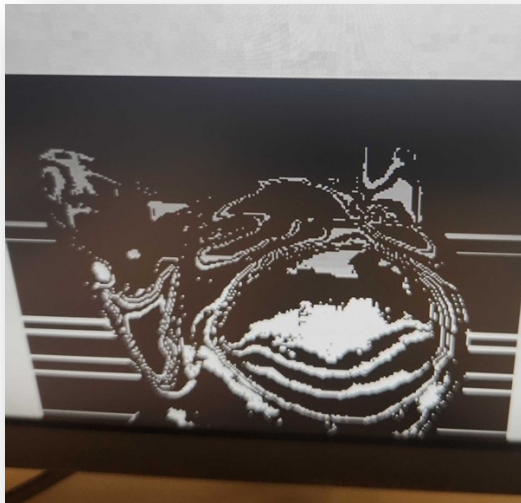


Pipelining

6.205



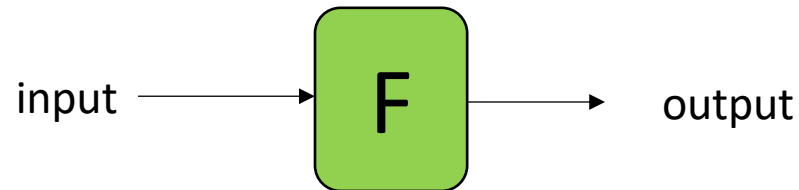
All happy popcats are alike; each unhappy popcat is unhappy in its own way

*-Leo Tolstoy
Anna Karenina (1877)*

Admin

- Week 05: due tomorrow
- Week 06 out on Thursday
- Final project teaming preferences due tonight! See piazza. No extensions

Performance Metrics



- Latency (L):
 - time between arrival of new input and generation of corresponding output.
 - For purely combinational circuits this is just t_{pD} .
 - For sequential circuits, it is the number of flops you travel through times the clock period
- Throughput (T):
 - Rate at which new outputs appear.
 - For purely combinational circuits this is just $1/t_{pD}$ or $1/L$.
 - For fully-pipelined circuits it is $1/1$

Division

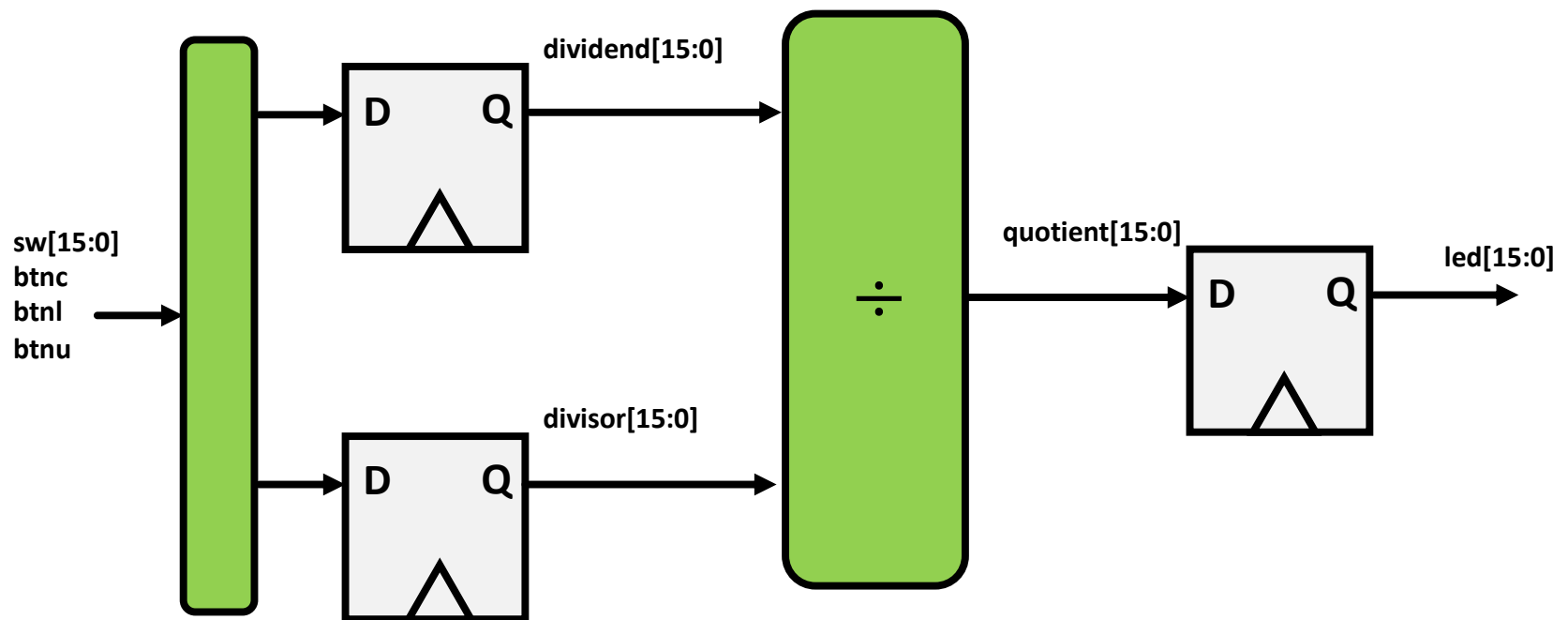
- The outlier in the + - * / set...
- Division is a significantly harder math operation to do compared to multiplication
- Where possible try to avoid
- Try to divide by powers of 2 (use right shift)!
- If you can't avoid we must do it.

One “Bad” Attempt at Division

- In previous lectures 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 (btnc & ~old_btnc)begin  
            dividend <= sw; //divide //load dividend  
        end  
        if (btnc & ~old_btnc)begin  
            divisor <= sw; //divide //load divisor  
        end  
    end  
endmodule
```

Circuit Built:



Build the Stupid Divider

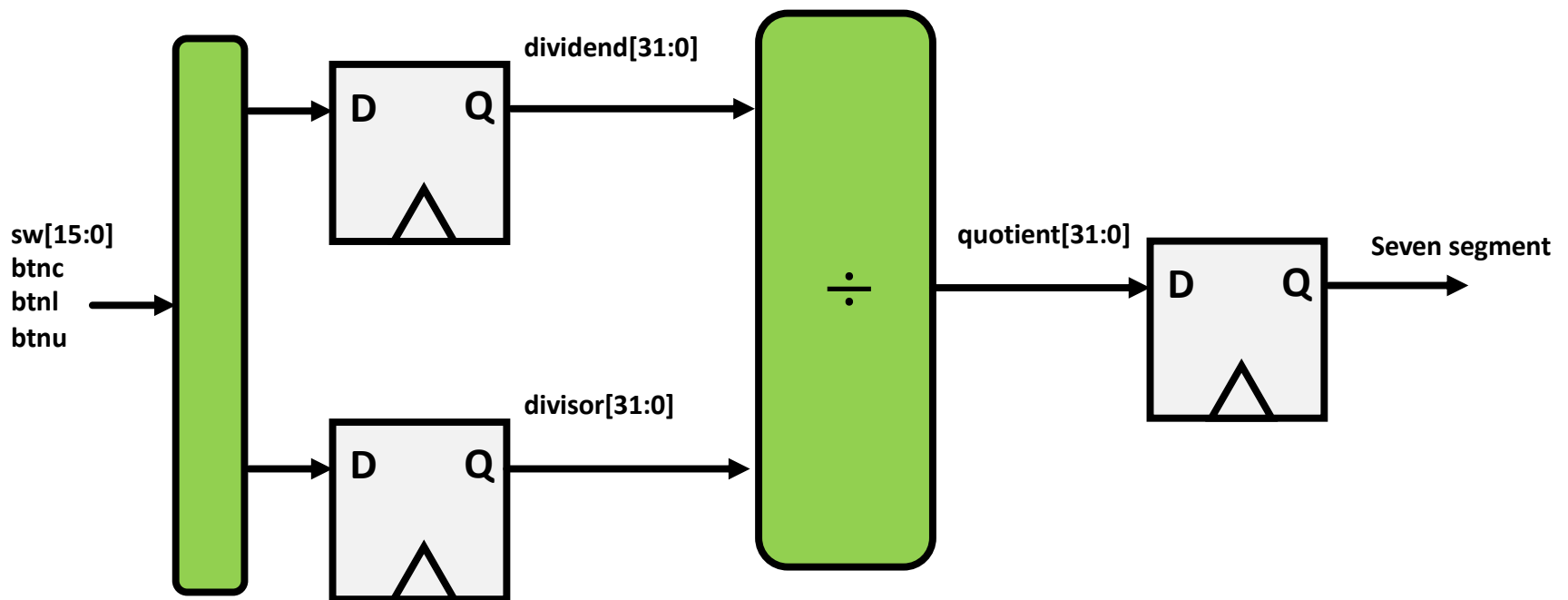
Violates timing!

