

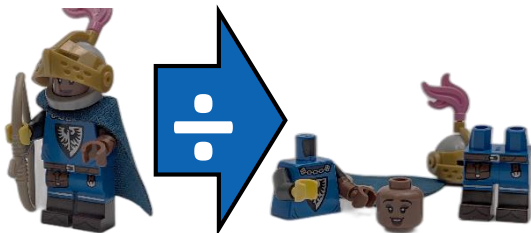


# The Forgotten Horseman

- Digital Implementation of Arithmetic  
Division and Resources to Learn and Teach Its Complexities

Peter Jamieson, Nathan Martin

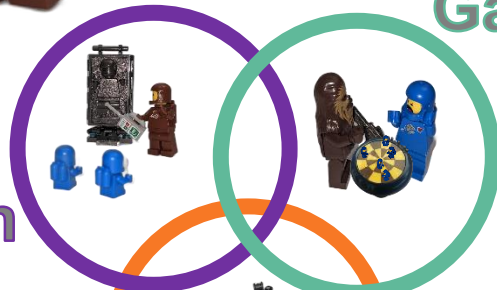
Miami University



# About Me



Education



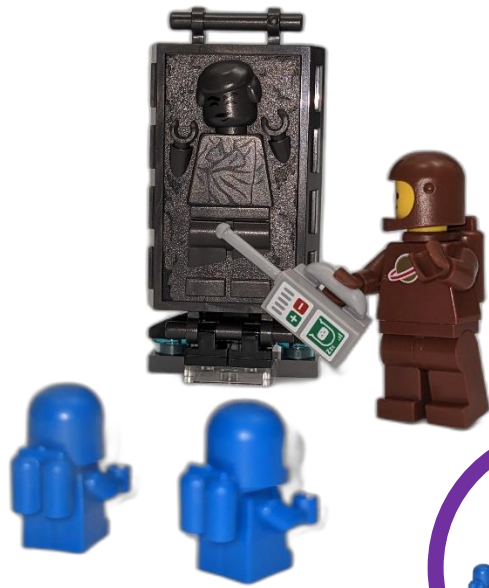
Games



Computer Engineering



# About Me



Education



Games



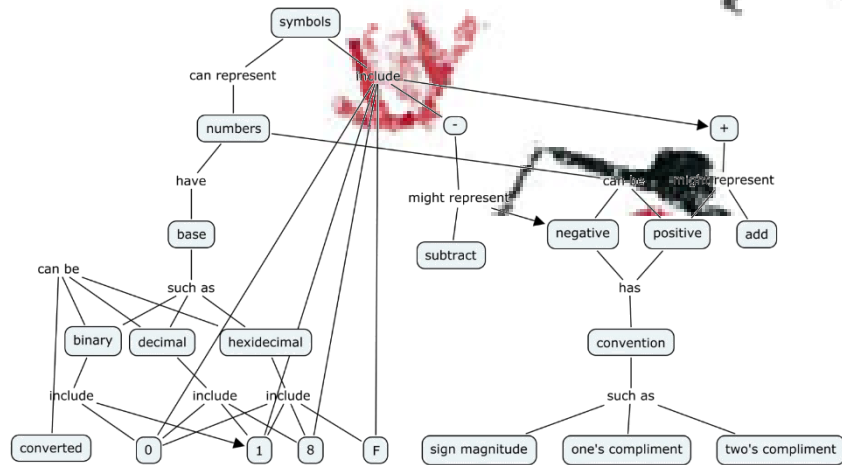
Computer Engineering





# Booth Multiplier

- Basic Adder
- Carry Ripple





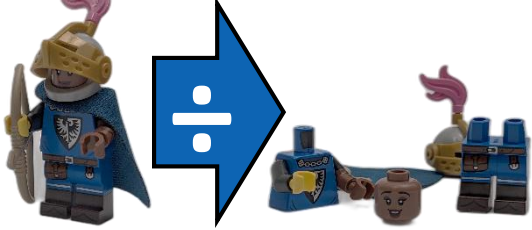
$$\frac{\text{Dividend } (D_{end})}{\text{Divisor } (D_{or})} = \text{Quotient } (Q) + \frac{\text{Remainder } (R)}{\text{Divisor } (D_{or})}$$



