wire data_valid_in,
                        output logic[WIDTH-1:0] quotient_out,
                        output logic[WIDTH-1:0] remainder_out,
                        output logic data_valid_out,
                        output logic error_out,
                        output logic busy_out);

  logic [31:0] p[31:0]; //32 stages
  logic [31:0] dividend [31:0];
  logic [31:0] divisor [31:0];
  logic data_valid [31:0];

  assign data_valid_out = data_valid[31];
  assign quotient_out = dividend[31];
  assign remainder_out = p[31];

  always @(*) begin
    data_valid[0] = data_valid_in;
    divisor[0] = divisor_in;
    if (data_valid_in)begin
      if ({31'b0,dividend_in[31]}>=divisor_in[31:0])begin
        p[0] = {31'b0,dividend_in[31]} - divisor_in[31:0];
        dividend[0] = {dividend_in[30:0],1'b1};
      end else begin
        p[0] = {31'b0,dividend_in[31]};
        dividend[0] = {dividend_in[30:0],1'b0};
      end
    end
    for (int i=2; i<32; i=i+2)begin
      data_valid[i] = data_valid[i-1];
      if ({p[i-1][30:0],dividend[i-1][31]}>=divisor[i-1][31:0])begin
        p[i] = {p[i-1][30:0],dividend[i-1][31]} - divisor[i-1][31:0];
        dividend[i] = {dividend[i-1][30:0],1'b1};
      end else begin
        p[i] = {p[i-1][30:0],dividend[i-1][31]};
        dividend[i] = {dividend[i-1][30:0],1'b0};
      end
      divisor[i] = divisor[i-1];
    end
  end

  always_ff @(posedge clk_in)begin
    for (int i=1; i<32; i=i+2)begin
      data_valid[i] <= data_valid[i-1];
      if ({p[i-1][30:0],dividend[i-1][31]}>=divisor[i-1][31:0])begin
        p[i] <= {p[i-1][30:0],dividend[i-1][31]} - divisor[i-1][31:0];
        dividend[i] <= {dividend[i-1][30:0],1'b1};
      end else begin
        p[i] <= {p[i-1][30:0],dividend[i-1][31]};
        dividend[i] <= {dividend[i-1][30:0],1'b0};
      end
      divisor[i] <= divisor[i-1];
    end
  end
endmodule
```

Build divider4

Over timing by 154 picoseconds

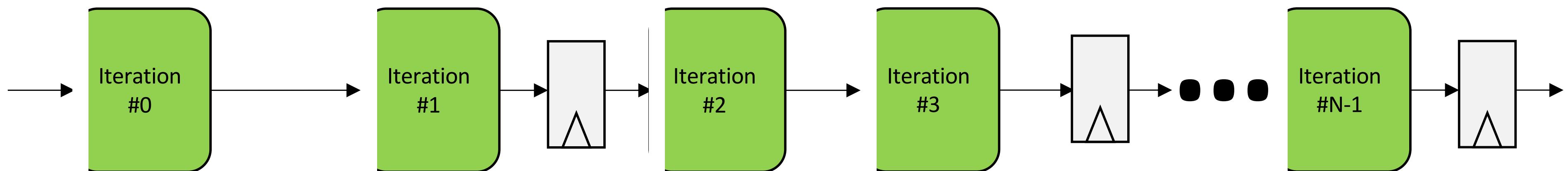
```
Phase 10 Post Router Timing
INFO: [Route 35-57] Estimated Timing Summary | WNS=-0.154 | TNS=-0.494 | WHS=0.083  | THS=0.000  |

