ECE 4514
Digital Design II
Spring 2008

Lecture 17:
Hardware Division

A *Design* Lecture

Patrick Schaumont

Overview

- Division algorithms/architectures
  - Division as an 'inverted' multiplication
  - Division one digit at-a-time: digit-recurrence
  - The restoring divider algorithm
  - The non-restoring divider algorithm
  - Designing the non-restoring divider architecture

Before Division .. look at Multiplication First

- Binary multiplication using shift - and - add

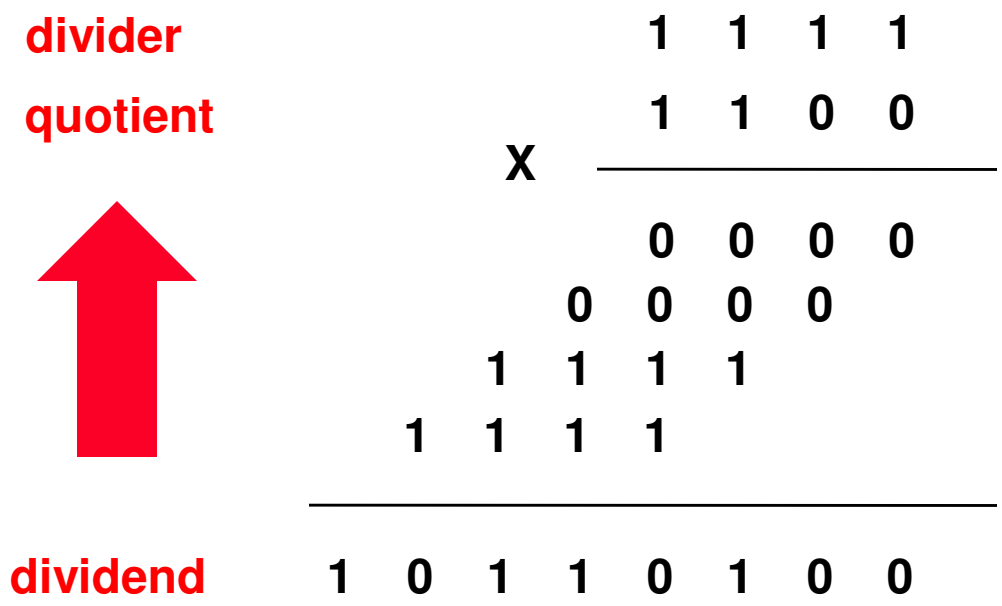
15					1	1	1	1
12					1	1	0	0
			X					
					0	0	0	0
				0	0	0	0	
			1	1	1	1		
		1	1	1	1			
180	1	0	1	1	0	1	0	0

Before Division .. look at Multiplication First

- Binary multiplication using shift - and - add
- 12 x 15 =
  - 0 x 1 x 15 +
  - 0 x 2 x 15 +
  - 1 x 4 x 15 +
  - 1 x 8 x 15
- In multiplication, we look at one bit of the multiplier at a time, and accumulate & shift the multiplicand

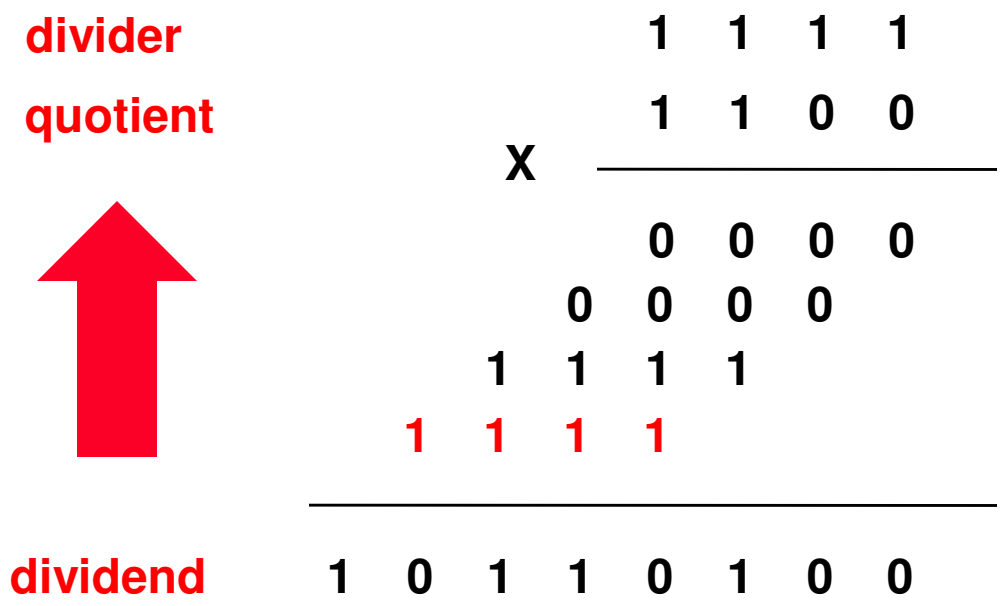
How does division work?

- more or less the opposite way ..



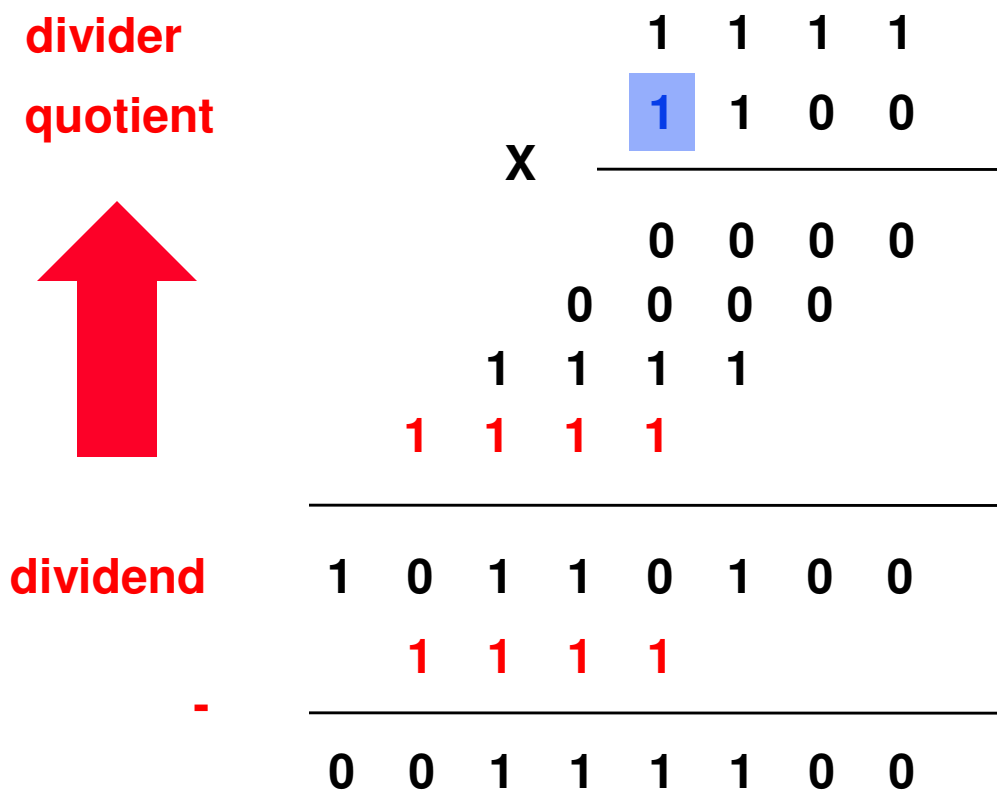
How does division work?

- We subtract the shifter divider from the dividend ..



