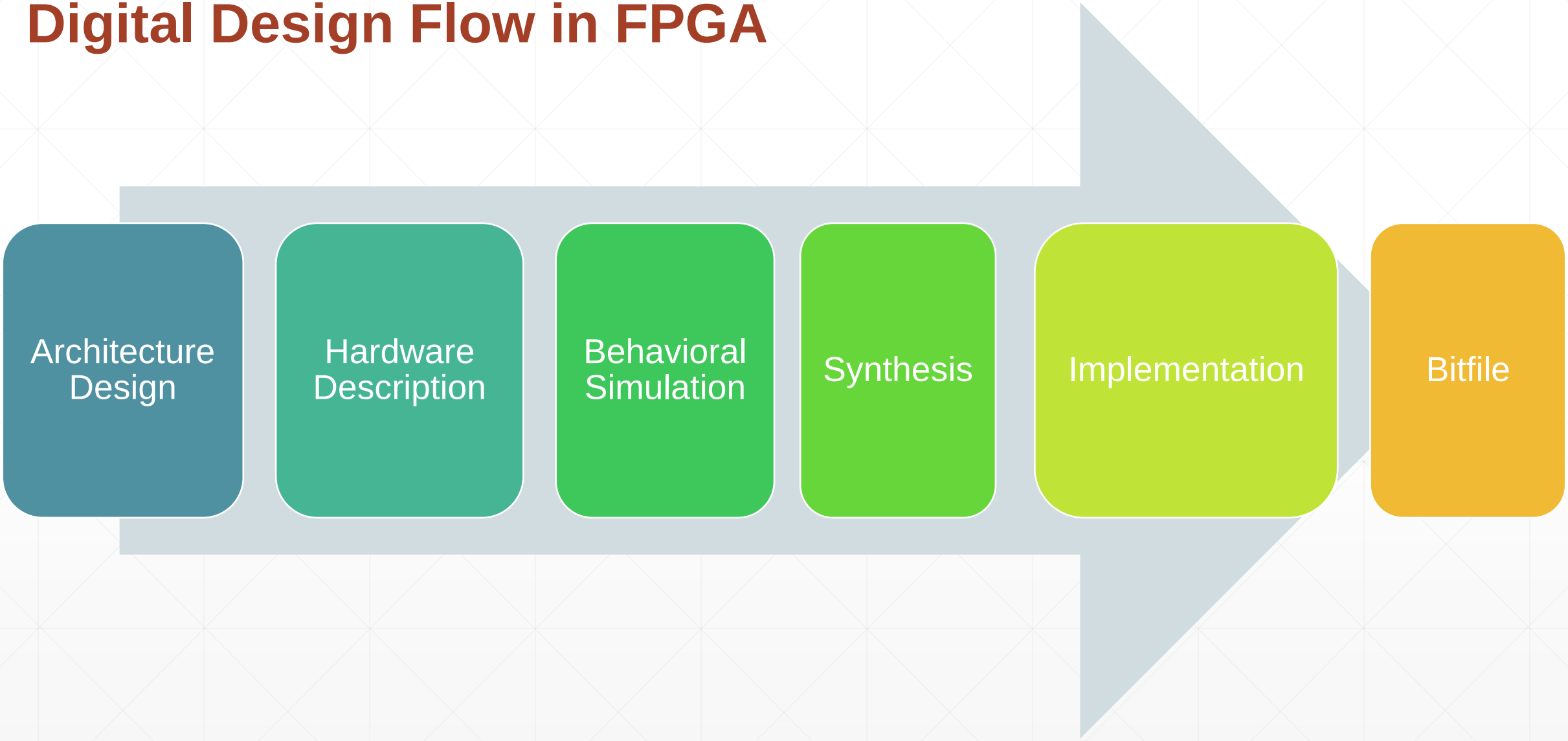
Hardware Design Overview

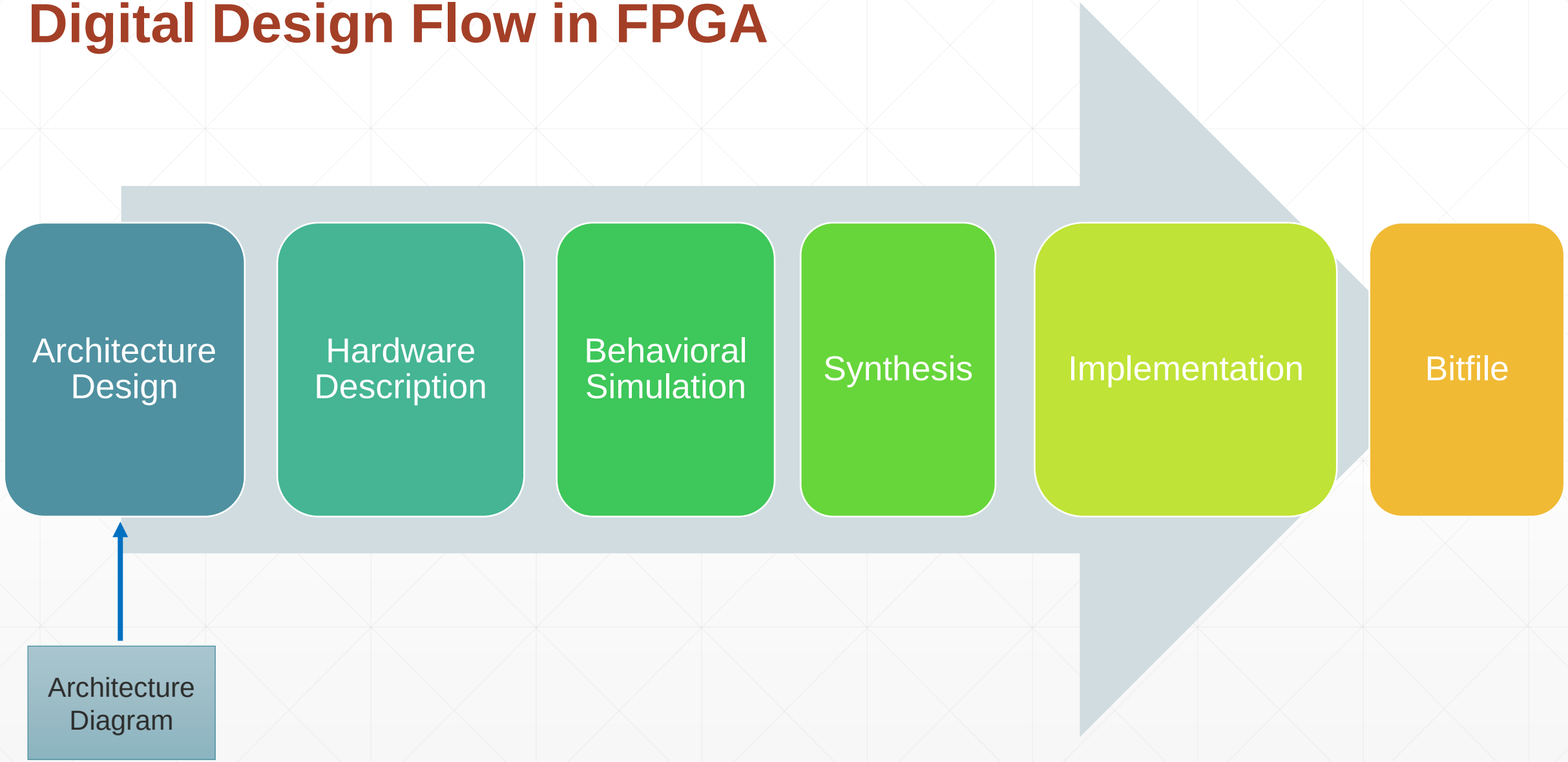
Siddhartha Chowdhury

Secured Embedded Architecture Laboratory (SEAL)
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