$$\frac{D_{end}}{D_{or}} = Q + \frac{R}{D_{or}}$$







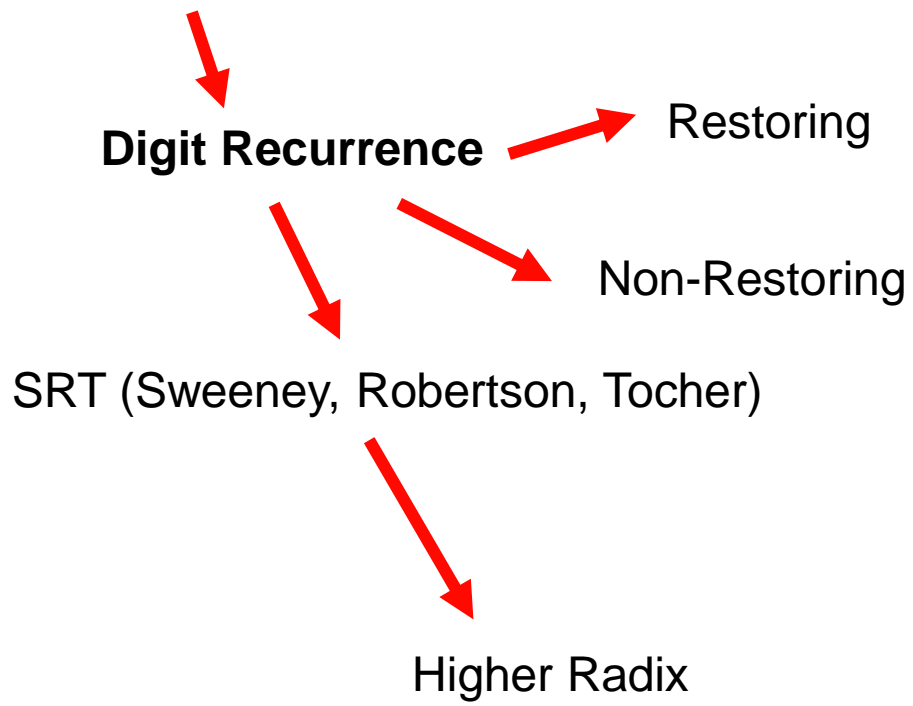
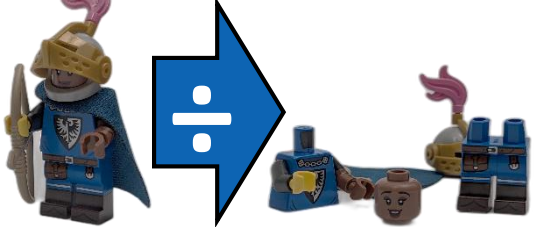
$$D_{or} = 1011_2 = 11_{10}$$

$$D_{end} = 10011001_2 = 153_{10}$$

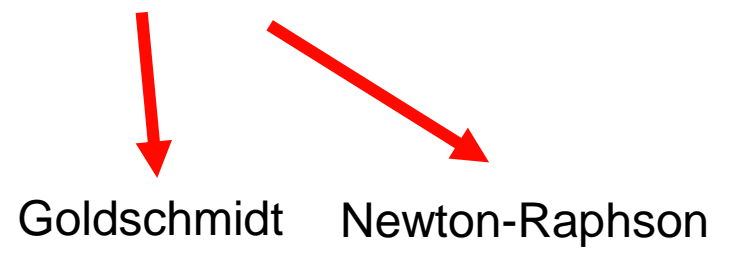
$$\begin{array}{r}
 1011 \overline{) 10011001} \\
 \underline{-1011} \phantom{0000} \\
 010000 \phantom{00} \rightarrow 1 \\
 \underline{-1011} \phantom{00} \\
 001010 \phantom{00} \rightarrow 1 \\
 \underline{-1011} \phantom{00} \\
 11111 \phantom{00} \rightarrow 0 \\
 \phantom{1}0010101 \\
 \underline{-1011} \\
 01010 \phantom{00} \rightarrow 1
 \end{array}$$

$$R = 01010_2 = 10_{10} \quad Q = 1101_2 = 13_{10}$$





## Functional Iteration





- 1) Bit-Width
- 2) Division Algorithm

Digit Recurrence

Functional Iteration

SRT

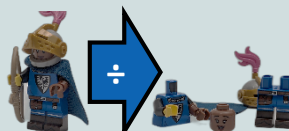
Non-Restoring

Goldschmidt

Radix-2

Radix-4

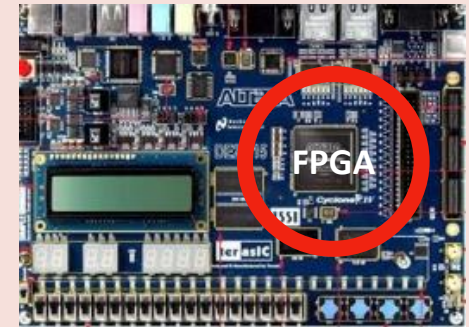
# Python Tool



```

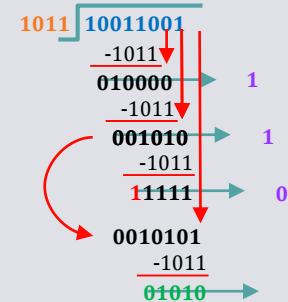
module Division (clk, reset, di, do, q, r);
input clk, reset, [bw:0]di, [bw:0]do;
output reg [bw:0]q, [bw:0]r;
always @(posedge clk or negedge reset)
begin
  if (reset == 1'b0)
    q <= BW'b0;
  else
    if (x == 1'b1)
      // calculations and iterations
      ....
  end
endmodule