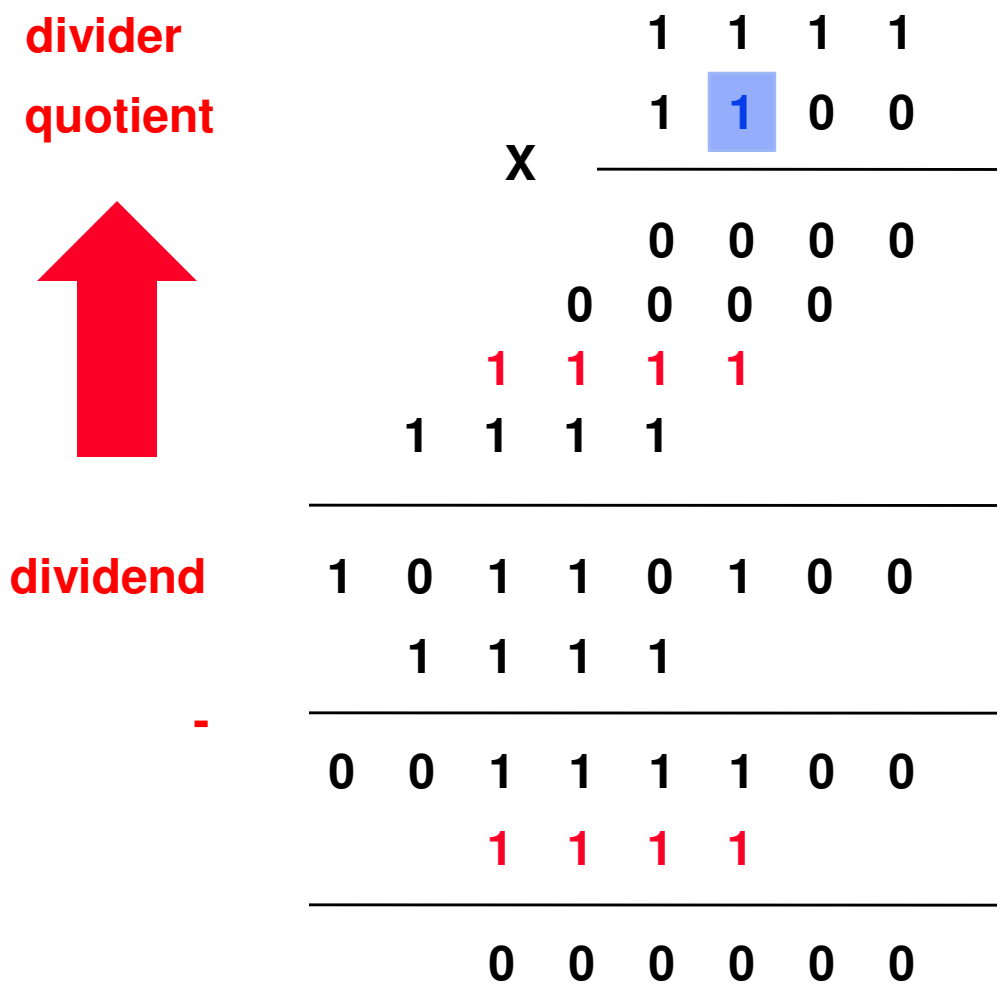
How does division work?

- We subtract the shifter divider from the dividend ..



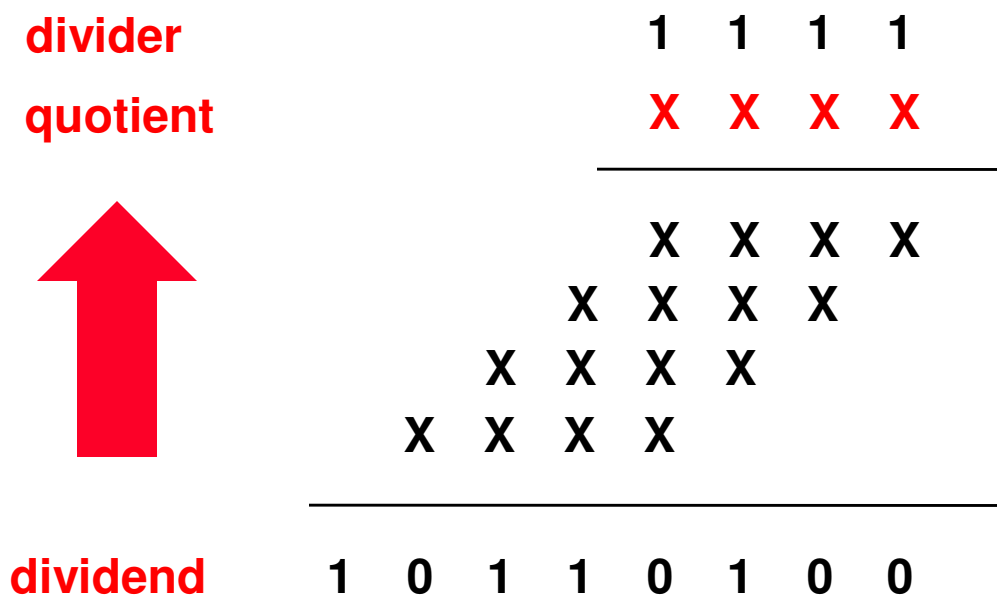
How does division work?

- We subtract the shifter divider from the dividend ..



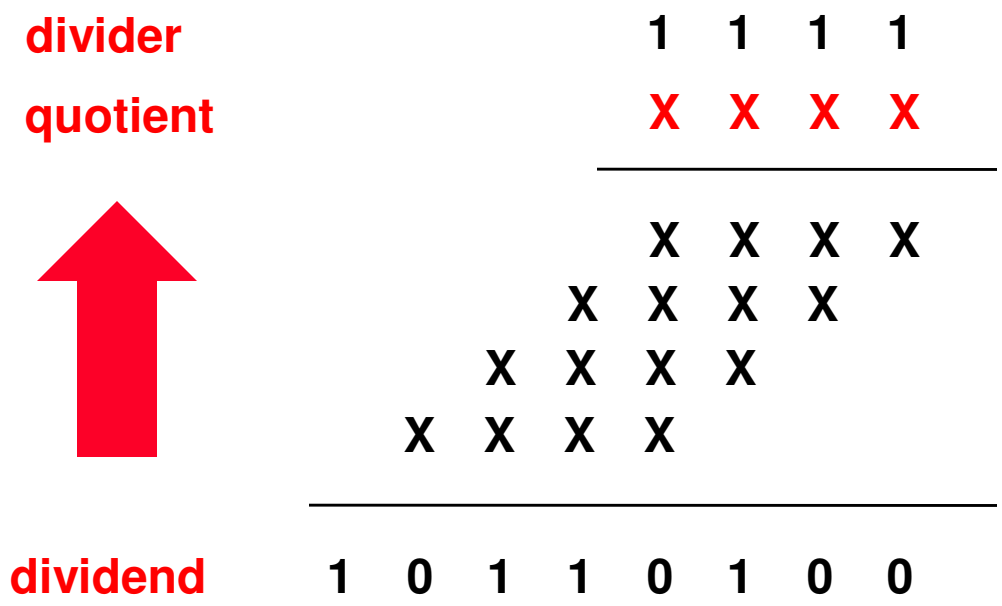
How does division work?

- The problem is: in a real division, we don't know the quotient ..



How does division work?

- Therefore, we make a tentative subtraction, and look each time if we are left with a positive number



How does division work?

- Example: Start at the MSB position

divider					**1**	**1**	**1**	**1**	
quotient				**0**	**X**	**X**	**X**	**X**	
dividend	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	
	1	**1**	**1**	**1**					
									subtract
..111	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	

This is an 'overflow' (when considering this as an unsigned number).