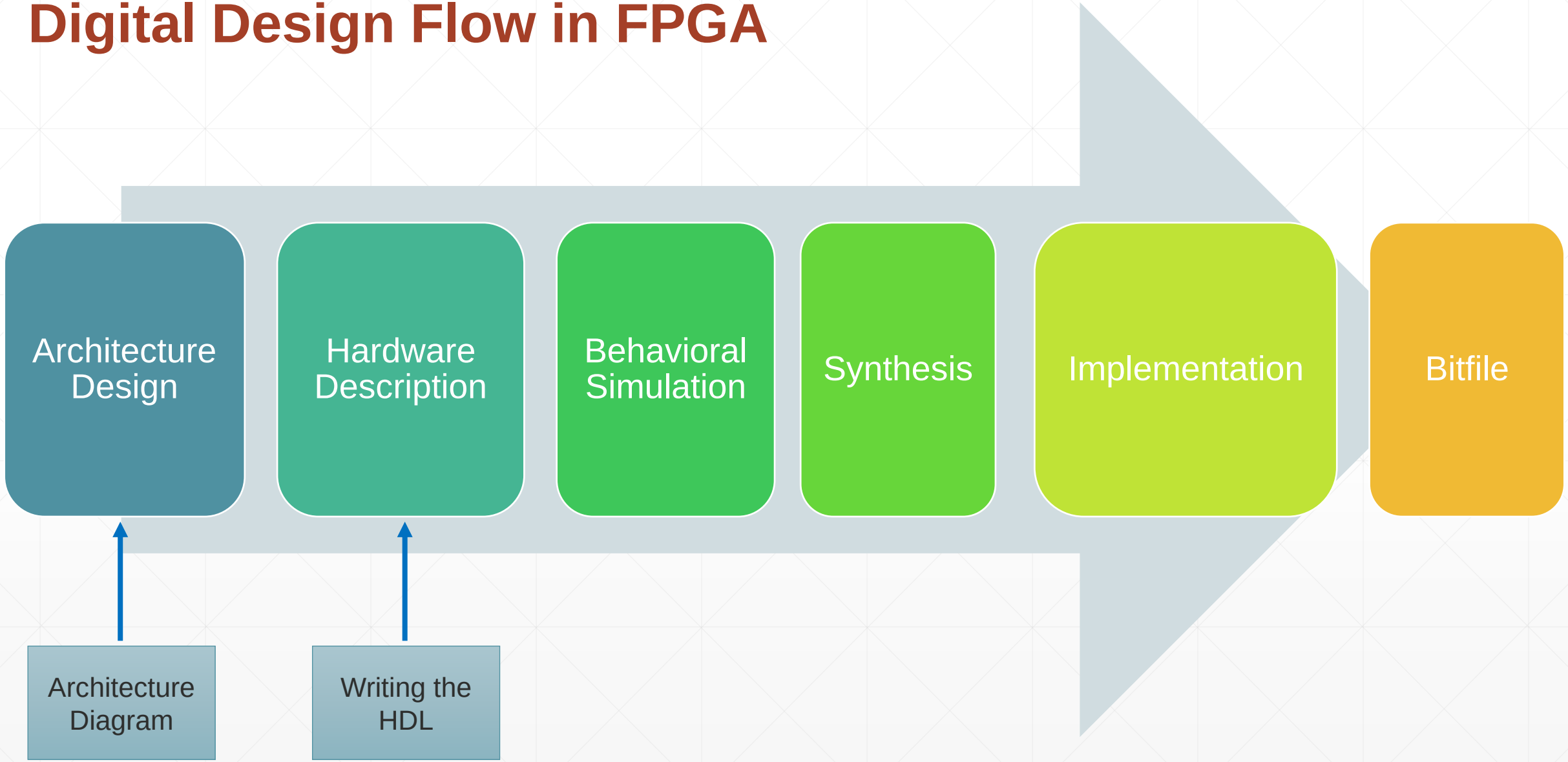
Digital Design Flow in FPGA



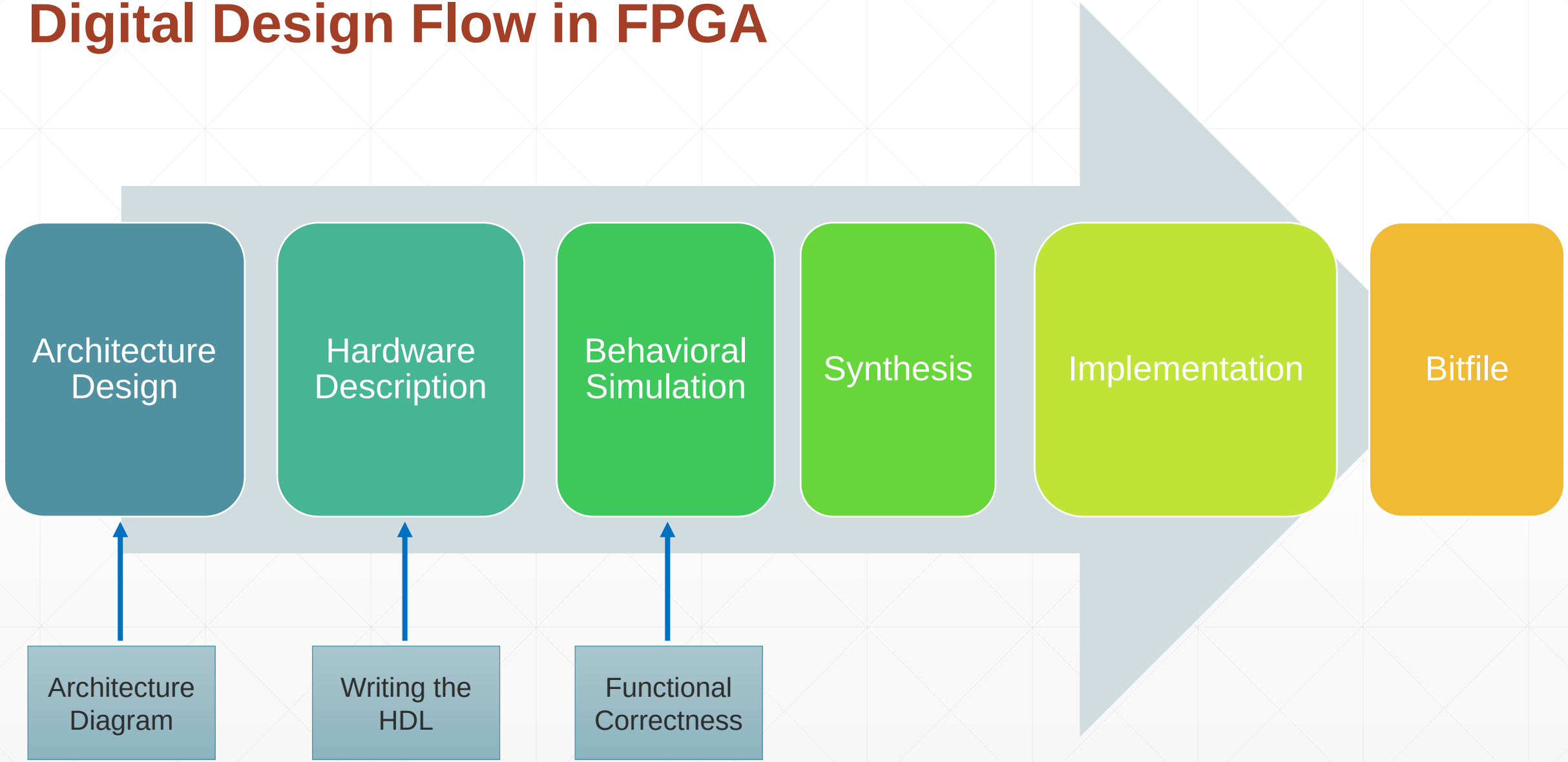
Digital Design Flow in FPGA



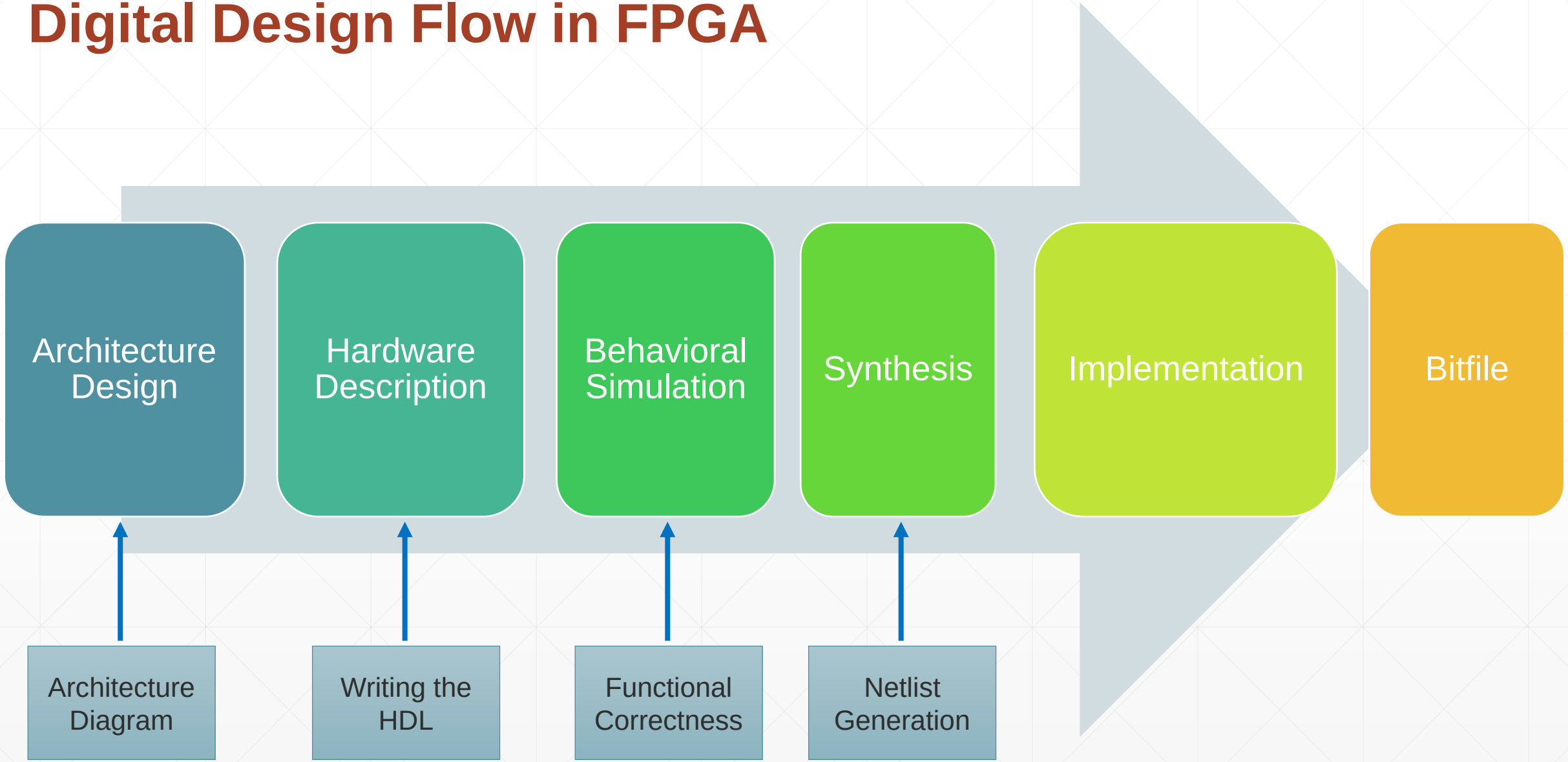
Digital Design Flow in FPGA



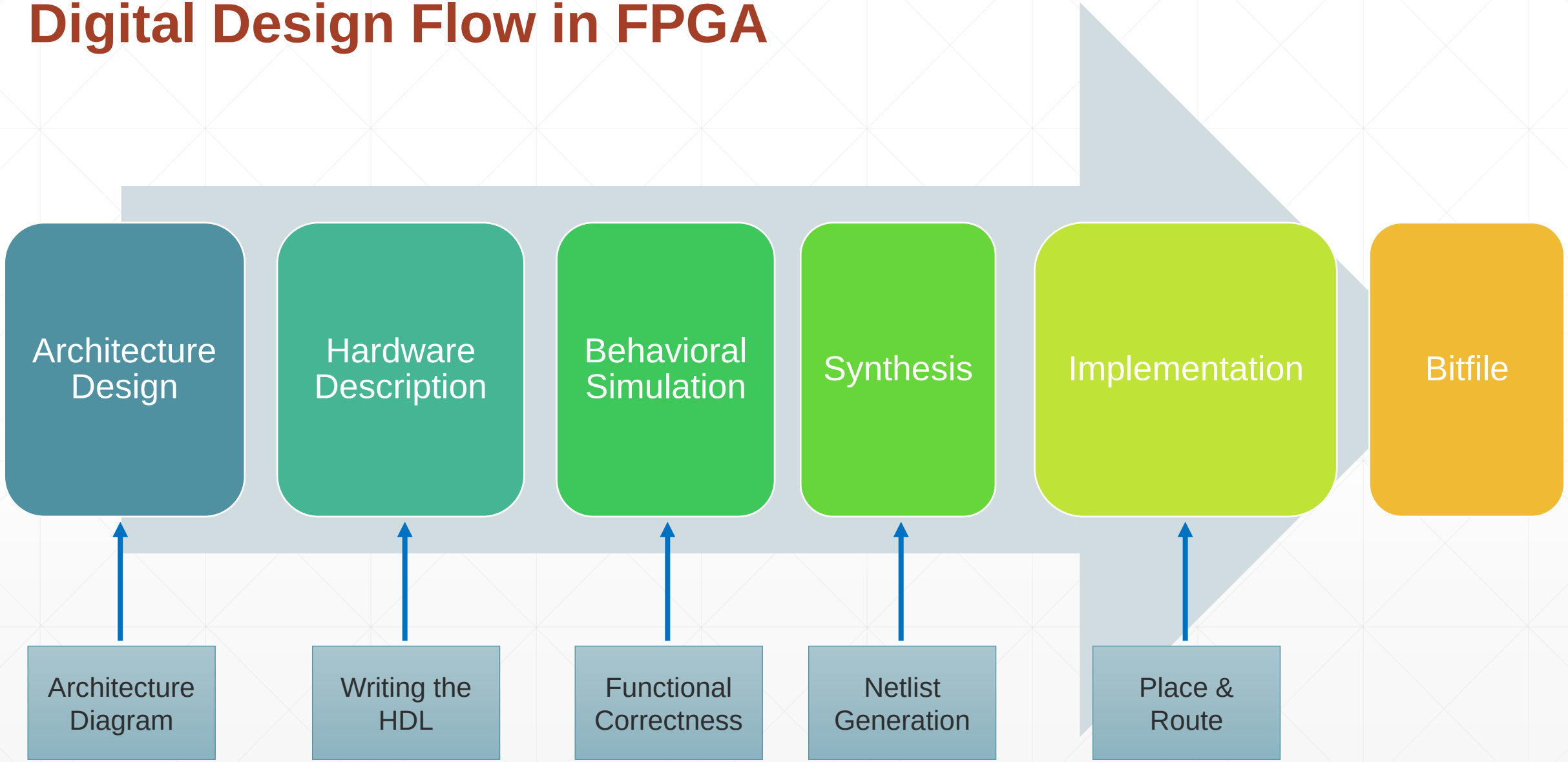
Digital Design Flow in FPGA



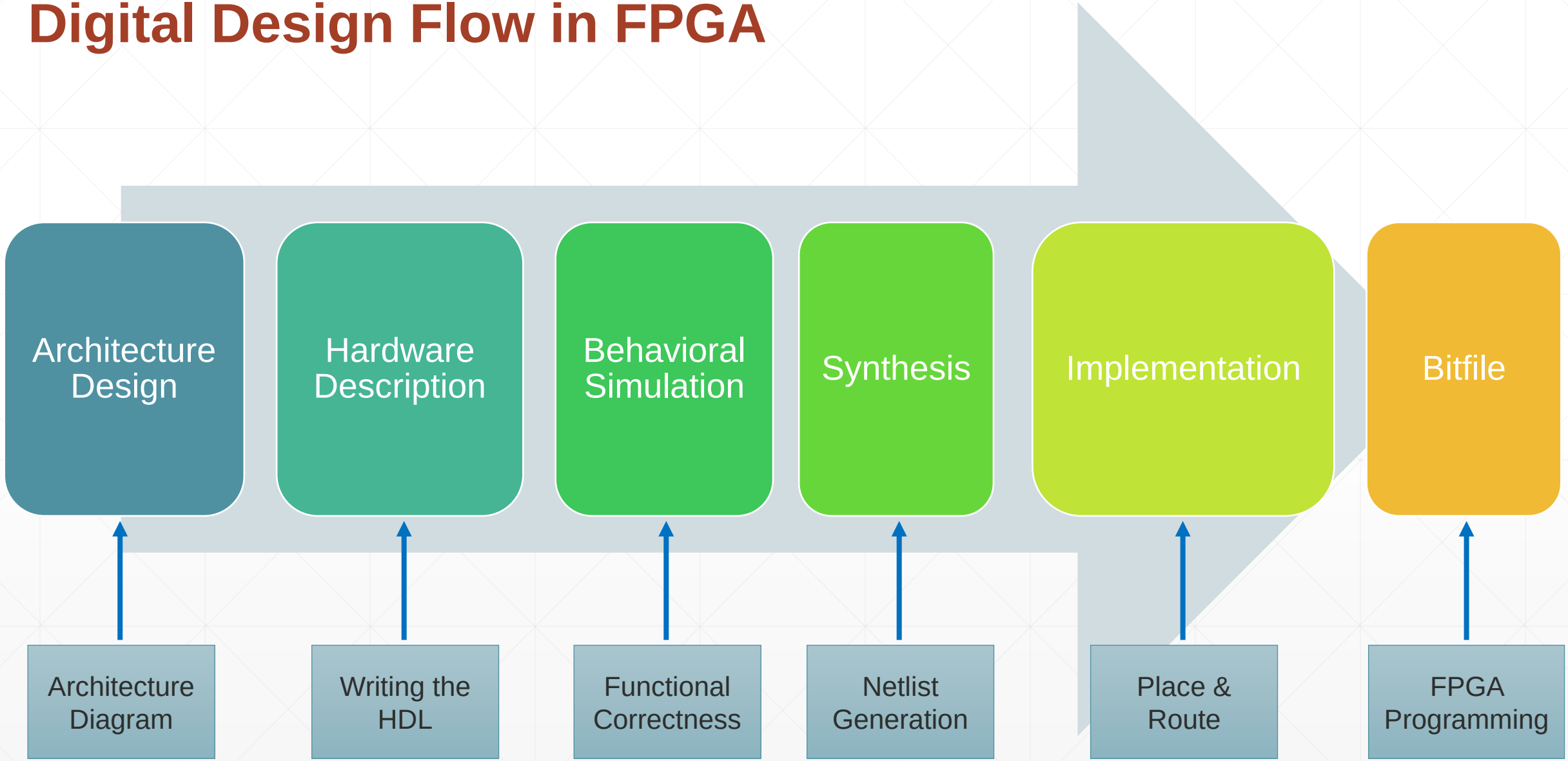
Digital Design Flow in FPGA



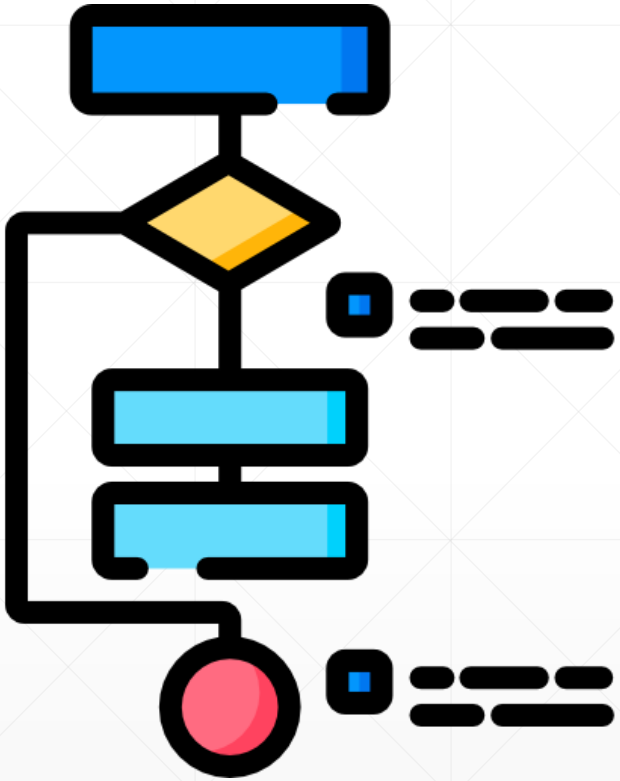
Digital Design Flow in FPGA



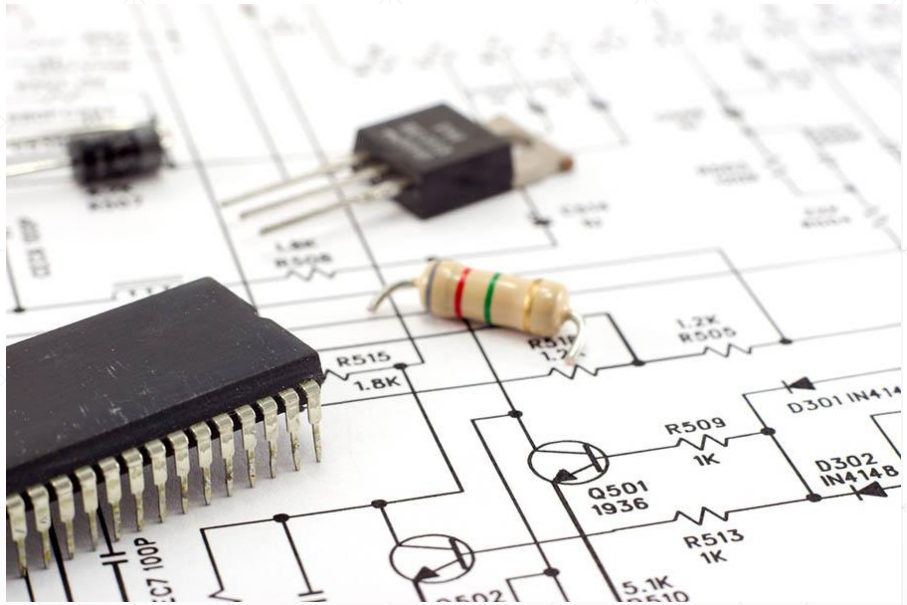
Digital Design Flow in FPGA



Architecture Design



Algorithm



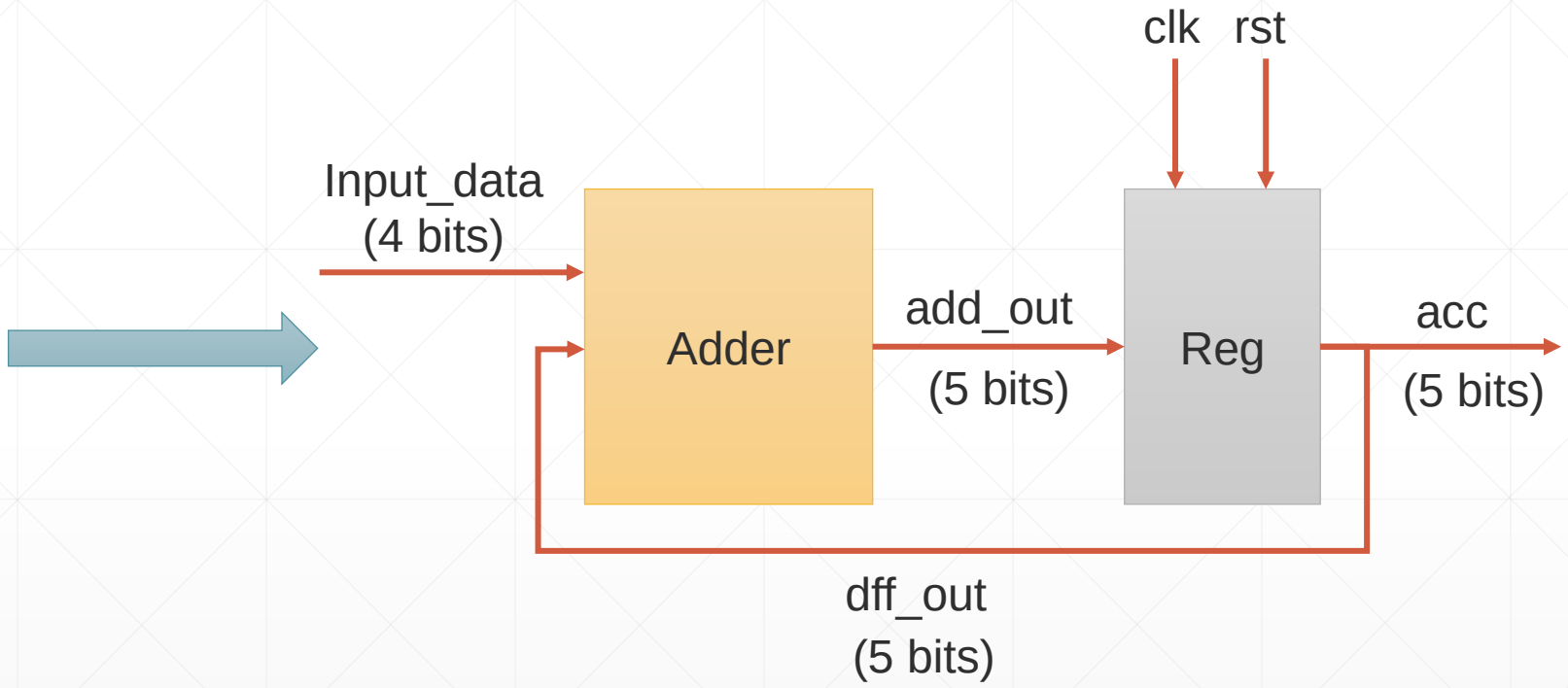
Hardware Design Architecture

- Design Datapath Elements
- Design Control Path

Architecture Design

```
# Accumulator Algorithm
accumulator = 0

# Loop for N data inputs
input_data = get_input() # Fetch new data

accumulator += input_data # Accumulate (accumulator = accumulator + input_data)

# Output the final accumulated value print(accumulator)
```

Algorithm of the Accumulator

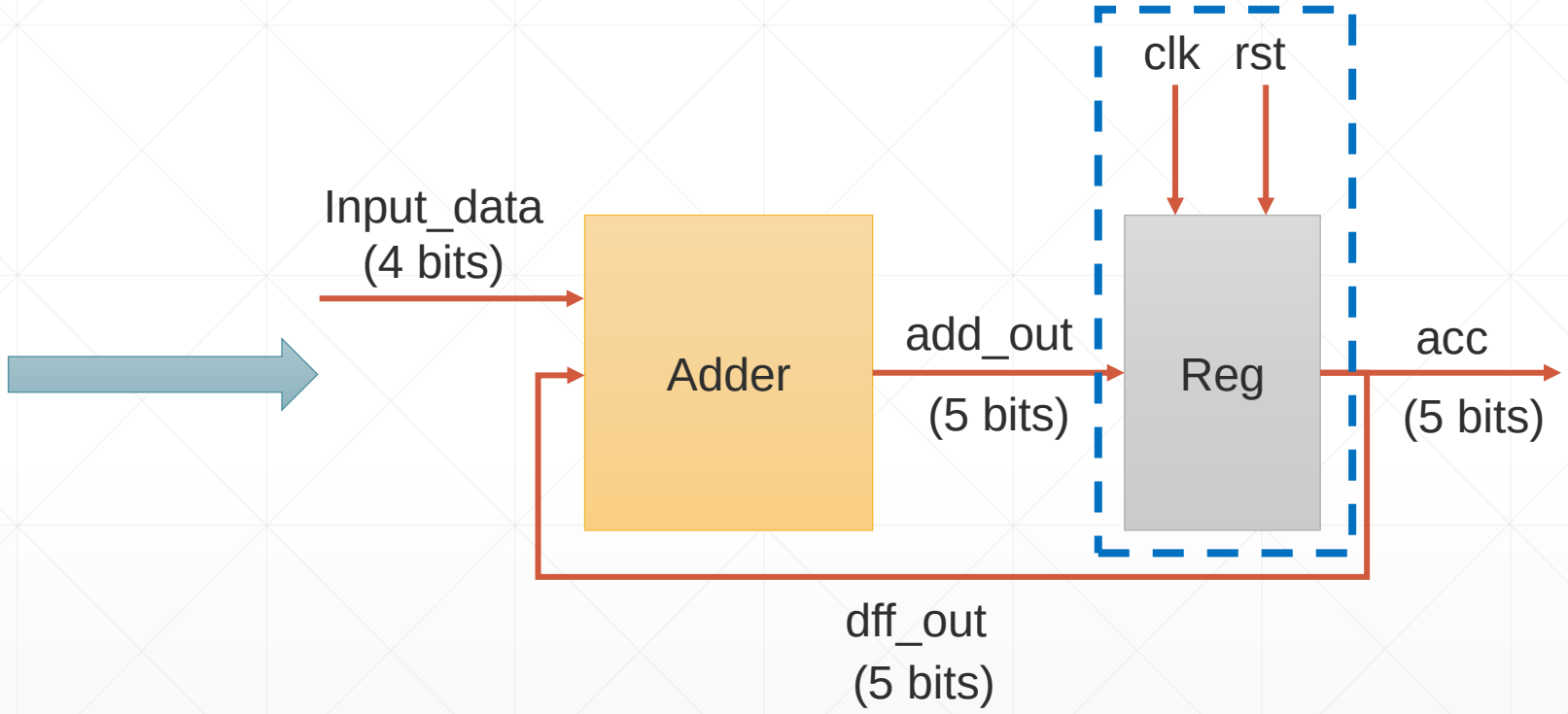


Hardware Design Architecture of the Accumulator

Architecture Design

```
