

# Introduction to CMOS VLSI Design

## The Essence of Logic Design using Verilog HDL

Young-Don Bae, Ph.D.  
([ceo@donny.co.kr](mailto:ceo@donny.co.kr))



Chungnam National University

**CMOS VLSI Design**

**Slide 1**

## Part – I



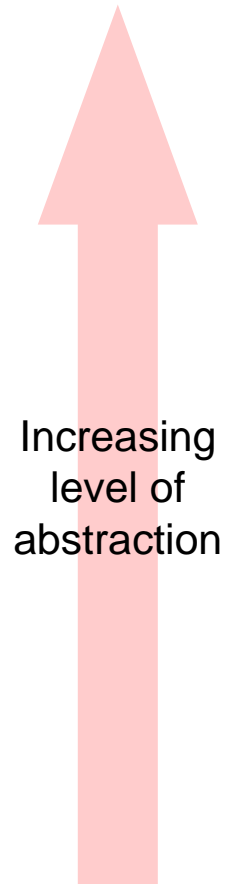
- Introduction
- Writing Fully-synthesizable Code
- Basic Description Styles
- Combination Logic Design Issues
  - **Describing Multiplexers**
- Sequential Logic Design Issues
  - **Blocking/Non-blocking Assignment**
- Modular Design

# Typical Applications of Verilog HDL

---

- Hardware Modeling
  - Behavioral Description
  - ex) Timing, Memory, Transistors
- Writing Testbenchs
  - Generate Test Patterns, File I/O, etc.
- Logic Design
  - Use only **Synthesizable** Description
- Netlist representation
  - Structural Description (Module and Port Connection)
  - Gate-level Netlist (Synthesis Results)

# Various Styles of Verilog Coding



Increasing level of abstraction

Architectural

```
always #($dis_poisson(seed,32))
begin
  if $q_full(qid)
    $q_remove(qid,job,job_id,status);
  else
    fill_queue;
end
```

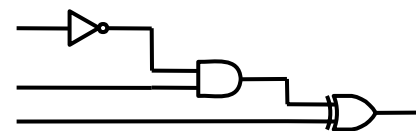
Algorithmic

```
always @(fetch_done)
begin
  casez(IR[7:6])
    2'b00: LDA(acc,IR[5:0]);
    2'b01: STR)acc,IR[5:0]);
    2'b10: JMP(IR[5:0]);
    2'b11: ; // NOP
  endcase
end
```

RTL

```
assign rt1=(i1 & buserr) | zero;
assign sub=rt1 ^| op;
assign out1=i1 & i2 | op;
```

Gate



*behavioral*

*structural*

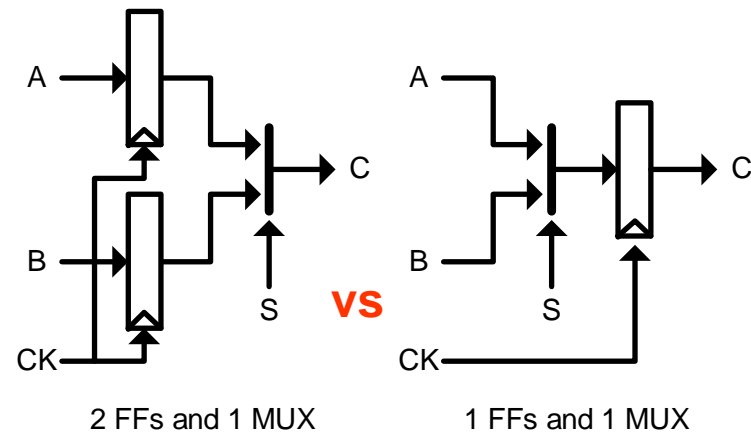
# A Good Code is (In aspect of Logic Design)

- Synthesizable
  - Independence on Synthesizer
    - ex) multiplication, division
- Performance
  - Small Area, High Speed and Low Power Consumption
  - ex) **number of registers**, architecture/micro-architecture
- Good Readability
  - Consistency of Description Style
  - **Precise Description**
  - Comments

assign zero\_detect = ~(x[3] | x[2] | x[1] | x[0]);

**VS**

assign zero\_detect = ~{{{x}}};



# Description Styles – Data flow

- Data flow Style
  - Modeling only **combinational functions**
  - Whenever any of the inputs changes, the output is recalculated and updated
  - Continuous assignment statement **assign**

```
// data flow
module AND2 (in1, in2, out);
    input in1;
    input in2;
    output out;
    assign out = in1 & in2;
endmodule
```

# Description Styles – Behavioral

- Behavioral Style
  - Behavioral instance: **always**
    - The expression **@(in1 or in2)** instruct the simulator to wait until either in1 or in2 has changed
  - Provides high-level language
    - Some of the modules may be unrealizable in hardware

```
// behavioral
module AND2 (in1, in2, out);
    input in1;
    input in2;
    output out;
    reg out;
    always @(in1 or in2)
        out = in1 & in2;
endmodule
```

