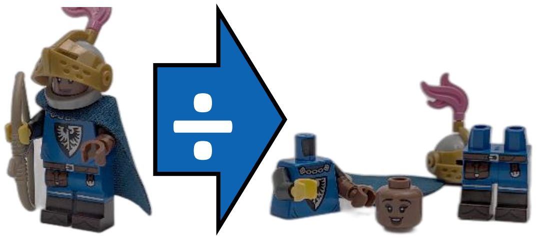
The Forgotten Horseman

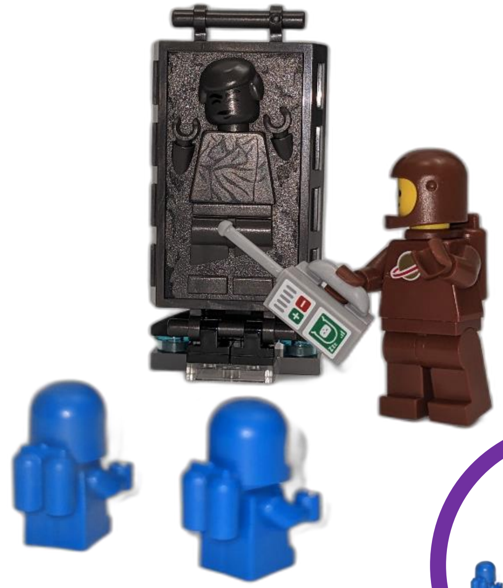
- Digital Implementation of Arithmetic Division and Resources to Learn and Teach Its Complexities

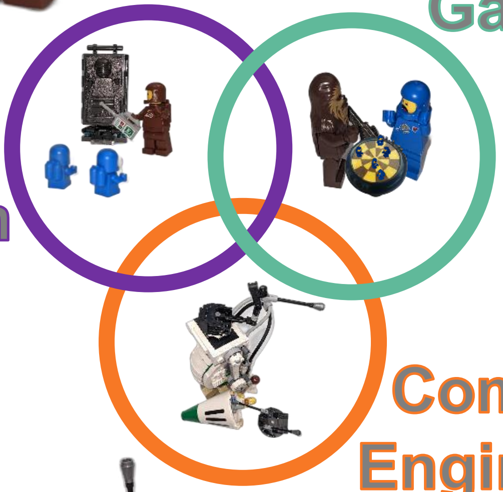
Peter Jamieson, Nathan Martin

Miami University

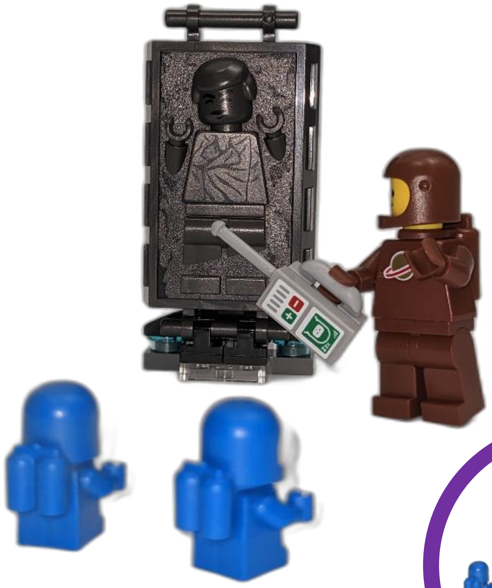


About Me





About Me

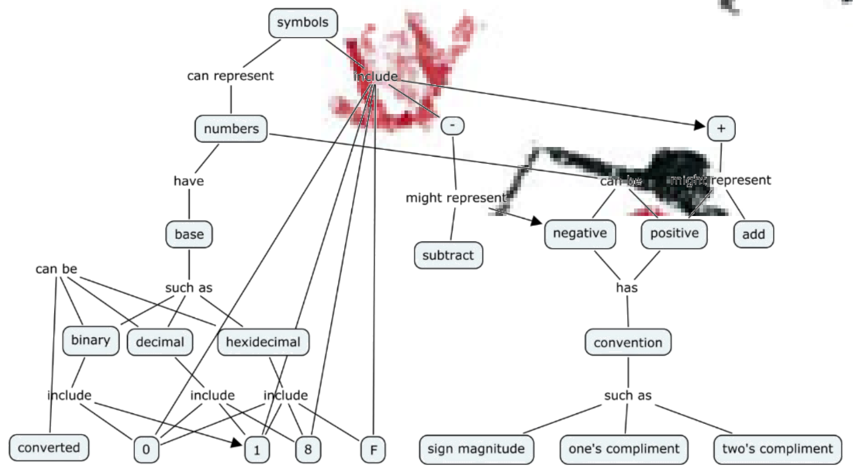


Education
Games
Computer Engineering

+, −, ÷, ×, ...
4508
?
or

Booth Multiplier

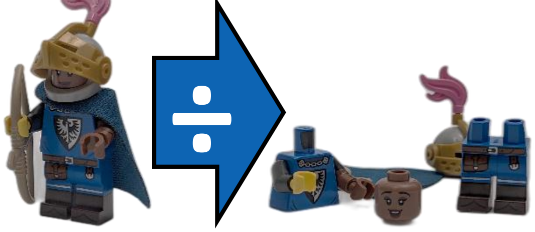
- Basic Adder
- Carry Ripple



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

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

$\div$

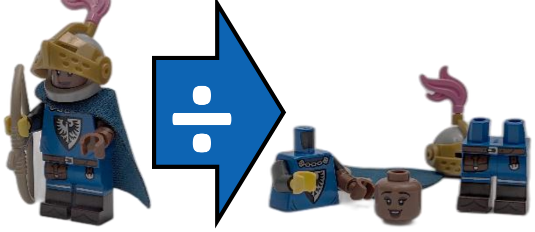


$D_{or} = 1011_2 = 11_{10}$

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

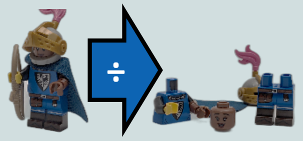
```
1011 | 10011001
        -1011
       010000    →  1
         -1011
         001010    →  1
            -1011
            11111    →  0

         0010101
             -1011
             01010    →  1
```

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

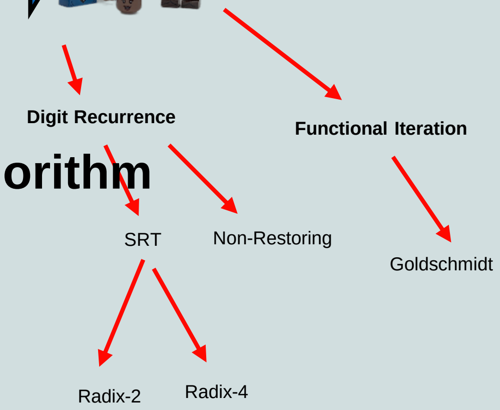


$$\begin{tikzcd}
 & & \textbf{Functional Iteration} \arrow[d] \arrow[dr] & \\
\textbf{Digit Recurrence} \arrow[r] \arrow[dr] \arrow[d] & \text{Restoring} & \text{Goldschmidt} & \text{Newton-Raphson} \\
\text{SRT (Sweeney, Robertson, Tocher)} \arrow[d] & \text{Non-Restoring} & & \\
\text{Higher Radix} & & &
\end{tikzcd}$$

```
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
```