```
Phase 22 Post Router Timing
INFO: [Route 35-20] Post Routing Timing Summary | WNS=-21.399 | TNS=-129.552 | WHS=0.090 |
THS=0.000 |
```

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 05 output only	0				
using 06 output only	274				
using 05 and 06	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 (pmod_pin & ~old_pmod_pin) begin
    quotient <= dividend/divisor;
end
```



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

Build the Stupider Divider

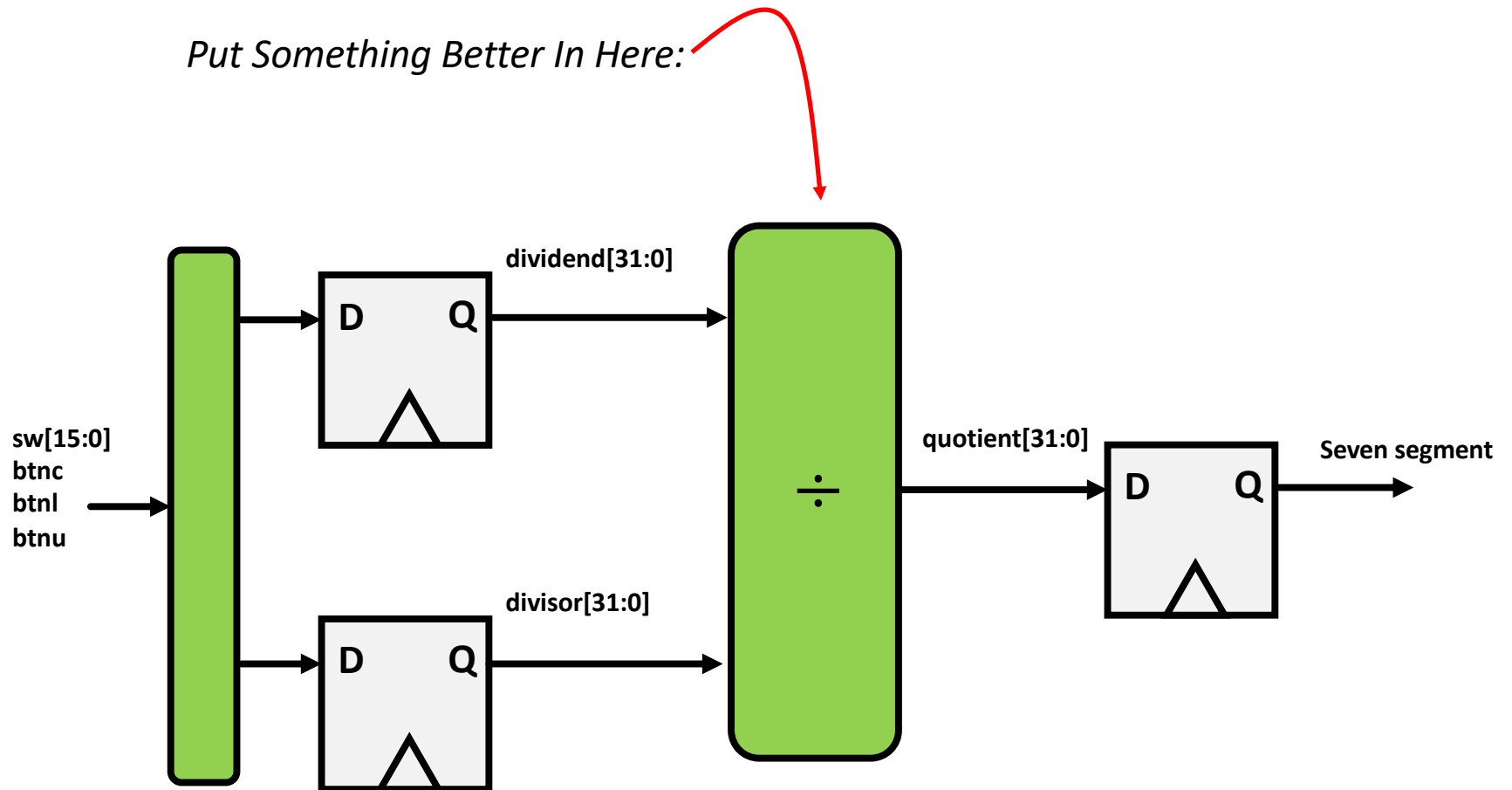
```
Phase 23 Post Router Timing  
INFO: [Route 35-20] Post Routing Timing Summary | WNS=-72.962 | TNS=-1017.985 | WHS=0.166 | THS=0.000 |
```

```
Phase 23 Post Router Timing | Checksum: 23fd227b7
```

2. Slice Logic Distribution. (from post_place_util.rpt)

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	303	0	0	8150	3.72
SLICEL	202	0			
SLICEM	101	0			
LUT as Logic	941	0	0	32600	2.89
using 05 output only	0				
using 06 output only	919				
using 05 and 06	22				
LUT as Memory	0	0	0	9600	0.00
LUT as Distributed RAM	0	0			
using 05 output only	0				
using 06 output only	0				
using 05 and 06	0				
LUT as Shift Register	0	0			
using 05 output only	0				
using 06 output only	0				
using 05 and 06	0				
Slice Registers	126	0	0	65200	0.19
Register driven from within the Slice	51				
Register driven from outside the Slice	75				
LUT in front of the register is unused	42				
LUT in front of the register is used	33				
Unique Control Sets	7		0	8150	0.09

A Better Divider?



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

Division (an example of an algorithm that takes an unknown amount of time)

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

Super efficient divider \s

A Divider (#1)

```
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
- Give you this 32 bit one in week05

```
module divider #(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;  
    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;  
        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;  
                    end  
                    data_valid_out <= 1'b0;  
                end  
                DIVIDING: begin  
                    if (dividend<=0)begin  
                        state <= RESTING; //similar to return statement  
                        remainder_out <= dividend;  
                        quotient_out <= quotient;  
                        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.  
                        state <= DIVIDING;  
                        quotient <= quotient + 1'b1;  
                        dividend <= dividend-divisor;  
                    end  
                end  
            endcase  
        end  
    end  
endmodule
```

Build divider1

Phase 12 Post Router Timing
INFO: [Route 35-20] Post Routing Timing Summary | WNS=4.533 | TNS=0.000 | WHS=0.164 | THS=0.000 |

2. Slice Logic Distribution

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	52	0	0	8150	0.64
SLICEL	39	0			
SLICEM	13	0			
LUT as Logic	140	0	0	32600	0.43
using 05 output only	0				
using 06 output only	107				
using 05 and 06	33				
LUT as Memory	0	0	0	9600	0.00
LUT as Distributed RAM	0	0			
using 05 output only	0				
using 06 output only	0				
using 05 and 06	0				
LUT as Shift Register	0	0			
using 05 output only	0				
using 06 output only	0				
using 05 and 06	0				
Slice Registers	192	0	0	65200	0.29
Register driven from within the Slice	85				
Register driven from outside the Slice	107				
LUT in front of the register is unused	49				
LUT in front of the register is used	58				
Unique Control Sets	9		0	8150	0.11

For divider1, what is the Good, the Bad, the Ugly?

- What are some nice features?
- What are some not-nice features?



Aside...



Original Italian poster 1967



Americanized poster with the Ugly and the Bad characters swapped



American DVD menu with the artwork appropriately reordered to match American name of movie

For divider1, though...what are good and bad?

- Good:
- ...
 - Meets timing
 - Resource usage is maybe small?
- Bad:
- ...
 - Blocking Implementation (low-throughput)
 - Variable throughput

So How to Fix...?

A Better Algorithm?

- This can't be how computers 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.
- 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.

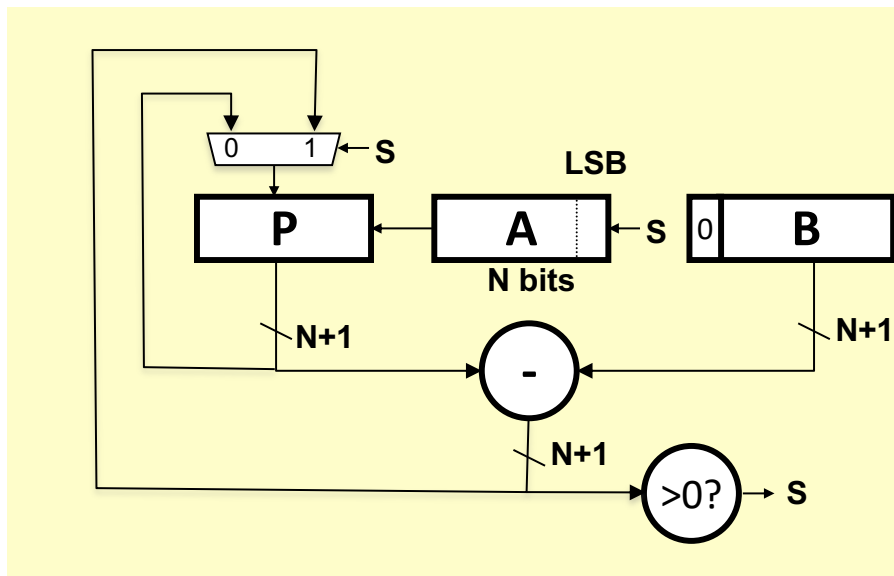
```
Init: P←0
Load A and B

Repeat N times {
  shift P,A left one bit
  temp = P-B
  if (temp >= 0){
    P←temp
    ALSB←1
  }else{
    ALSB←0
  }
}