This is a negative number (when considered as 2's complement).

This means that the q-bit for the MSB position must be 0.

How does division work?

- Example: let's try the MSB-1 position

divider				1	1	1	1	
quotient			0	1	X	X	X	
dividend	1	0	1	1	0	1	0	0
		1	1	1	1			
								subtract
	0	0	1	1	1	1	0	0

This is not an 'overflow':
the result is < then the dividend.

This means that the q-bit for the MSB-1 position must be 1.

How does division work?

- Example: let's try the MSB-2 position

divider				1	1	1	1	
quotient			0	1	1	X	X	
dividend	1	0	1	1	0	1	0	0
		1	1	1	1			
								subtract
	0	0	1	1	1	1	0	0
			1	1	1	1		
								subtract
	0	0	0	0	0	0	0	0

This is 0 - we are done.

This means that the q-bit for the MSB-2 position must be 1, all other positions must be 0.

How does division work?

- Example: let's try the MSB-2 position

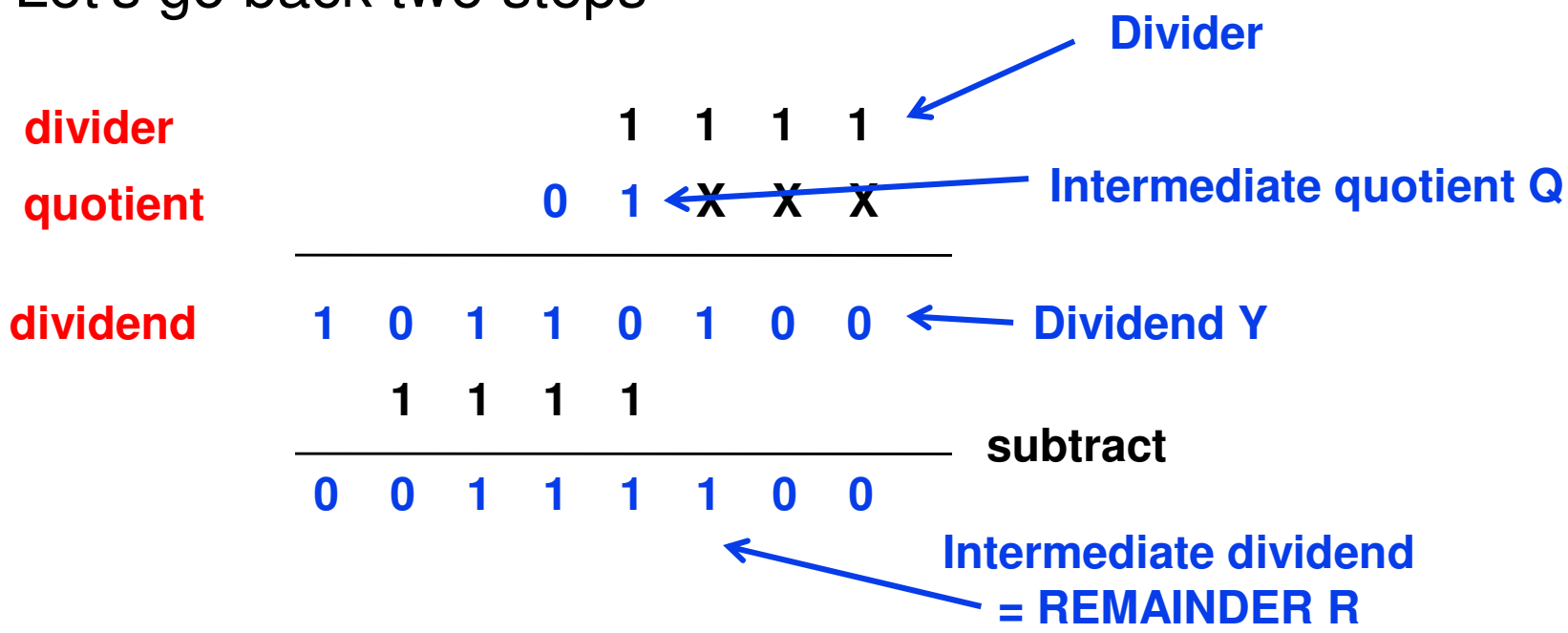
divider				1	1	1	1	
quotient			0	1	1	0	0	
dividend	1	0	1	1	0	1	0	0
		1	1	1	1			
								subtract
	0	0	1	1	1	1	0	0
			1	1	1	1		
								subtract
	0	0	0	0	0	0	0	0

This is 0 - we are done.

This means that the q-bit for the MSB-2 position must be 1, all other positions must be 0.

How does division work?

- Let's go back two steps



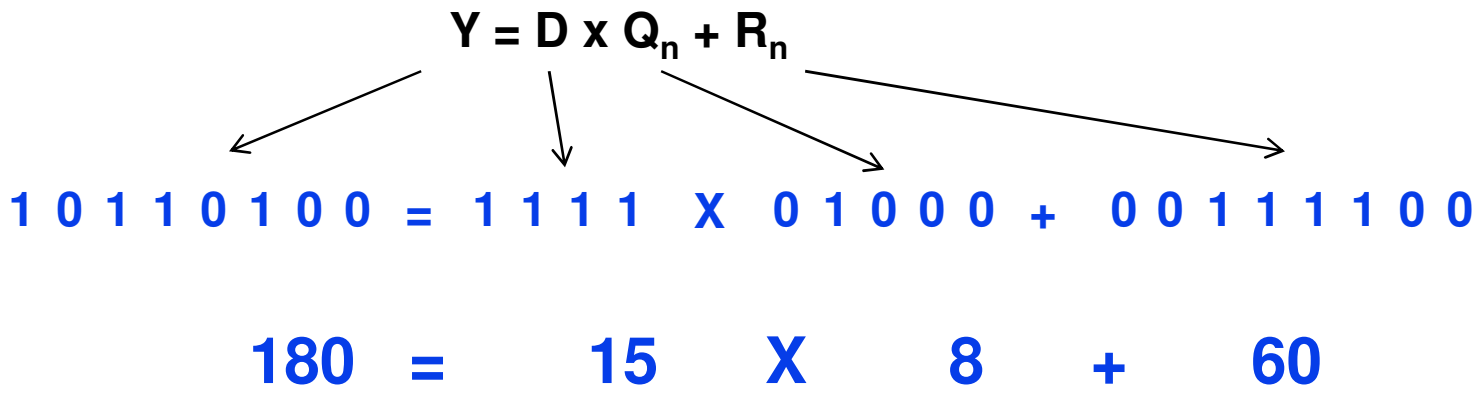
There is a well defined relation between the dividend Y, divider D, remainder R and the Quotient Q

$$Y = D \times Q_n + R_n$$

Quotient at step n **Remainder at step n**

Dividend/Divider, Quotient, Remainder

- Indeed



Dividend/Divider, Quotient, Remainder

- For an n-bit Dividend Y, an n-bit Divider D, an n-bit Quotient Q, and a Remainder R we can write:

$$2^n \times Y = D \times Q + R$$

The lsb weight of Q and R equals 1

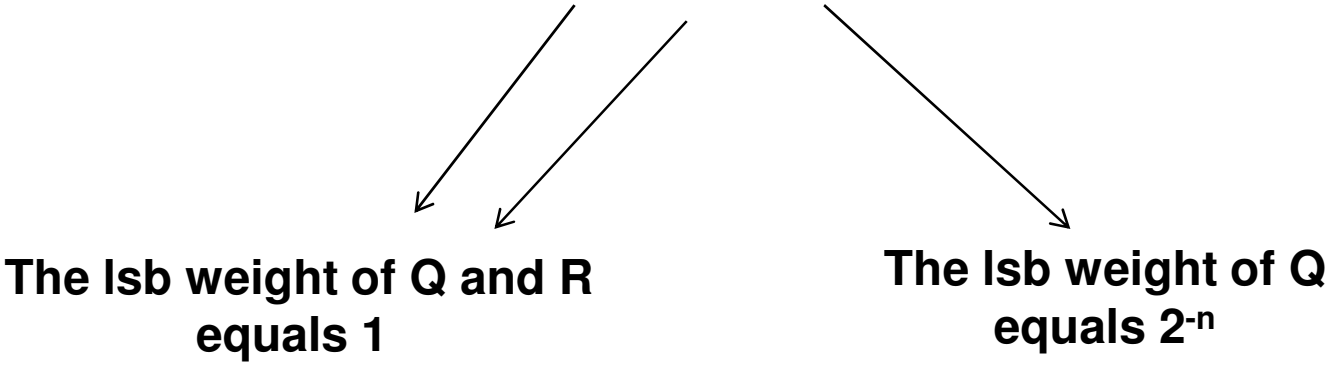
The lsb weight of Q equals 2^{-n}

i.e. the Q 'absorbs' the 2^n and the lsb weight of Q needs to decrease to keep the expression balanced

Dividend/Divider, Quotient, Remainder

- For an n-bit Dividend Y, an n-bit Divider D, an n-bit Quotient Q, and a Remainder R we can write:

$$2^n \times Y = D \times Q + R$$



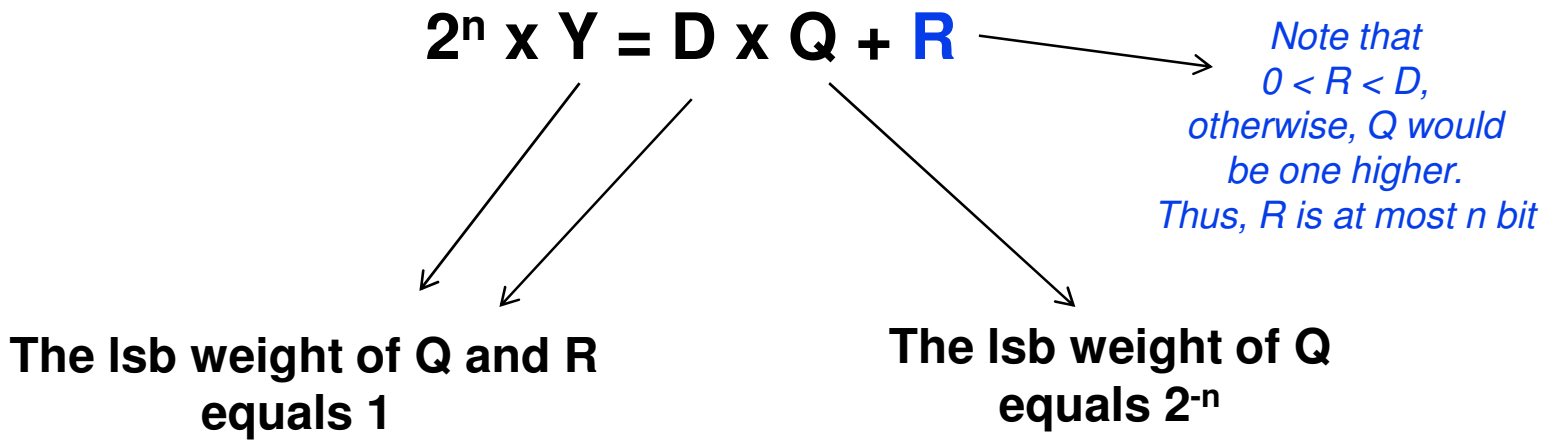
Example: Find 8 / 15 on 8 bits of precision for Q

Y = 8, D = 15
8 / 15 = 0.5333333...

Q = int (0.533333... x 256) = 136
R = (8 x 256 - 136 x 15) = 8

Dividend/Divider, Quotient, Remainder

- For an n-bit Dividend Y, an n-bit Divider D, an n-bit Quotient Q, and a Remainder R we can write:



Example: Find 8 / 15 on 8 bits of precision for Q

Y = 8, D = 15
8 / 15 = 0.5333333...

Q = int (0.533333... x 256) = 136
R = (8 x 256 - 136 x 15) = 8

Let's use this relationship for a 1-bit divider

- Assume that we are looking for a 1-bit quotient
- The 1-bit quotient q, remainder R, dividend Y and divider D are related as follows:

$$2 \times Y = D \times q + R$$

n bit *n bit* *1 bit* *n bit*

q = {0, 1}, corresponding to a real value of {0, 0.5}

1-bit divider: How do we choose q ?

- Since we don't know q nor R, we will just assume q is 1, and make a trial subtraction

$$2 \times Y - D = R$$

- If this relationship is correct, the subtraction should yield a positive result (since we want R to be positive).
  If we would work with unsigned numbers, the subtraction should not cause overflow.

2 x Y - D must be > 0

or

2 x Y - D must not overflow

1-bit divider

- After trial subtraction, if the relationship is true, then:

$$q = 1$$
$$R = 2 \times Y - D$$

- After trial subtraction, if this relationship is not true, then:

$$q = 0$$
$$R = 2 \times Y$$

Example: a 1-bit division of 4 (Y) by 6 (D):

$2 \times 4 - 6 \times 1 = 2$

This is > 0, so q = 1 and R = 2

1-bit divider

- After trial subtraction, if the relationship is true, then:

q = 1
R = 2 x Y - D

- After trial subtraction, if this relationship is not true, then:

q = 0
R = 2 x Y

Example 2: a 1-bit division of 2 (Y) by 6 (D):