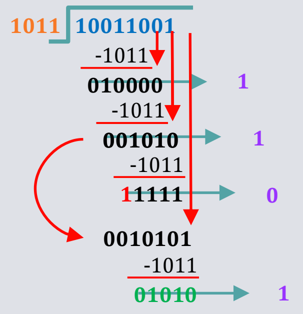


1) **Bit-Width**
2) **Division Algorithm**

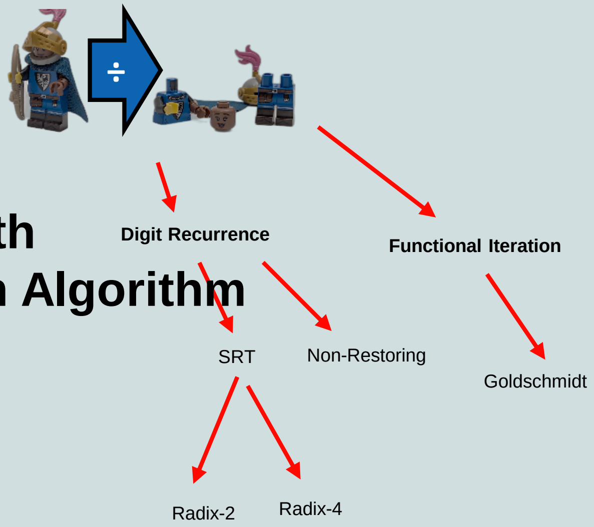


Python Tool

$D_{or} = 1011_2 = 11_{10}$
$D_{end} = 10011001_2 = 153_{10}$



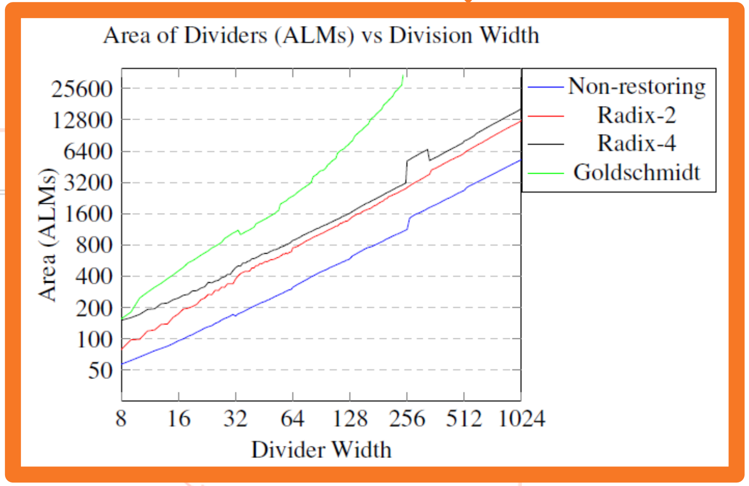
$R = 01010_2 = 10_{10}$ $Q = 1101_2 = 13_{10}$

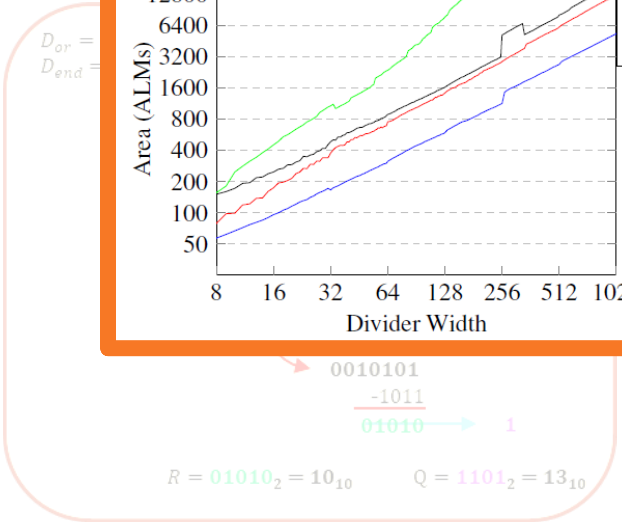


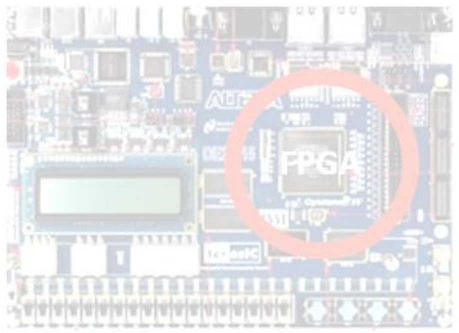
1) **Bit-Width**
2) **Division Algorithm**

Python Tool

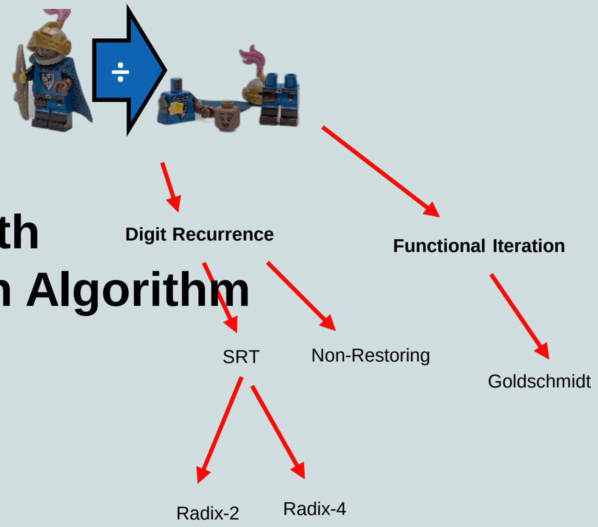
```
module Division (clk, reset, di, do, q, r);
input clk, reset, [bw:0]di, [bw:0]do;
output reg [bw:0]q, [bw:0]r;
always @(posedge clk or negedge reset)
begin
   if (reset == 1'b0)
      q <= BW'b0;
   else
      if (x == 1'b1)
         // calculations and iterations
         ....
end
endmodule
```







```
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
```



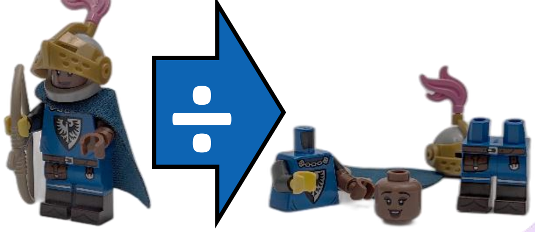
1) **Bit-Width**
2) **Division Algorithm**

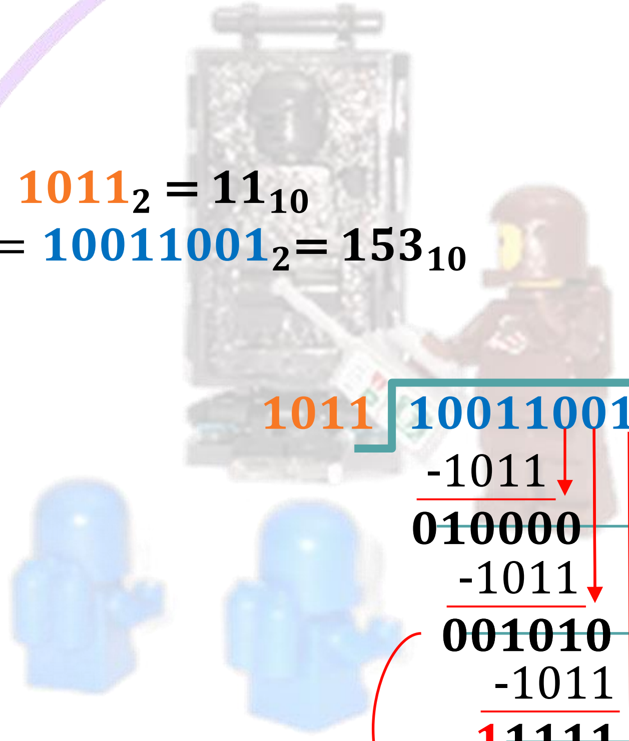
Python Tool

$D_{or} = 1011_2 = 11_{10}$
$D_{end} = 10011001_2 = 153_{10}$

```
1011 | 10011001
         -1011
        010000      1
          -1011
         001010     1
          -1011
          11111     0
         0010101
           -1011
           01010    1
```

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



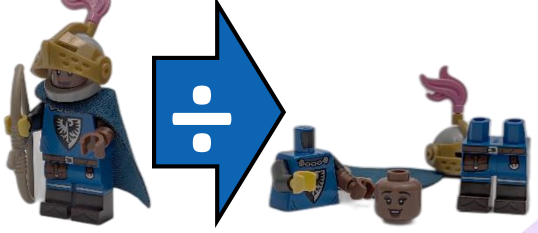


$D_{or} = 1011_2 = 11_{10}$

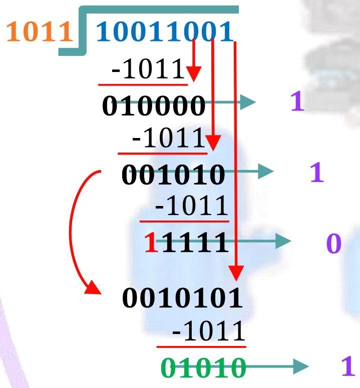
$D_{end} = 10011001_2 = 153_{10}$

$$
\begin{array}{rl}
1011 \,\big) & 10011001 \\
 & \text{-}1011 \\
 & \overline{010000} \rightarrow 1 \\
 & \text{-}1011 \\
 & \overline{001010} \rightarrow 1 \\
 & \text{-}1011 \\
 & \overline{11111} \rightarrow 0 \\
 & 0010101 \\
 & \text{-}1011 \\
 & \overline{01010} \rightarrow 1
\end{array}
$$