```



$$D_{or} = 1011_2 = 11_{10}$$

$$D_{end} = 10011001_2 = 153_{10}$$



$$R = 01010_2 = 10_{10} \quad Q = 1101_2 = 13_{10}$$



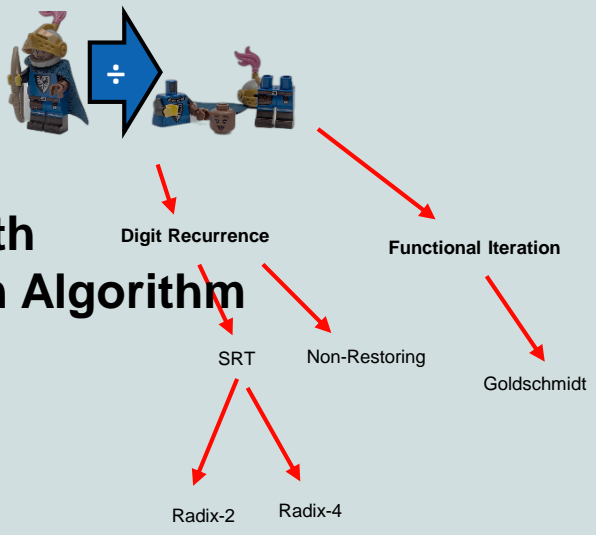


```

module Division (clk, reset, di, do, q, r);
input clk, reset, [bw:0]di, [bw:0]do;
output reg [bw:0]q, [bw:0]r;
always @(posedge clk or negedge reset)
begin
  if (reset == 1'b0)
    q <= BW'b0;
  else
    if (x == 1'b1)
      // calculations and iterations
      ....
  end
endmodule
  
```



- 1) Bit-Width
- 2) Division Algorithm



# Python Tool



$D_{or} =$   
 $D_{end} =$

$$\begin{array}{r}
 0010101 \\
 -1011 \\
 \hline
 01010 \rightarrow 1
 \end{array}$$

$R = 01010_2 = 10_{10}$        $Q = 1101_2 = 13_{10}$

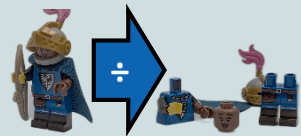
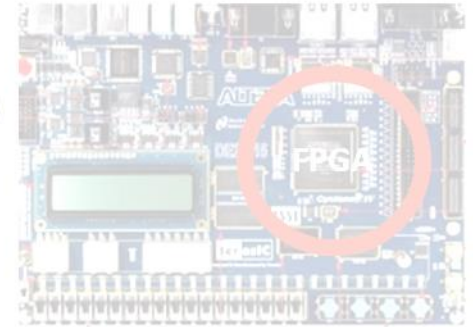




```

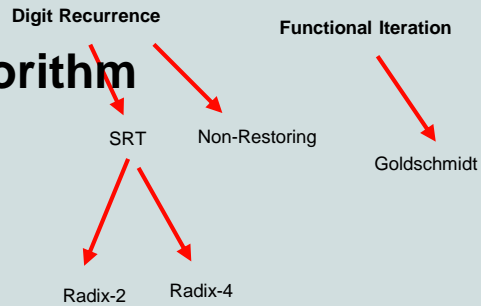
module Division (clk, reset, di, do, q, r);
input clk, reset, [bw:0]di, [bw:0]do;
output reg [bw:0]q, [bw:0]r;
always @(posedge clk or negedge reset)
begin
    if (reset == 1'b0)
        q <= BW'b0;
    else
        if (x == 1'b1)
            // calculations and iterations
            ...
end
endmodule

```



1) Bit-Width

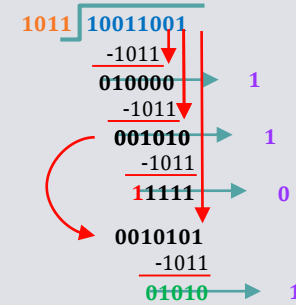
2) Division Algorithm



# Python Tool

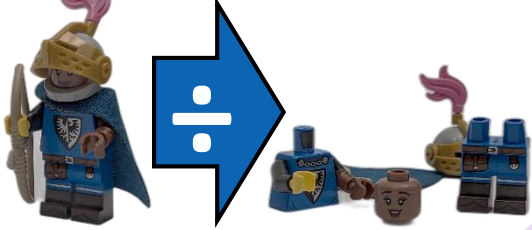
$$D_{or} = 1011_2 = 11_{10}$$

$$D_{end} = 10011001_2 = 153_{10}$$



$$R = 01010_2 = 10_{10} \quad Q = 1101_2 = 13_{10}$$





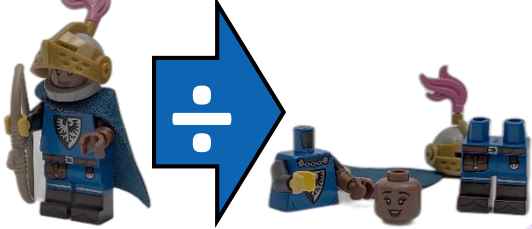
$$D_{or} = 1011_2 = 11_{10}$$

$$D_{end} = 10011001_2 = 153_{10}$$

$$\begin{array}{r}
 1011 \overline{) 10011001} \\
 \underline{-1011} \phantom{0000} \\
 010000 \phantom{00} \rightarrow 1 \\
 \underline{-1011} \phantom{00} \\
 001010 \phantom{00} \rightarrow 1 \\
 \underline{-1011} \phantom{00} \\
 11111 \phantom{00} \rightarrow 0 \\
 \phantom{1}0010101 \\
 \underline{-1011} \\
 01010 \phantom{00} \rightarrow 1
 \end{array}$$

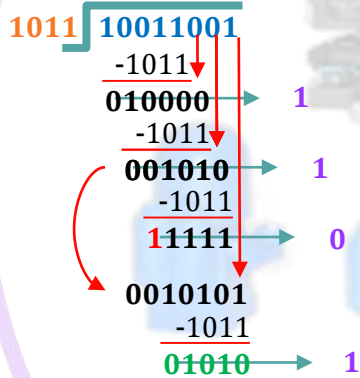
$$R = 01010_2 = 10_{10} \quad Q = 1101_2 = 13_{10}$$