# Describing Sequential Logic

- **always** statement is used to describe sequential logic
- **posedge** and **negedge** are used to specify the edge-triggered function

```
// D flipflop
module D-FF (ck, d, q);
    input  d, ck;
    output q;
    reg q;
    always @(posedge ck)
        q = d;
endmodule
```

```
// D flipflop with reset
module D-FF (ck, rst, d, q);
    input  d, rst, ck;
    output q;
    reg q;
    always @(posedge ck)
        if(~rst) q = 0;
        else q = d;
endmodule
```

# Combinational Logic Design Issues

# Describing Multiplexer using 'assign' and 'always'

- Multiplexer can be described using both the dataflow and behavioral style
- As a multiplexer is combinational element, assign can be a good choice

```
// dataflow  
assign out = (sel) ? in1 : in0;
```

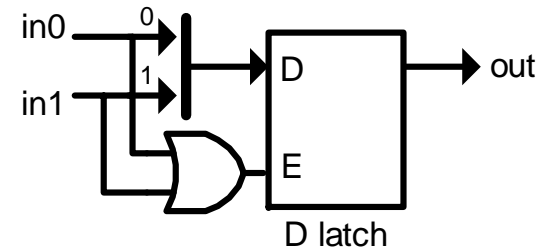
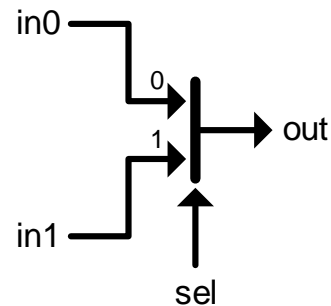
```
// behavioral  
reg out;  
always@(in0 or in1 or sel)  
    case(sel)  
        0: out = in0;  
        1: out = in1;  
    endcase
```

or

```
if(sel) out = in1  
else out = in0;
```

# A Frequently Found Mistake

- Missing inputs in the sensitivity list will synthesize latches



```
// behavioral (missing sel in s-list)  
reg out;  
always@(in0 or in1)  
    case(sel)  
        0: out = in0;  
        1: out = in1;  
    endcase
```

# A Frequently Found Mistake

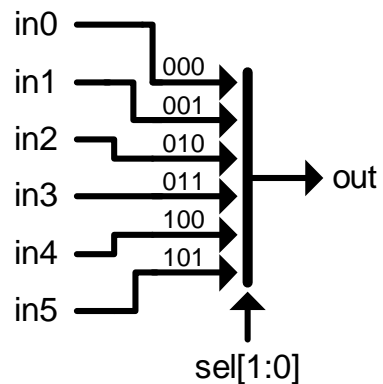
---

- Verilog-2001 supports @\* for combinational logic description

```
// behavioral (* in s-list)
reg out;
always@(*)
    case(sel)
    0: out = in0;
    1: out = in1;
    endcase
```

# Describing Multiplexer– Large Input Size

- Multiplexer with a number of inputs



```
// dataflow style (not optimized)
assign out = (sel == 3'b000) ? in0 :
             (sel == 3'b001) ? in1 :
             (sel == 3'b010) ? in2 :
             (sel == 3'b011) ? in3 :
             (sel == 3'b100) ? in4 :
             in5;
```

```
// behavioral style (latch induced)
reg out;
always @(*)
    case(sel)
        3'b000: out = in0;
        3'b001: out = in1;
        3'b010: out = in2;
        3'b011: out = in3;
        3'b100: out = in4;
        3'b101: out = in5;
    endcase
```

# Optimization Issue

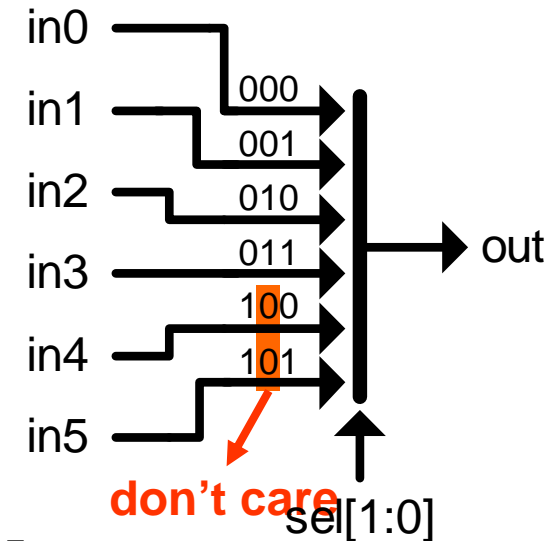
```
// dataflow style (not optimized)
assign out = (sel == 3'b000) ? in0 :
             (sel == 3'b001) ? in1 :
             (sel == 3'b010) ? in2 :
             (sel == 3'b011) ? in3 :
             (sel == 3'b100) ? in4 :
             in5;
```

rewrite using minterms:

```
out = out_in0 | out_in1 | ... | out_in4 | out_in5;
out_in4 = in4 & sel[2] & ~sel[1] & sel[0];
```

```
// dataflow style (optimized)
assign out = sel[2] ? (sel[0] ? in5 : in4) :
               ((sel[1:0] == 2'b00) ? in0 :
                (sel == 2'b001) ? in1 :
                (sel == 2'b010) ? in2 :
                in3);
```

```
out_in4 = in4 & sel[2] & sel[0];
```