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

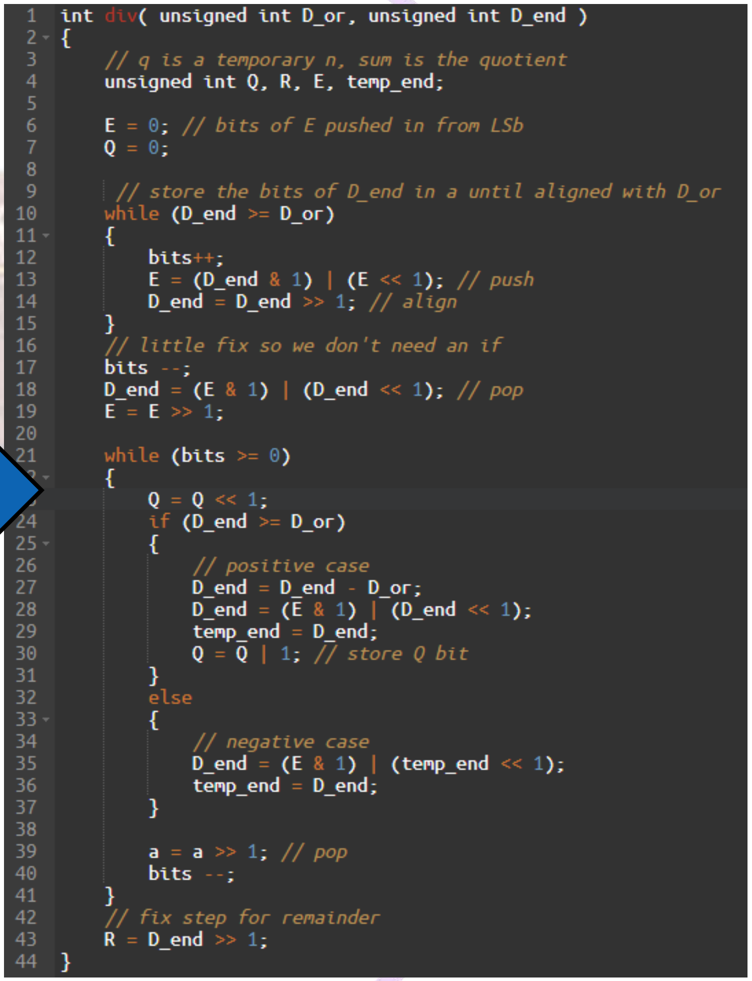


$D_{or} = 1011_2 = 11_{10}$
$D_{end} = 10011001_2 = 153_{10}$

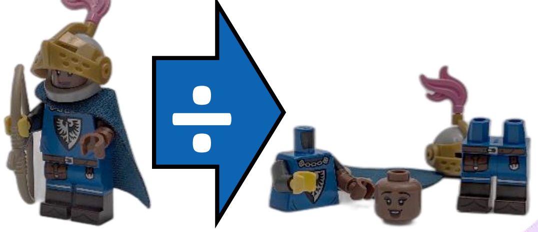


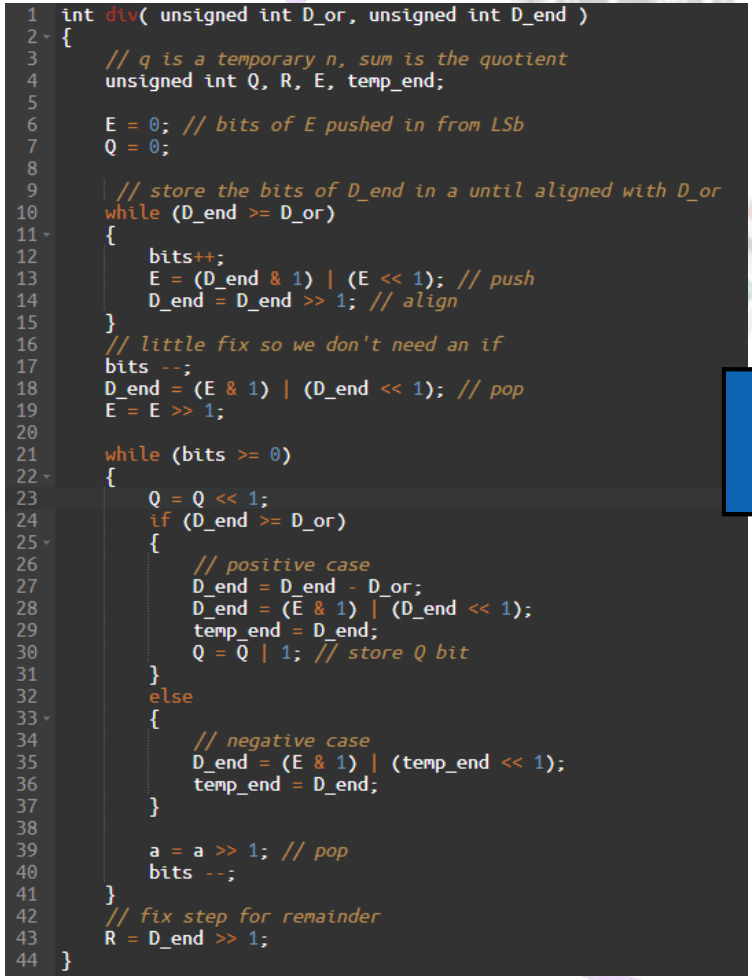
$R = 01010_2 = 10_{10}$ $\quad Q = 1101_2 = 13_{10}$

C code



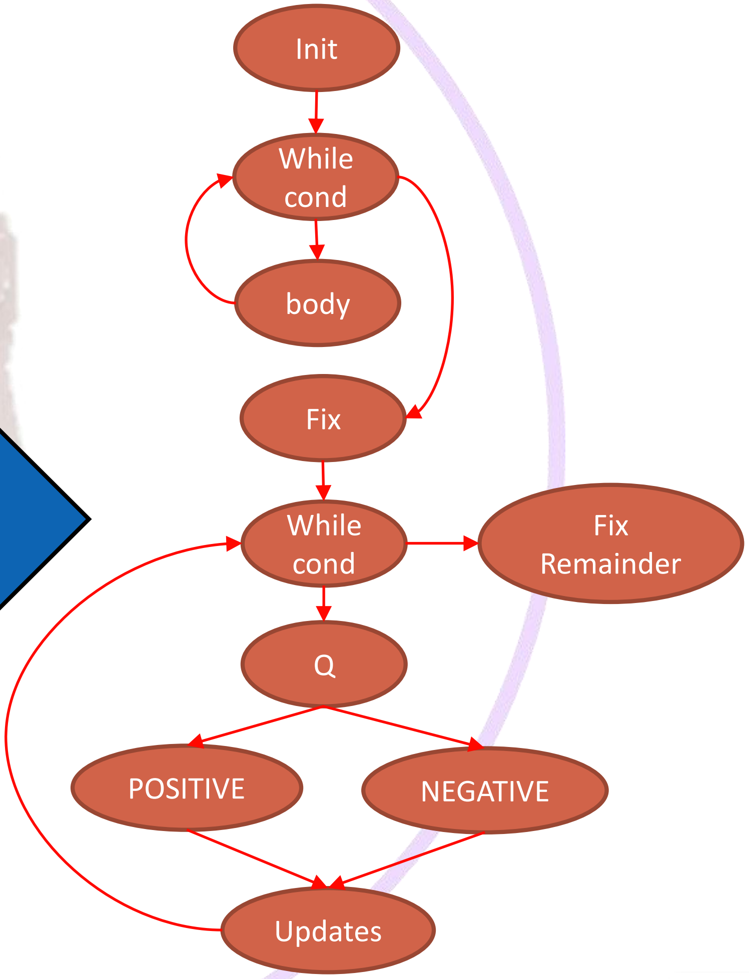
```
int div( unsigned int D_or, unsigned int D_end )
{
    // q is a temporary n, sum is the quotient
    unsigned int Q, R, E, temp_end;

    E = 0; // bits of E pushed in from LSb
    Q = 0;

     // store the bits of D_end in a until aligned with D_or
    while (D_end >= D_or)
    {
        bits++;
        E = (D_end & 1) | (E << 1); // push
        D_end = D_end >> 1; // align
    }
    // little fix so we don't need an if
    bits --;
    D_end = (E & 1) | (D_end << 1); // pop
    E = E >> 1;

    while (bits >= 0)
    {
        Q = Q << 1;
        if (D_end >= D_or)
        {
            // positive case
            D_end = D_end - D_or;
            D_end = (E & 1) | (D_end << 1);
            temp_end = D_end;
            Q = Q | 1; // store Q bit
        }
        else
        {
            // negative case
            D_end = (E & 1) | (temp_end << 1);
            temp_end = D_end;
        }

        a = a >> 1; // pop
        bits --;
    }
    // fix step for remainder
    R = D_end >> 1;
}
```