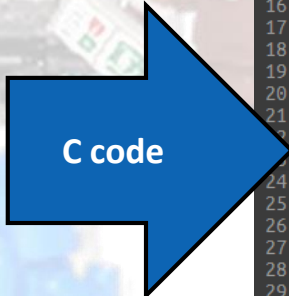


$$D_{or} = 1011_2 = 11_{10}$$

$$D_{end} = 10011001_2 = 153_{10}$$



$$R = 01010_2 = 10_{10} \quad Q = 1101_2 = 13_{10}$$

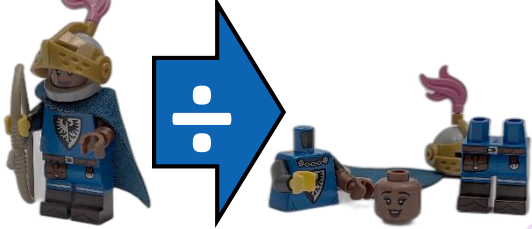


```

1 int div( unsigned int D_or, unsigned int D_end )
2 {
3     // q is a temporary n, sum is the quotient
4     unsigned int Q, R, E, temp_end;
5
6     E = 0; // bits of E pushed in from LSB
7     Q = 0;
8
9     // store the bits of D_end in a until aligned with D_or
10    while (D_end >= D_or)
11    {
12        bits++;
13        E = (D_end & 1) | (E << 1); // push
14        D_end = D_end >> 1; // align
15    }
16    // little fix so we don't need an if
17    bits--;
18    D_end = (E & 1) | (D_end << 1); // pop
19    E = E >> 1;
20
21    while (bits >= 0)
22    {
23        Q = Q << 1;
24        if (D_end >= D_or)
25        {
26            // positive case
27            D_end = D_end - D_or;
28            D_end = (E & 1) | (D_end << 1);
29            temp_end = D_end;
30            Q = Q | 1; // store Q bit
31        }
32        else
33        {
34            // negative case
35            D_end = (E & 1) | (temp_end << 1);
36            temp_end = D_end;
37        }
38
39        a = a >> 1; // pop
40        bits--;
41    }
42    // fix step for remainder
43    R = D_end >> 1;
44 }

```

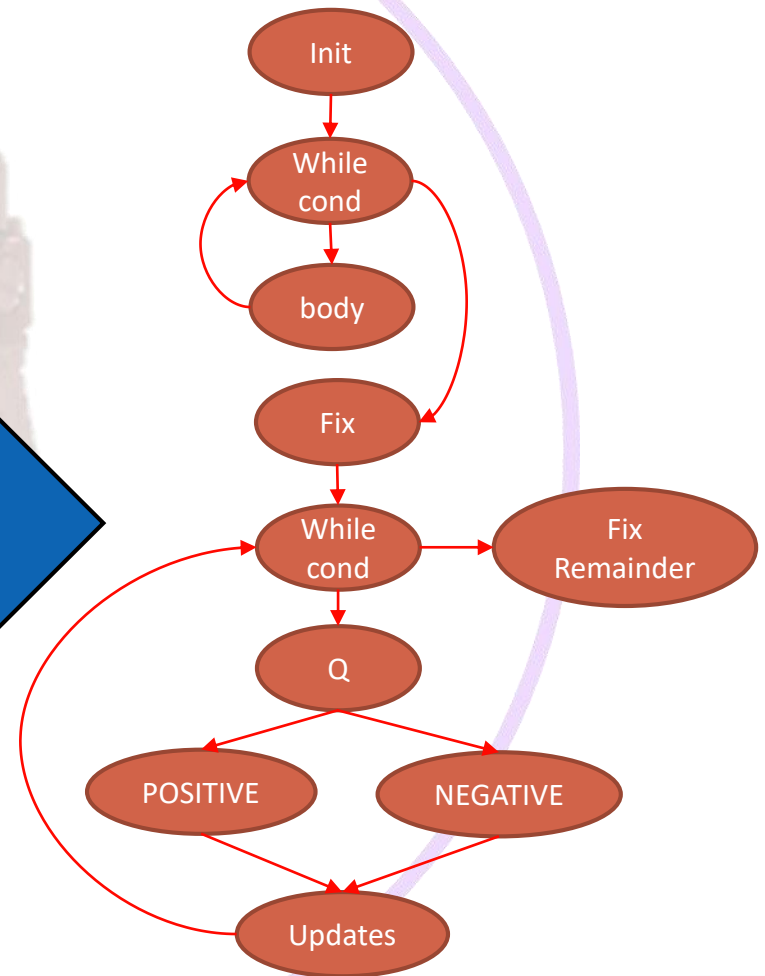
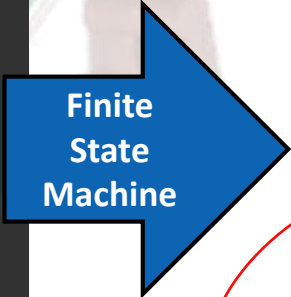