WARNING: [Route 35-328] Router estimated timing not met.
Resolution: For a complete and accurate timing signoff, report_timing_summary must be run after
route_design. Alternatively, route_design can be run with the -timing_summary option to enable a
complete timing signoff at the end of route_design.
Phase 10 Post Router Timing | Checksum: 1479b4e85
```

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	438	0	0	15850	2.76
SLICEL	259	0			
SLICEM	179	0			
LUT as Logic	1509	0	0	63400	2.38
using O5 output only	0				
using O6 output only	1004				
using O5 and O6	505				
LUT as Memory	54	0	0	19000	0.28
LUT as Distributed RAM	0	0			
LUT as Shift Register	54	0			
using O5 output only	0				
using O6 output only	54				
using O5 and O6	0				
Slice Registers	934	0	0	126800	0.74
Register driven from within the Slice	603				
Register driven from outside the Slice	331				
LUT in front of the register is unused	33				
LUT in front of the register is used	298				
Unique Control Sets	7		0	15850	0.04

Look for Optimizations

- Need to shave a little logic off our stages to fit it
- Only looking for 157 picoseconds
- Not too much
- …



Look for Optimizations

- We only feed the start of the data pipeline if data valid in
- This saves energy (since circuit isn't dividing garbage every clock cycle).
- Makes our logic a bit thicker
- Let's just ***always*** process the inputs

CUT

```
always @(*) begin //always_comb (just keeps iverilog happy)
  data_valid[0] = data_valid_in;
  divisor[0] = divisor_in;
  if (data_valid_in)begin
    if({31'b0,dividend_in[31]}>=divisor_in[31:0])begin
      p[0] = {31'b0,dividend_in[31]} - divisor_in[31:0];
      dividend[0] = {dividend_in[30:0],1'b1};
    end else begin
      p[0] = {31'b0,dividend_in[31]};
      dividend[0] = {dividend_in[30:0],1'b0};
    end
  end
```

Look for Optimizations

- We only feed the start of the data pipeline if data valid in
- This saves energy (since circuit isn't dividing garbage every clock cycle).
- Makes our logic a bit thicker
- Let's just ***always*** process the inputs

```
always @(*) begin //always_comb (just keeps iverilog happy)
  data_valid[0] = data_valid_in;
  divisor[0] = divisor_in;
  if({31'b0,dividend_in[31]}>=divisor_in[31:0])begin
    p[0] = {31'b0,dividend_in[31]} - divisor_in[31:0];
    dividend[0] = {dividend_in[30:0],1'b1};
  end else begin
    p[0] = {31'b0,dividend_in[31]};
    dividend[0] = {dividend_in[30:0],1'b0};
  end
```

General Guideline

- The deeper, more nested your if/else logic is, the more complicated and thicker your intermediate logic will be.
- There's no clear quantitative measure for how much an "if/else" costs since it is very context dependent.
- Good place to start cutting if needed.

divider5

- Fully pipelined variant of our 32-step division algorithm.
- Two steps per clock cycle
- Always processing data
- Throughput: 1/cycle
- Latency: 16 cycles
- Simulate to verify it works….
- Then build…

```
module divider5 #(parameter WIDTH = 32) (input wire clk_in,
                   input wire rst_in,
                   input wire[WIDTH-1:0] dividend_in,
                   input wire[WIDTH-1:0] divisor_in,
                   input wire data_valid_in,
                   output logic[WIDTH-1:0] quotient_out,
                   output logic[WIDTH-1:0] remainder_out,
                   output logic data_valid_out,
                   output logic error_out,
                   output logic busy_out);

  logic [31:0] p[31:0]; //32 stages
  logic [31:0] dividend [31:0];
  logic [31:0] divisor [31:0];
  logic data_valid [31:0];

  assign data_valid_out = data_valid[31];
  assign quotient_out = dividend[31];
  assign remainder_out = p[31];

  always @(*) begin
    data_valid[0] = data_valid_in;
    divisor[0] = divisor_in;
    if ({31'b0,dividend_in[31]}>=divisor_in[31:0])begin
      p[0] = {31'b0,dividend_in[31]} - divisor_in[31:0];
      dividend[0] = {dividend_in[30:0],1'b1};
    end else begin
      p[0] = {31'b0,dividend_in[31]};
      dividend[0] = {dividend_in[30:0],1'b0};
    end
    for (int i=2; i<32; i=i+2)begin
      data_valid[i] = data_valid[i-1];
      if ({p[i-1][30:0],dividend[i-1][31]}>=divisor[i-1][31:0])begin
        p[i] = {p[i-1][30:0],dividend[i-1][31]} - divisor[i-1][31:0];
        dividend[i] = {dividend[i-1][30:0],1'b1};
      end else begin
        p[i] = {p[i-1][30:0],dividend[i-1][31]};
        dividend[i] = {dividend[i-1][30:0],1'b0};
      end
      divisor[i] = divisor[i-1];
    end
  end