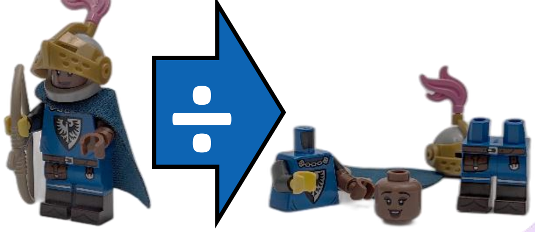


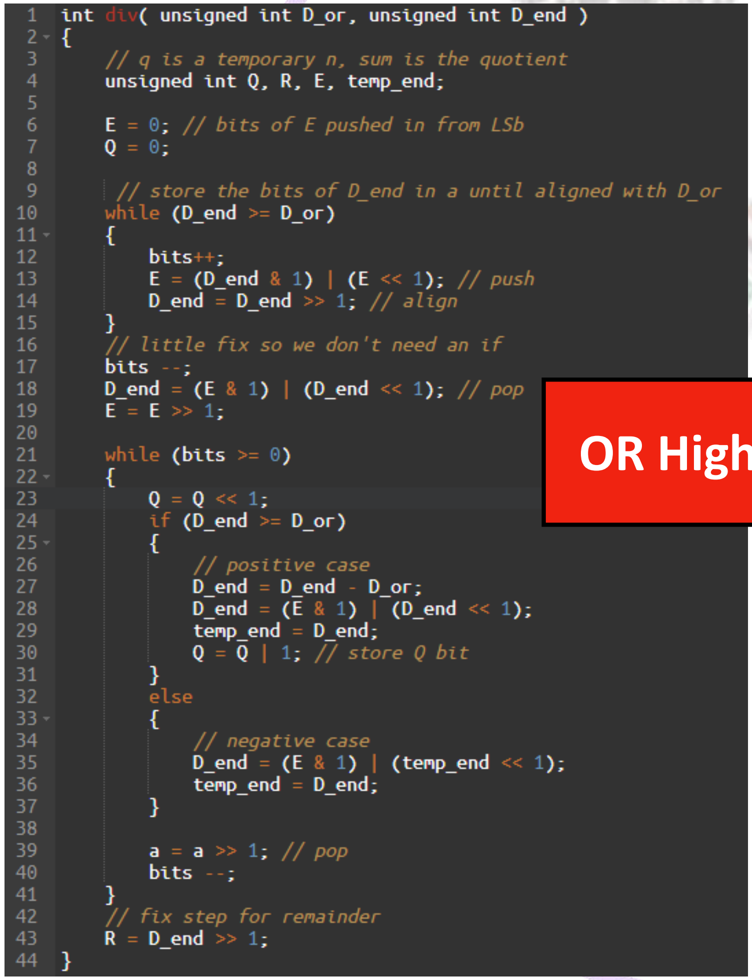


```c
int div( unsigned int D_or, unsigned int D_end )
{
    // q is a temporary n, sum is the quotient
    unsigned int Q, R, E, temp_end;

    E = 0; // bits of E pushed in from LSb
    Q = 0;

     // store the bits of D_end in a until aligned with D_or
    while (D_end >= D_or)
    {
        bits++;
        E = (D_end & 1) | (E << 1); // push
        D_end = D_end >> 1; // align
    }
    // little fix so we don't need an if
    bits --;
    D_end = (E & 1) | (D_end << 1); // pop
    E = E >> 1;

    while (bits >= 0)
    {
        Q = Q << 1;
        if (D_end >= D_or)
        {
            // positive case
            D_end = D_end - D_or;
            D_end = (E & 1) | (D_end << 1);
            temp_end = D_end;
            Q = Q | 1; // store Q bit
        }
        else
        {
            // negative case
            D_end = (E & 1) | (temp_end << 1);
            temp_end = D_end;
        }

        a = a >> 1; // pop
        bits --;
    }
    // fix step for remainder
    R = D_end >> 1;
}
```

Finite State Machine

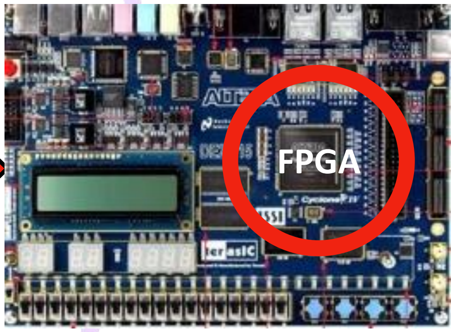


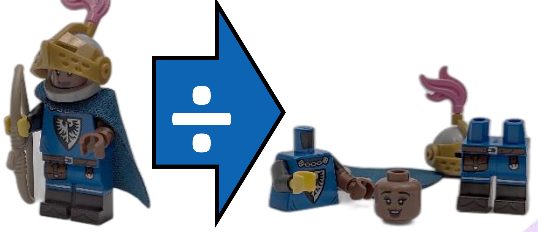


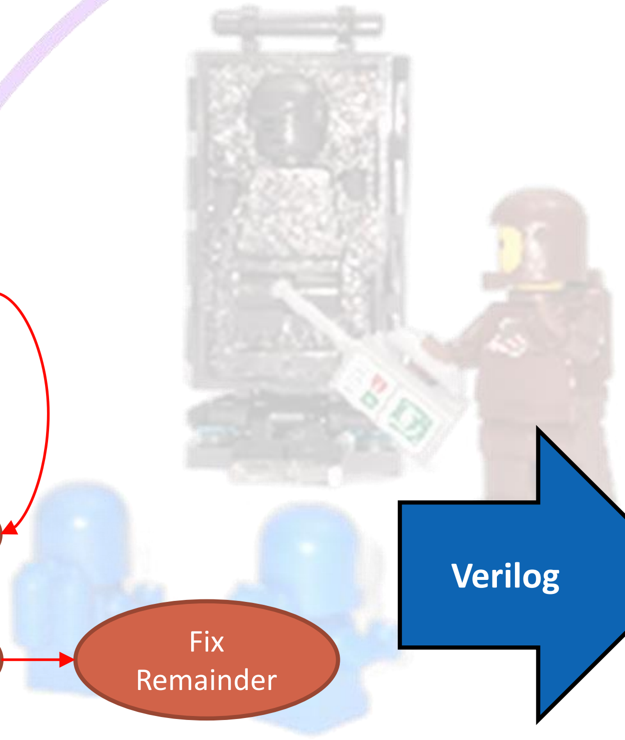


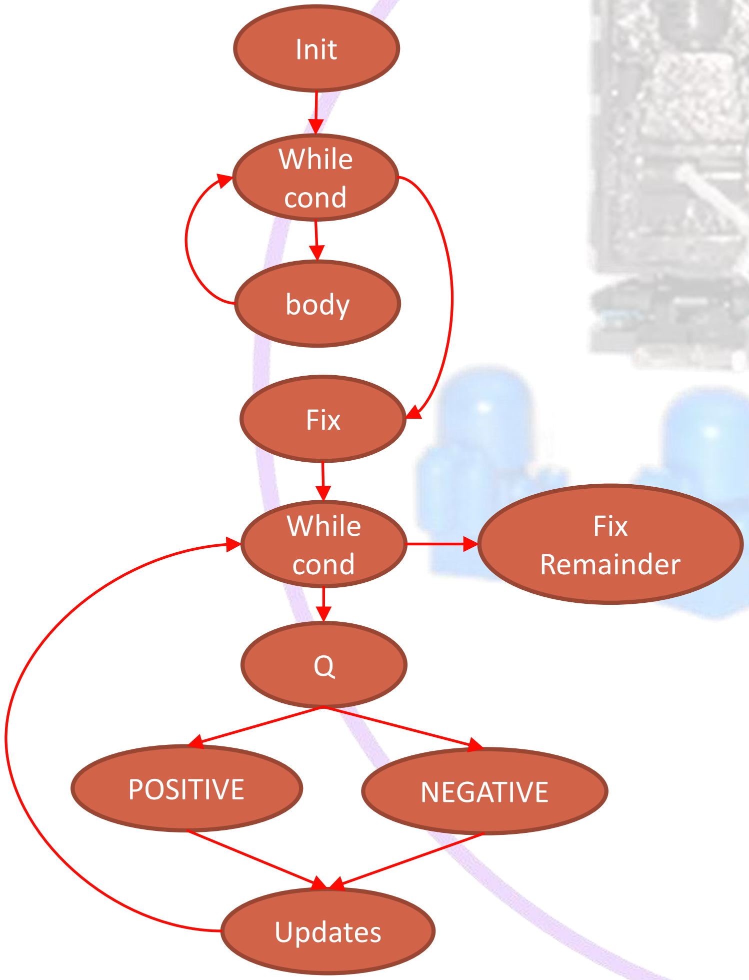
```
int div( unsigned int D_or, unsigned int D_end )
{
    // q is a temporary n, sum is the quotient
    unsigned int Q, R, E, temp_end;

    E = 0; // bits of E pushed in from LSb
    Q = 0;

     // store the bits of D_end in a until aligned with D_or
    while (D_end >= D_or)
    {
        bits++;
        E = (D_end & 1) | (E << 1); // push
        D_end = D_end >> 1; // align
    }
    // little fix so we don't need an if
    bits --;
    D_end = (E & 1) | (D_end << 1); // pop
    E = E >> 1;

    while (bits >= 0)
    {
        Q = Q << 1;
        if (D_end >= D_or)
        {
            // positive case
            D_end = D_end - D_or;
            D_end = (E & 1) | (D_end << 1);
            temp_end = D_end;
            Q = Q | 1; // store Q bit
        }
        else
        {
            // negative case
            D_end = (E & 1) | (temp_end << 1);
            temp_end = D_end;
        }

        a = a >> 1; // pop
        bits --;
    }
    // fix step for remainder
    R = D_end >> 1;
}
```