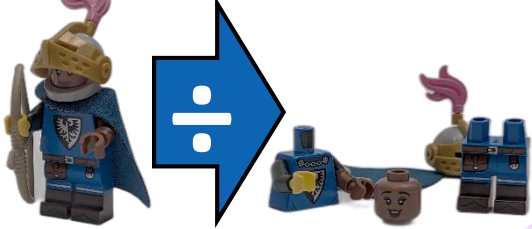


```

1 int div( unsigned int D_or, unsigned int D_end )
2 {
3     // q is a temporary n, sum is the quotient
4     unsigned int Q, R, E, temp_end;
5
6     E = 0; // bits of E pushed in from LSb
7     Q = 0;
8
9     // store the bits of D_end in a until aligned with D_or
10    while (D_end >= D_or)
11    {
12        bits++;
13        E = (D_end & 1) | (E << 1); // push
14        D_end = D_end >> 1; // align
15    }
16    // little fix so we don't need an if
17    bits --;
18    D_end = (E & 1) | (D_end << 1); // pop
19    E = E >> 1;
20
21    while (bits >= 0)
22    {
23        Q = Q << 1;
24        if (D_end >= D_or)
25        {
26            // positive case
27            D_end = D_end - D_or;
28            D_end = (E & 1) | (D_end << 1);
29            temp_end = D_end;
30            Q = Q | 1; // store Q bit
31        }
32        else
33        {
34            // negative case
35            D_end = (E & 1) | (temp_end << 1);
36            temp_end = D_end;
37        }
38
39        a = a >> 1; // pop
40        bits --;
41    }
42    // fix step for remainder
43    R = D_end >> 1;
44 }

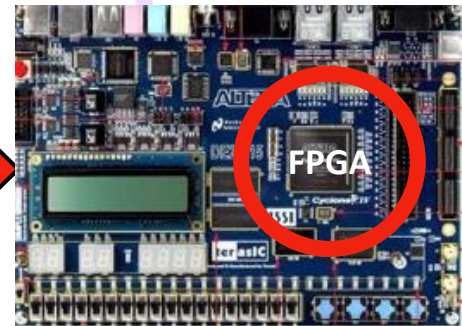
```



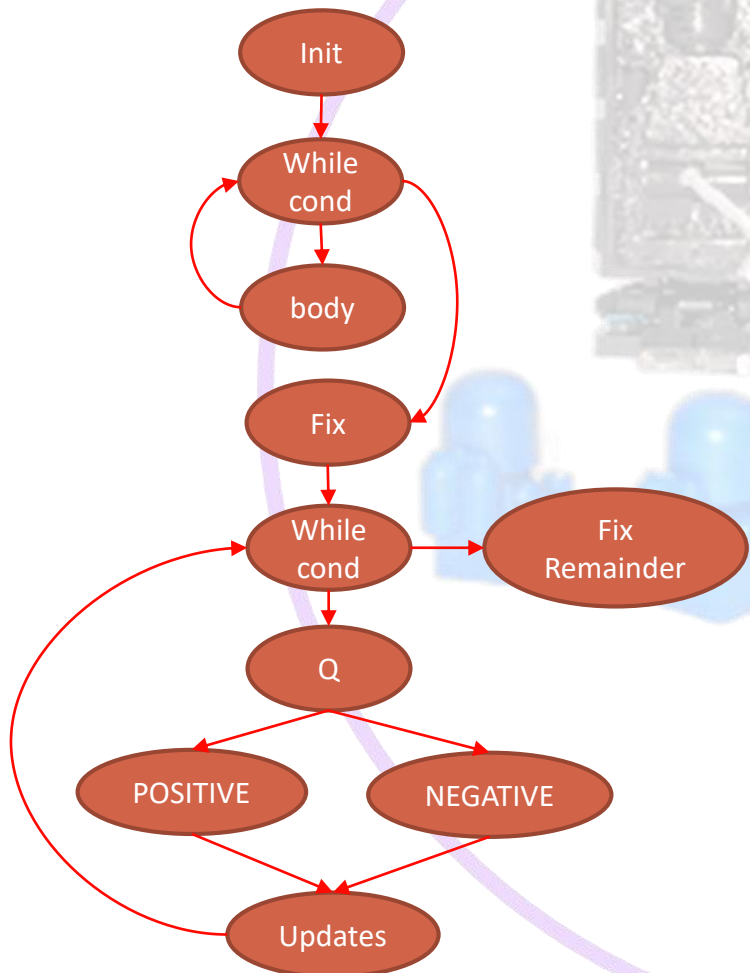
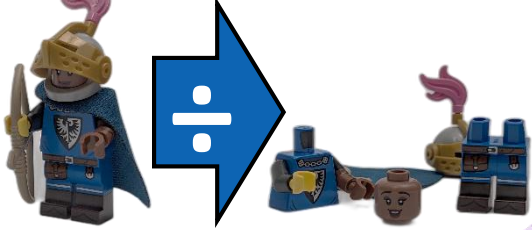


```
1 int div( unsigned int D_or, unsigned int D_end )
2 {
3     // q is a temporary n, sum is the quotient
4     unsigned int Q, R, E, temp_end;
5
6     E = 0; // bits of E pushed in from LSB
7     Q = 0;
8
9     // store the bits of D_end in a until aligned with D_or
10    while (D_end >= D_or)
11    {
12        bits++;
13        E = (D_end & 1) | (E << 1); // push
14        D_end = D_end >> 1; // align
15    }
16    // little fix so we don't need an if
17    bits --;
18    D_end = (E & 1) | (D_end << 1); // pop
19    E = E >> 1;
20
21    while (bits >= 0)
22    {
23        Q = Q << 1;
24        if (D_end >= D_or)
25        {
26            // positive case
27            D_end = D_end - D_or;
28            D_end = (E & 1) | (D_end << 1);
29            temp_end = D_end;
30            Q = Q | 1; // store Q bit
31        }
32        else
33        {
34            // negative case
35            D_end = (E & 1) | (temp_end << 1);
36            temp_end = D_end;
37        }
38
39        a = a >> 1; // pop
40        bits --;
41    }
42    // fix step for remainder
43    R = D_end >> 1;
44 }
```