## 'case' should cover the full case - 1

- Not fully described condition of case statement can induce latches
  - in that condition, the output is maintained
- **casex** can be used to specify don't care value
- **default** can be used to cover the non-specified conditions

```
// behavioral style (using casex)
reg      out;
always @(*)
    casex(sel)
        3'b000: out = in0;
        3'b001: out = in1;
        3'b010: out = in2;
        3'b011: out = in3;
        3'b1x0: out = in4;
        3'b1x1: out = in5;
    endcase
```

```
// behavioral style (using default)
reg      out;
always @(*)
    casec(sel)
        3'b000: out = in0;
        3'b001: out = in1;
        3'b010: out = in2;
        3'b011: out = in3;
        3'b100: out = in4;
        default: out = in5;
    endcase
```

## 'case' should cover the full case - 2

- using synopsys directives
  - comment '**// synopsys parallel case full case**' denote that not-defined cases can be regarded as don't care
- Not recommended
  - synthesis directives often cause functional mismatches

```
// behavioral style (using synopsys directives)
reg      out;
always @(*)
    case(sel) // synopsys parallel case full case
        3'b000: out = in0;
        3'b001: out = in1;
        3'b010: out = in2;
        3'b011: out = in3;
        3'b100: out = in4;
        3'b101: out = in5;
    endcase
```

# Sequential Logic Design Issues

## Block / Non-block Assignment in **always** blocks (Q)

```
// blocking assignment
module test_block;
reg a, b;

always #10 begin
    a = ~a;
    b = a;
end

initial begin
    a = 0;
    $monitor("%b %b", a, b);
end
```

```
// non-blocking assignment
// aka cuncurrent assignment
module test_block;
reg a, b;

always #10 begin
    a <= ~a;
    b <= a;
end

initial begin
    a = 0;
    $monitor("%b %b", a, b);
end
```

**Guess the results :-)**

# Block / Non-block Assignment in **always** blocks (A)

```
// blocking assignment
module test_block;
reg a, b;

always #10 begin
    a = ~a;
    b = a;
end

initial begin
    a = 0;
    $monitor("a:%b b:%b", a, b);
end
```

```
a:0 b:0
a:1 b:1
a:0 b:0
a:1 b:1
```

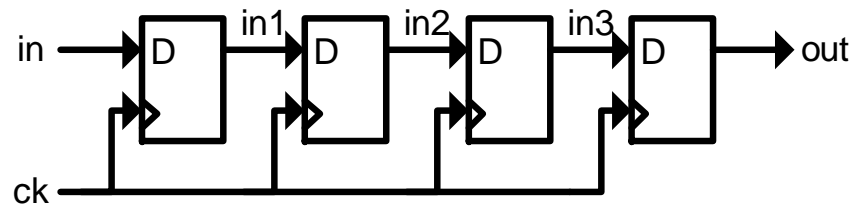
```
// non-blocking assignment
module test_nonblock;
reg a, b;

always #10 begin
    a <= ~a;
    b <= a;
end

initial begin
    a = 0;
    $monitor("a:%b b:%b", a, b);
end
```

```
a:0 b:1
a:1 b:0
a:0 b:1
a:1 b:0
```

# Describing a Shift Register – Block Assignment 1

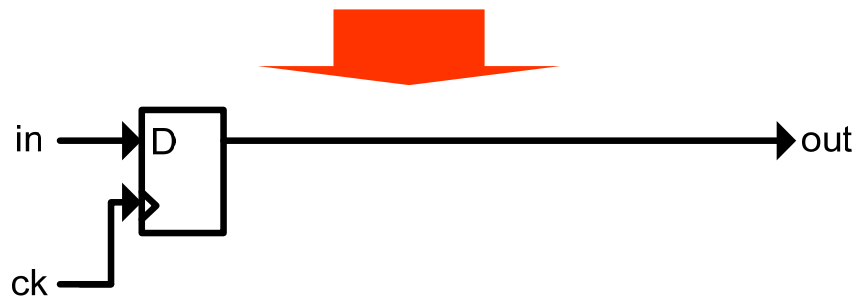
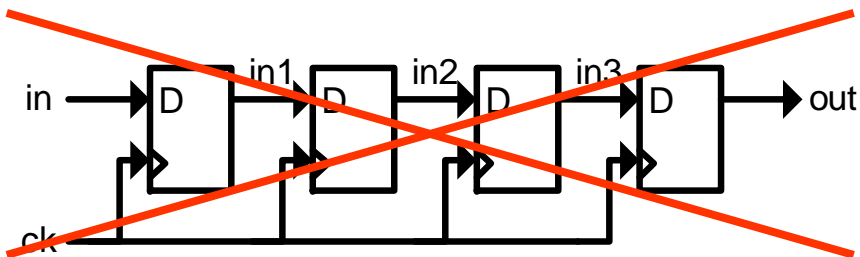


```
// blocking assignment
module shift_register(ck, in, out);
input ck, in;
output out;
reg in1, in2, in3, out;

always@(posedge ck)begin
    out = in3;
    in3 = in2;
    in2 = in1;
    in1 = in;
end

endmodule
```