Done: Q in A, R in P
```

Divider (Fixed # of Steps)

Assume the Dividend (**A**) and the divisor (**B**) have **N** bits. 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 and correct sign of result.



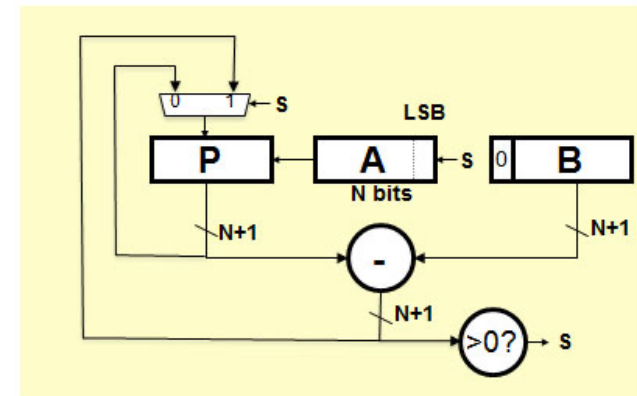
```
Init: P ← 0
Load A and B
```

```
Repeat N times {
  shift P, A left one bit
  temp = P - B
  if (temp ≥ 0) {
    P ← temp
    ALSB ← 1
  } else {
    ALSB ← 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: $P \leftarrow 0$
Load A and B

```
Repeat N times {
  shift P,A left one bit
  temp = P-B
  if (temp >= 0){
    P ← temp
    ALSB ← 1
  }else{
    ALSB ← 0
  }
}
```

Done: Q in A, R in P

divider2

- This is an FSM implementation of the “smarter” algorithm just shown:
- Latency:
 - 32 clock cycles (one for each bit)
- 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 12 Post Router Timing
INFO: [Route 35-20] Post Routing Timing Summary | **WNS=5.214** | TNS=0.000 | WHS=0.167 | THS=0.000 |

2. Slice Logic Distribution

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	55	0	0	8150	0.67
SLICEL	40	0			
SLICEM	15	0			
LUT as Logic	125	0	0	32600	0.38
using 05 output only	0				
using 06 output only	67				
using 05 and 06	58				
LUT as Memory	0	0	0	9600	0.00
LUT as Distributed RAM	0	0			
using 05 output only	0				
using 06 output only	0				
using 05 and 06	0				
LUT as Shift Register	0	0			
using 05 output only	0				
using 06 output only	0				
using 05 and 06	0				
Slice Registers	197	0	0	65200	0.30
Register driven from within the Slice	94				
Register driven from outside the Slice	103				
LUT in front of the register is unused	46				
LUT in front of the register is used	57				
Unique Control Sets	10		0	8150	0.12

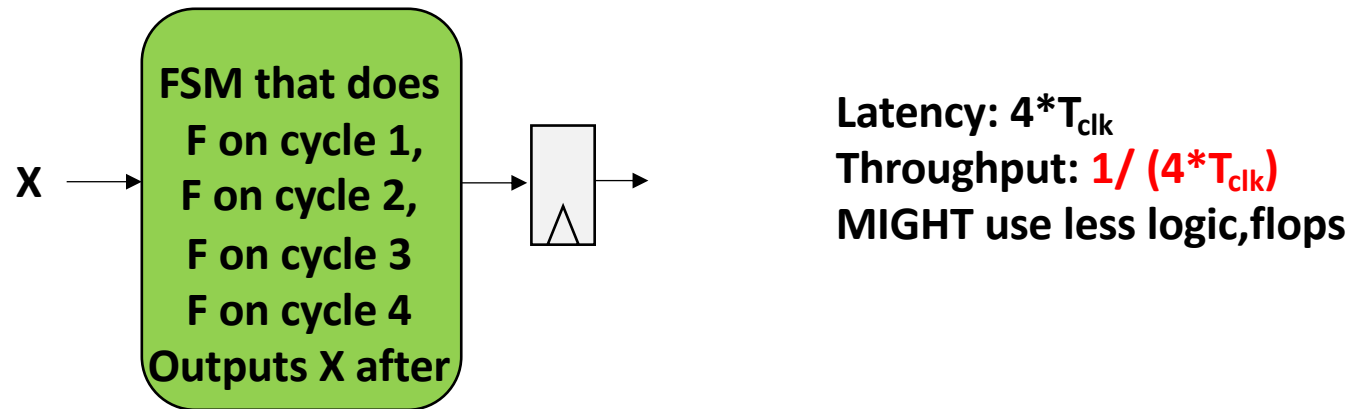
For divider2, ...what are good and bad?

- Good:
- ...
 - Meets timing
 - Nominally the same resource usage as before
 - Runs in fixed number of cycles!
- Bad:
- ...
 - Blocking Implementation (low-throughput)

Can We Make it Better?

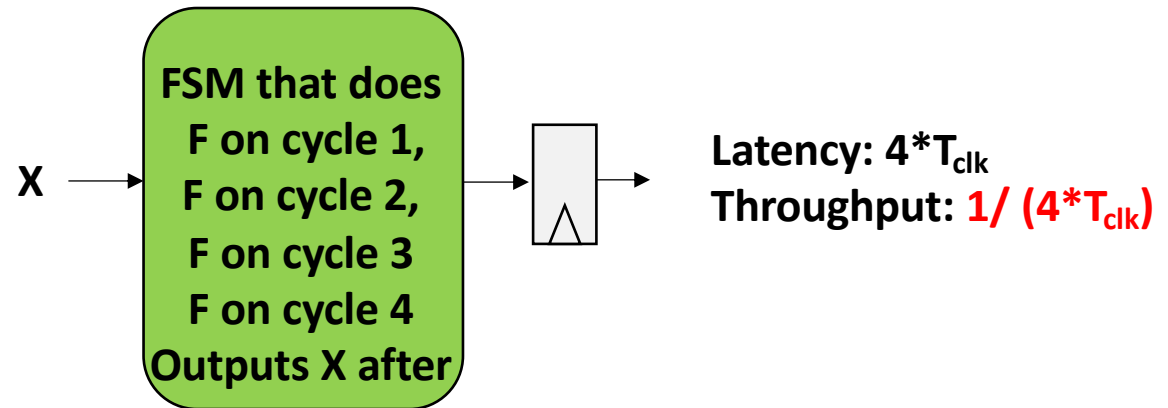
Phase 12 Post Router Timing
INFO: [Route 35-20] Post Routing Timing Summary WNS=5.214 | TNS=0.000 | WHS=0.167 | THS=0.000 |

- We have a lot of slack with this current design.
- Currently kinda doing something like this (but 32 cycles rather than 4 cycles):

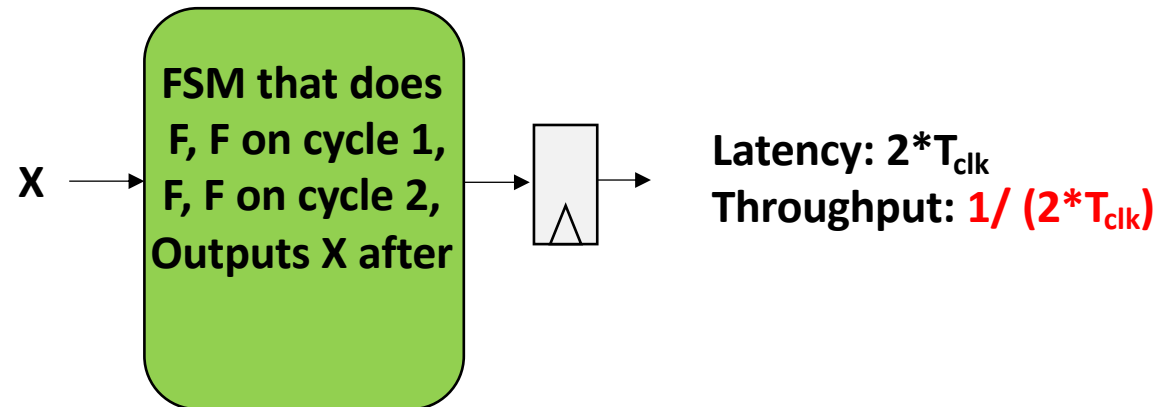


Could We Instead...