OR High-Level Synthesis (HLS)



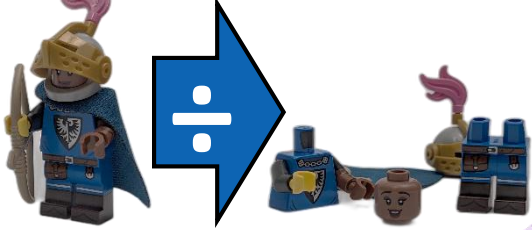




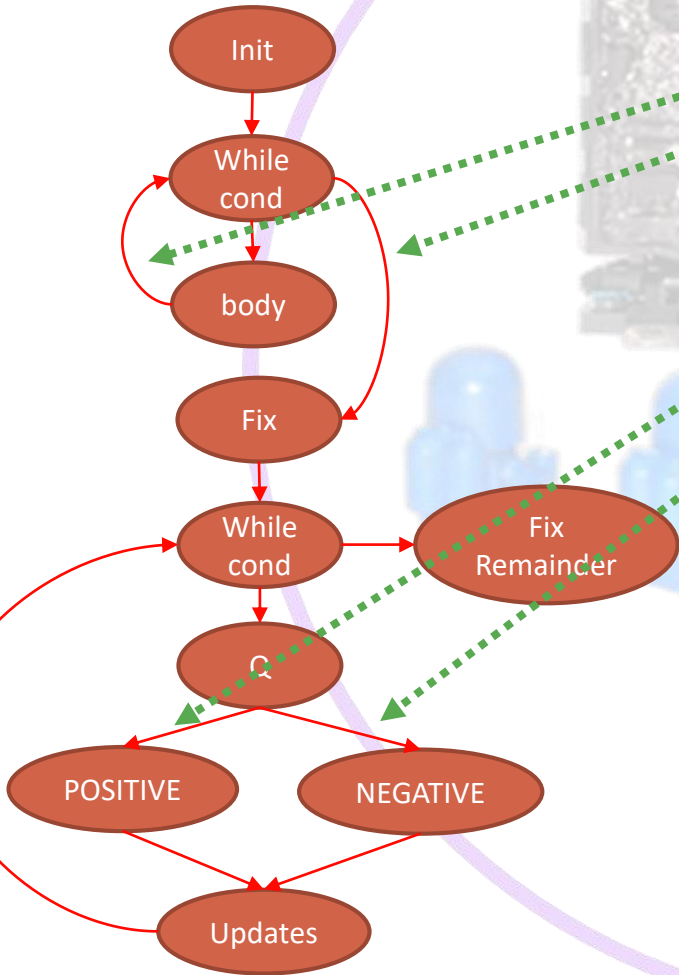
```
1 module div(  
2     input rst, clk,  
3     input [7:0]D_or,  
4     input [7:0]D_end,  
5     input start,  
6     output reg [7:0]Q,  
7     output reg [7:0]R,  
8     output reg done);  
9  
10    reg [7:0] E;  
11    reg [7:0] tD_end;  
12    reg [3:0] bits;  
13    reg [7:0] old_D_end;  
14  
15    reg [3:0] S;  
16    reg [3:0] NS;  
17    parameter INIT = 4'd0,  
18                WHILE_COND_1 = 4'd1,  
19                WHILE_BODY_1 = 4'd2,  
20                FIX = 4'd3,  
21                WHILE_COND_2 = 4'd4,  
22                WHILE_2_Q = 4'd5,  
23                WHILE_2_POS = 4'd6,  
24                WHILE_2_NEG = 4'd7,  
25                WHILE_2_UPDATE = 4'd8,  
26                FIX_REM = 4'd9,  
27                DONE = 4'd10,  
28                ERROR = 4'hF;  
29
```

```
30    always @(posedge clk or negedge rst)  
31        if (rst == 1'b0)  
32            S <= INIT;  
33        else  
34            S <= NS;  
35
```