## Describing a Shift Register – Block Assignment 2

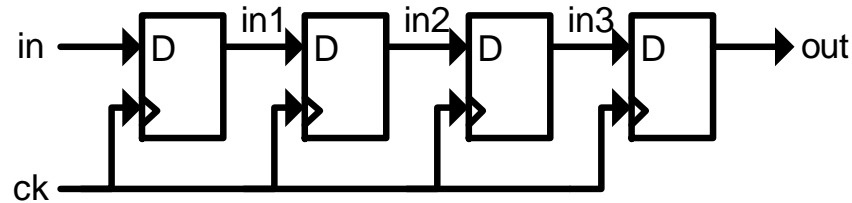


```
// blocking assignment
module shift_register(ck, in, out);
input ck, in;
output out;
reg in1, in2, in3, out;

always@(posedge ck)begin
    in1 = in;
    in2 = in1;
    in3 = in2;
    out = in3;
end

endmodule
```

# Describing a Shift Register – Non-blocking Assignment 1



**blocking = sequential**  
**non-blocking = concurrent**

```
// blocking assignment
module shift_register(ck, in, out);
input ck, in;
output out;
reg in1, in2, in3, out;

always@(posedge ck)begin
    in1 <= in;
    out <= in3;
    in2 <= in1;
    in3 <= in2;
end

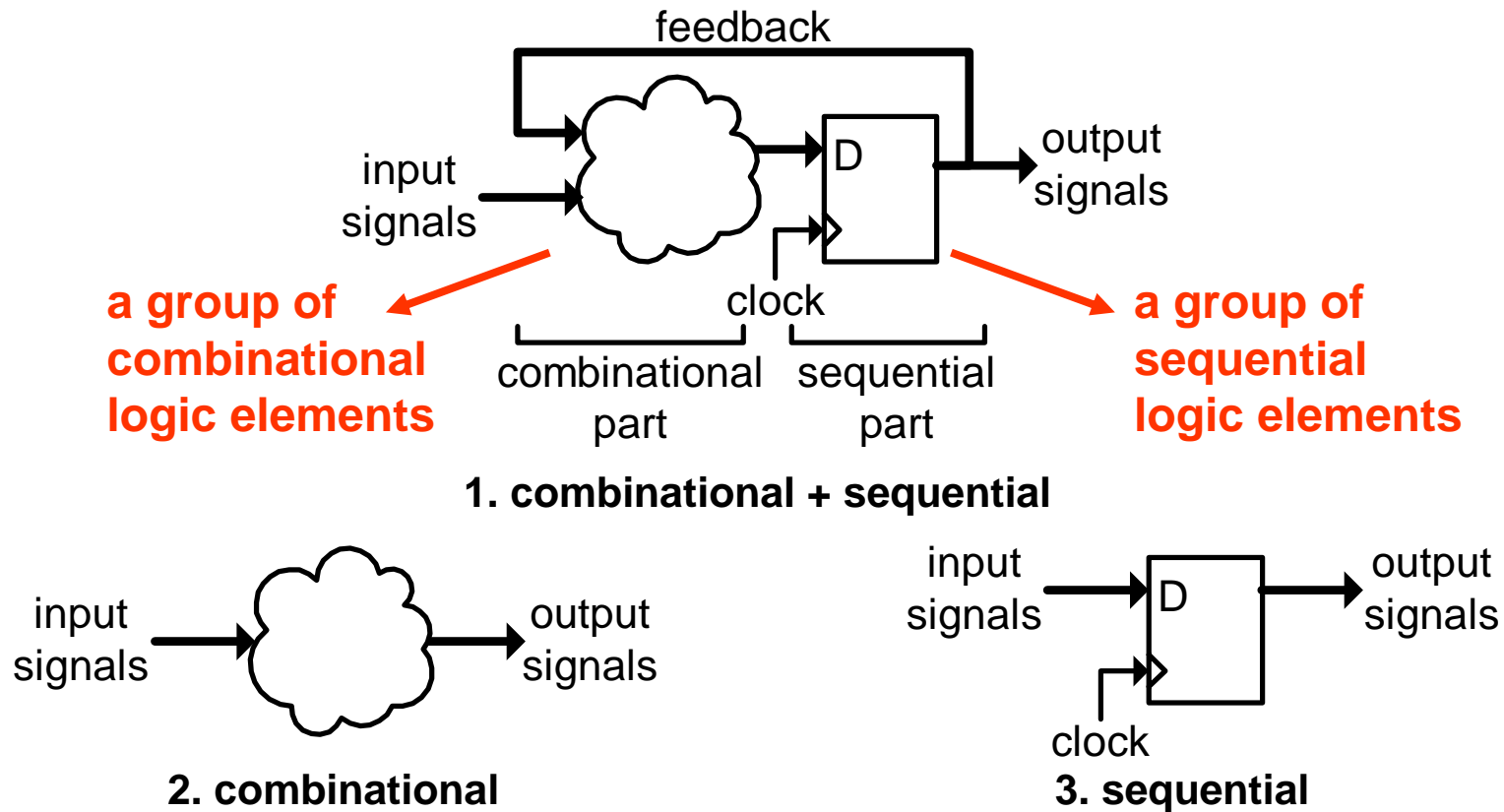
endmodule
```