# Accumulator Algorithm
accumulator = 0

# Loop for N data inputs
input_data = get_input() # Fetch new data

accumulator += input_data # Accumulate (accumulator = accumulator + input_data)

# Output the final accumulated value print(accumulator)
```

Algorithm of the Accumulator

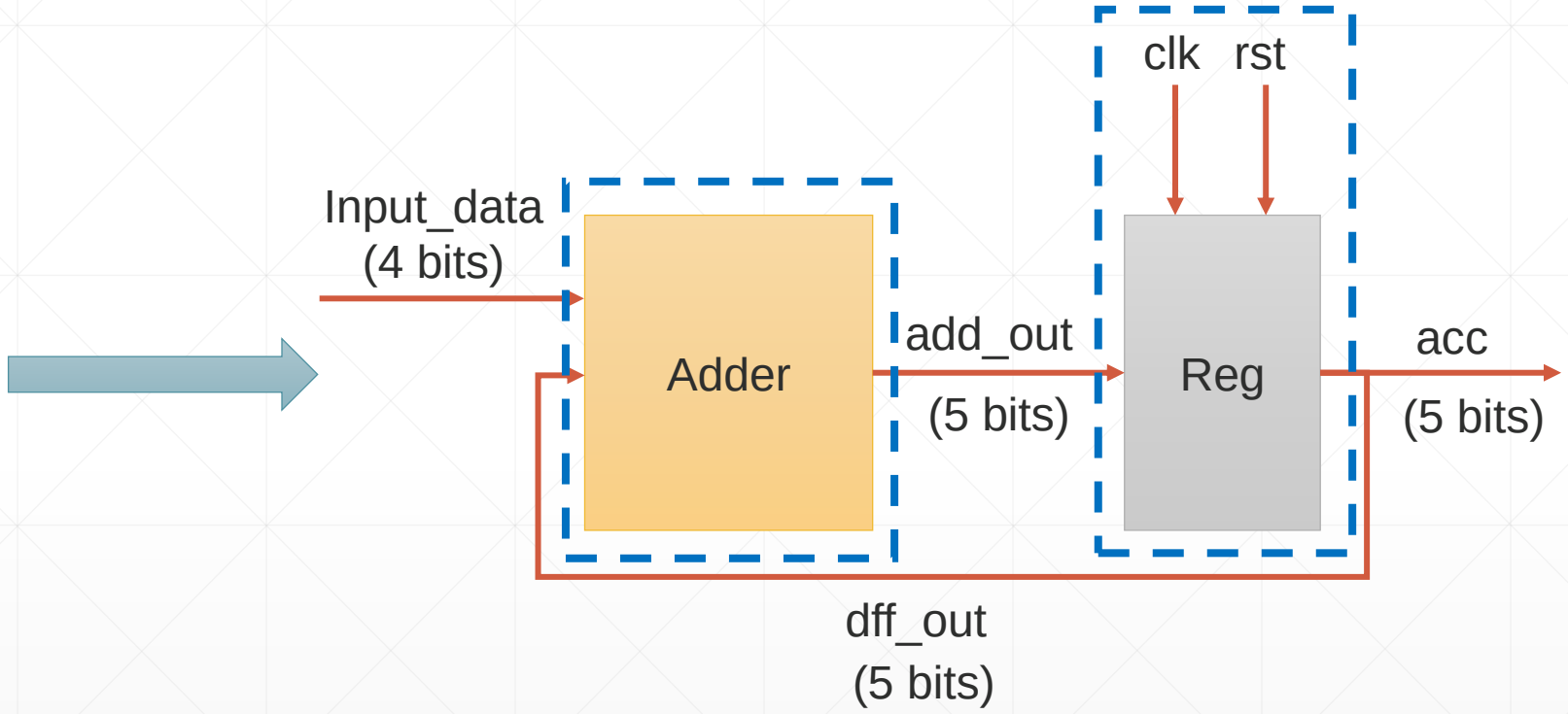


Hardware Design Architecture of the Accumulator

Architecture Design

```
# Accumulator Algorithm
accumulator = 0

# Loop for N data inputs
input_data = get_input() # Fetch new data

accumulator += input_data # Accumulate (accumulator = accumulator + input_data)

# Output the final accumulated value print(accumulator)
```

Algorithm of the Accumulator

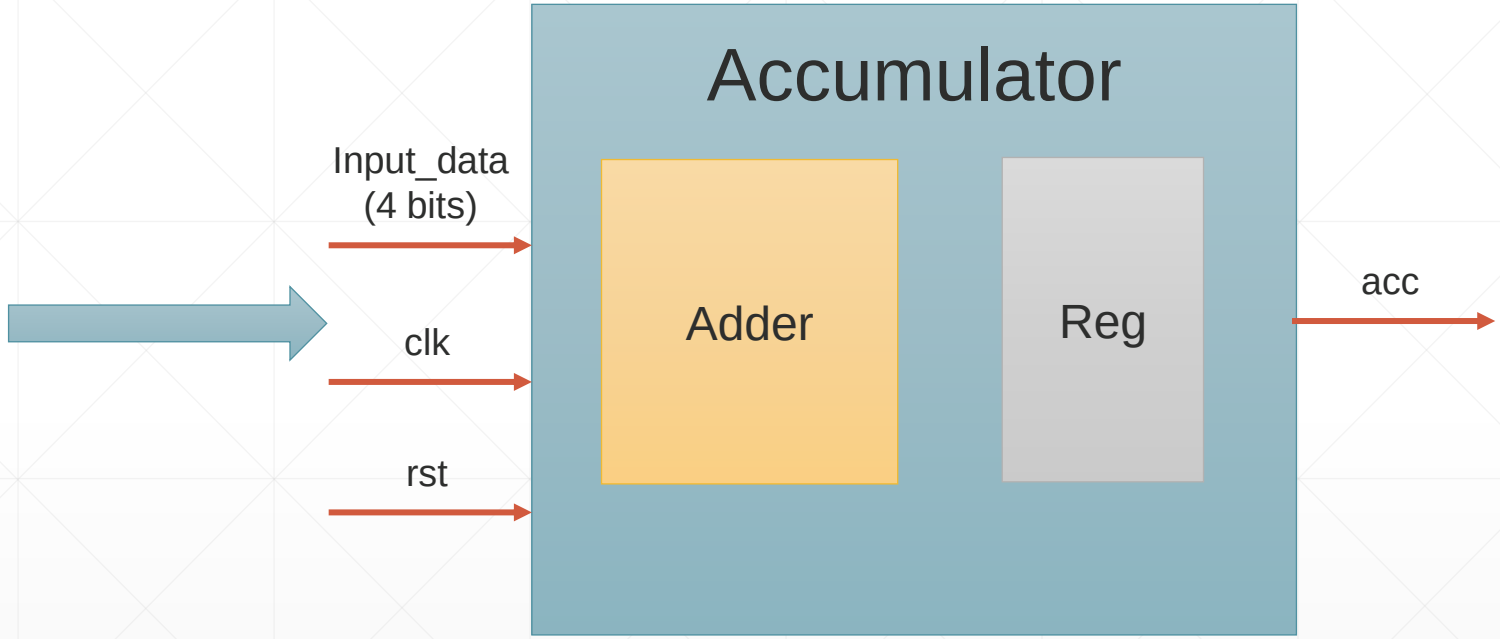


Hardware Design Architecture of the Accumulator

Architecture Design

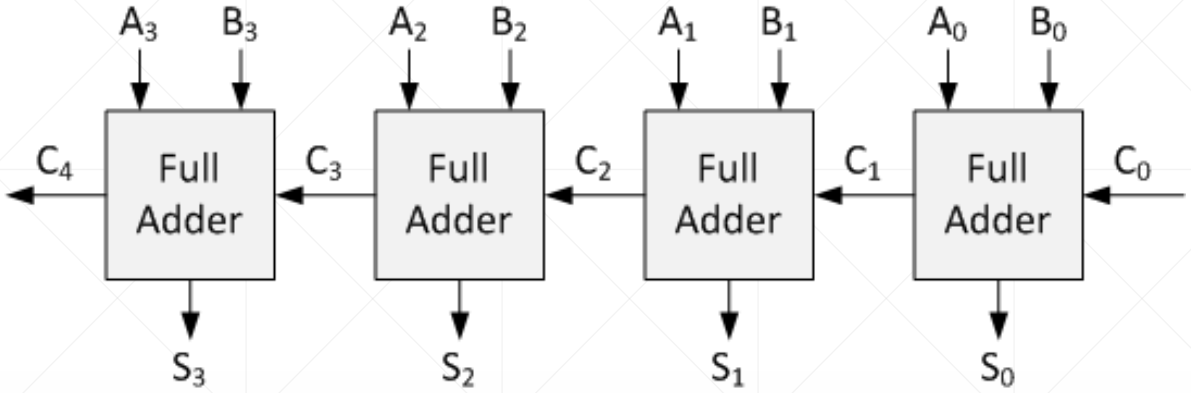
```
# Accumulator Algorithm
accumulator = 0

# Loop for N data inputs
input_data = get_input() # Fetch new data

accumulator += input_data # Accumulate (accumulator = accumulator + input_data)

# Output the final accumulated value print(accumulator)
```