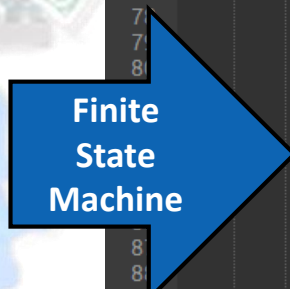
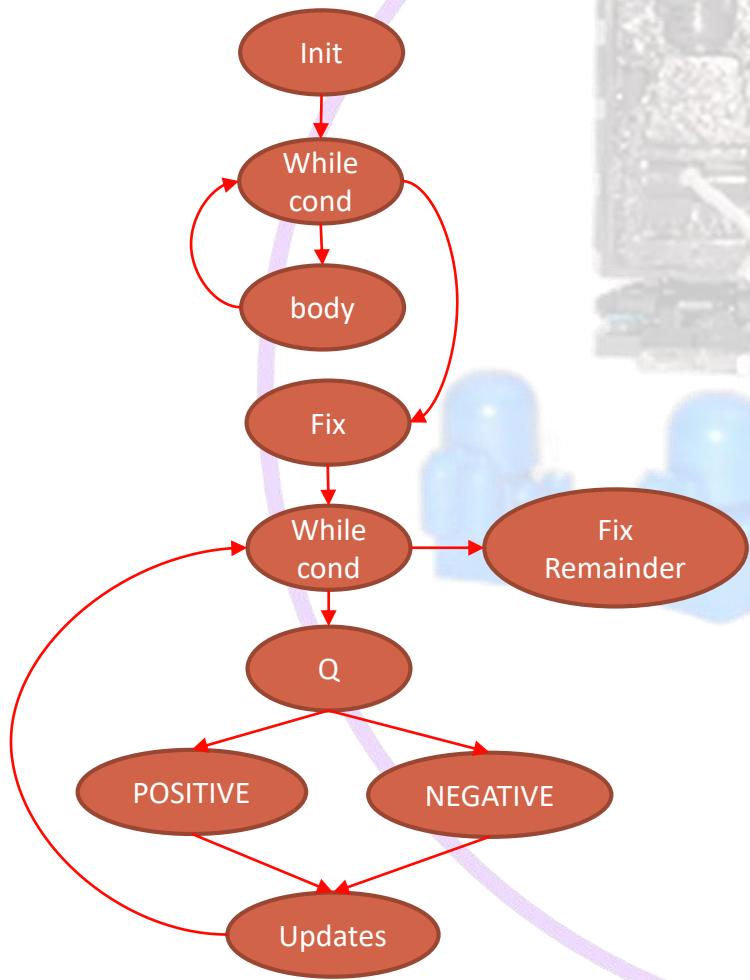
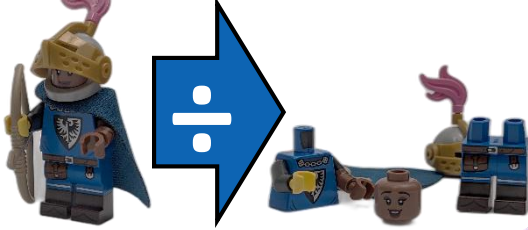


```
35
36 always @(*)
37     case (S)
38         INIT: if (start == 1'b1) NS = WHILE_COND;
39              else NS = INIT;
40         WHILE_COND_1: if (tD_end >= D_or) NS = WHILE_BODY_1;
41                      else NS = FIX;
42         WHILE_BODY_1: NS = WHILE_COND_1;
43         FIX: NS = WHILE_COND_2;
44         WHILE_COND_2: if (bits >= 0) NS = WHILE_2_Q;
45                    else NS = FIX_REM;
46         WHILE_2_Q: if (tD_end >= D_or) NS = WHILE_2_POS;
47                  else NS = WHILE_2_NEG;
48         WHILE_2_POS: NS = WHILE_2_UPDATE;
49         WHILE_2_NEG: NS = WHILE_2_UPDATE;
50         WHILE_2_UPDATE: NS = WHILE_COND_2;
51         FIX_REM: NS = DONE;
52         DONE: NS = DONE;
53         default: NS = ERROR;
54     endcase
```



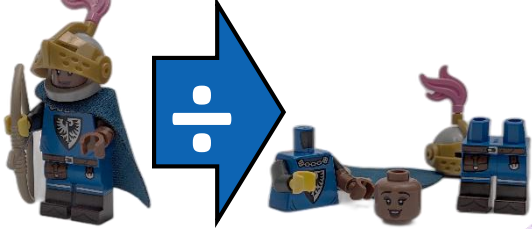
Finite State Machine



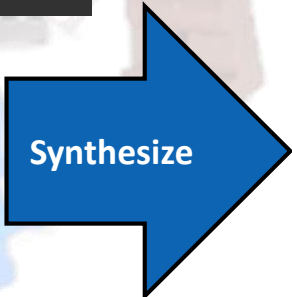


```

55
56 always @(posedge clk or negedge rst)
57   if (rst == 1'b0)
58     begin
59       bits <= 4'd0;
60       Q <= 8'd0;
61       R <= 8'd0;
62       done <= 1'b0;
63       E <= 8'd0;
64     end
65   else
66     case (S)
67     INIT:
68       begin
69         bits <= 4'd0;
70         Q <= 8'd0;
71         R <= 8'd0;
72         done <= 1'b0;
73         E <= 8'd0;
74       end
75     WHILE_BODY_1:
76       begin
77         bits <= bits + 1'b1;
78         E <= (tD_end && 8'h01) | (E << 1);
79         tD_end <= tD_end >> 1;
80       end
81     FIX:
82       begin
83         bits <= bits - 1'b1;
84         tD_end <= (E && 8'h01) | (tD_end << 1);
85         E <= E >> 1;
86       end
87     WHILE_2_Q: Q <= Q << 1;
88     WHILE_2_POS:
89       begin
90         tD_end <= (E && 8'h01) | ((tD_end - D_or) << 1);
91         old_D_end <= (E && 8'h01) | ((tD_end - D_or) << 1);
92         Q <= Q | 8'h01;
93       end
94     WHILE_2_NEG:
95       begin
96         tD_end = (E && 8'h01) | (old_D_end << 1);
97         old_D_end = (E && 8'h01) | (old_D_end << 1);
98       end
99     WHILE_2_UPDATE:
100      begin
101        E <= E >> 1;
102        bits <= bits - 8'd1;
103      end
104     FIX_REM: R <= tD_end >> 1;
105   endcase
  