**Always use non-blocking assignment  
in 'always' block**

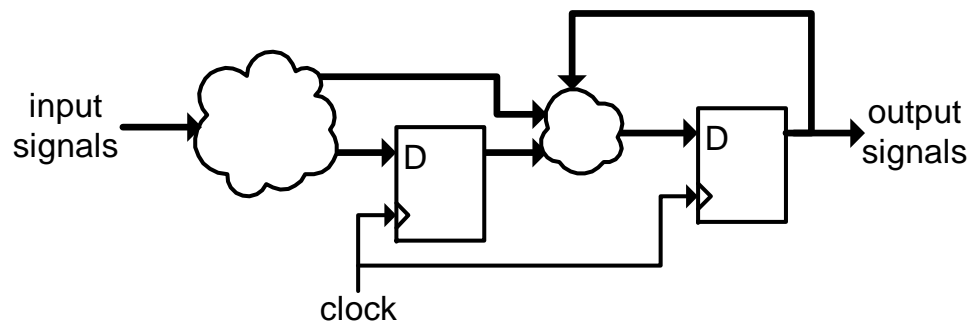
# Modular Design Issues

# Modular Design

- A subblock can be classified into three categories:



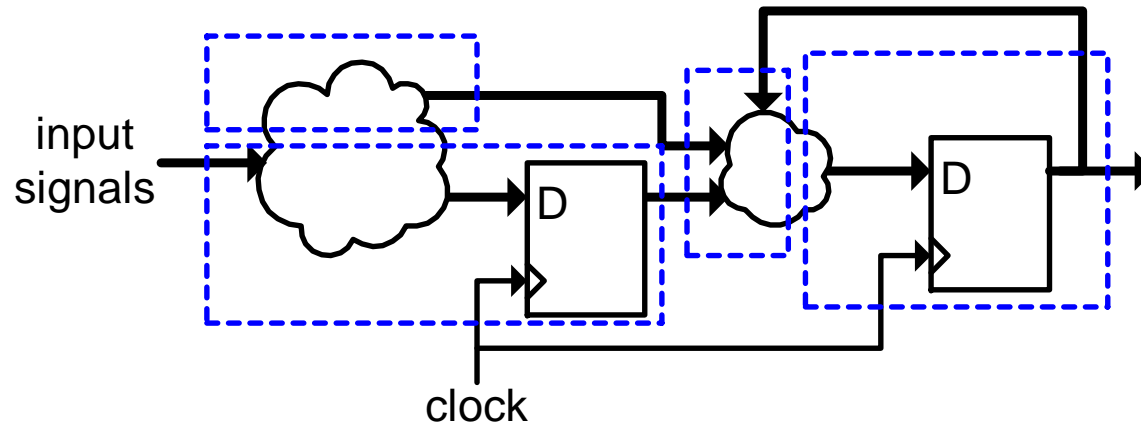
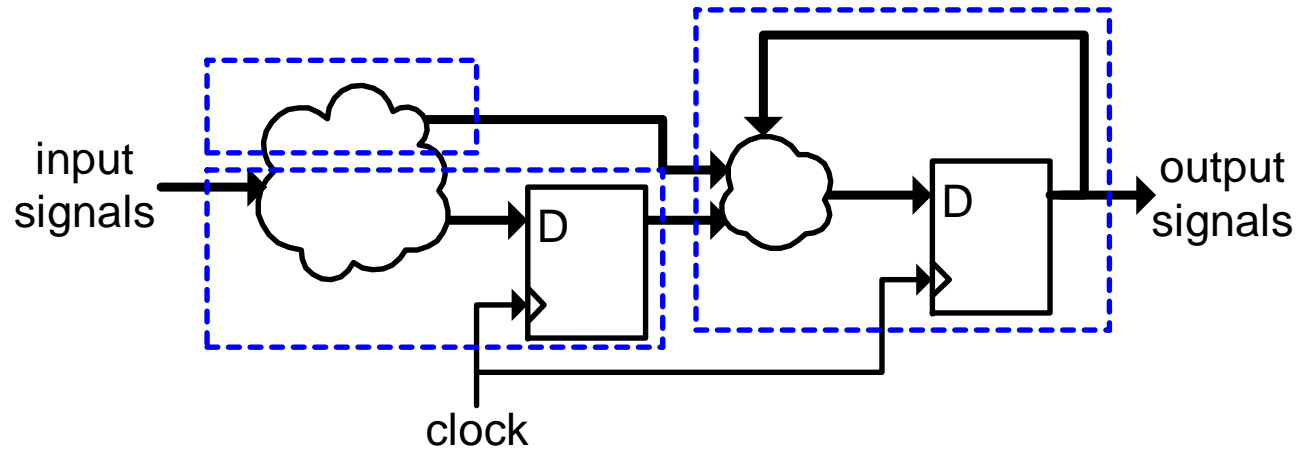
# Example: A Bad Case



```
always@(posedge clock)
begin
  if(a == b)
    c = d;
  else if(d == e)
    d = a;

  if(a == c)
    d = b;
  ...
  ...
end
```