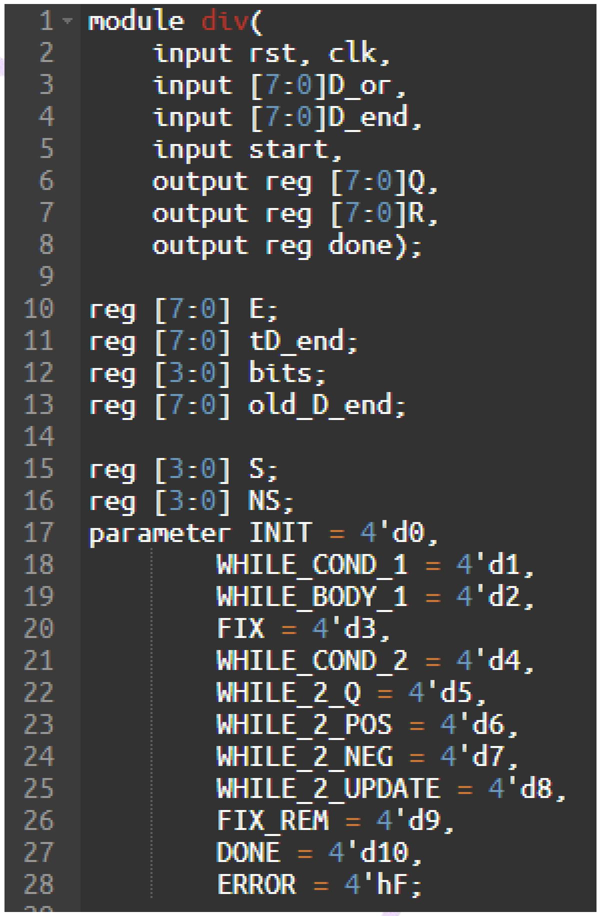
OR High-Level Synthesis (HLS)