- Instead of this:



- Do this:



Divider2b: Wedge a second iteration into each clock cycle:

- Did not mix-n-match blocking/non-blocking in my `always_ff` because I realize that to err is human and this will lead to my downfall, if not today, then tomorrow
- Instead made an `always_comb` with some “temp” variables to hold the result of the first iteration

```
end
DIVIDING: begin
  if (count==1)begin
    state <= RESTING;
    if ({p_temp[30:0],div_temp[31]}>=divisor[31:0])begin
      remainder_out <= {p_temp[30:0],div_temp[31]} - divisor[31:0];
      quotient_out <= {div_temp[30:0],1'b1};
    end else begin
      remainder_out <= {p_temp[30:0],div_temp[31]};
      quotient_out <= {div_temp[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_temp[30:0],div_temp[31]}>=divisor[31:0])begin
      p <= {p_temp[30:0],div_temp[31]} - divisor[31:0];
      dividend <= {div_temp[30:0],1'b1};
    end else begin
      p <= {p_temp[30:0],div_temp[31]};
      dividend <= {div_temp[30:0],1'b0};
    end
    count <= count-2;
  end
end
endcase
end
end
//extra:
logic [31:0] p_temp;
logic [31:0] div_temp;
always_comb begin
  if ({p[30:0],dividend[31]}>=divisor[31:0])begin
    p_temp = {p[30:0],dividend[31]} - divisor[31:0];
    div_temp = {dividend[30:0],1'b1};
  end else begin
    p_temp = {p[30:0],dividend[31]};
    div_temp = {dividend[30:0],1'b0};
  end
end
end
endmodule
```

Build divider2b:

Phase 12 Post Router Timing
INFO: [Route 35-20] Post Routing Timing Summary | **WNS=1.561** | TNS=0.000 | WHS=0.160 | THS=0.000 |

2. Slice Logic Distribution

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	67	0	0	8150	0.82
SLICEL	44	0			
SLICEM	23	0			
LUT as Logic	185	0	0	32600	0.57
using 05 output only	0				
using 06 output only	125				
using 05 and 06	60				
LUT as Memory	0	0	0	9600	0.00
LUT as Distributed RAM	0	0			
using 05 output only	0				
using 06 output only	0				
using 05 and 06	0				
LUT as Shift Register	0	0			
using 05 output only	0				
using 06 output only	0				
using 05 and 06	0				
Slice Registers	197	0	0	65200	0.30
Register driven from within the Slice	92				
Register driven from outside the Slice	105				
LUT in front of the register is unused	35				
LUT in front of the register is used	70				
Unique Control Sets	9		0	8150	0.11

For divider2b, ...what are good and bad?

- Good:
- ...
 - Meets timing
 - Improved, fixed throughput (2X)
 - Latency improved (1/2X)
- Bad:
- ...
 - Blocking Implementation (low-throughput)
 - Resource usage a little bit higher

Make it Better?

- Easiest thing to try is to shove *three steps or four steps* of the algorithm into one clock cycle?
- Maybe?
- lunno
- Maybe?

Attempt at divider2c...try to do three layers of the algorithm on one clock cycle

- Used some more poorly named variables to act as intermediaries
- But should work “in theory”*

**to use the terminology of a student trying to convince me that they achieved what they set out to do on their final project when they did not.*

```
case (state)
  DIVIDING: begin
    if (count==1)begin
      state <= RESTING;
      if ({p_temp2[30:0],div_temp2[31]}>=divisor[31:0])begin
        remainder_out <= {p_temp2[30:0],div_temp2[31]} - divisor[31:0];
        quotient_out <= {div_temp2[30:0],1'b1};
      end else begin
        remainder_out <= {p_temp2[30:0],div_temp2[31]};
        quotient_out <= {div_temp2[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_temp2[30:0],div_temp2[31]}>=divisor[31:0])begin
        p <= {p_temp2[30:0],div_temp2[31]} - divisor[31:0];
        dividend <= {div_temp2[30:0],1'b1};
      end else begin
        p <= {p_temp2[30:0],div_temp2[31]};
        dividend <= {div_temp2[30:0],1'b0};
      end
      count <= count-2;
    end
  end
endcase
end
//extra:
logic [31:0] p_temp;
logic [31:0] div_temp;
logic [31:0] p_temp2;
logic [31:0] div_temp2;
always_comb begin
  if ({p[30:0],dividend[31]}>=divisor[31:0])begin
    p_temp = {p[30:0],dividend[31]} - divisor[31:0];
    div_temp = {dividend[30:0],1'b1};
  end else begin
    p_temp = {p[30:0],dividend[31]};
    div_temp = {dividend[30:0],1'b0};
  end
  if ({p_temp[30:0],div_temp[31]}>=divisor[31:0])begin
    p_temp2 = {p_temp[30:0],div_temp[31]} - divisor[31:0];
    div_temp2 = {div_temp[30:0],1'b1};
  end else begin
    p_temp2 = {p_temp[30:0],div_temp[31]};
    div_temp2 = {div_temp[30:0],1'b0};
  end
end
end
```

Build divider2c:

Phase 25 Post Router Timing
INFO: [Route 35-20] Post Routing Timing Summary

WNS=-2.343

TNS=-66.953 | WHS=0.159 | THS=0.000 |

2. Slice Logic Distribution

Timing Failed...got greedy...tried to fly too close to the sun, Icarus

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	88	0	0	8150	1.08
SLICEL	66	0			
SLICEM	22	0			
LUT as Logic	234	0	0	32600	0.72
using 05 output only	0				
using 06 output only	160				
using 05 and 06	74				
LUT as Memory	0	0	0	9600	0.00
LUT as Distributed RAM	0	0			
using 05 output only	0				
using 06 output only	0				
using 05 and 06	0				
LUT as Shift Register	0	0			
using 05 output only	0				
using 06 output only	0				
using 05 and 06	0				
Slice Registers	197	0	0	65200	0.30
Register driven from within the Slice	96				
Register driven from outside the Slice	101				
LUT in front of the register is unused	31				
LUT in front of the register is used	70				
Unique Control Sets	9		0	8150	0.11

Another interesting feature

- Notice this number:

```
Phase 25 Post Router Timing  
INFO: [Route 35-20] Post Routing Timing Summary | WNS=-2.343 | TNS=-66.953 | WHS=0.159 | THS=0.000 |
```

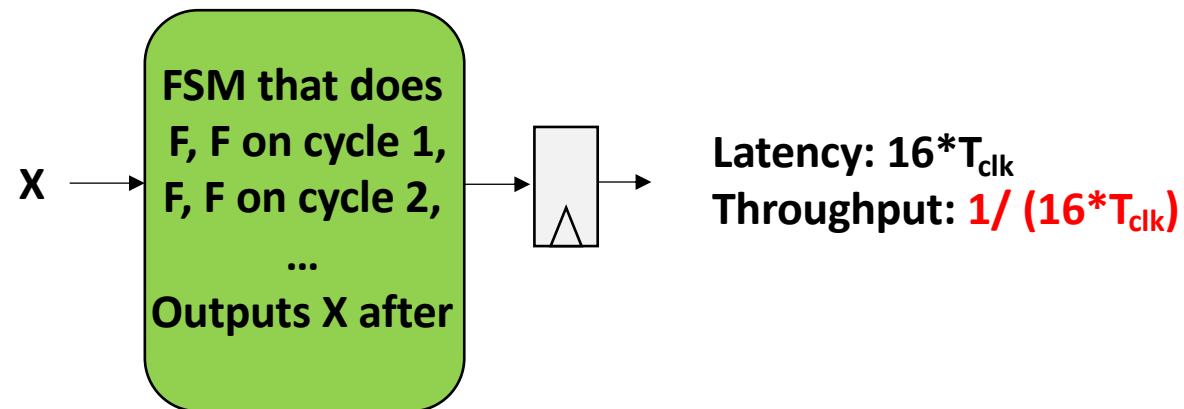
- Whereas a design that worked earlier is this:

```
Phase 12 Post Router Timing  
INFO: [Route 35-20] Post Routing Timing Summary | WNS=1.561 | TNS=0.000 | WHS=0.160 | THS=0.000 |
```

- Designs which fit timing easily will go through fewer phases of optimization
- Vivado will give up after too many phases and can't achieve

Summary so far...

- So we've made some gains by:
 - picking a better algorithm (something suited to base 2)
 - Shoving more iterations of the cycle between the clock edges...



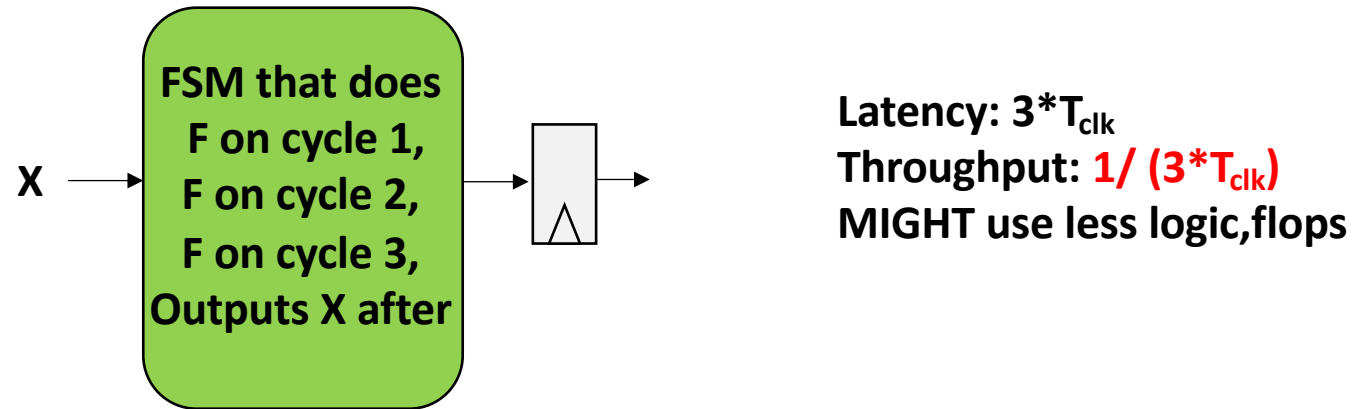
- Latency still bad though :/

Can We Make it Better?