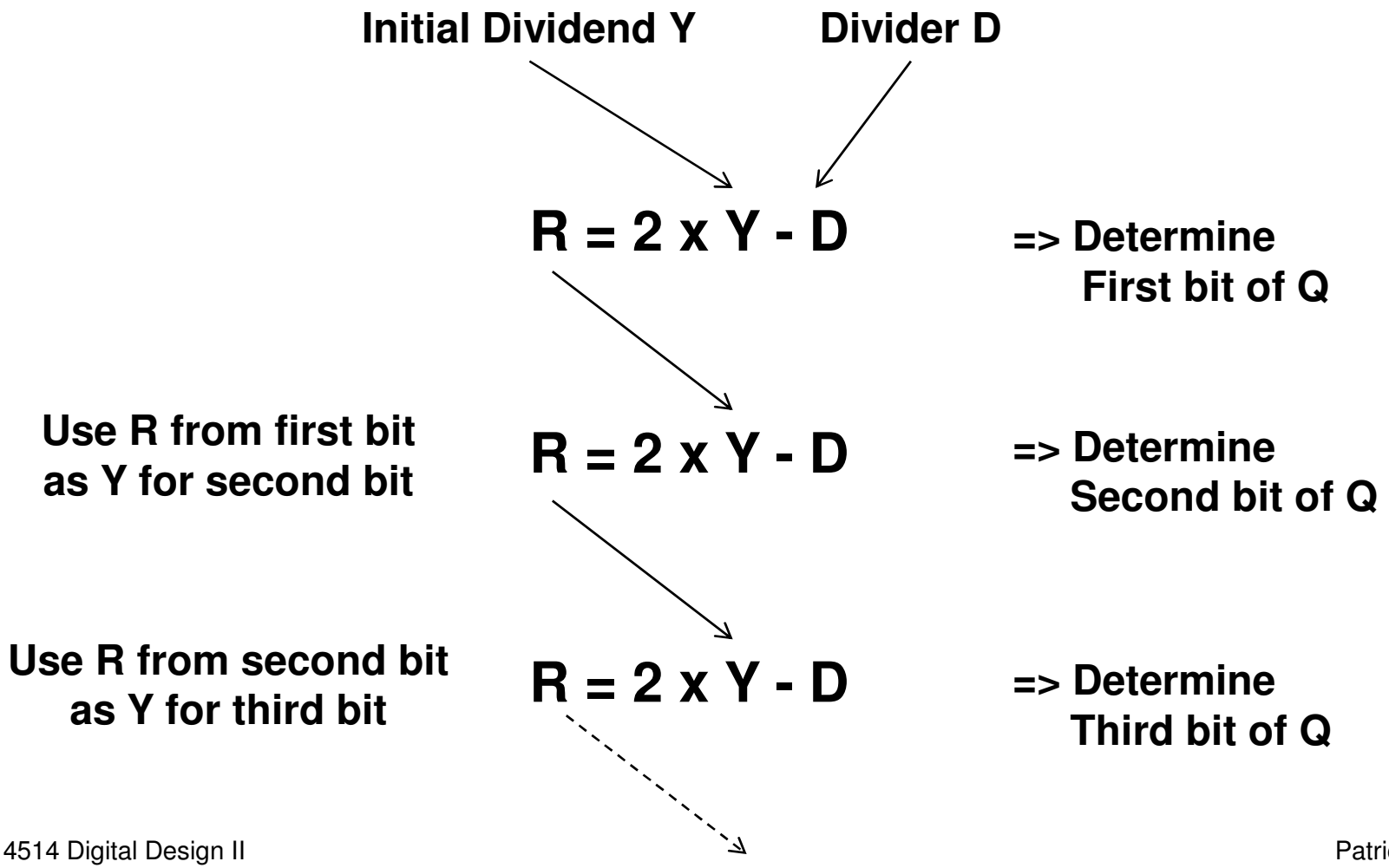
2 x 2 - 6 x 1 = -2, this is < 0

In unsigned arithmetic, we would find 4 - 6 = overflow

Thus q = 0 and R = 4

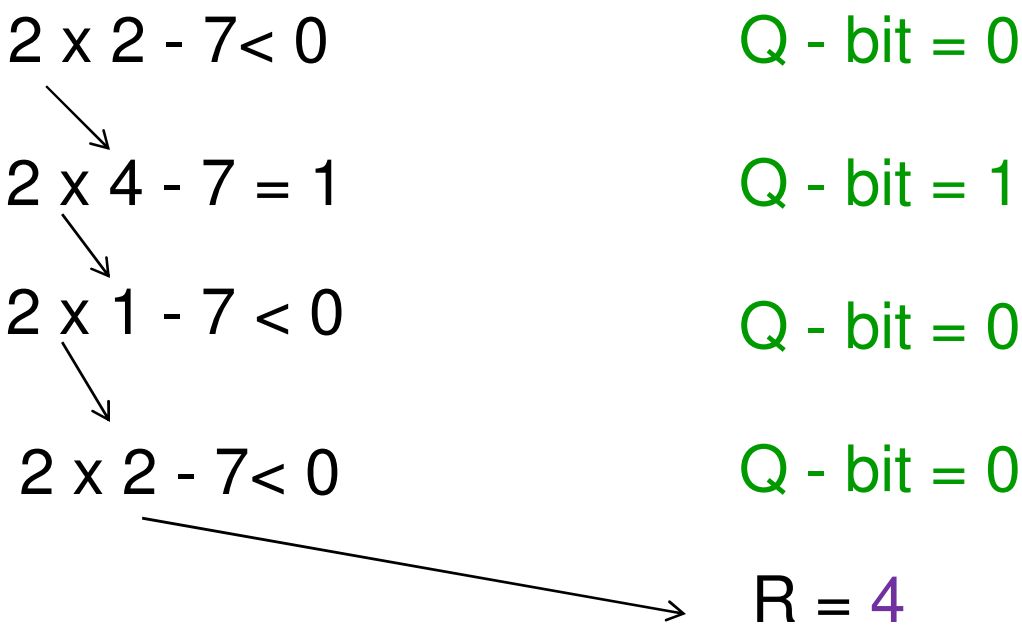
Extend to an n-bit divider

- Use the 1-bit divider in a ***digit-recurrence*** **formula**



Example 2/7 on 4 bit precision ..

- Y = 2, D = 7



So

$$2^4 \times \underset{Y}{2} = \underset{D}{7} \times \underset{Q}{4} + \underset{R}{4}$$

$$32 = 28 + 4$$

Example 2/7 on 4 bit precision ..

- Y = 2, D = 7

So

$$2^4 \times \underset{Y}{2} = \underset{D}{7} \times \underset{Q}{4} + \underset{R}{4}$$

The real Q (unscaled) would be: $4 / 2^4$

Indeed, 2/7 = 0.2857 ~ 0.25

Example 23/45 on 4 bit precision ..

- Y = 23, D = 45

2 x 23 - 45 = 1	Q - bit = 1
2 x 1 - 45 < 0	Q - bit = 0
2 x 2 - 45 < 0	Q - bit = 0
2 x 4 - 45 < 0	Q - bit = 0
	R = 8

So

$$2^4 \times 23 = 45 \times 8 + 8$$

$$368 = 45 \times 8 + 8$$

Example 23/45 on 4 bit precision ..

- Y = 23, D = 45

So

$$2^4 \times 23 = 45 \times 8 + 8$$

$$368 = 45 \times 8 + 8$$

The real Q (unscaled) would be: $8 / 2^4$

Indeed, 23/45 ~ 0.5

Pseudocode for a *restoring* divider

- This pseudocode assumes sequential execution

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R

R =  X
for i in 1 .. p loop
  tmp = 2 * R - Y;
  if (tmp < 0)
    q(i) = 0;
    R = tmp + Y;
  else
    q(i) = 1;
    R = tmp;
```

- Each iteration does 1 bit
  q(i) are each 1 bit
- The '< 0' condition determines if a q-bit is 0 or 1
- The name 'restoring' divider comes from the fact that the loop 'restores' the remainder after Y has been subtracted when it shouldn't.

Pseudocode for a *restoring* divider

- This pseudocode assumes sequential execution

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R

R =  X
for i in 1 .. p loop
  tmp = 2 * R - Y;
  if (tmp < 0)
    q(i) = 0;
    R = tmp + Y;
  else
    q(i) = 1;
    R = tmp;
```