# Modularize into subblocks



## Part – II



- Introduction: HDL Design is Not Programming
- Example Design: A Simple Command Processor
- Separating the design into Datapath and Control Logic
- Command Decoding
- Constructing Datapath
- Implementing Datapath
- Implementing Control Logic
- Implementation Results
- HDL Design vs. Schematic Capture

# HDL Design is Not Programming

- When a designer write a HDL statement, he can guess the resulting hardware
  - cf) **Good programmers also can know the resulting assembly code in advance**
- **Quiz: Guess the equivalent logic!**

```
always@(posedge clk)
begin
  if(~reset)
    out <= 0;
  else
    out <= in;
end
```

```
always@(posedge clk)
begin
  if(~reset)
    out <= 0;
  else if(enable)
    out <= in;
end
```

## Guess the equivalent logic

---

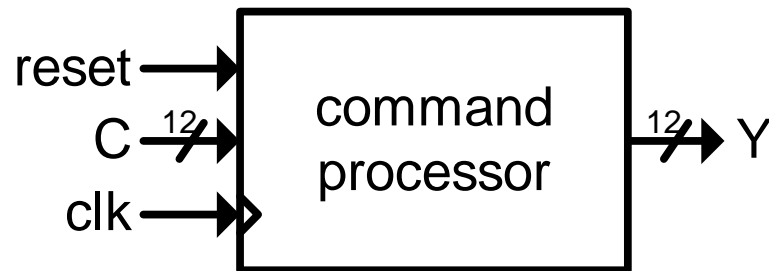
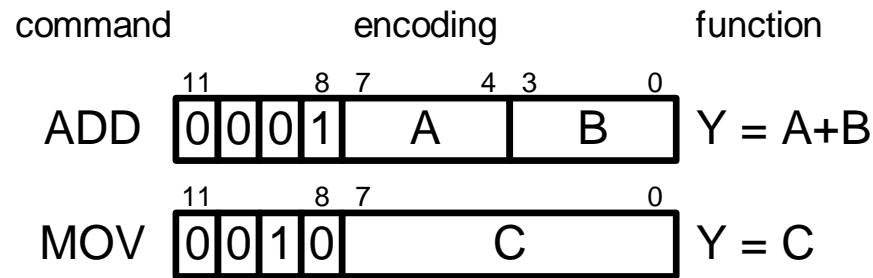
```
always@(posedge clk)
begin
  if(~reset)
    out <= 0;
  else
    out <= in;
end
```

## Guess the equivalent logic

---

```
always@(posedge clk)
begin
  if(~reset)
    out <= 0;
  else if(enable)
    out <= in;
end
```

# Example Design: Very Simple Processor



```

module (clk, C, Y);
input clk;
input [7:0] C;
reg [11:0] Y;

```

```

always@(posedge clk) begin
    if(C[11:8] == 4'b0001)
        Y = C[7:4] + C[3:0];
    else if(C[11:8] == 4'b0010)
        Y = C[7:0];
end
endmodule

```

**Is it a Good Code?**

# More Commands...

command	encoding						function
	11	8	7	4	3	0	
ADD	0	0	0	1	A	B	$Y = A+B$
MOV	0	0	1	0	C		$Y = C$
ADU	0	1	0	D		0	$Y = D+E$ //unsigned
ADS	0	1	0	D		1	$Y = D+E$ //signed

- Lots of if...else statement will be used in this manner

```

module (clk, C, Y);
input clk;
input [7:0] C;
reg [11:0] Y;

always@(posedge clk) begin
  if(C[11:8] == 4'b0000)
    Y = C[7:4] + C[3:0];
  else if(C[11:8] == 4'b0010)
    Y = C[7:0];
  else if(C[11:9] == 3'b010) begin
    if(C[4]) Y = C[8:5] + C[3:0];
    else Y = C[8:5] + {{4{C[4]}},C[3:0]};
  end
end
endmodule

```

**Is it still a Good Code?**

# Even Variable Length Command

command	encoding											
	11	8	7	4	3	0						
ADD	0	0	0	1	A		B					$Y = A+B$
MOV	0	0	1	0	C							$Y = C$
ADU	0	1	0	D			0	E				$Y = D+E$ //unsigned
ADS	0	1	0	D			1	E				$Y = D+E$ //signed
ADL	1	F									$Y = F+G$	
	G											