```

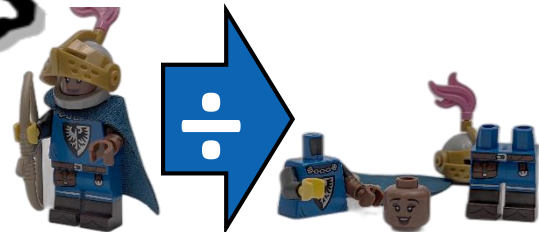


```
1 module div(  
2   input rst, clk, 35  
3   input [7:0]D_or, 36  
4   input [7:0]D_end, 37  
5   input start, 38  
6   output reg [7:0]Q, 39  
7   output reg [7:0]R, 40  
8   output reg done); 41  
9  
10  reg [7:0] E; 42  
11  reg [7:0] tD_end; 43  
12  reg [3:0] bits; 44  
13  reg [7:0] old_D_end; 45  
14  
15  reg [3:0] S; 46  
16  reg [3:0] NS; 47  
17  parameter INIT = 4'd0, 48  
18  WHILE_COND_1 = 49  
19  WHILE_BODY_1 = 50  
20  FIX = 4'd3, 51  
21  WHILE_COND_2 = 52  
22  WHILE_2_Q = 4'd5, 53  
23  WHILE_2_POS = 4'd6, 54  
24  WHILE_2_NEG = 4'd7,  
25  WHILE_2_UPDATE = 4'd8,  
26  FIX_REM = 4'd9,  
27  DONE = 4'd10,  
28  ERROR = 4'hF;  
29  
30  always @(posedge clk or negedge rst)  
31  if (rst == 1'b0)  
32  S <- INIT;  
33  else  
34  S <- NS;  
35  
36  always @(*)  
37  case (S)  
38  INIT: if (start == 1'b1) NS = WHILE_COND;  
39  else NS = INIT;  
40  WHILE_COND_1: if (tD_end >= D_or) NS = WHILE_BODY_1;  
41  else NS = FIX;  
42  WHILE_BODY_1: NS = WHILE_COND_1;  
43  FIX: NS = WHILE_COND_2;  
44  WHILE_COND_2: if (bits >= 0) NS = WHILE_2_Q;  
45  else NS = FIX_REM;  
46  WHILE_2_Q: if (tD_end >= D_or) NS = WHILE_2_POS;  
47  else NS = WHILE_2_NEG;  
48  WHILE_2_POS: NS = WHILE_2_UPDATE;  
49  WHILE_2_NEG: NS = WHILE_2_UPDATE;  
50  WHILE_2_UPDATE: NS = WHILE_COND_2;  
51  FIX_REM: NS = DONE;  
52  DONE: NS = DONE;  
53  default: NS = ERROR;  
54  endcase  
55  
56  FIX:  
57  begin  
58  bits <= bits - 1'b1;  
59  tD_end <= (E && 8'h01) | (tD_end << 1);  
60  E <= E >> 1;  
61  end  
62  WHILE_2_Q: Q <= Q << 1;  
63  WHILE_2_POS:  
64  begin  
65  tD_end <= (E && 8'h01) | ((tD_end - D_or) << 1);  
66  old_D_end <= (E && 8'h01) | ((tD_end - D_or) << 1);  
67  Q <= Q | 8'h01;  
68  end  
69  WHILE_2_NEG:  
70  begin  
71  tD_end = (E && 8'h01) | (old_D_end << 1);  
72  old_D_end = (E && 8'h01) | (old_D_end << 1);  
73  end  
74  WHILE_2_UPDATE:  
75  begin  
76  E <= E >> 1;  
77  bits <= bits - 8'd1;  
78  end  
79  FIX_REM: R <= tD_end >> 1;  
80  endcase  
81  
82  done <= 1'b0;  
83  F <= 8'hA;
```



## MORE IDEAS -

- Fixed Sized Loop and Unroll
- Optimizations
- Other Division Algorithms



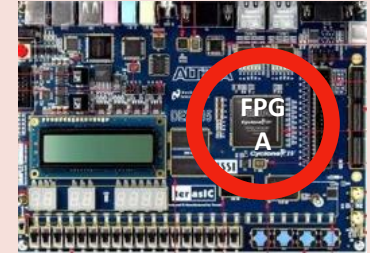




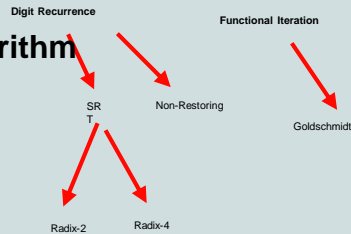
```

module Division (clk, reset, di, do, q, r);
input clk, reset, [bw:0]di, [bw:0]do;
output reg [bw:0]q, [bw:0]r;
always @(posedge clk or negedge reset)
begin
    if (reset == 1'b0)
        q <= BW'b0;
    else
        if (x == 1'b1)
            // calculations and iterations
            ....
        end
    endmodule

```



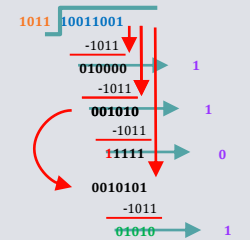
- 1) Bit-Width
- 2) Division Algorithm



# Python Tool

$$D_{div} = 1011_2 = 11_{10}$$

$$D_{end} = 10011001_2 = 153_{10}$$



$$R = 01010_2 = 10_{10} \quad Q = 1101_2 = 13_{10}$$