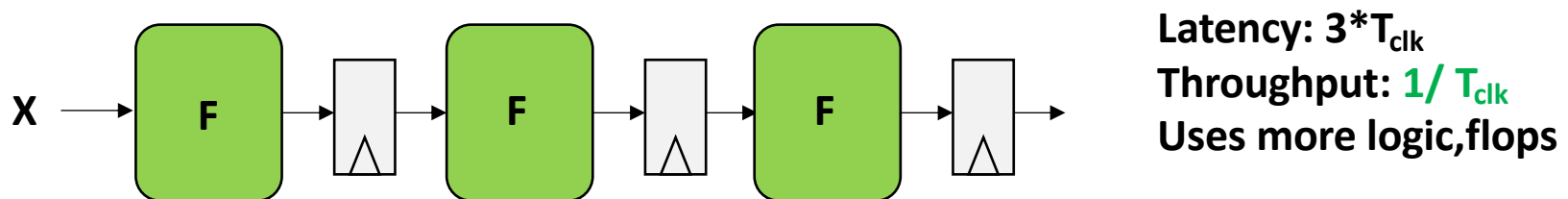
- We have an algorithm that takes a fixed amount of cycles per divide (32 in our case)
- Because of this we know exactly how many calculations we need to do.
- This allows us to set up a fully-pipelined system (can't easily do in a variable-run-time algorithm)

What?

- Currently we're doing something like this:



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



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
```

How to “Unwrap/Unroll”?

Variables at each point on an algorithm's iteration are now separate

```
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
```

```
Init: P←0
Load A and B

Repeat N times {
  shift P,A left one bit
  temp = P-B
  if (temp >= 0){
    P←temp
    ALSB←1
  }else{
    ALSB←0
  }
}

Done: Q in A, R in P
```

Compare to the FSM-based approach before where there was one p variable, for example

```
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
```

How to “Unroll”?

Have your initial stage of variables being processed

```
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
end
```

Use index variables to refer backwards in “history” of the algorithm...not in time, but space

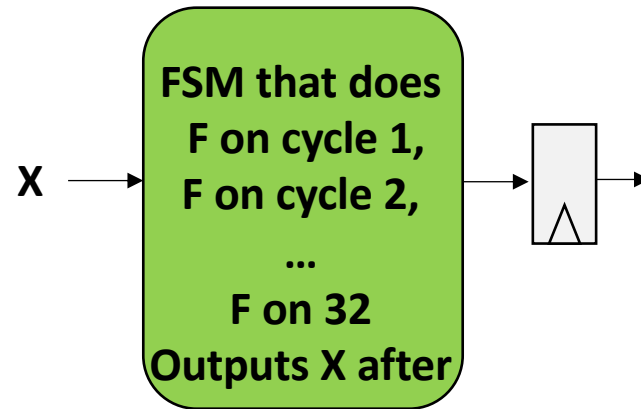
```
p[i] <= {p[i-1][30:0],dividend[i-1][31]};
```

Previously in FSM, p referred to itself!

```
p <= {p[30:0],dividend[31]};
```

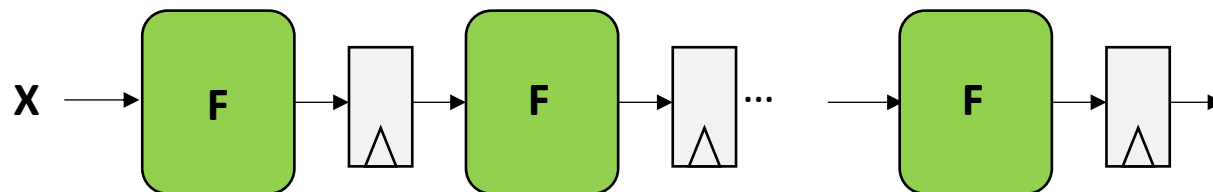
Nice...fully unrolled

- Were previously doing this:



Latency: $32 * T_{clk}$
Throughput: $1 / (32 * T_{clk})$
MIGHT use less logic, flops

- Now instead do this (“unwrap the loop”):



Latency: $32 * T_{clk}$
Throughput: $1 / T_{clk}$
Uses more logic, flops

Build divider3

Phase 12 Post Router Timing
 INFO: [Route 35-20] Post Routing Timing Summary | WNS=3.738 | TNS=0.000 | WHS=0.054 | THS=0.000 |

2. Slice Logic Distribution

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	492	0	0	8150	6.04
SLICEL	336	0			
SLICEM	156	0			
LUT as Logic	1504	0	0	32600	4.61
using 05 output only	0				
using 06 output only	999				
using 05 and 06	505				
LUT as Memory	58	0	0	9600	0.60
LUT as Distributed RAM	0	0			
using 05 output only	0				
using 06 output only	0				
using 05 and 06	0				
LUT as Shift Register	58	0			
using 05 output only	17				
using 06 output only	41				
using 05 and 06	0				
Slice Registers	1632	0	0	65200	2.50
Register driven from within the Slice	1052				
Register driven from outside the Slice	580				
LUT in front of the register is unused	194				
LUT in front of the register is used	386				
Unique Control Sets	7		0	8150	0.09

Resource usage went way up! Why?

We should expect increased resource usage!!

- We've traded resources for throughput
- Now can do 100 million divisions per second as opposed to ~3 million per second from before

For divider3, ...what are good and bad?

- Good:
- ...
 - Meets timing
 - Improved, fixed throughput (32X) compared to v2
 - Latency the same (compared to v2)
- Bad:
- ...
 - Resource usage significantly higher

Do it Better?

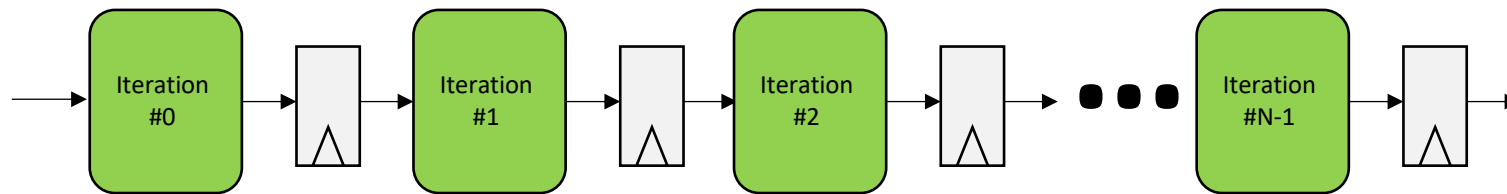
- Can I get my result faster?

```
Init: P ← 0
Load A and B

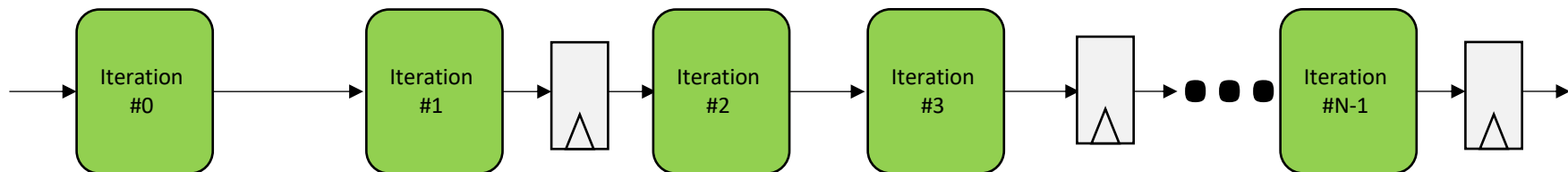
Repeat N times {
  shift P, A left one bit
  temp = P - B
  if (temp ≥ 0) {
    P ← temp
    A_LSB ← 1
  } else {
    A_LSB ← 0
  }
}