  always_ff @(posedge clk_in)begin
    for (int i=1; i<32; i=i+2)begin
      data_valid[i] <= data_valid[i-1];
      if ({p[i-1][30:0],dividend[i-1][31]}>=divisor[i-1][31:0])begin
        p[i] <= {p[i-1][30:0],dividend[i-1][31]} - divisor[i-1][31:0];
        dividend[i] <= {dividend[i-1][30:0],1'b1};
      end else begin
        p[i] <= {p[i-1][30:0],dividend[i-1][31]};
        dividend[i] <= {dividend[i-1][30:0],1'b0};
      end
      divisor[i] <= divisor[i-1];
    end
  end
endmodule
```

divider5: The Big Reveal

Passed by 77 picoseconds!!!

```
Phase 10 Post Router Timing
INFO: [Route 35-57] Estimated Timing Summary | WNS=0.077 | TNS=0.000  | WHS=0.083  | THS=0.000  |

INFO: [Route 35-327] The final timing numbers are based on the router estimated timing analysis. For
a complete and accurate timing signoff, please run report_timing_summary.
Phase 10 Post Router Timing | Checksum: 1a748b8ad
```

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	426	0	0	15850	2.69
SLICEL	257	0			
SLICEM	169	0			
LUT as Logic	1492	0	0	63400	2.35
using O5 output only	0				
using O6 output only	987				
using O5 and O6	505				
LUT as Memory	54	0	0	19000	0.28
LUT as Distributed RAM	0	0			
LUT as Shift Register	54	0			
using O5 output only	0				
using O6 output only	54				
using O5 and O6	0				
Slice Registers	863	0	0	126800	0.68
Register driven from within the Slice	559				
Register driven from outside the Slice	304				
LUT in front of the register is unused	30				
LUT in front of the register is used	274				
Unique Control Sets	6		0	15850	0.04

Summary of the Journey

Divider	Resource Usage %LUT/%FF	Latency	Throughput
32 bit /	1.91/0.10	FAIL (78.215 ns)	FAIL (1/L)
divider 1 (lab04 og)	0.41/0.16	Variable	Variable
divider 2	0.33/0.17	32	1/32
divider 3	2.91/1.3	32	1/1
divider 4	2.76/0.74	**FAIL (barely)**	**FAIL (barely)**
divider 5	2.69/0.68	16	1/1

which to use?

Conclusions

- First: Use a good algorithm!
  - Doing things stupidly can only work out so well (not well)
- Second:
  - Figure out what we (you, customer) actually need…
  - Need to divide every clock cycle?
  - Need to divide every million clock cycles?

More Conclusions

- Some tasks can be parallelized:
  - (adding an array up…See Lecture 02 with big_adder)
- Some tasks cannot be parallelized and steps must be done sequentially:
  - 10 violinists cannot play a violin solo ten times as fast
  - Division is an iterative process inherently
- If must be done sequentially:
  - Variable-length or Fixed-length Algorithm?

Algorithms

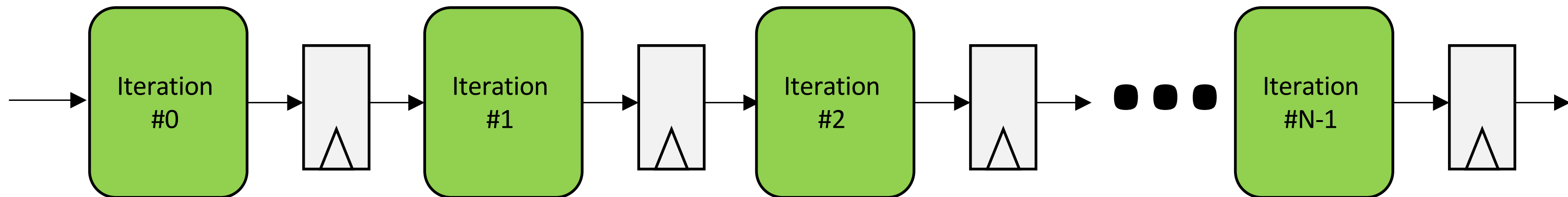
- Variable-length algorithm are generally implemented as type of state machine

Optimize for Need!

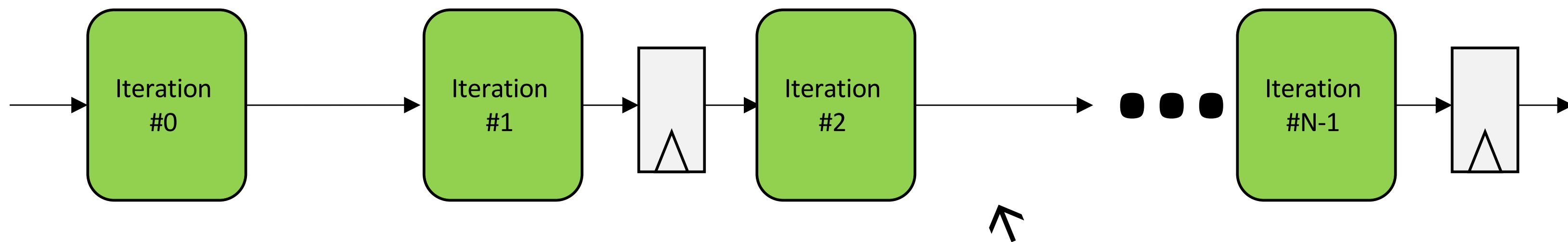
- All those options allow one to vary between amounts of pipelining and iterative behavior