Algorithm of the Accumulator

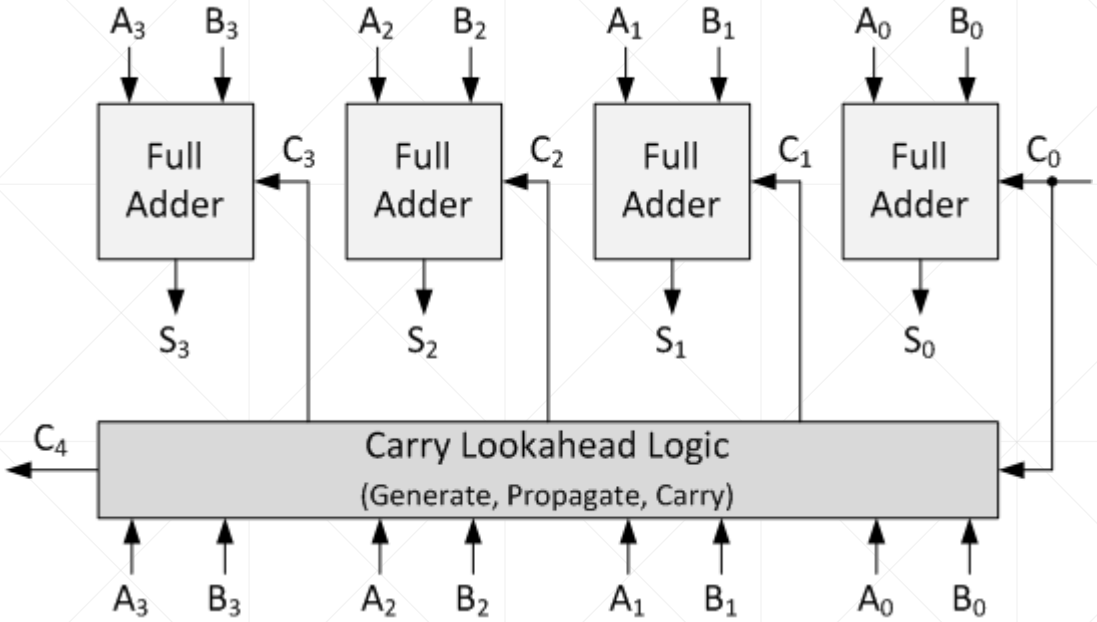


Hardware Design Architecture of the Accumulator

Datapath Element Design

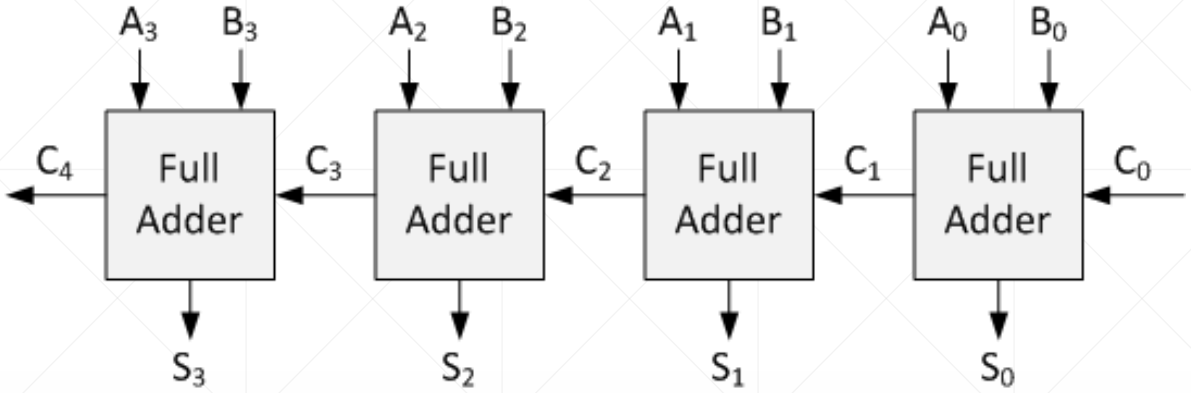


Ripple Carry Adder

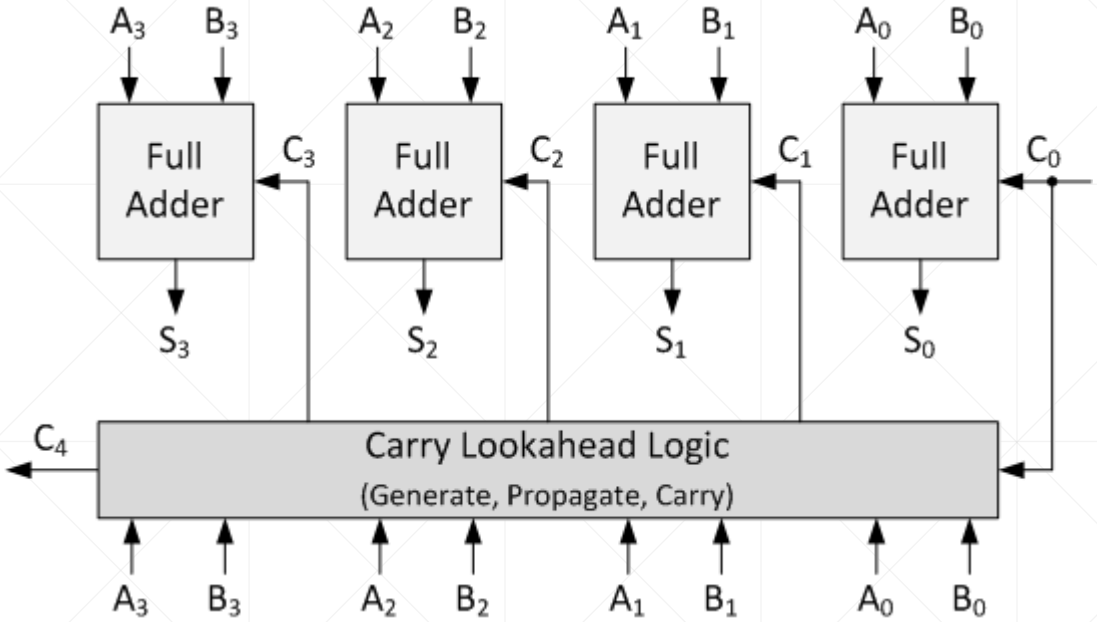


Carry Look Ahead Adder

Datapath Element Design



Ripple Carry Adder



Carry Look Ahead Adder

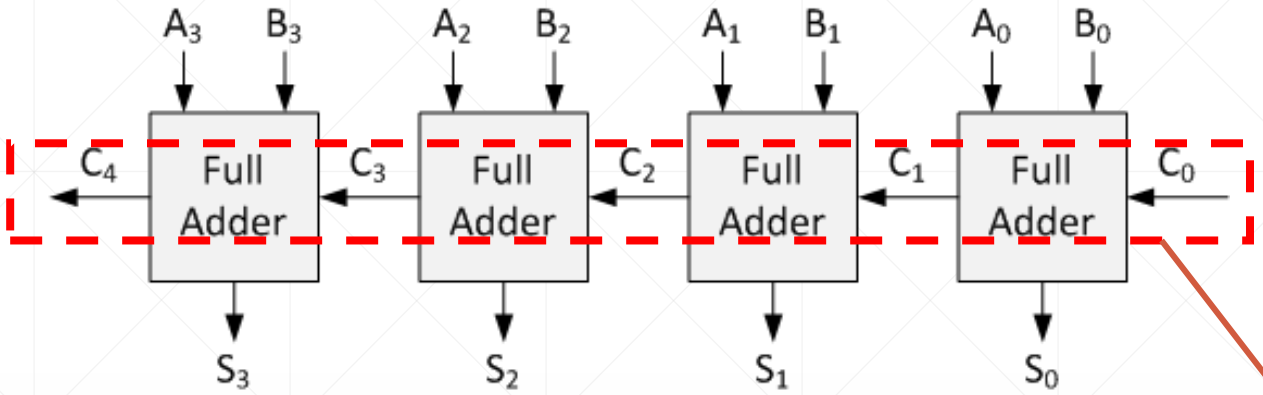
Pros:
- Simple design and easy to implement.
- Low area and power consumption for small bit-width adders.

Cons:
- Slow for large bit-widths due to the sequential carry propagation (linear delay).
- Propagation Delay: Increases linearly with the number of bits.

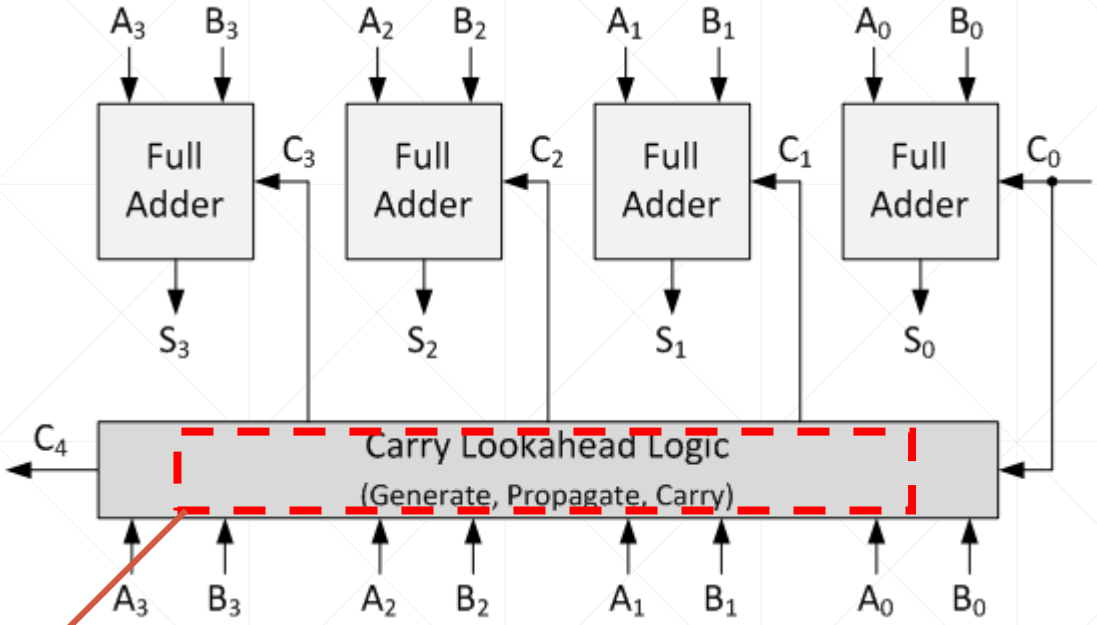
Pros:
- Much faster than RCA, especially for wide adders.
- Parallel carry computation reduces critical path delay.

Cons:
- More complex design (requires additional logic for generate/propagate).
- Higher area and power consumption due to extra gates.

Datapath Element Design



Ripple Carry Adder



Carry Look Ahead Adder

Critical Path

Pros:
- Simple design and easy to implement.
- Low area and power consumption for small bit-width adders.

Cons:
- Slow for large bit-widths due to the sequential carry propagation (linear delay).
- Propagation Delay: Increases linearly with the number of bits.

Pros:
- Much faster than RCA, especially for wide adders.
- Parallel carry computation reduces critical path delay.

Cons:
- More complex design (requires additional logic for generate/propagate).
- Higher area and power consumption due to extra gates.

Revisit GCD

- Input: Integers u and v
- Output: Greatest Common Divisor of u and v, z=gcd(u,v)
- while (u!=v) do
  - If (u and v are even)
  - z=2gcd(u/2,v/2)
  - else if (u is even and v is odd)
  - z=gcd(u/2,v)
  - else if(u is odd and v is even)
  - z=gcd(u,v/2)

```
else
    if(u ≥ v)
        z=gcd((u-v)/2,v)
    else
        z=gcd(u,(v-u)/2)
```

We need to realize a co-processor on FPGA to compute gcd of two given numbers

References: Hardware Security: Design, Threats, and Safeguards, CRC Press.

Revisit GCD

- Input: Integers u and v
- Output: Greatest Common Divisor of u and v, z=gcd(u,v)

```
register u and v

XR=u, YR=v, count=0

while (XR!=YR) do
    If (!XR[0] and !YR[0])
        XR=RIGHT_SHIFT(XR)
        YR=RIGHT_SHIFT(YR)
        Count=Count+1
    else if(XR[0] and !YR[0])
        YR=RIGHT_SHIFT(YR)
    else if(!XR[0] and YR[0])
        XR=RIGHT_SHIFT(XR)
    else
     if(XR ≥YR)
        XR=RIGHT_SHIFT(XR-YR)
    else
        YR=RIGHT_SHIFT(YR-XR)
while(count > 0)
    XR=LEFT_SHIFT(XR)
    count=count-1
```

References: Hardware Security: Design, Threats, and Safeguards, CRC Press.

Identification of the Data Path Elements

- Subtractor
- Complementer
- Right Shifter
- Left Shifter
- Counter
- Multiplexer:
  - Required in large numbers for the switching necessary for the computations done in the datapath.
  - Selection lines in the multiplexer are configured by the control circuitry, which is essentially a state machine.

References: Hardware Security: Design, Threats, and Safeguards, CRC Press.

Identification of the State Machine of the Control Path

- Control Path is a sequential design.
- It can be represented by a state machine.
- In this example, there are 6 states.
- The controller receives inputs from the partial computations of the datapath.
- Based on the current state and input, it performs state transitions.
- It also produces control signals which configures the datapath elements or switches the multiplexers to sequence the dataflow.

References: Hardware Security: Design, Threats, and Safeguards, CRC Press.

State Machine of the Controller

Present State	Next State					Output Signals										
	0____	100_	110_	101_	111_	*load uv*	*update* X_R	*update* Y_R	*load* X_R	*load* Y_R	*load_*X_R *after_sub*	*load_*Y_R *after_sub*	*Update counter*	*Inc /Dec*	*left shift*	*count zero*
S_0	S_5	S_1	S_2	S_3	S_4	1	0	0	1	1	0	0	0	_	_	_
S_1	S_5	S_1	S_2	S_3	S_4	0	1	1	1	1	0	0	1	1	_	_
S_2	S_5	S_1	S_2	S_3	S_4	0	0	1	0	1	0	0	0	_	_	_
S_3	S_5	S_1	S_2	S_3	S_4	0	1	0	1	0	0	0	0	_	_	_
S_4 $(X_R \geq Y_R)$	S_5	S_1	S_2	S_3	S_4	0	1	0	1	0	1	0	0	_	_	_
S_4 $(X_R < Y_R)$	S_5	S_1	S_2	S_3	S_4	0	0	1	0	1	0	1	0	_	_	_
S_5	S_5	S_5	S_5	S_5	S_5	0	0	0	0	0	0	0	1	0	1	0

There are 6 States of the Controller.
Controller receives 4 inputs from the data-path: [(XR!=YR), XR[0], YR[0], XR≥YR]

Example:

Present State: S0

load_uv=1, load_XR=load_YR=1.

Input=(0xxx)=>XR=YR=>Next State is S5.

Input=(100x)=>XR!=YR, both XR and YR are even=>Next State is S1.