Done: Q in A, R in P
```

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



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



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

Pass timing by 143 picoseconds

Phase 12 Post Router Timing
 INFO: [Route 35-20] Post Routing Timing Summary | **WNS=0.173** | TNS=0.000 | WHS=0.057 | THS=0.000 |

2. Slice Logic Distribution

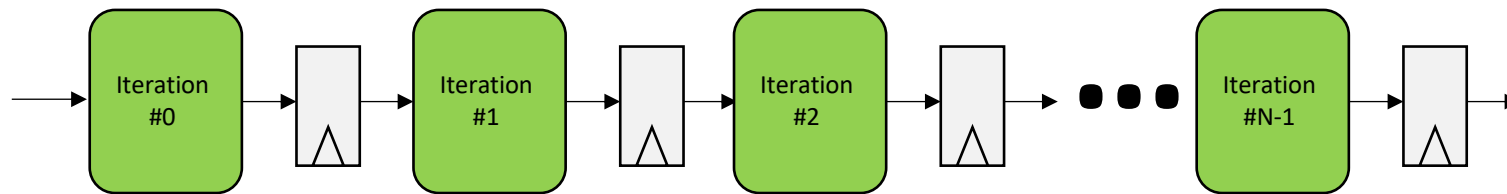
Flip flop usage dropped by a lot! (prev 2.50%) Why?

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	460	0	0	8150	5.64
SLICEL	321	0			
SLICEM	139	0			
LUT as Logic	1504	0	0	32600	4.61
using 05 output only	0				
using 06 output only	999				
using 05 and 06	505				
LUT as Memory	54	0	0	9600	0.56
LUT as Distributed RAM	0	0			
using 05 output only	0				
using 06 output only	0				
using 05 and 06	0				
LUT as Shift Register	54	0			
using 05 output only	0				
using 06 output only	54				
using 05 and 06	0				
Slice Registers	919	0	0	65200	1.41
Register driven from within the Slice	574				
Register driven from outside the Slice	345				
LUT in front of the register is unused	113				
LUT in front of the register is used	232				
Unique Control Sets	7		0	8150	0.09

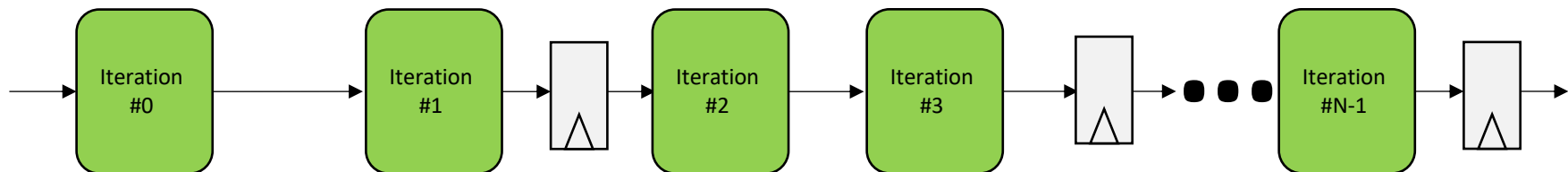
Fewer Flip Flops

- Should be expected!

Previously:



Currently:



Summary of the Journey

Divider	Resource Usage %LUT/%FF	Latency	Throughput
32 bit / divider 1 (lec06/week5)	3.72/0.29	FAIL (-72.004 ns)	FAIL (1/L)
divider 2	0.64/0.29	Variable	Variable
divider 2b	0.67/0.30	32	1/32
divider 2c	0.82/0.30	16	1/16
divider 3	1.08/0.30	FAIL (-2.3ns)	FAIL (1/L)
divider 4	6.04/2.50	32	1/1
	5.64/1.41	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
- Fixed-length algorithms can be more flexible:
 - FSM (blocking)
 - Fully pipeline (assembly-line)
 - Mixture in between

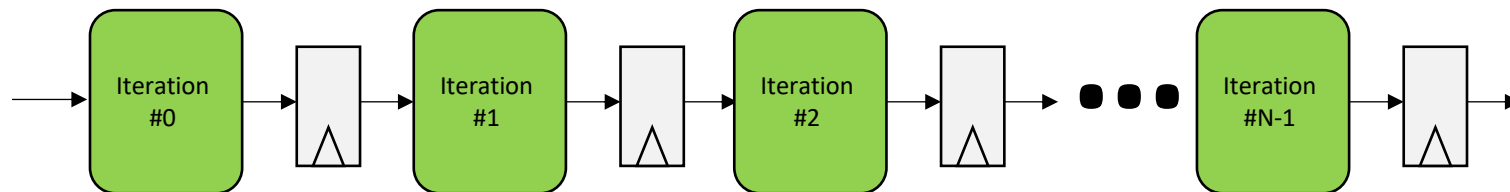
Optimize for Need!

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

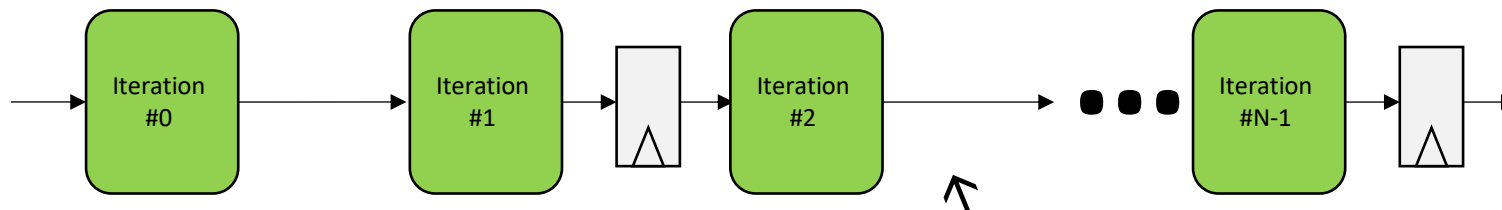
```

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 minimal benefit to this at least for divider since it now barely passes timing

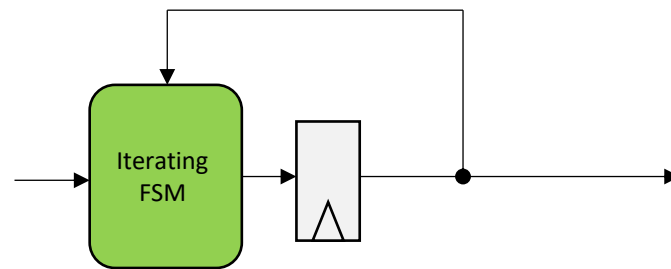
Optimize for Need!

- 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:

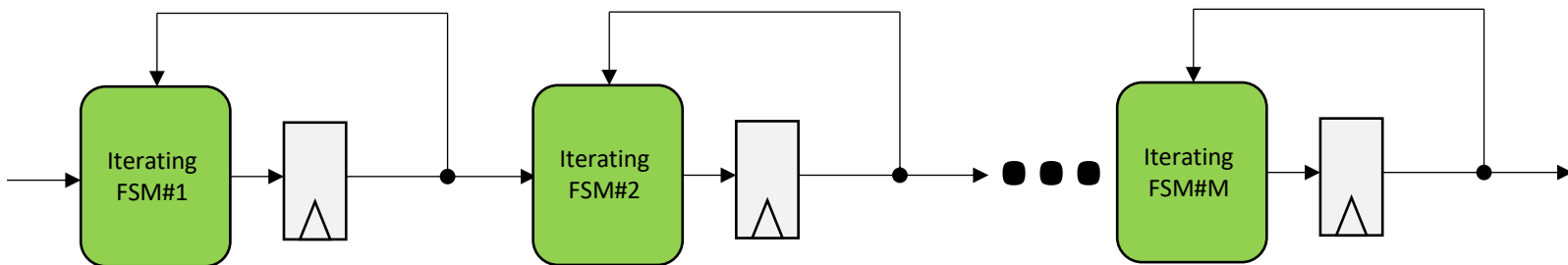


← Takes N cycles to divide and can't accept new inputs during that time

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

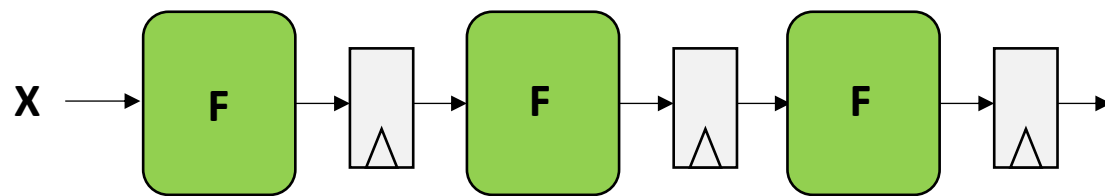
↙ N/M cycles

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



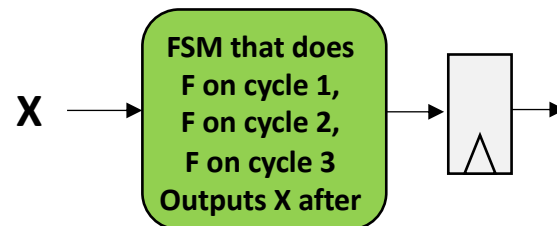
A lot of Algorithms are Repetition-Based though

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



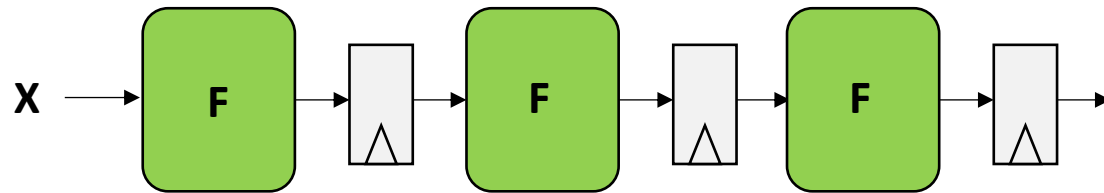
Latency: $3 \cdot T_{\text{clk}}$
Throughput: $1 / T_{\text{clk}}$
Uses more resources

- Or like this:?



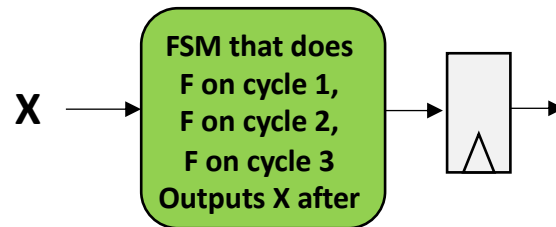
Latency: $3 \cdot T_{\text{clk}}$
Throughput: $1 / (3 \cdot T_{\text{clk}})$
Likely uses fewer resources

This is the Great Tradeoff!