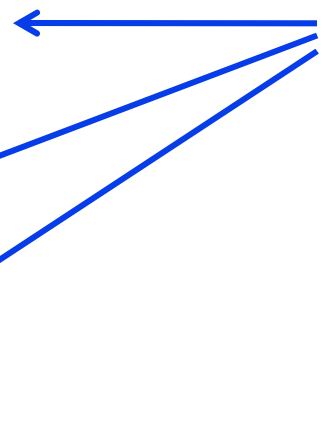
trial subtraction

restore subtraction, since remainder becomes < 0

Pseudocode for a *restoring* divider

- This pseudocode assumes sequential execution

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R

R =  X
for i in 1 .. p loop
  tmp = 2 * R - Y;
  if (tmp < 0)
    q(i) = 0;
    R = tmp + Y;
  else
    q(i) = 1;
    R = tmp;
```



As a further optimization, we will rewrite the code to overlap the execution of this expression with the execution of the if-then-else statement

This will result in the NON-RESTORING DIVIDER

Pseudocode for a *non-restoring* divider

- This pseudocode assumes sequential execution

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R

R = 2 * X - Y;
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;

if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

Does already the first iteration

Restore, and prepare for the next iteration
2 * (R + X) - X

Continue to the next iteration

Make sure remainder is positive

Pseudocode for a *non-restoring* divider

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R

R = 2 * X - Y;
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;

if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

- This code is 90% equivalent to the project-3 assignment
- The algorithm used for project 3 had extra features:
  - No initial expression before the loop
  - Support for signed numbers X, Y
- For now, we stick to the algorithm shown on the left

How to map this into Verilog ?

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R

R = 2 * X - Y;
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;

if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

- Decide on Module Interface
- Decide on Schedule
  (how to map into clock cycles)
- Choose a precision

How to map this into Verilog ?

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R

R = 2 * X - Y;
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;

if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

- Decide on Module Interface
  Inputs: X, Y, start
  Outputs: Z, R, done
- Decide on Schedule
  (how to map into clock cycles)
  One iteration of for-loop per clock cycle
- Choose a precision
  X, Y, R: 16-bit
  Z: 8-bit

How to map this into Verilog ?

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R

R = 2 * X - Y;
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;

if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

- Start by choosing registers

  What registers do you need?
  (Assume inputs are stable)

How to map this into Verilog ?

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R


R = 2 * X - Y;
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;

if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

- Start by choosing registers

  What registers do you need?
  (Assume inputs are stable)

  R is a 16-bit register
  Q is an 8-bit register

  + additional control registers (counters and flags)

Algorithm has three phases

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R
```

Initialization (start=1)

```
R = 2 * X - Y;
```

Processing in clock cycle 1 .. 7

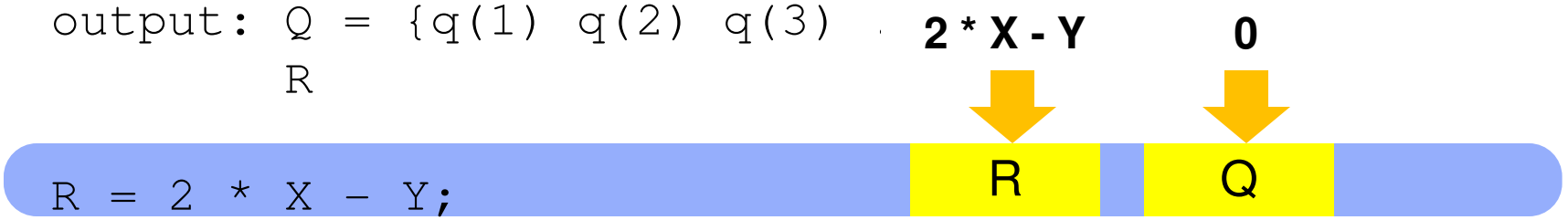
```
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;
```

Wrap-up clock cycle 8

```
if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

Assignments on reg in initialization

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) ...}
        R
```



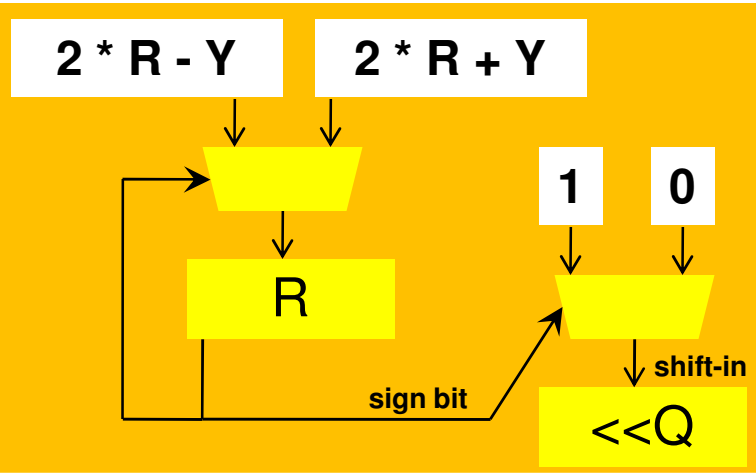

```
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;
```

```
if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

Assignments on reg during processing

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R
```

```
R = 2 * X - Y;
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;
```



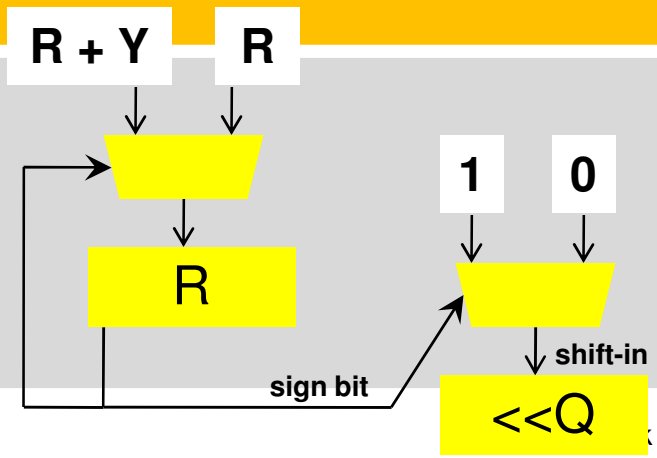

```
if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

Assignments on reg at wrap-up

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R
```

```
R = 2 * X - Y;
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;
```

```
if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```



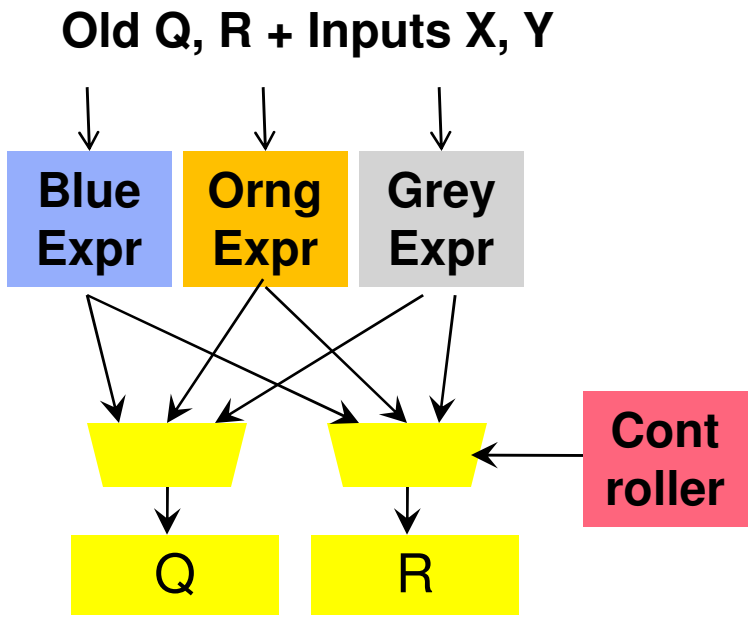
Birds-eye view on module structure

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R
```

```
R = 2 * X - Y;
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;
```

```
if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

Datapath Design:



Verilog for R register assignments

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R
```

start

```
R = 2 * X - Y;
```

```
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;
```

last

```
if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

Assignments on R Register:

```
always @(posedge clk)
  if (start)
    R <= 2 * X - Y;
  else if (~last & R[15])
    R <= (R << 1) + Y;
  else if (~last)
    R <= (R << 1) - Y;
  else if (last & R[15])
    R <= R + Y;
  else
    R <= R;
```

Verilog for Q Register Assignments

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R
```

start

```
R = 2 * X - Y;
```

```
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;
```

last

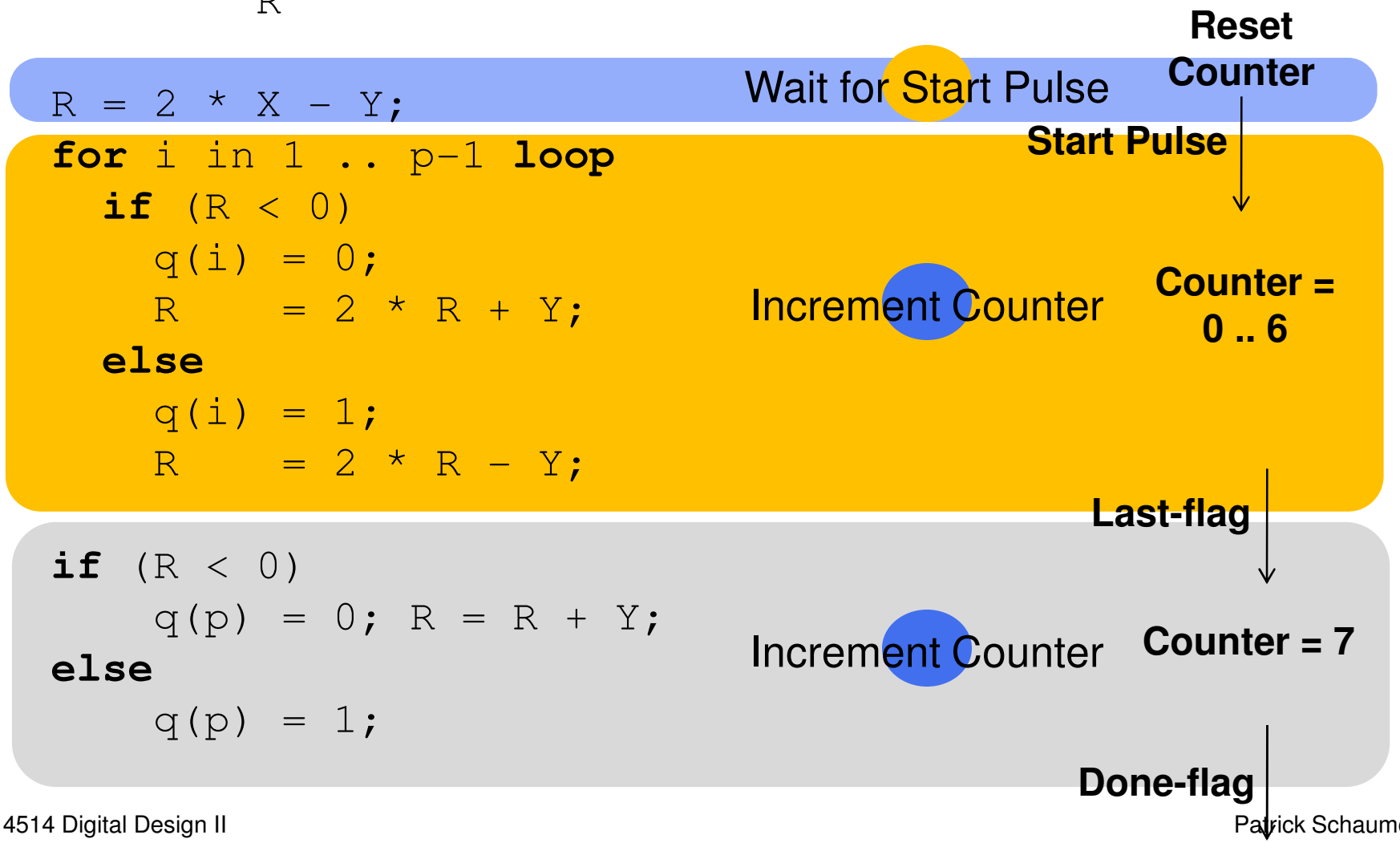
```
if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

Assignments on Q Register:

```
always @(posedge clk)
  if (start)
    Q <= 0;
  else if (~last)
    Q <= R[15] ?
         {Q[6:0], 1'b0} :
        {Q[6:0], 1'b1};
  else if (last & R[15])
    Q <= {Q[6:0], 1'b0};
  else if (last & ~R[15])
    Q <= {Q[6:0], 1'b1};
  else
    Q <= Q;
```

Controller

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R
```



Verilog for controller

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R
```

```
R = 2 * X - Y;
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;
```

```
if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

Controller:

```
reg done;
reg last;

always @(posedge clk) begin
  done <= 0;
  last <= 0;
  if (start)
    cnt <= 0;
  else
    cnt <= cnt + 1;
  if (cnt == 6'd6)
    last <= 1;
  else if (cnt == 6'd7)
    done <= 1;
 end
```

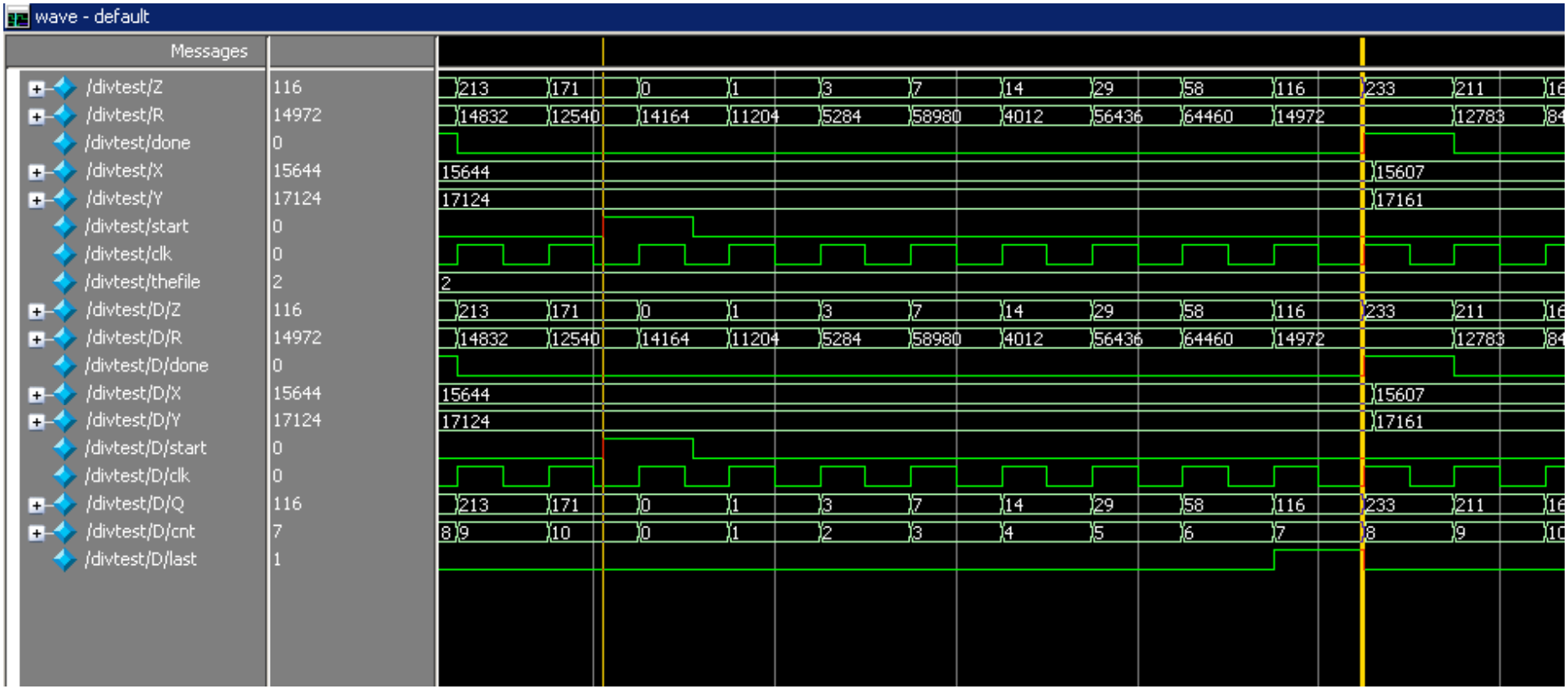
Entire Design

```
module divider(Z, R, done, X, Y, start, clk);
  output [7:0] Z;
  output [15:0] R;
  output done;
  input  [15:0] X;
  input  [15:0] Y;
  input  start;
  input  clk;

  reg [7:0] Q;
  reg [15:0] R;

  reg [5:0] cnt;
  reg done;
  reg last;

  always @(posedge clk) begin
    done <= 0;
    last <= 0;
    if (start)
      cnt <= 0;
    else
      cnt <= cnt + 1;
    if (cnt == 6'd6)
      last <= 1;
    else if (cnt == 6'd7)
      done <= 1;
  end
```

```
always @(posedge clk)
   if (start)
      R <= 2 * X - Y;
   else if (~last & R[15])
      R <= (R << 1) + Y;
   else if (~last)
      R <= (R << 1) - Y;
   else if (last & R[15])
      R <= R + Y;
   else
      R <= R;

always @(posedge clk)
   if (start)
      Q <= 0;
   else if (~last)
      Q <= R[15] ? {Q[6:0], 1'b0} :
                   {Q[6:0], 1'b1};
   else if (last & R[15])
      Q <= {Q[6:0], 1'b0};
    else if (last & ~R[15])
      Q <= {Q[6:0], 1'b1};
    else
      Q <= Q;

  assign Z = Q;

endmodule
```