```
Init: P←0, load A and B
Repeat N times {
    shift P,A left one bit
    temp = P-B
    if (temp >= 0){
      P←temp
      A_LSB←1
    }else{
        A_LSB←0
    }
}
Done: Q in A, R in P
```

$L = N*t_{clk}$, $T = 1/t_{clk}$ At small t_{clk} ***But use lots of resources:***



$L = 0.5*N*t_{clk}$, $T = 1/t_{clk}$ At ***larger t_{clk}*** *But uses slightly fewer of resources:*



Honestly there's not too much benefit to this

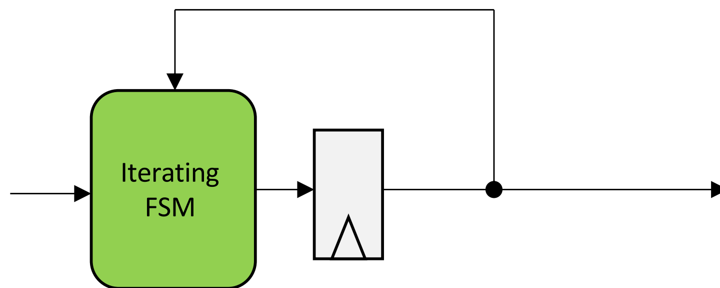
Optimize for Need!

- All those options allow one to vary between amounts of pipelining and iterative behavior