State Machine of the Controller

Present State	Next State					Output Signals										
	0___	100_	110_	101_	111_	*load uv*	*update* X_R	*update* Y_R	*load* X_R	*load* Y_R	*load_*X_R *after_sub*	*load_*Y_R *after_sub*	*Update counter*	*Inc /Dec*	*left shift*	*count zero*
S_0	S_5	S_1	S_2	S_3	S_4	1	0	0	1	1	0	0	0	_	_	_
S_1	S_5	S_1	S_2	S_3	S_4	0	1	1	1	1	0	0	1	1	_	_
S_2	S_5	S_1	S_2	S_3	S_4	0	0	1	0	1	0	0	0	_	_	_
S_3	S_5	S_1	S_2	S_3	S_4	0	1	0	1	0	0	0	0	_	_	_
S_4 $(X_R \geq Y_R)$	S_5	S_1	S_2	S_3	S_4	0	1	0	1	0	1	0	0	_	_	_
S_4 $(X_R < Y_R)$	S_5	S_1	S_2	S_3	S_4	0	0	1	0	1	0	1	0	_	_	_
S_5	S_5	S_5	S_5	S_5	S_5	0	0	0	0	0	0	0	1	0	1	0

There are 6 States of the Controller.

Controller receives 4 inputs from the data-path: **[(XR!=YR), XR[0], YR[0], XR≥YR]**

Example:

Present State: S0

load_uv=1, load_XR=load_YR=1.

Input=(0xxx)=>XR=YR=>Next State is S5.

Input=(100x)=>XR!=YR, both XR and YR are even=>Next State is S1.

State Machine of the Controller

Present State	Next State					Output Signals										
	0____	100_	110_	101_	111_	*load uv*	*update* X_R	*update* Y_R	*load* X_R	*load* Y_R	*load_*X_R *after_sub*	*load_*Y_R *after_sub*	*Update counter*	*Inc /Dec*	*left shift*	*count zero*
S_0	S_5	S_1	S_2	S_3	S_4	1	0	0	1	1	0	0	0	_	_	_
S_1	S_5	S_1	S_2	S_3	S_4	0	1	1	1	1	0	0	1	1	_	_
S_2	S_5	S_1	S_2	S_3	S_4	0	0	1	0	1	0	0	0	_	_	_
S_3	S_5	S_1	S_2	S_3	S_4	0	1	0	1	0	0	0	0	_	_	_
S_4 $(X_R \geq Y_R)$	S_5	S_1	S_2	S_3	S_4	0	1	0	1	0	1	0	0	_	_	_
S_4 $(X_R < Y_R)$	S_5	S_1	S_2	S_3	S_4	0	0	1	0	1	0	1	0	_	_	_
S_5	S_5	S_5	S_5	S_5	S_5	0	0	0	0	0	0	0	1	0	1	0

Present State: S1

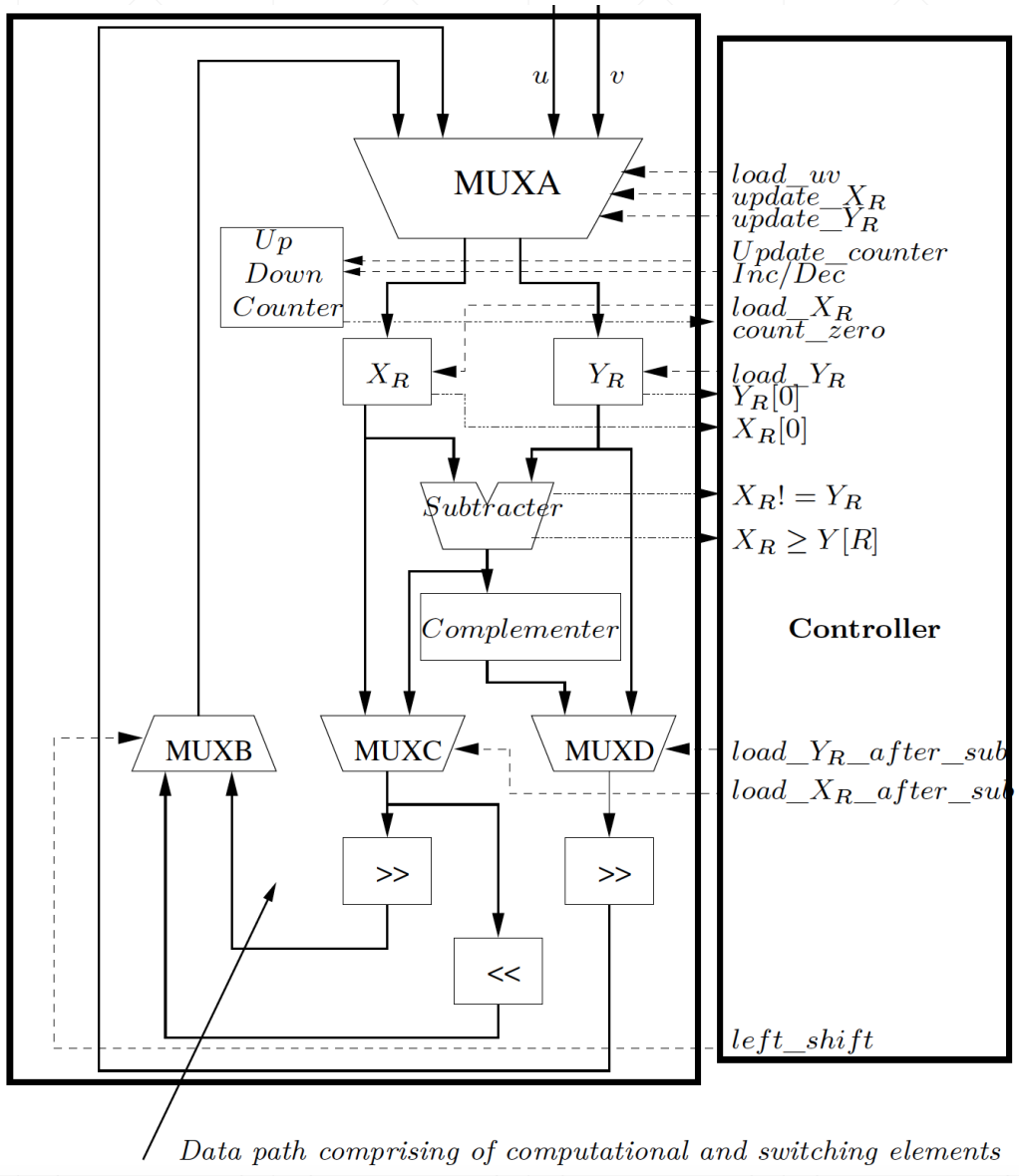
load_uv=0, load_XR=load_YR=1, update_XR=update_YR=1, Update Counter=1, Inc/Dec=1.

Input=(0xxx)=>XR=YR=>Next State is S5.

Input=(100x)=>XR!=YR, both XR and YR are even=>Next State is S1.

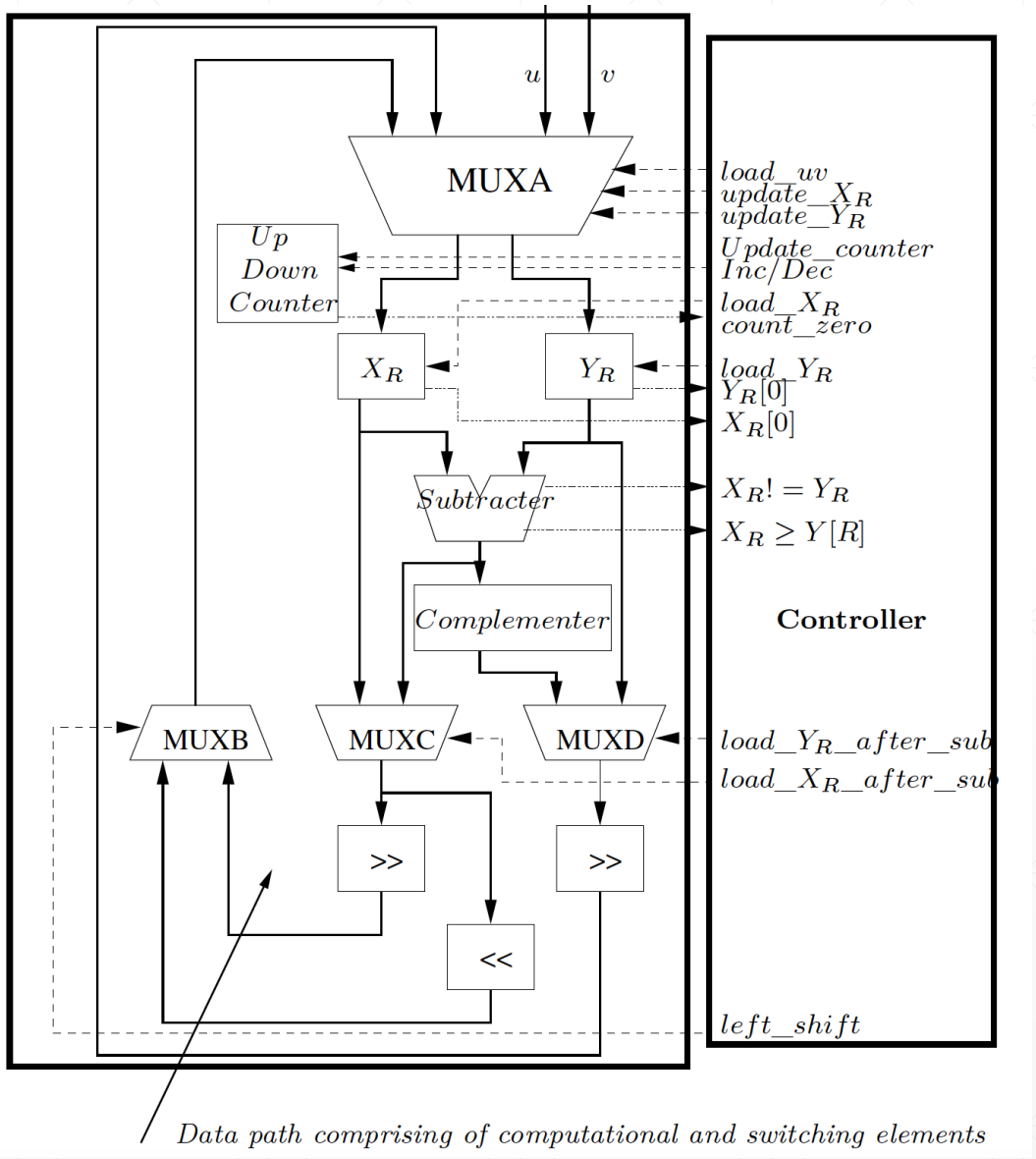
Input=(110x)=>XR!=YR, XR is odd, YR is even=>Next State is S2.

State Machine of the Controller



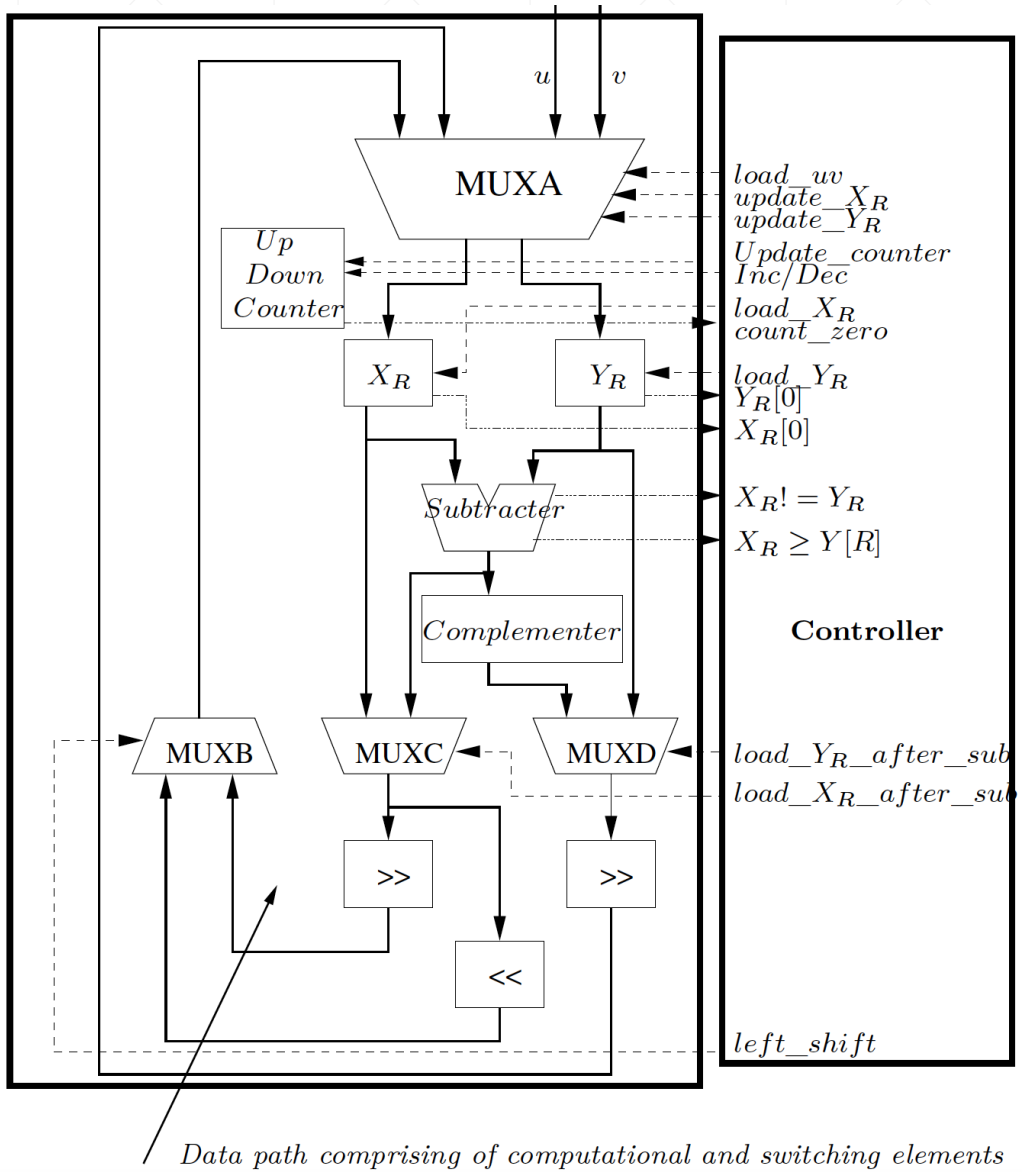
1. State Initialization and State Transition
2. Next State Logic
3. Control Signal Logic at Present State