**More resources,
Better Throughput
Same Latency**

OR



**Fewer resources,
Worse Throughput
Same Latency**

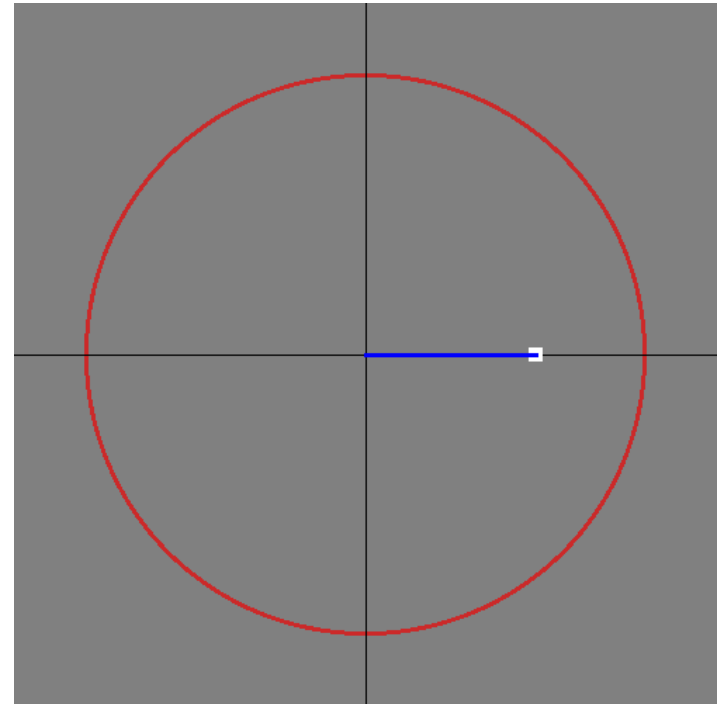
- Base on what you need for the design!

Most of what you may need to do can be framed in this way

- What about the “other” math operations?
- Square root?
- Trig functions?
- Exponents?
- Anything else?
- There’s usually a “smart” way to do it.

CORDIC

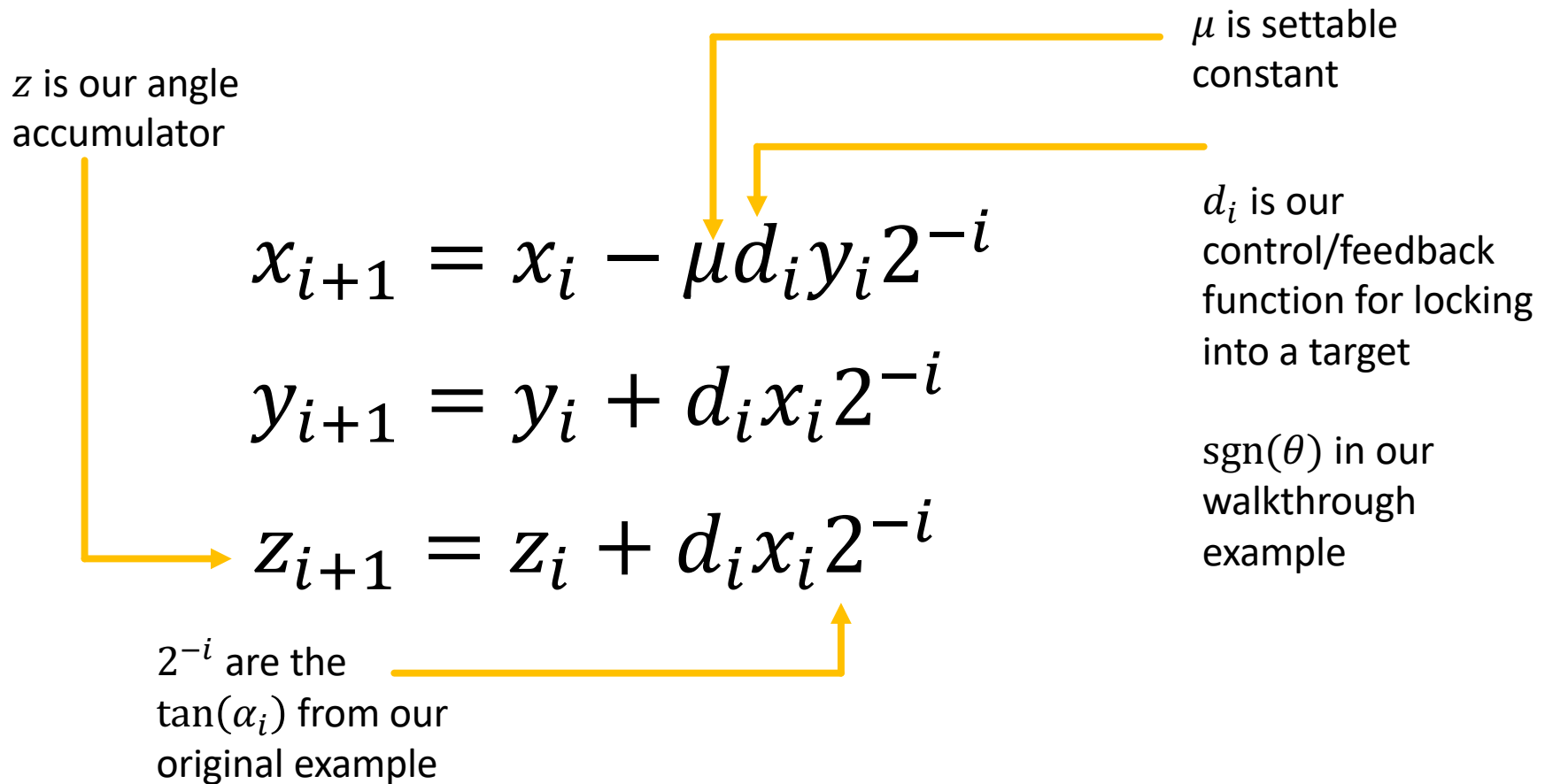
- **Coordinate Rotation Digital Computer**
- Super versatile class of iterative algorithms that are used widely in hardware because they are relatively simple to implement (mostly just shifts and adds and compares)
- Can operate quite efficiently using a minimal amount of resources



<https://www.remcycles.net/blog/cordic.html>

Generalized CORDIC

- The three equations are iterated



Different Modes

Mode	Rotation	Vectoring
	$d_i = \text{sgn}(z_i), \quad z \rightarrow 0$	$d_i = -\text{sgn}(y_i), \quad y \rightarrow 0$
Circular $\mu = 1$ $\alpha_i = \tan^{-1}2^{-i}$		
Linear $\mu = 0$ $\alpha_i = 2^{-i}$		
Hyperbolic $\mu = -1$ $\alpha_i = \tanh^{-1}2^{-i}$		

- In hyperbolic mode, iterations 4, 13, 40, 121, ..., $j, 3j+1, \dots$ must be repeated. The constant K' given below accounts for this.
- $K = 1.646760258121\dots$
- $1/K = 0.607252935009\dots$
- $K' = 0.8281593609602\dots$
- $1/K' = 1.207497067763\dots$

CORDIC

- What can you compute with CORDIC?

Directly computable functions [\[edit | edit source \]](#)

$\sin z$	$\cos z$
$\tan^{-1} z$	$\sinh z$
$\cosh z$	$\tanh^{-1} z$
y/x	xz
$\tan^{-1}(y/x)$	$\sqrt{x^2 + y^2}$
$\sqrt{x^2 - y^2}$	$e^z = \sinh z + \cosh z$

Indirectly computable functions [\[edit | edit source \]](#)

In addition to the above functions, a number of other functions can be produced by combining the results of previous computations:

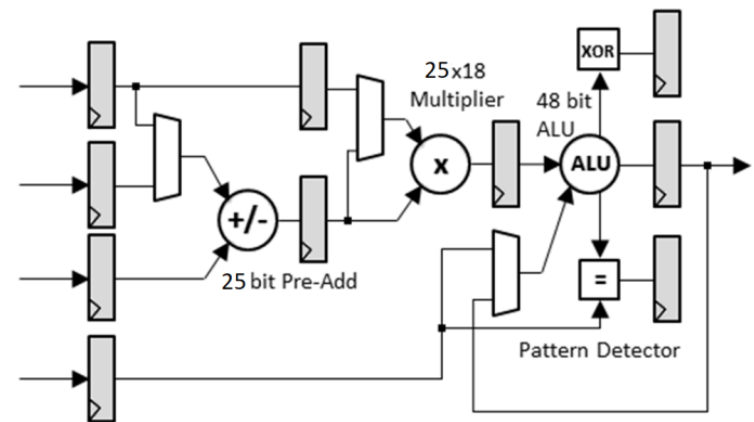
$\tan z = \frac{\sin z}{\cos z}$	$\cos^{-1} w = \tan^{-1} \frac{\sqrt{1-w^2}}{w}$
$\tanh z = \frac{\sinh z}{\cosh z}$	$\sin^{-1} w = \tan^{-1} \frac{w}{\sqrt{1-w^2}}$
$\ln w = 2 \tanh^{-1} \frac{w-1}{w+1}$	$\log_b w = \frac{\ln w}{\ln b}$
$w^t = e^{t \ln w}$	$\cosh^{-1} = \ln(w + \sqrt{w^2 - 1})$
$\tan^{-1}(y/x)$	$\sinh^{-1} = \ln(w + \sqrt{w^2 + 1})$
$\sqrt{x^2 - y^2}$	$\sqrt{w} = \sqrt{(w+1/4)^2 - (w-1/4)^2}$

Often Use more “Primitive” algorithms on an FPGA

- Along with things like `srli` or `add`, modern processors will often have:
 - 32-bit integer multiply instructions
 - Floating-point instructions
 - Etc...
- If both `srli` and `mult` cost the same in terms of instructions, then you might as well use a `mult` if it gets you more performance
 - And many algorithms for certain things can be done more quickly using `mult` than just `srli`

In an FPGA, accelerator, etc...