```
Init: P←0, load A and B
Repeat N times {
    shift P,A left one bit
    temp = P-B
    if (temp >= 0){
      P←temp
      A_LSB←1
    }else{
        A_LSB←0
    }
}
Done: Q in A, R in P
```

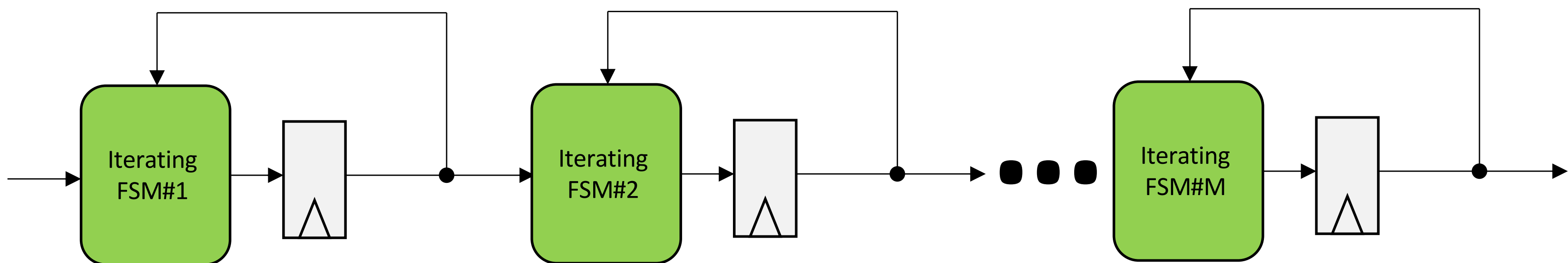
*$L = N*t_{clk}$, **$T = 1/(N*t_{clk})$** At small t_{clk} But use very few resources:*

← Takes N cycles to divide and can't accept new inputs during that time



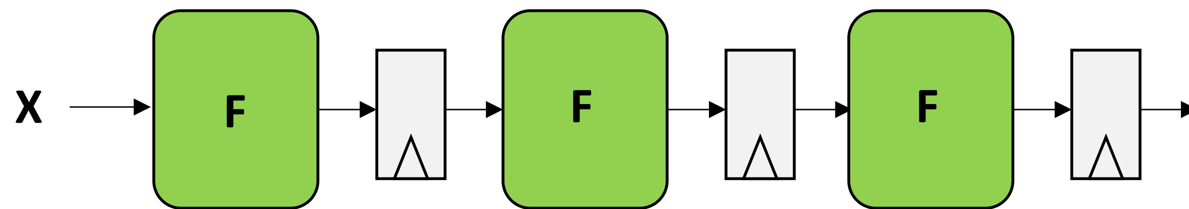
Takes N cycles to divide but can take a new input every N/M cycles

*$L = N*t_{clk}$ $T = 1/(N/M*t_{clk})$ At small t_{clk} **But uses more of resources:***



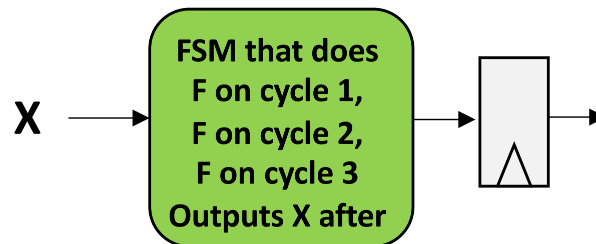
A lot of Algorithms are Repeition-Based though

- Let's say we need to compute F(F(F(X))). Do we build our hardware like this?:



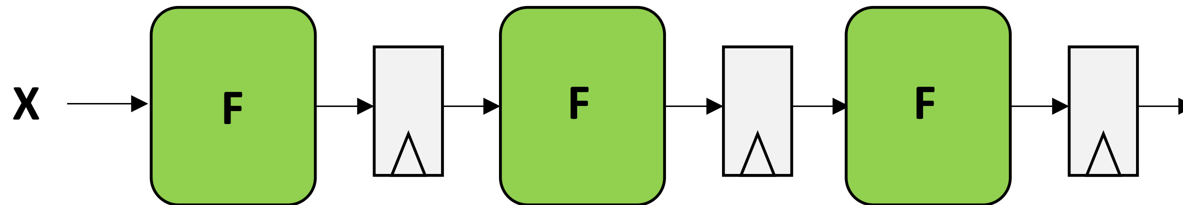
Latency: 3*T_{clk}
Throughput: 1/ T_{clk}
Uses more resources

- Or like this:?



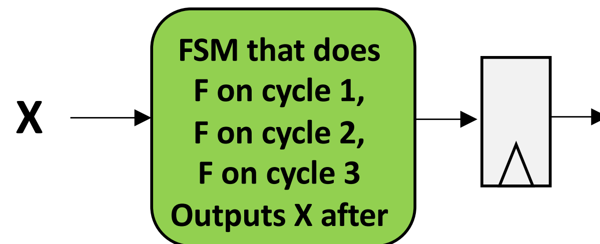
Latency: 3*T_{clk}
Throughput: 1/ (3*T_{clk})
Likely uses fewer resources

This is the Great Tradeoff!



More resources, Better Throughput Same Latency

OR



Fewer resources, Worse Throughput Same Latency

- Base on what you need for the design!