State Machine of the Controller



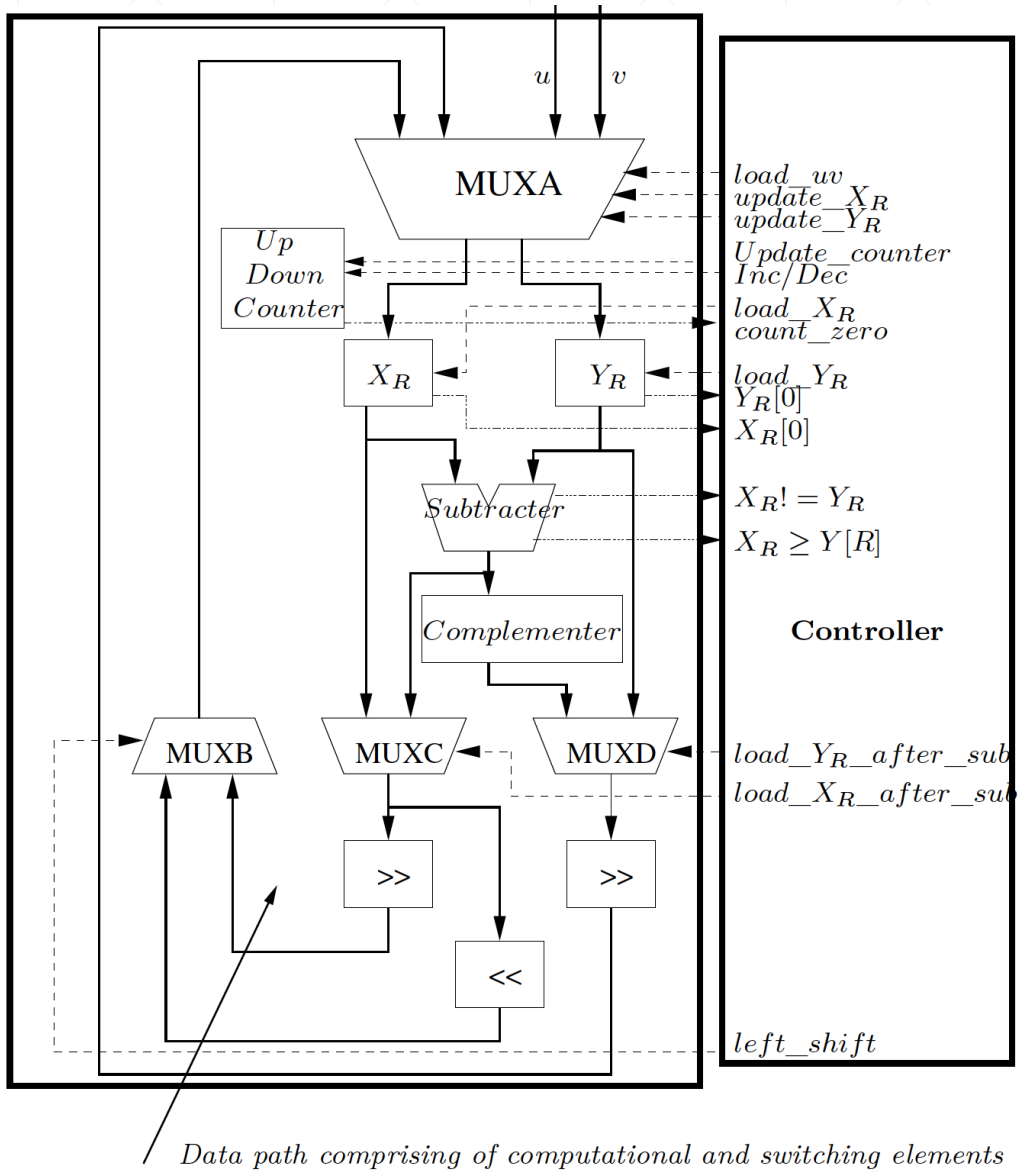

```
always @(posedge clk)
    begin
        if(reset)
            state<=S0;
        else
            state<=next_state;
    end
endmodule
```

State Machine of the Controller




```
localparam S0 = 3'd0;
localparam S1 = 3'd1;
localparam S2 = 3'd2;
localparam S3 = 3'd3;
localparam S4 = 3'd4;
localparam S5 = 3'd5;
always @(state or in)
    begin
        if(!in[0])
            next_state <=S5;
        else if(in[0] && in[3]==0 && in[2]==0)
            next_state <= S1;
        else if(in[0] && in[3]==1 && in[2]==0)
            next_state <= S2;
        else if(in[0] && in[3]==0 && in[2]==1)
            next_state <= S3;
        else if(in[0] && in[3]==1 && in[2]==1)
            next_state <= S4;
        else if(state == S5)
            next_state <= S5;
        else
            next_state <= S0;
        if(state == S5)
            next_state <= S5;
    end
```

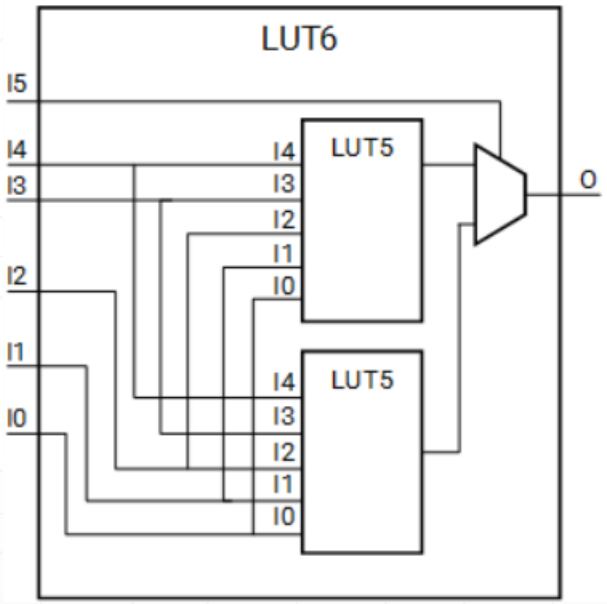
State Machine of the Controller




```
always @(state or in or done or cnt_zero)
   begin
      case(state)
         S0:
            begin
               ld_in1_in2<=1;
               update_Xr<=0;
               update_Yr<=0;
               ld_Xr<=1;
               ld_Yr<=1;
               ld_Xr_aftr_subs<=0;
               ld_Yr_aftr_subs<=0;
               lShift<=0;
            end
         S1:
            begin
               ld_in1_in2<=0;
               update_Xr<=1;
               update_Yr<=1;
               update_Counter<=1;
               incr_dec<=1;
               ld_Xr<=1;
               ld_Yr<=1;
               ld_Xr_aftr_subs<=0;
               ld_Yr_aftr_subs<=0;
               lShift<=0;
            end
```

```
         S2:
            begin
               ld_in1_in2<=0;
               update_Xr<=0;
               update_Yr<=1;
               update_Counter<=0;
               incr_dec<=0;
               ld_Xr<=0;
               ld_Yr<=1;
               ld_Xr_aftr_subs<=0;
               ld_Yr_aftr_subs<=0;
               lShift<=0;
            end
         S3:
            begin
               ld_in1_in2<=0;
               update_Xr<=1;
               update_Yr<=0;
               update_Counter<=0;
               incr_dec<=0;
               ld_Xr<=1;
               ld_Yr<=0;
               ld_Xr_aftr_subs<=0;
               ld_Yr_aftr_subs<=0;
               lShift<=0;
            end
```

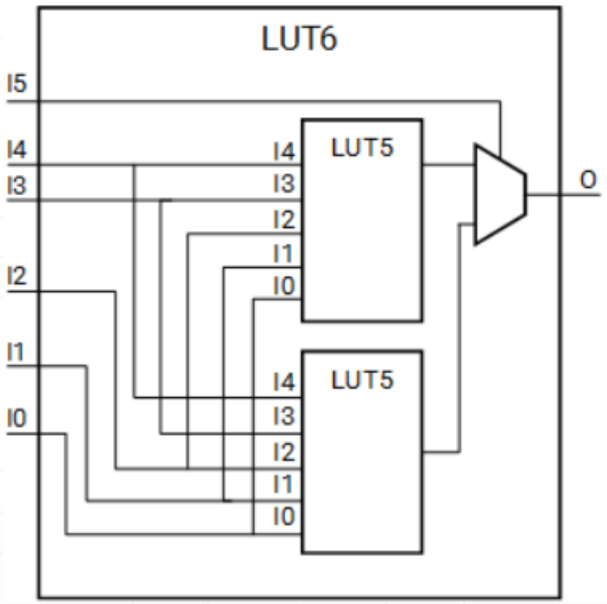
Lookup Tables



6-Input Look-Up Table with General Output

6-input, 1-output look-up table (LUT) that can either act as an asynchronous 64-bit ROM (with 6-bit addressing) or implement any 6-input logic function.

Lookup Tables



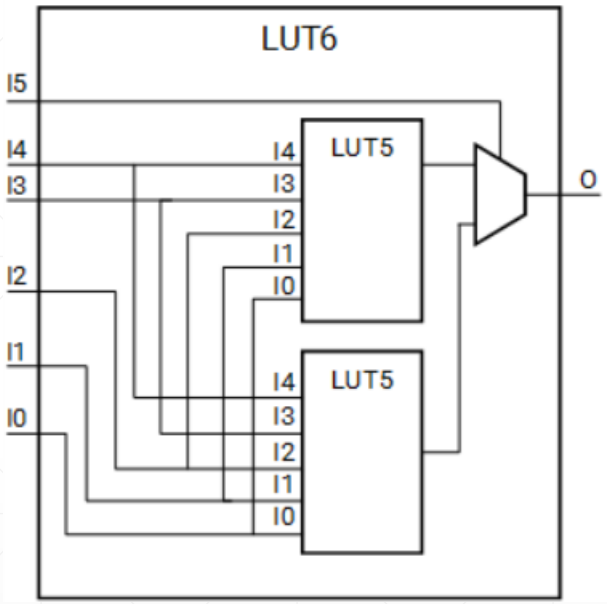
6-Input Look-Up Table with General Output

6-input, 1-output look-up table (LUT) that can either act as an asynchronous 64-bit ROM (with 6-bit addressing) or implement any 6-input logic function.

SRAM Cells: Each bit in a LUT is stored in a small SRAM cell that holds either a '0' or a '1'.

The LUT acts like a simple **multiplexer (MUX)**, selecting the output based on the current input combination. The stored SRAM bits are the MUX's inputs.

Lookup Tables



6-Input Look-Up Table with General Output

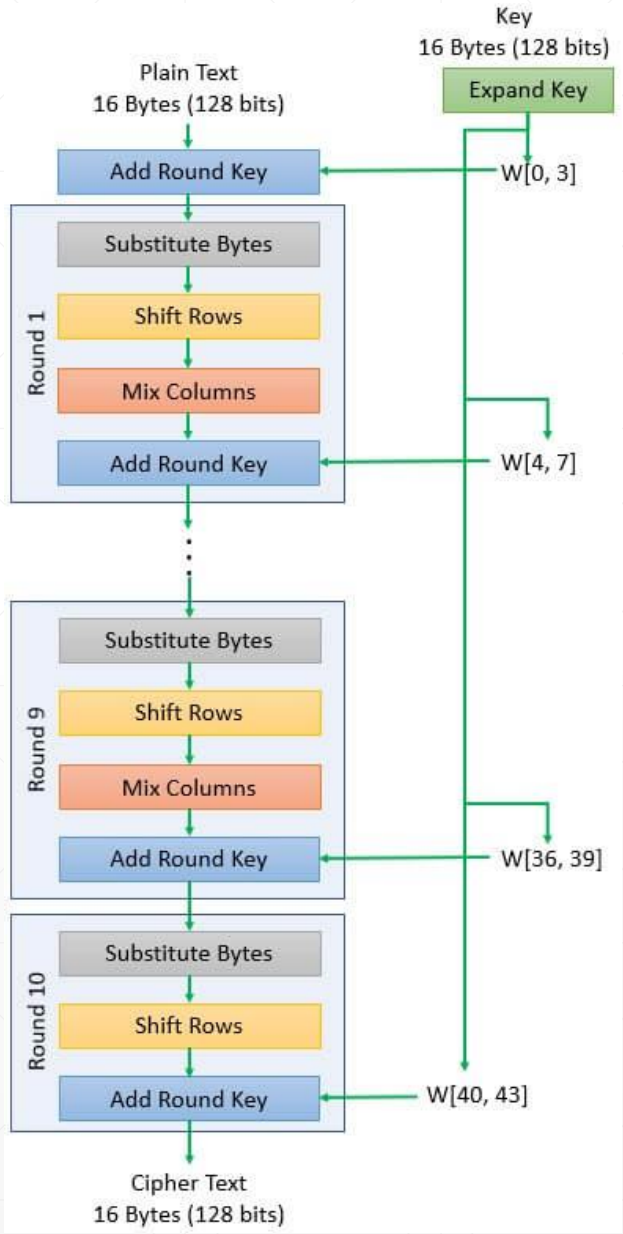
6-input, 1-output look-up table (LUT) that can either act as an asynchronous 64-bit ROM (with 6-bit addressing) or implement any 6-input logic function.

SRAM Cells: Each bit in a LUT is stored in a small SRAM cell that holds either a '0' or a '1'.

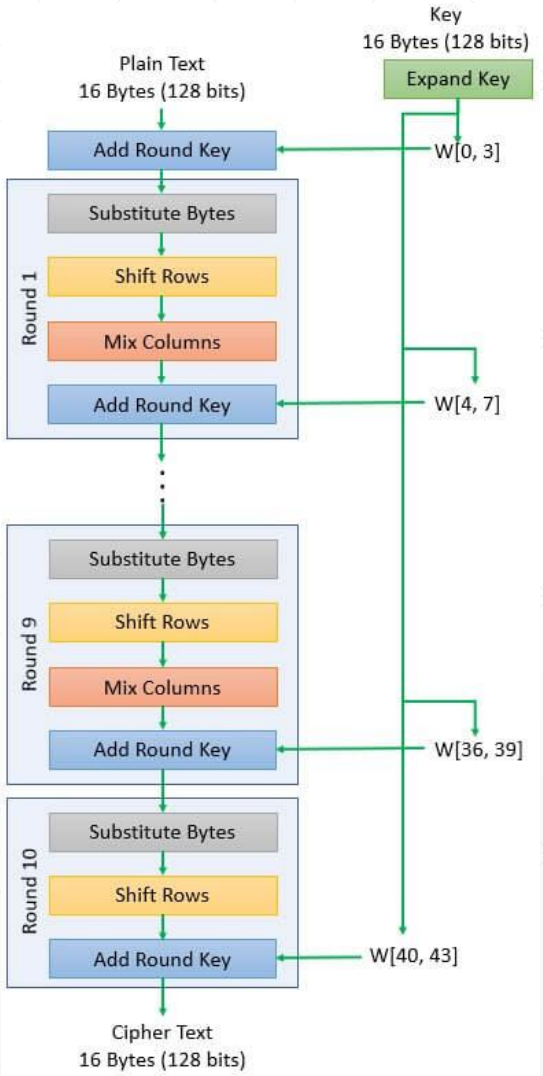
The LUT acts like a simple **multiplexer (MUX)**, selecting the output based on the current input combination. The stored SRAM bits are the MUX's inputs.

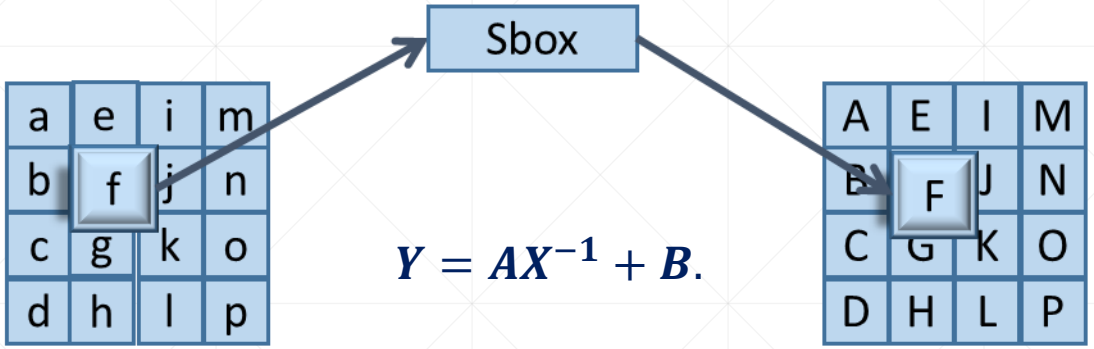
Some FPGAs allow LUTs to be used as small RAMs (**LUTRAM**), where dynamic data can be written during operation.