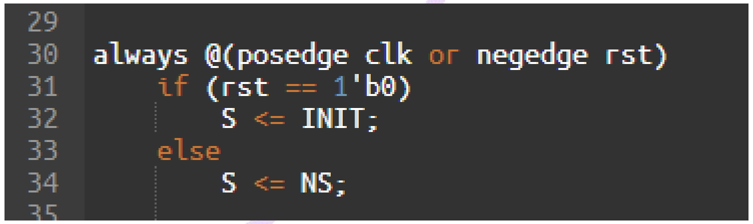




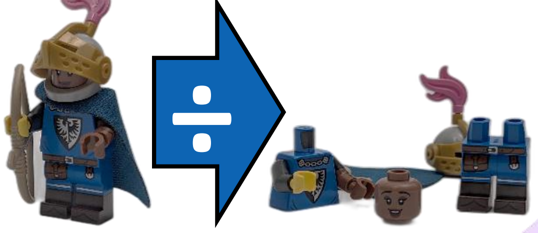


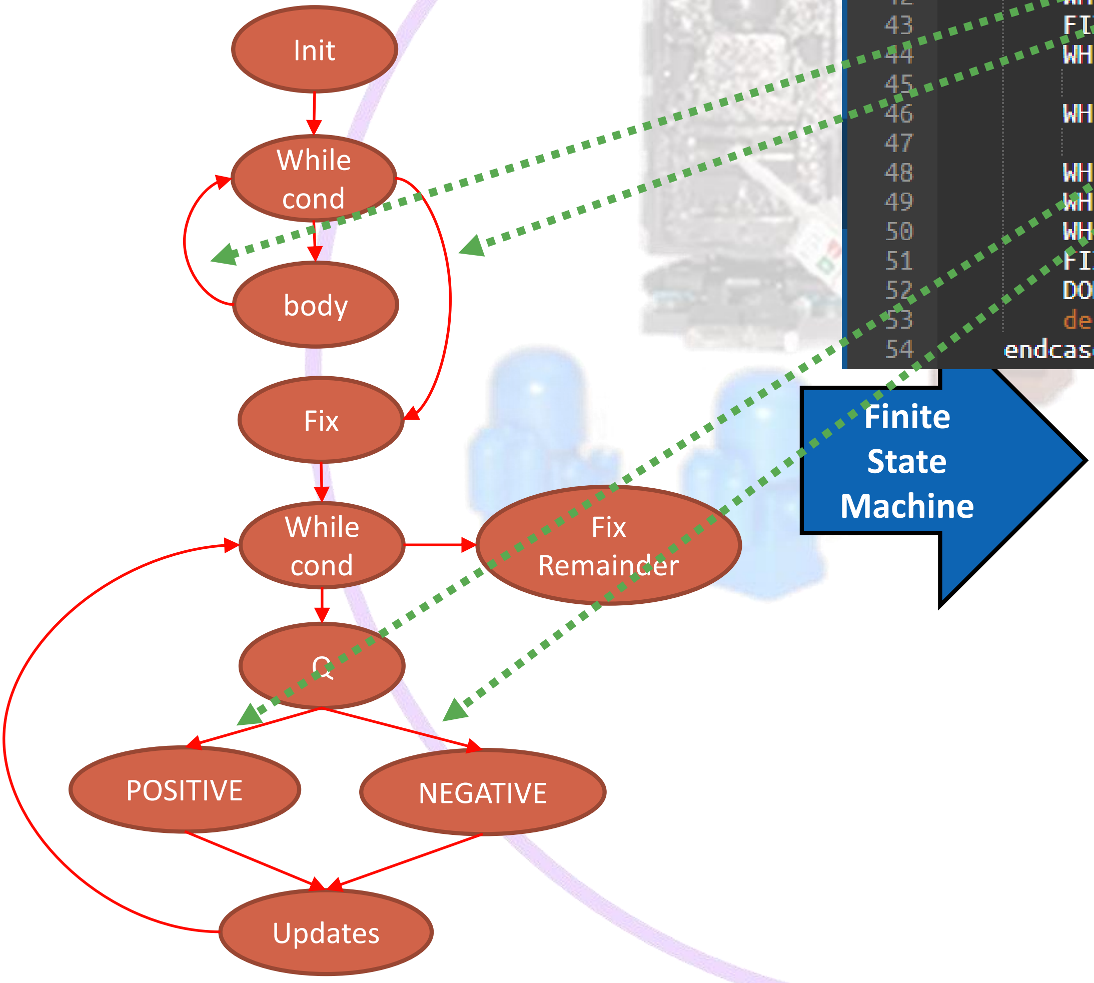


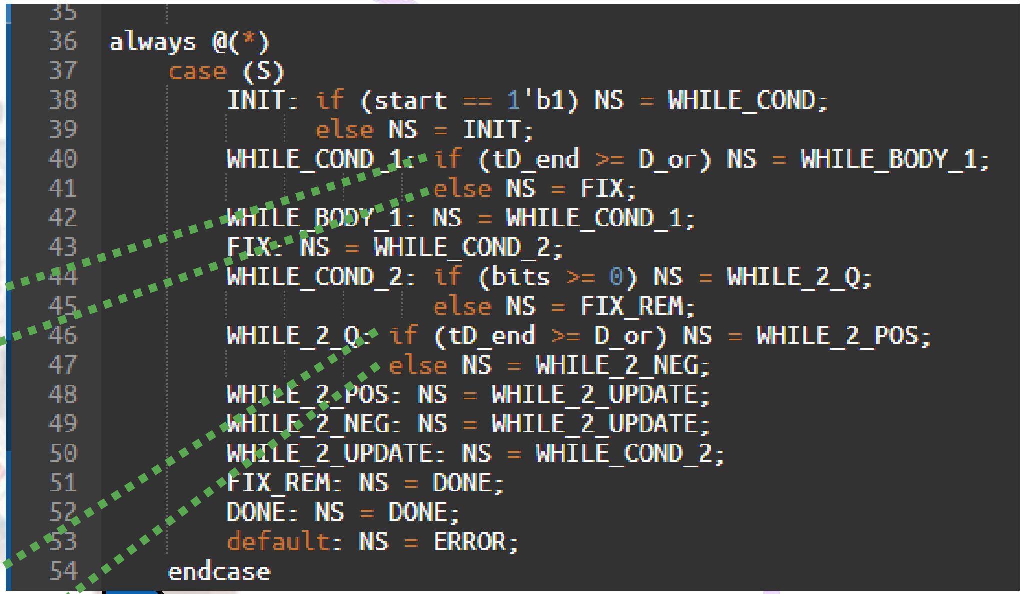
```
module div(
      input rst, clk,
      input [7:0]D_or,
      input [7:0]D_end,
      input start,
      output reg [7:0]Q,
      output reg [7:0]R,
      output reg done);

reg [7:0] E;
reg [7:0] tD_end;
reg [3:0] bits;
reg [7:0] old_D_end;

reg [3:0] S;
reg [3:0] NS;
parameter INIT = 4'd0,
            WHILE_COND_1 = 4'd1,
            WHILE_BODY_1 = 4'd2,
            FIX = 4'd3,
            WHILE_COND_2 = 4'd4,
            WHILE_2_Q = 4'd5,
            WHILE_2_POS = 4'd6,
            WHILE_2_NEG = 4'd7,
            WHILE_2_UPDATE = 4'd8,
            FIX_REM = 4'd9,
            DONE = 4'd10,
            ERROR = 4'hF;
```



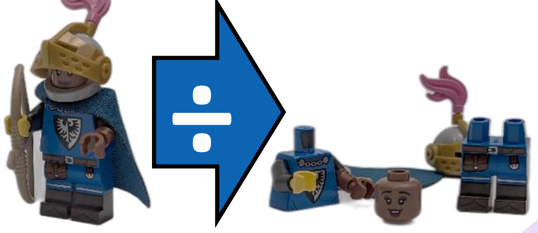
```

always @(posedge clk or negedge rst)
      if (rst == 1'b0)
            S <= INIT;
      else
            S <= NS;
```

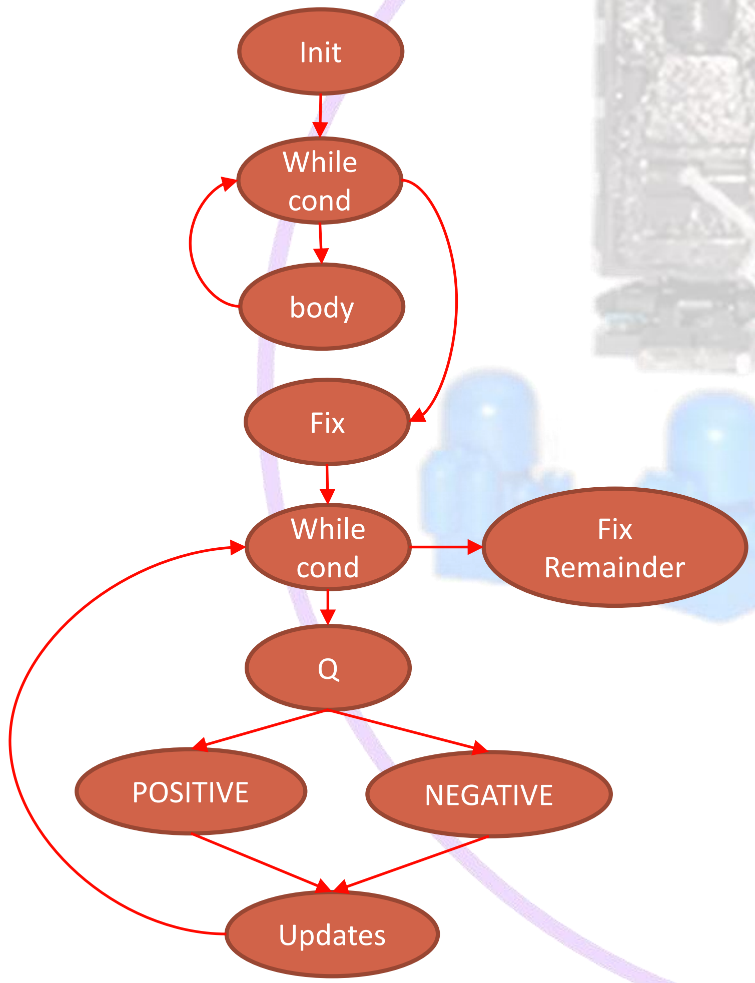




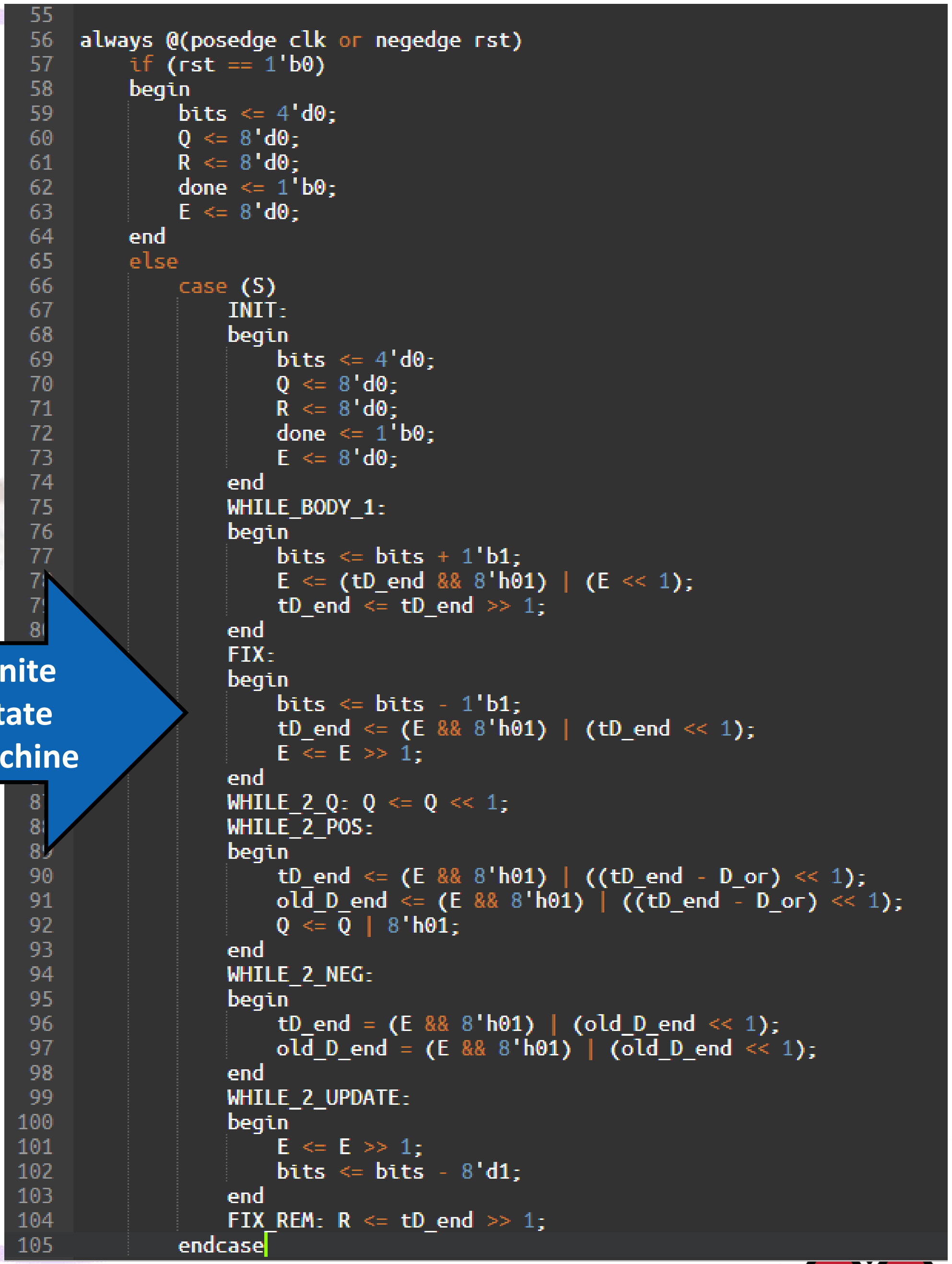


```
always @(*)
    case (S)
        INIT: if (start == 1'b1) NS = WHILE_COND;
              else NS = INIT;
        WHILE_COND_1: if (tD_end >= D_or) NS = WHILE_BODY_1;
                      else NS = FIX;
        WHILE_BODY_1: NS = WHILE_COND_1;
        FIX: NS = WHILE_COND_2;
        WHILE_COND_2: if (bits >= 0) NS = WHILE_2_Q;
                      else NS = FIX_REM;
        WHILE_2_Q: if (tD_end >= D_or) NS = WHILE_2_POS;
                   else NS = WHILE_2_NEG;
        WHILE_2_POS: NS = WHILE_2_UPDATE;
        WHILE_2_NEG: NS = WHILE_2_UPDATE;
        WHILE_2_UPDATE: NS = WHILE_COND_2;
        FIX_REM: NS = DONE;
        DONE: NS = DONE;
        default: NS = ERROR;
    endcase
```