- If you have the freedom to not use `mult`, and it has a benefit (perhaps in terms of resource usage)...
- Then you should consider it as another degree in which to optimize.
- Have finite number of multiplier blocks on FPGA...
- Spending some on an algorithm that doesn't need it could hurt you elsewhere



So make sure you explore your algorithms

- Don't necessarily do it the software way or even the "C-way" since those are often optimized to a different set of constraints.

Data Type Sizes

- In a traditional processor, instructions are optimized for particular data type sizes:
 - 32 or 64 bit integers
 - 32 or 64 bit floats
- Don't need to do that necessarily anymore
- Can be the difference between making timing and not making timing

The Ongoing 8-bit debate in the ML field

FP8 FORMATS FOR DEEP LEARNING

Paulius Micikevicius, Dusan Stosic, Patrick Judd, John Kamalu, Stuart Oberman, Mohammad Shoeybi, Michael Siu, Hao Wu

NVIDIA

{paulium, dstosic, pjudd, jkamalu, soberman, mshoeybi, msiu, skyw}@nvidia.com

Neil Burgess, Sangwon Ha, Richard Grisenthwaite

Arm

{n.burgess, s.ha, richard.grisenthwaite}@arm.com

Sam Cornea, Alexander Heinecke, Pradeep Dubey

Intel

{s.cornea, alexander.heinecke, pradeep.dubey}@intel.com

ABSTRACT

Enabling deep learning training inference beyond the 16-bit
In this paper we propose an 8-bit floating point (FP8) binary

FP8 Quantization: The Power of the Exponent

Andrey Kuzmin*, Mart Van Baalen*, Yuwei Ren,
Markus Nagel, Jorn Peters, Tijmen Blankevoort

Qualcomm AI Research†

{akuzmin, mart, ren, markusn, jpeters, tijmen}@qti.qualcomm.com

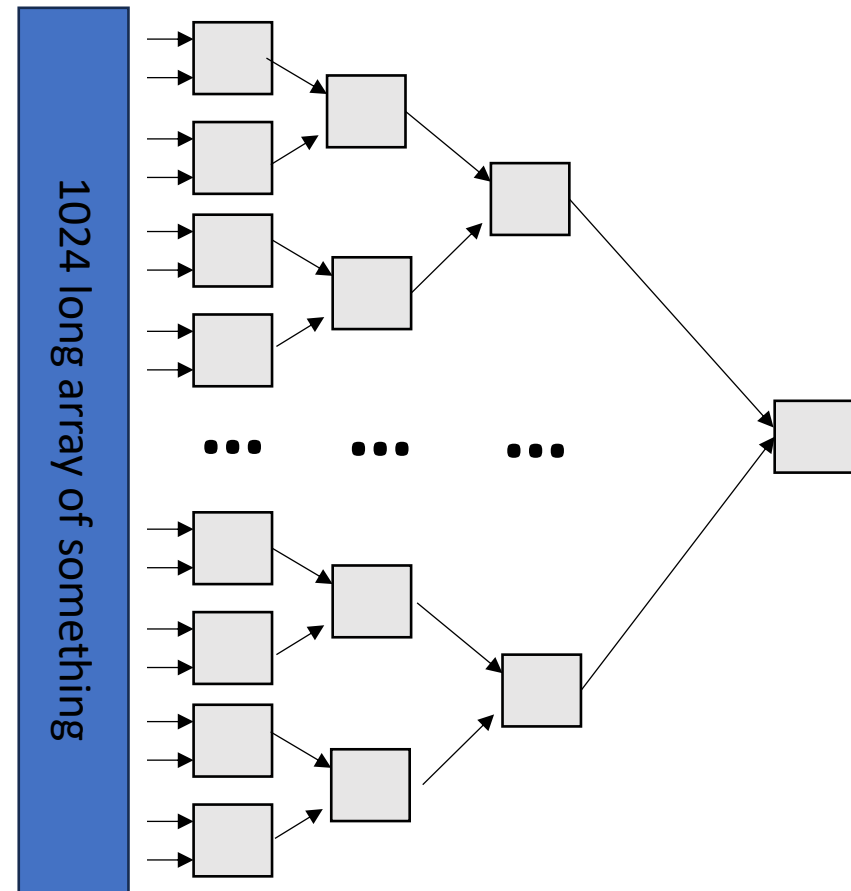
Abstract

When quantizing neural networks for efficient inference, low-bit integers are the go-to format for efficiency. However, low-bit floating point numbers have an extra degree of freedom, assigning some bits to work on an exponential scale instead. This paper in-depth investigates this benefit of the floating point format for neural network inference. We detail the choices that can be made for the FP8 format, including the important choice of the number of bits for the mantissa and exponent,

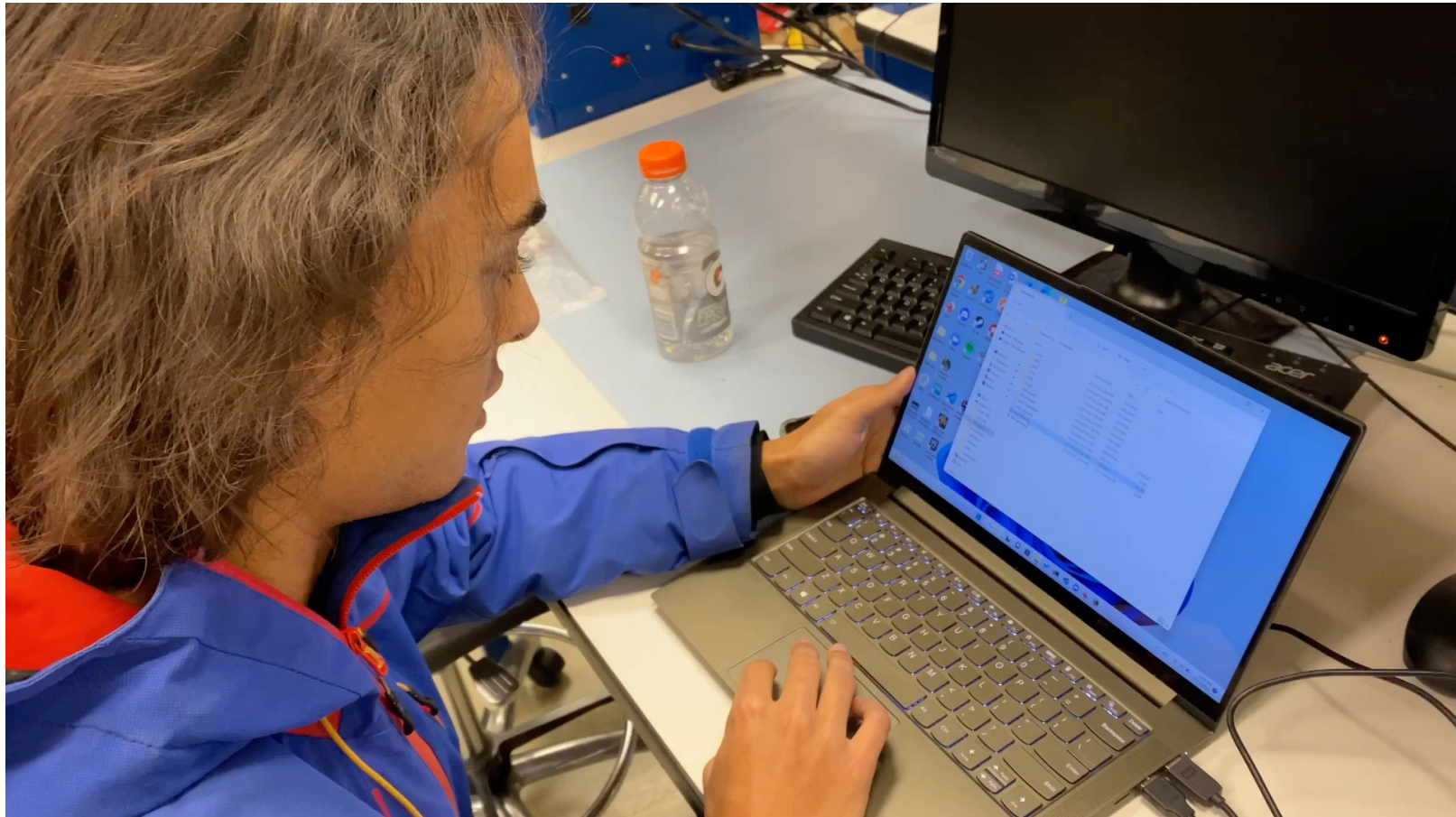
23 Feb 2024

And of course...remember memory is often a limiting factor!

- Few years ago...team built a fully-pipelined search implementation.
- Could search 1024 elements 100 million times per second.
- But we couldn't give it data fast enough to take advantage of it.



Project Video



Project Video

Mirror Me
6.205 Final Project Fall 2022

Janette (Jan) Park and Thienan Nguyen

Final Project Teams/preferences by tonight

- Must submit tonight
- Do it
- See Piazza announcement