AES Algorithm



Sub Byte Operation

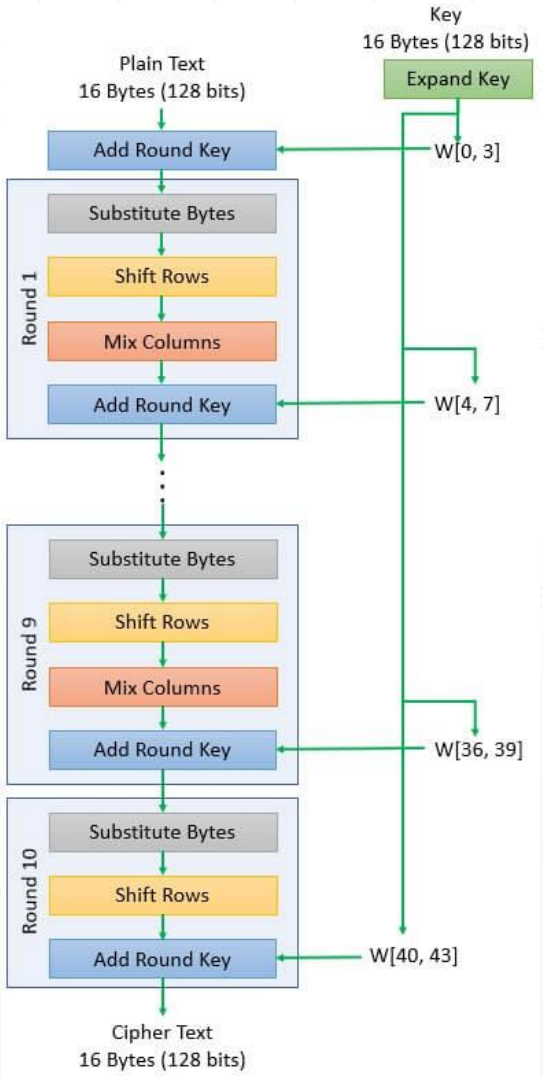


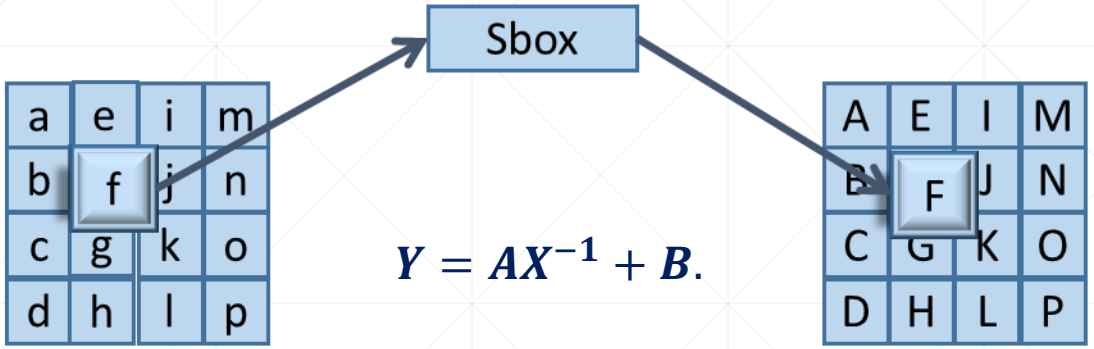


$$Y = AX^{-1} + B.$$

```
module sbox(a,c);

input  [7:0] a;
output [7:0] c;

reg [7:0] c;


  always @(a)
   case (a)
     8'h00: c=8'h63;
                    8'h01: c=8'h7c;
                    8'h02: c=8'h77;
                    8'h03: c=8'h7b;
                    8'h04: c=8'hf2;
                    8'h05: c=8'h6b;
                    8'h06: c=8'h6f;
```

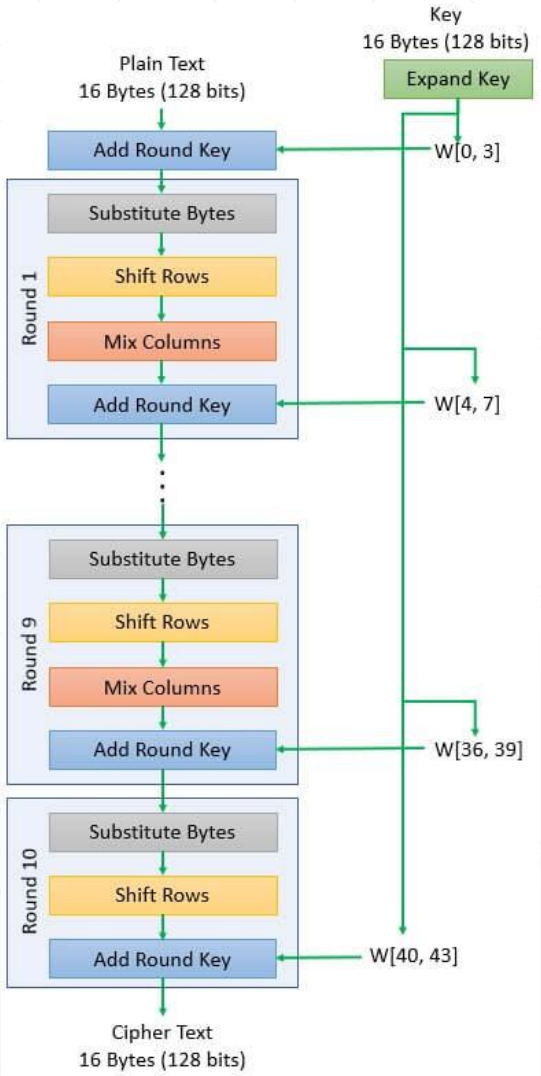
Sub Byte Operation






```
module sbox(input[7:0] sboxIn,output[7:0] sboxOut);
wire[7:0] inverterOut,Aout;

inverter invMod(sboxIn,inverterOut);

A_mult Amod(inverterOut,Aout);

assign sboxOut[7]=Aout[7];
assign sboxOut[6]=Aout[6]^1'b1;
assign sboxOut[5]=Aout[5]^1'b1;
assign sboxOut[4]=Aout[4];
assign sboxOut[3]=Aout[3];
assign sboxOut[2]=Aout[2];
assign sboxOut[1]=Aout[1]^1'b1;
assign sboxOut[0]=Aout[0]^1'b1;

endmodule
```

Shift Row Operation



a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

⊕

k_0	k_4	k_8	k_{12}
k_1	k_5	k_9	k_{13}
k_2	k_6	k_{10}	k_{14}
k_3	k_7	k_{11}	k_{15}

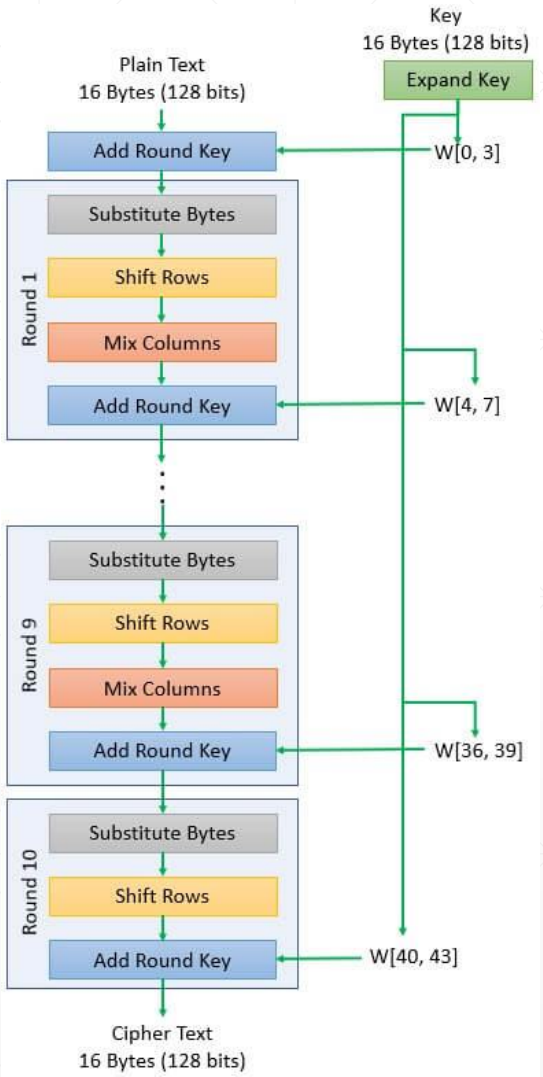
→

$a+k_0$	$e+k_4$	$i+k_8$	$m+k_{12}$
$b+k_1$	$f+k_5$	$j+k_9$	$n+k_{13}$
$c+k_2$	$g+k_6$	$k+k_{10}$	$o+k_{14}$
$d+k_3$	$h+k_7$	$l+k_{11}$	$p+k_{15}$

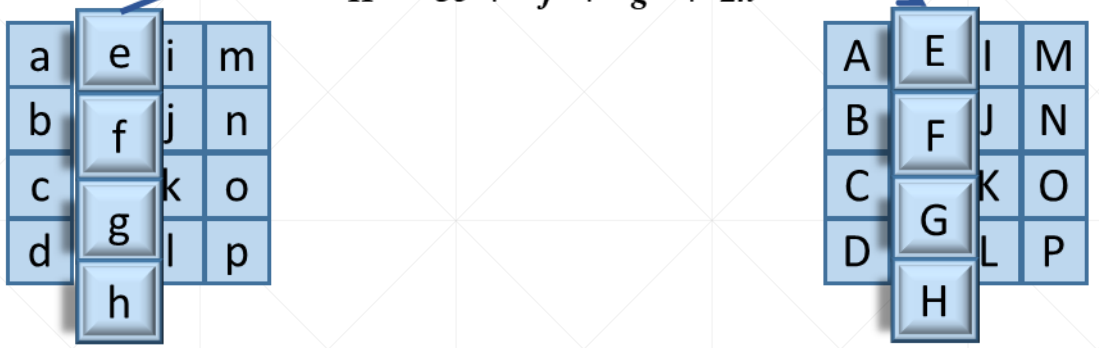
```
assign      sr[127:120] = sb[127:120];
assign      sr[119:112] = sb[87:80];
assign      sr[111:104] = sb[47:40];
assign      sr[103:96] = sb[7:0];

assign       sr[95:88] = sb[95:88];
assign      sr[87:80] = sb[55:48];
assign      sr[79:72] = sb[15:8];
assign      sr[71:64] = sb[103:96];
```

```
assign      sr[63:56] = sb[63:56];
assign      sr[55:48] = sb[23:16];
assign      sr[47:40] = sb[111:104];
assign      sr[39:32] = sb[71:64];

assign      sr[31:24] = sb[31:24];
assign      sr[23:16] = sb[119:112];
assign      sr[15:8] = sb[79:72];
assign      sr[7:0] = sb[39:32];
```

Mix Column Operation

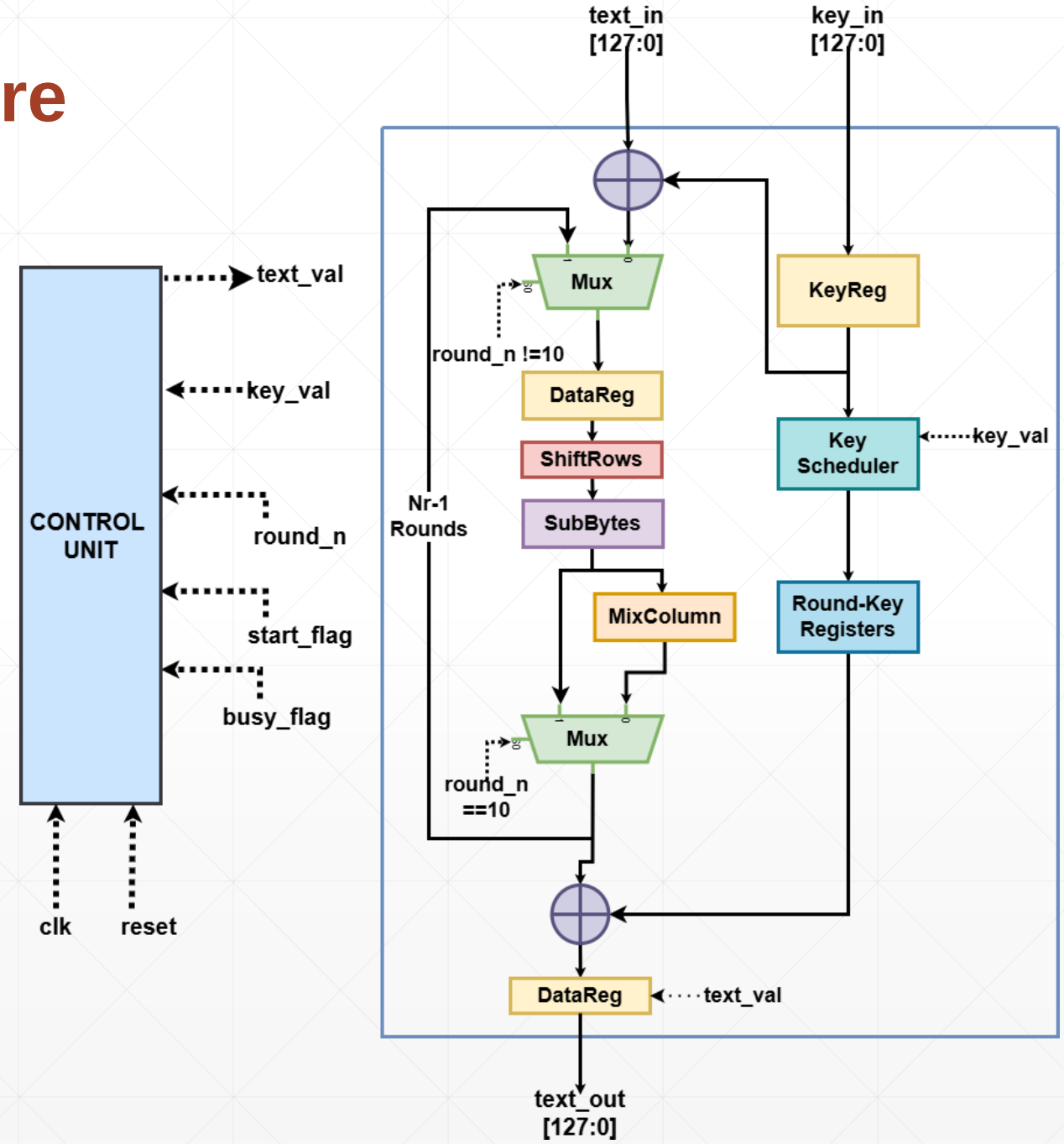


$$E = 2e + 3f + g + h$$
$$F = e + 2f + 3g + h$$
$$G = e + f + 2g + 3h$$
$$H = 3e + f + g + 2h$$