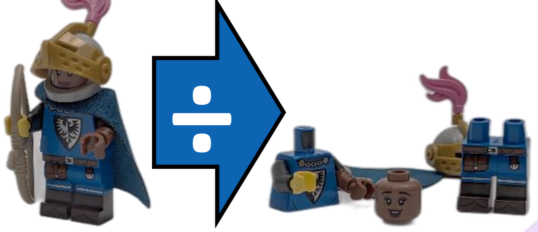


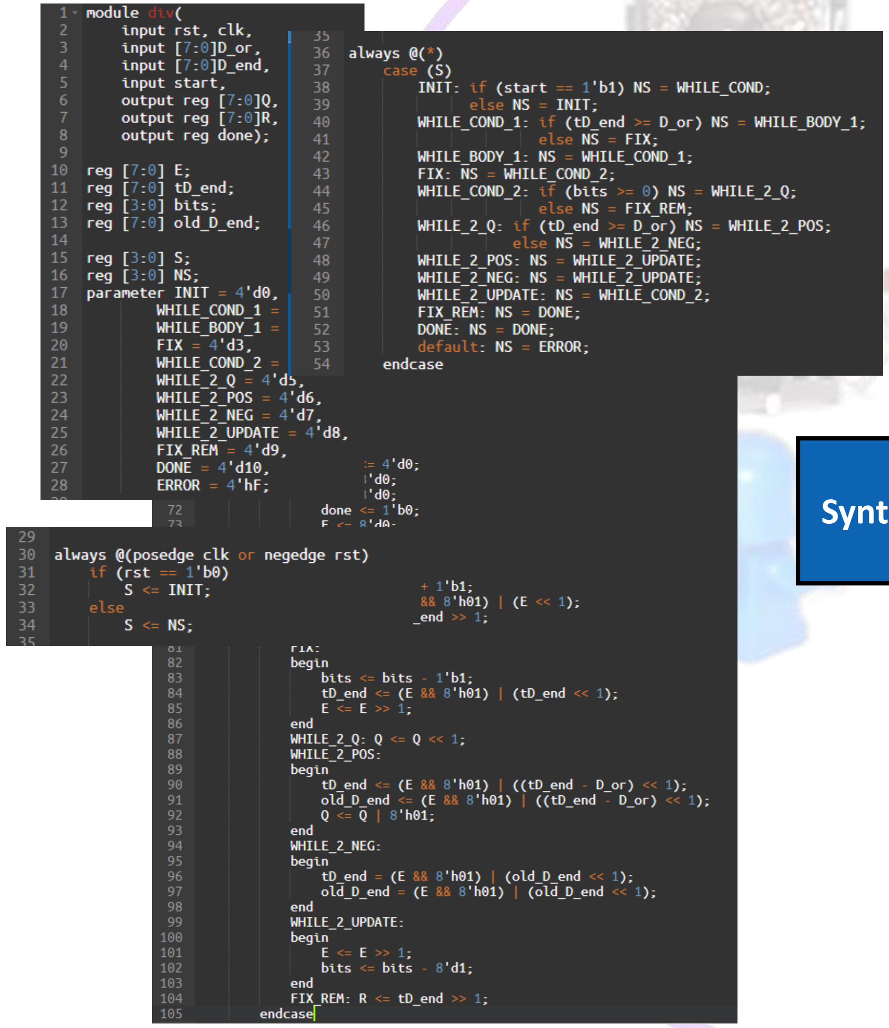


Finite State Machine



```
always @(posedge clk or negedge rst)
    if (rst == 1'b0)
    begin
        bits <= 4'd0;
        Q <= 8'd0;
        R <= 8'd0;
        done <= 1'b0;
        E <= 8'd0;
    end
    else
        case (S)
            INIT:
            begin
                bits <= 4'd0;
                Q <= 8'd0;
                R <= 8'd0;
                done <= 1'b0;
                E <= 8'd0;
            end
            WHILE_BODY_1:
            begin
                bits <= bits + 1'b1;
                E <= (tD_end && 8'h01) | (E << 1);
                tD_end <= tD_end >> 1;
            end
            FIX:
            begin
                bits <= bits - 1'b1;
                tD_end <= (E && 8'h01) | (tD_end << 1);
                E <= E >> 1;
            end
            WHILE_2_Q: Q <= Q << 1;
            WHILE_2_POS:
            begin
                tD_end <= (E && 8'h01) | ((tD_end - D_or) << 1);
                old_D_end <= (E && 8'h01) | ((tD_end - D_or) << 1);
                Q <= Q | 8'h01;
            end
            WHILE_2_NEG:
            begin
                tD_end = (E && 8'h01) | (old_D_end << 1);
                old_D_end = (E && 8'h01) | (old_D_end << 1);
            end
            WHILE_2_UPDATE:
            begin
                E <= E >> 1;
                bits <= bits - 8'd1;
            end
            FIX_REM: R <= tD_end >> 1;
        endcase
```





```
module div(
    input rst, clk,
    input [7:0]D_or,
    input [7:0]D_end,
    input start,
    output reg [7:0]Q,
    output reg [7:0]R,
    output reg done);

reg [7:0] E;
reg [7:0] tD_end;
reg [3:0] bits;
reg [7:0] old_D_end;

reg [3:0] S;
reg [3:0] NS;
parameter INIT = 4'd0,
        WHILE_COND_1 =
        WHILE_BODY_1 =
        FIX = 4'd3,
        WHILE_COND_2 =
        WHILE_2_Q = 4'd5,
        WHILE_2_POS = 4'd6,
        WHILE_2_NEG = 4'd7,
        WHILE_2_UPDATE = 4'd8,
        FIX_REM = 4'd9,
        DONE = 4'd10,
        ERROR = 4'hF;
```

```
always @(*)
    case (S)
        INIT: if (start == 1'b1) NS = WHILE_COND;
                else NS = INIT;
        WHILE_COND_1: if (tD_end >= D_or) NS = WHILE_BODY_1;
                        else NS = FIX;
        WHILE_BODY_1: NS = WHILE_COND_1;
        FIX: NS = WHILE_COND_2;
        WHILE_COND_2: if (bits >= 0) NS = WHILE_2_Q;
                        else NS = FIX_REM;
        WHILE_2_Q: if (tD_end >= D_or) NS = WHILE_2_POS;
                    else NS = WHILE_2_NEG;
        WHILE_2_POS: NS = WHILE_2_UPDATE;
        WHILE_2_NEG: NS = WHILE_2_UPDATE;
        WHILE_2_UPDATE: NS = WHILE_COND_2;
        FIX_REM: NS = DONE;
        DONE: NS = DONE;
        default: NS = ERROR;
    endcase
```

```
:= 4'd0;
'd0;
'd0;
            done <= 1'b0;
```

```
always @(posedge clk or negedge rst)
    if (rst == 1'b0)
        S <= INIT;
    else
        S <= NS;
```