- Usually a counter(or flag) is used to denote it's in a multi-clock state
- However, these multi-level if...else statements is very hard to handle
- Some conditions are ambiguous and even unreachable

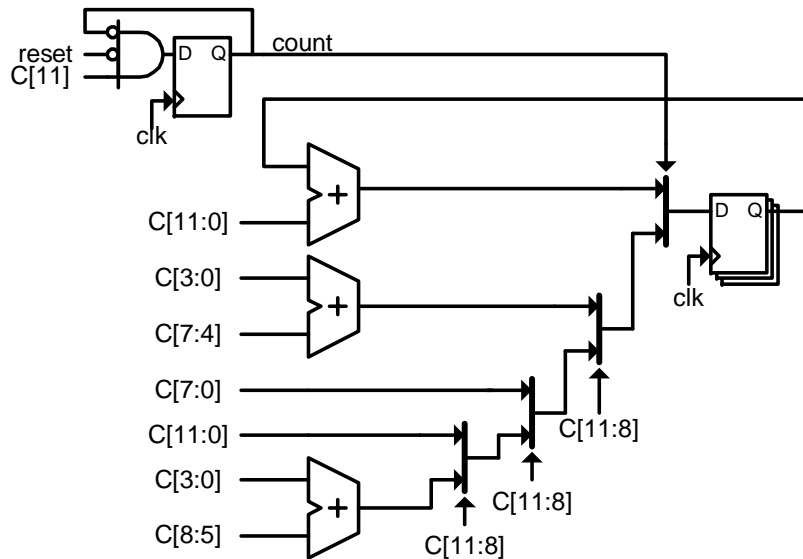
**So, it is a BAD CODE**

```
input clk;
input [7:0] C;
reg [11:0] Y;
reg count;
```

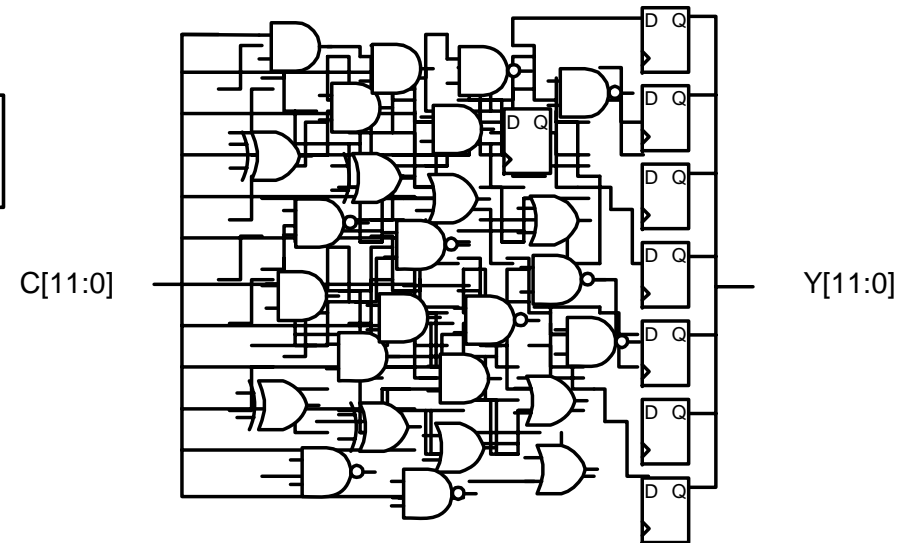
```
always@(posedge clk) begin
  if(reset) count = 0;
  else if(count) begin
    Y = Y+C;
    count = 0;
  end
  else begin
    if(C[11:8] == 4'b0000)
      Y = C[7:4] + C[3:0];
    else if(C[11:8] == 4'b0010)
      Y = C[7:0];
    else if(C[11:9] == 3'b010)
      if(C[4]) Y = C[8:5] + C[3:0];
      else if(~C[4]) Y = C[8:5] + {{4{C[4]}},C[3:0]};
    else if(C[11]) begin
      Y = C[10:0];
      count = 1;
    end
  end
end
end
```

# Equivalent Logic and Synthesis Results

- Synthesis results of the previous example will be like (b)
  - **All the intermediate signals are merged into a random logic**
    - No debugging information