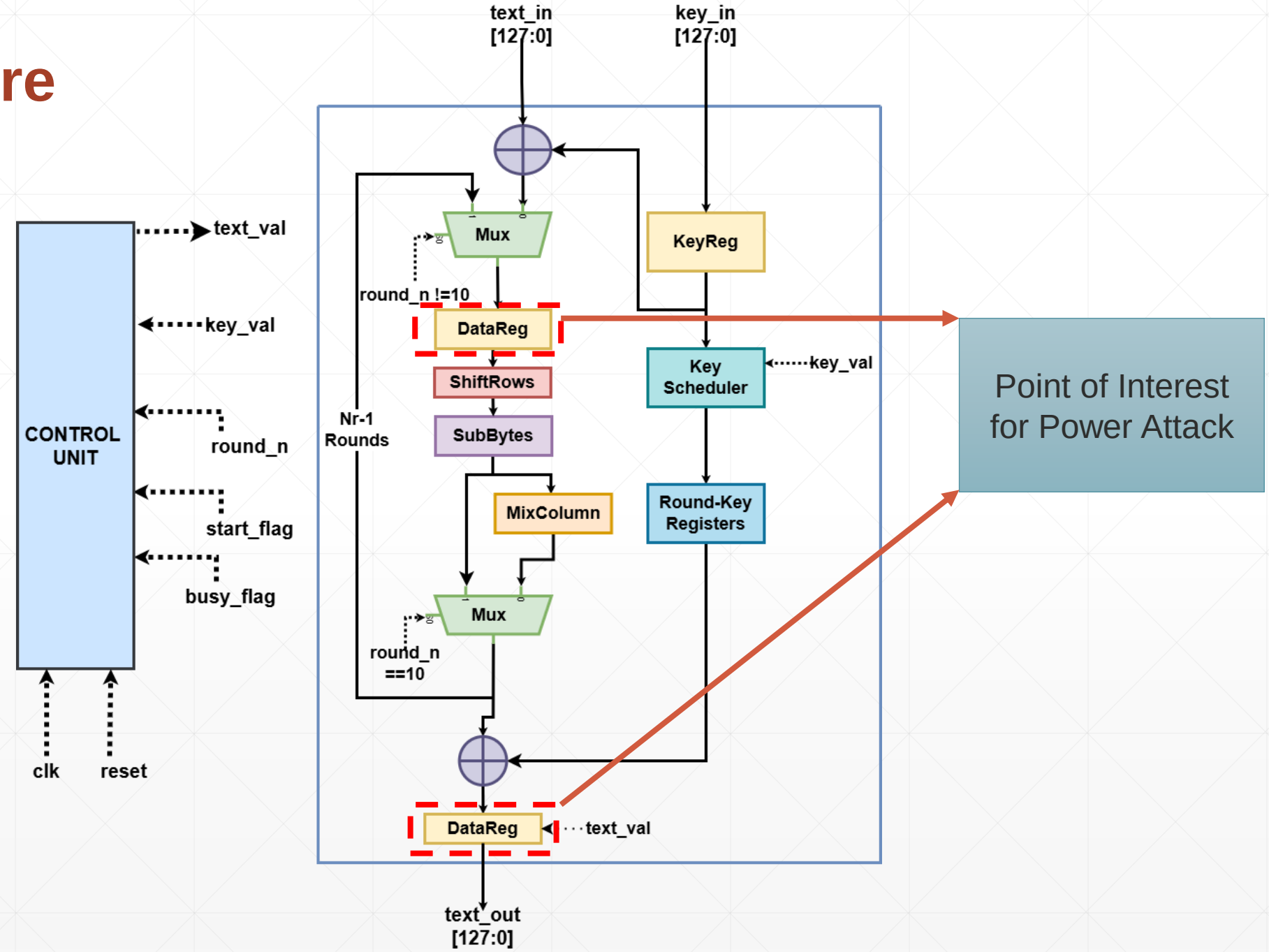



```
function [7:0] mixcolumn32;
input [7:0] i1,i2,i3,i4;
begin
mixcolumn32[7]=i1[6] ^ i2[6] ^ i2[7] ^ i3[7] ^ i4[7];
mixcolumn32[6]=i1[5] ^ i2[5] ^ i2[6] ^ i3[6] ^ i4[6];
mixcolumn32[5]=i1[4] ^ i2[4] ^ i2[5] ^ i3[5] ^ i4[5];
mixcolumn32[4]=i1[3] ^ i1[7] ^ i2[3] ^ i2[4] ^ i2[7] ^ i3[4] ^ i4[4];
mixcolumn32[3]=i1[2] ^ i1[7] ^ i2[2] ^ i2[3] ^ i2[7] ^ i3[3] ^ i4[3];
mixcolumn32[2]=i1[1] ^ i2[1] ^ i2[2] ^ i3[2] ^ i4[2];
mixcolumn32[1]=i1[0] ^ i1[7] ^ i2[0] ^ i2[1] ^ i2[7] ^ i3[1] ^ i4[1];
mixcolumn32[0]=i1[7] ^ i2[7] ^ i2[0] ^ i3[0] ^ i4[0];
end
endfunction
```

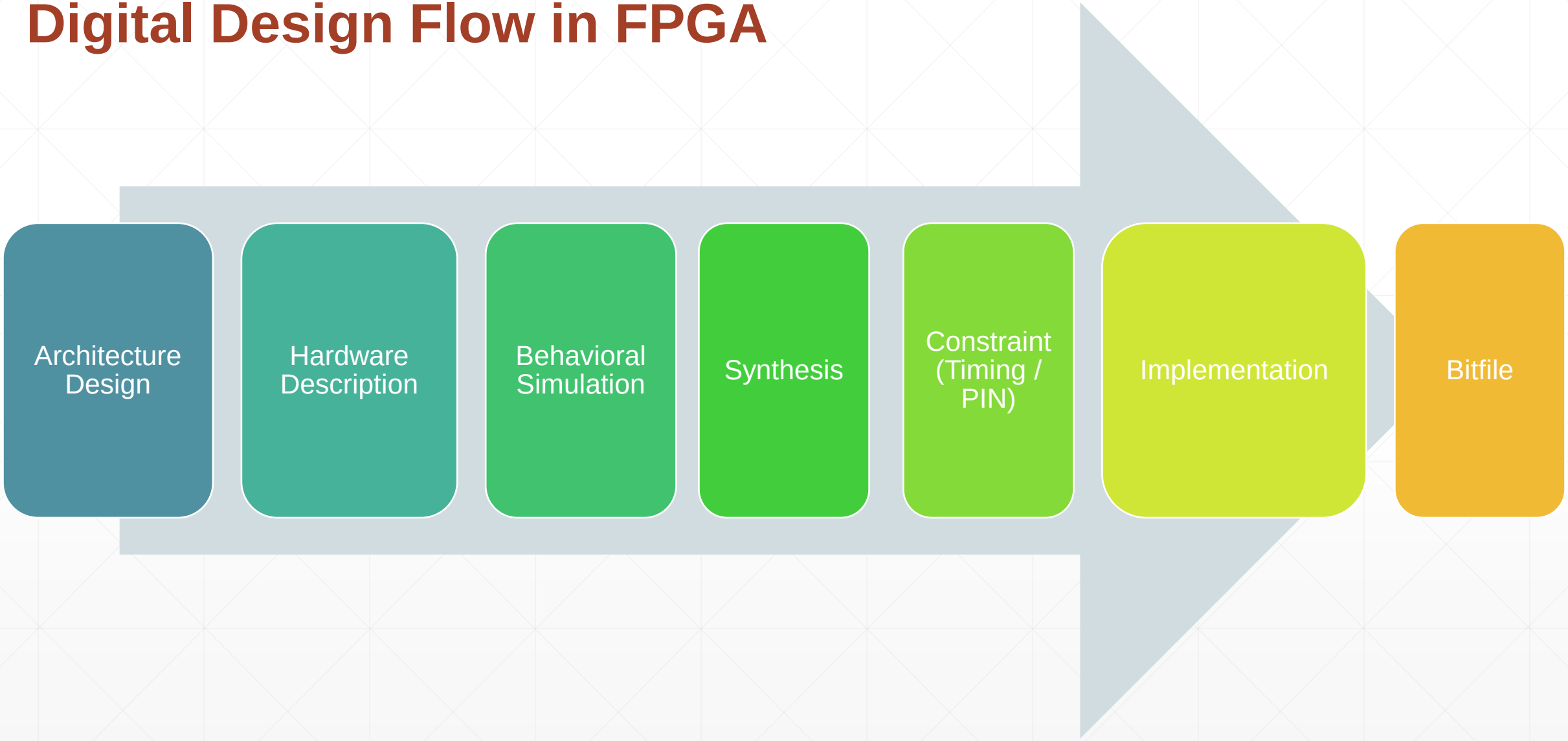
AES Hardware



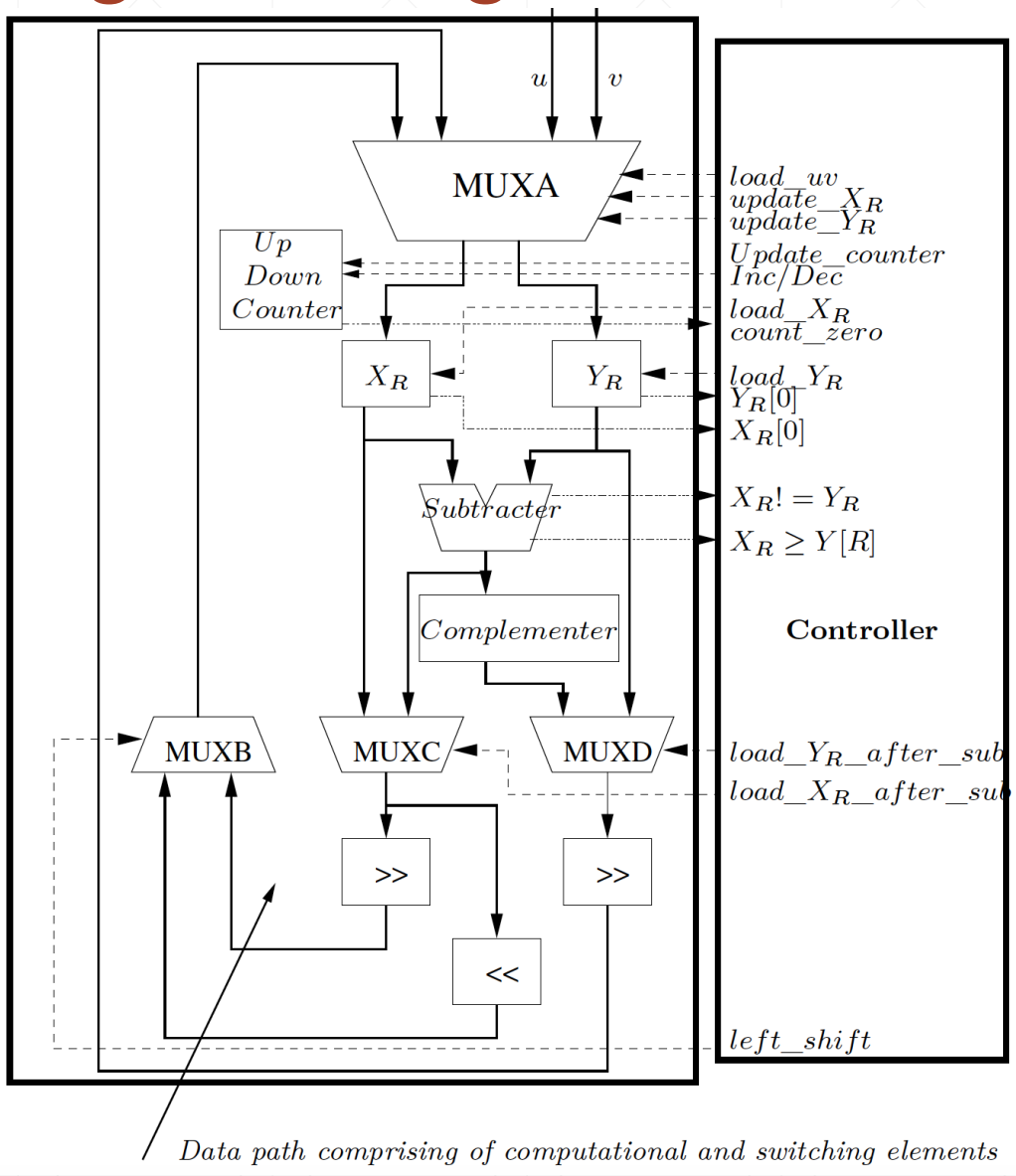
AES Hardware



Digital Design Flow in FPGA

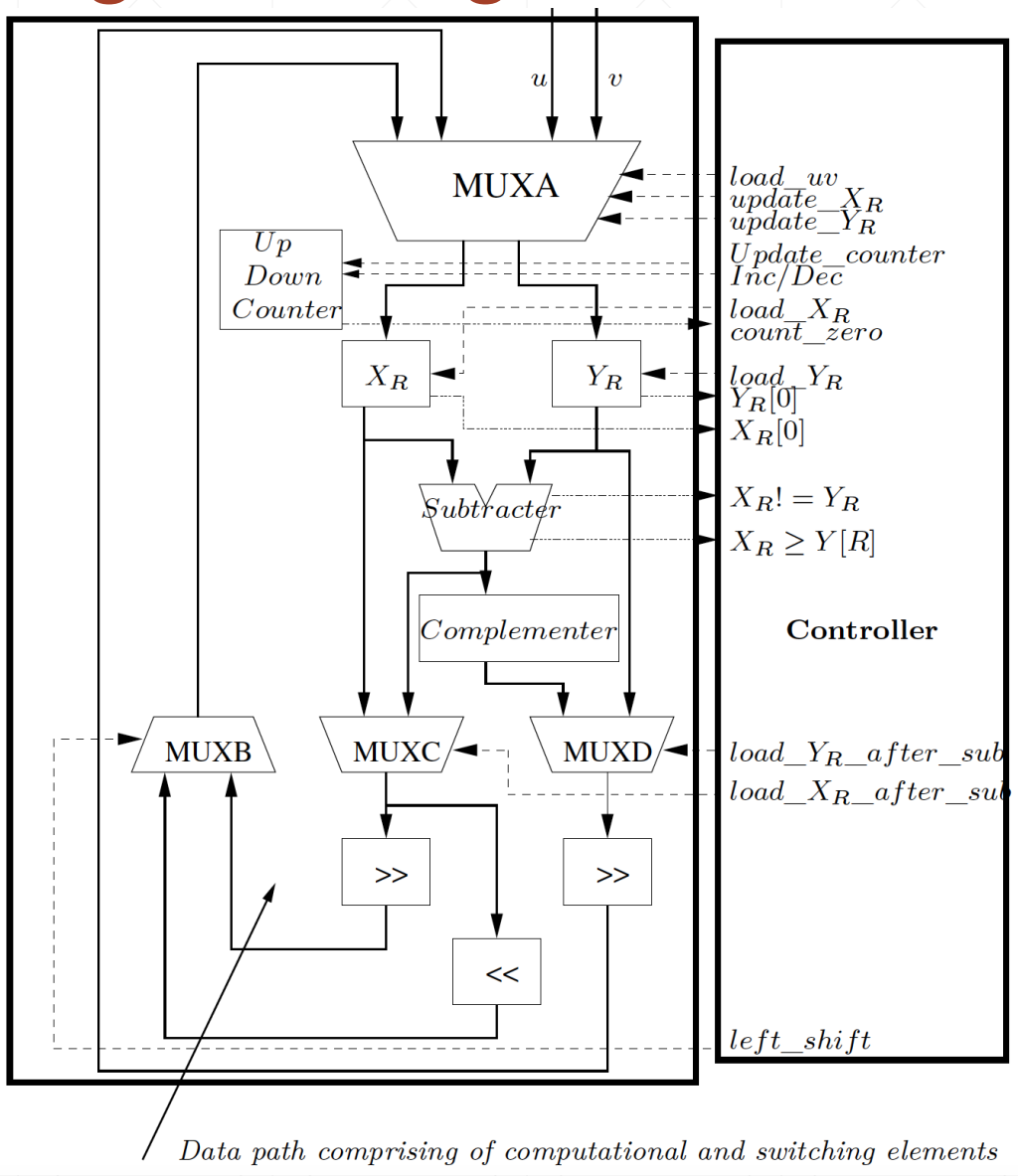


Digital Design Flow in FPGA




```
module gcd
#(parameter WIDTH =32) // Bit
width
(
input[WIDTH-1:0] in1,
input[WIDTH-1:0] in2,
input clk,
input reset,
output reg[WIDTH-1:0] out,
output reg done
   );
```

Digital Design Flow in FPGA




```
module gcd
#(parameter WIDTH =32) // Bit
width
(
input[WIDTH-1:0] in1,
input[WIDTH-1:0] in2,
input clk,
input reset,
output reg[WIDTH-1:0] out,
output reg done
    );
```

```
create_clock -period 2.2 -name clk -
waveform {0.000 1.1} [get_ports {clk}]
```

Thank You