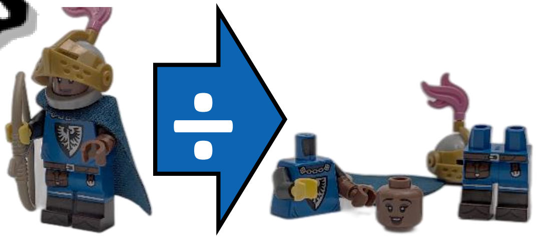
```
 + 1'b1;
 && 8'h01) | (E << 1);
_end >> 1;
```

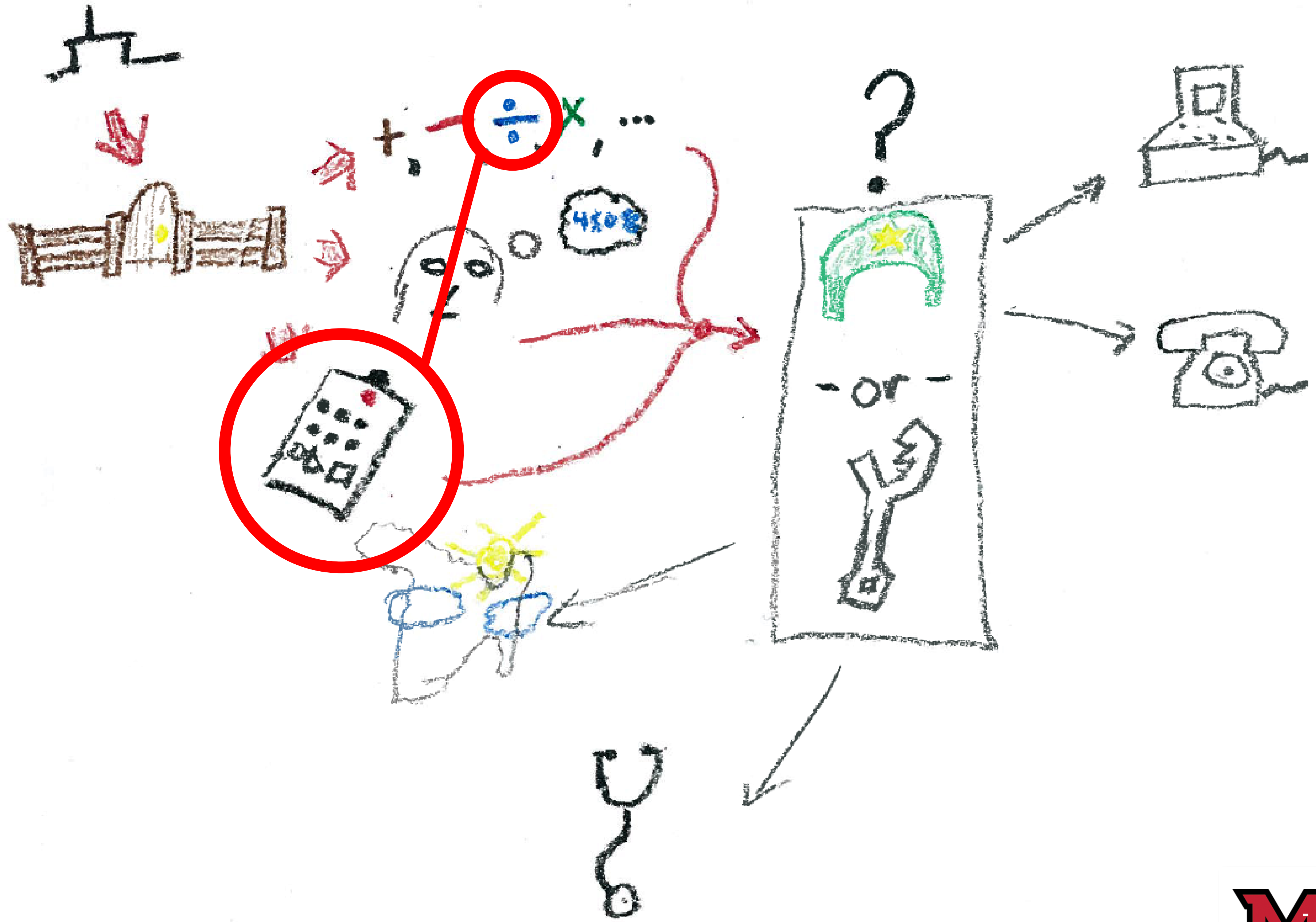
```
            begin
                bits <= bits - 1'b1;
                tD_end <= (E && 8'h01) | (tD_end << 1);
                E <= E >> 1;
            end
            WHILE_2_Q: Q <= Q << 1;
            WHILE_2_POS:
            begin
                tD_end <= (E && 8'h01) | ((tD_end - D_or) << 1);
                old_D_end <= (E && 8'h01) | ((tD_end - D_or) << 1);
                Q <= Q | 8'h01;
            end
            WHILE_2_NEG:
            begin
                tD_end = (E && 8'h01) | (old_D_end << 1);
                old_D_end = (E && 8'h01) | (old_D_end << 1);
            end
            WHILE_2_UPDATE:
            begin
                E <= E >> 1;
                bits <= bits - 8'd1;
            end
            FIX_REM: R <= tD_end >> 1;
        endcase
```

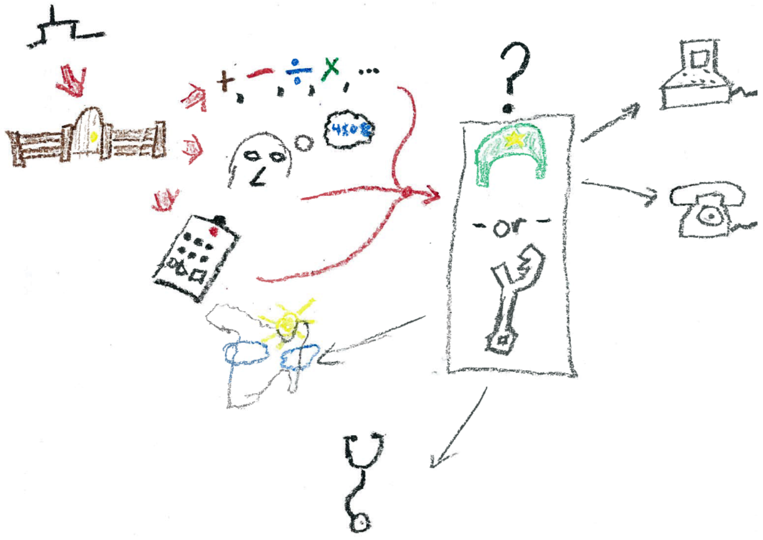
Synthesize

MORE IDEAS -

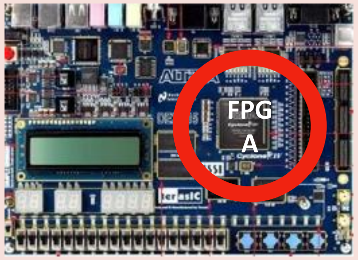
- Fixed Sized Loop and Unroll
- Optimizations
- Other Division Algorithms

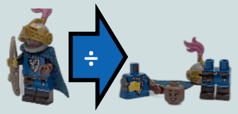



+, −, ÷, ×, ...
4508
?
- or -

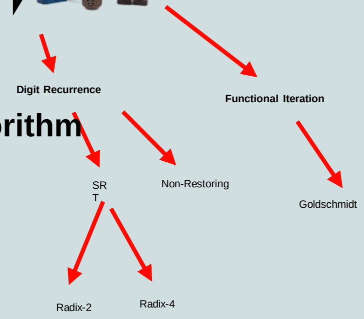


```
module Division (clk, reset, di, do, q, r);
input clk, reset, [bw:0]di, [bw:0]do;
output reg [bw:0]q, [bw:0]r;
always @(posedge clk or negedge reset)
begin
   if (reset == 1'b0)
      q <= BW'b0;
   else
      if (x == 1'b1)
         // calculations and iterations
         ….
end
endmodule
```

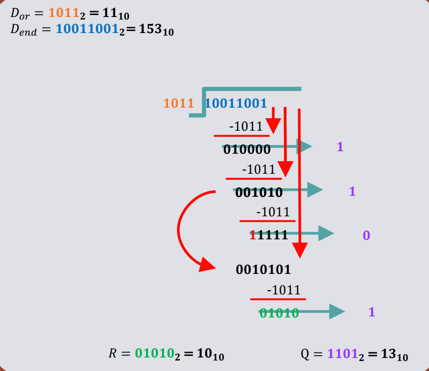




1) Bit-Width
2) Division Algorithm



Python Tool



$D_{or} = 1011_2 = 11_{10}$
$D_{end} = 10011001_2 = 153_{10}$

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

https://github.com/marti693/Variable-Width-Division-Scripts