Simulation Output

start **stop**

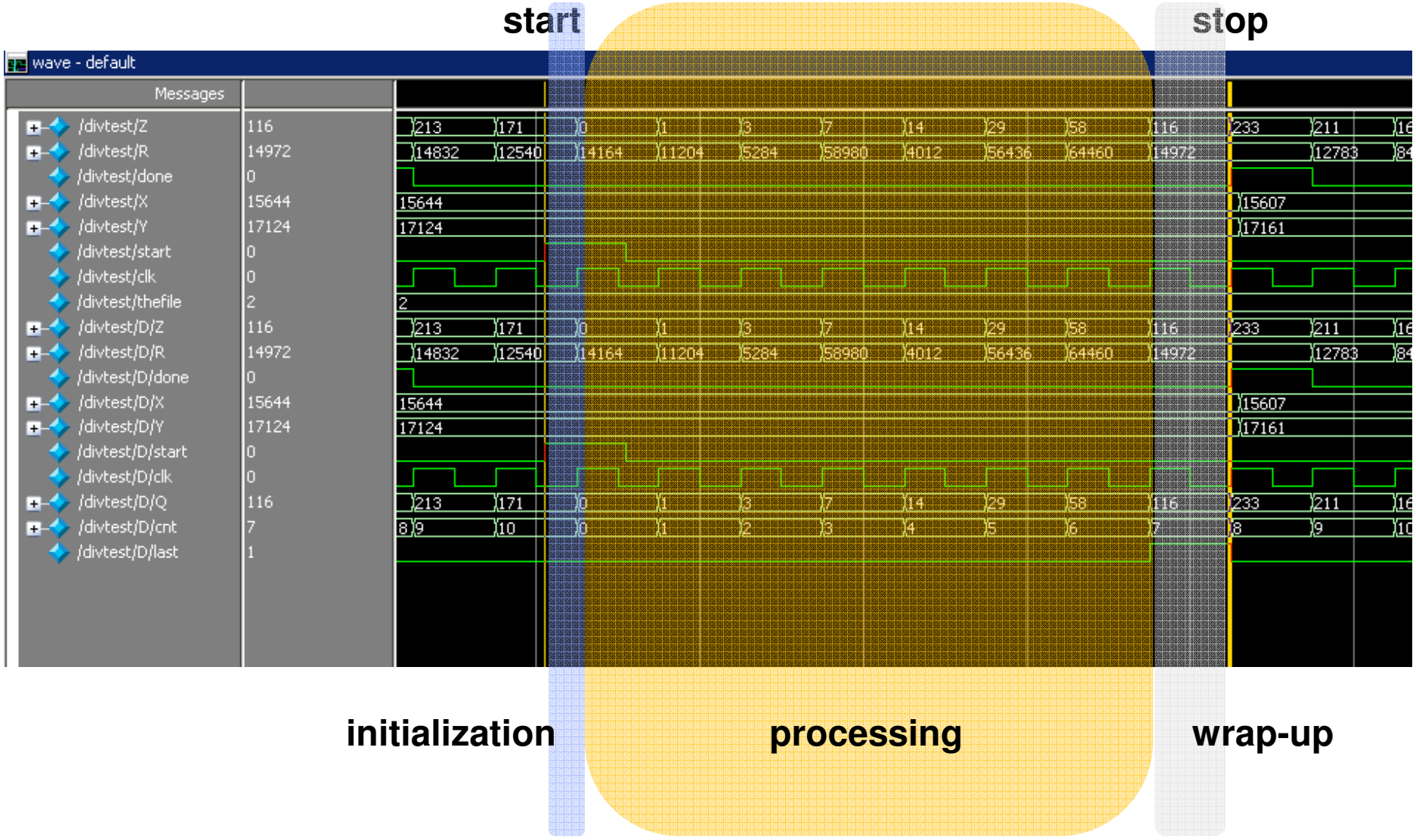


X = 15644
Y = 17124

Z = 233
R = 14972

check: 15644 * 256 - 233 * 17124 = 14972

Simulation Output



Synthesis Output

```
=========================================================================
*                            HDL Synthesis                              *
=========================================================================

Performing bidirectional port resolution...

Synthesizing Unit <divider>.
    Related source file is "div2.v".
WARNING:Xst:647 - Input <X<15>> is never used.
WARNING:Xst - Property "use_dsp48" is not applicable for this technology.
    Found 16-bit register for signal <R>.
    Found 1-bit register for signal <done>.
    Found 6-bit up counter for signal <cnt>.
    Found 1-bit register for signal <last>.
    Found 8-bit register for signal <Q>.
    Found 16-bit addsub for signal <R$mux0000>.
    Summary:
        inferred   1 Counter(s).
        inferred  26 D-type flip-flop(s).
        inferred   1 Adder/Subtractor(s).
Unit <divider> synthesized.

INFO:Xst:1767 - HDL ADVISOR - Resource sharing has identified that some
arithmetic operations in this design can share the same physical resources
for reduced device utilization. For improved clock frequency you may try to
disable resource sharing.
```

Expected

Indeed

Resource Sharing

```
input: X, Y, evaluate X/Y
output: Q = {q(1) q(2) q(3) .. q(p)}
        R
```

```
R = 2 * X - Y;
for i in 1 .. p-1 loop
  if (R < 0)
    q(i) = 0;
    R    = 2 * R + Y;
  else
    q(i) = 1;
    R    = 2 * R - Y;
```

```
if (R < 0)
    q(p) = 0; R = R + Y;
else
    q(p) = 1;
```

4 Expressions, 1 Adder/Sub

```
always @(posedge clk)
  if (start)
    R <= 2 * X - Y;
  else if (~last & R[15])
    R <= (R << 1) + Y;
  else if (~last)
    R <= (R << 1) - Y;
  else if (last & R[15])
    R <= R + Y;
  else
    R <= R;
```

Summary

- Division algorithms/architectures
  - Division as an 'inverted' multiplication
  - Division one digit at-a-time: digit-recurrence
  - The restoring divider algorithm
  - The non-restoring divider algorithm
  - Designing an architecture for a non-restoring divider