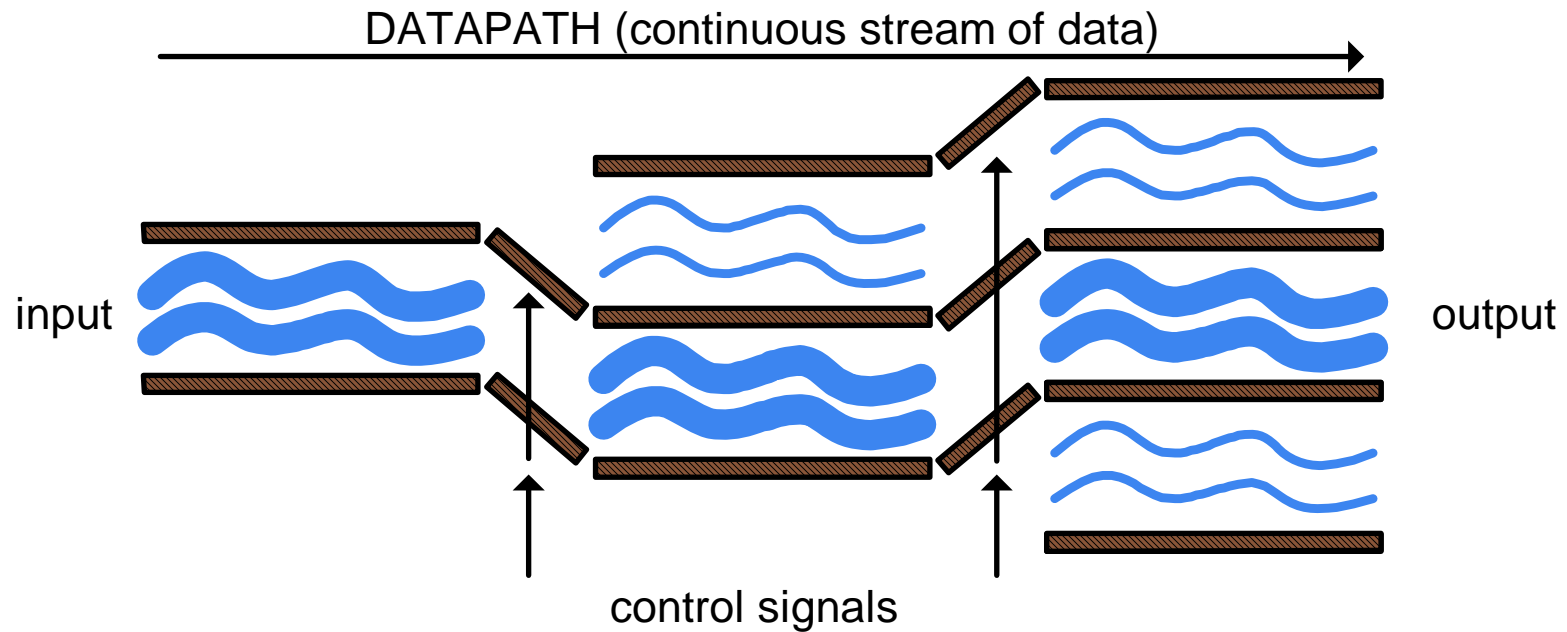


a) Equivalent Logic



b) Synthesis Result (random logic)

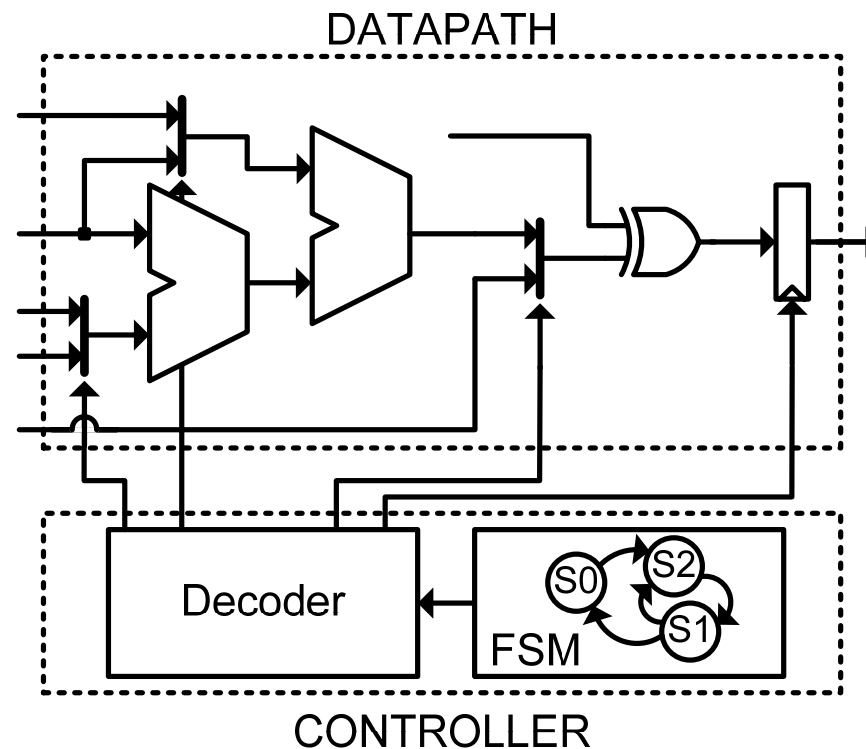
# Orthogonal design of Datapath and Control Logic



- Datapath
  - is like the water flows continuously
- Control Logic
  - simply close or opens the floodgates to select the path

# Orthogonal design of Datapath and Control Logic (cont'd)

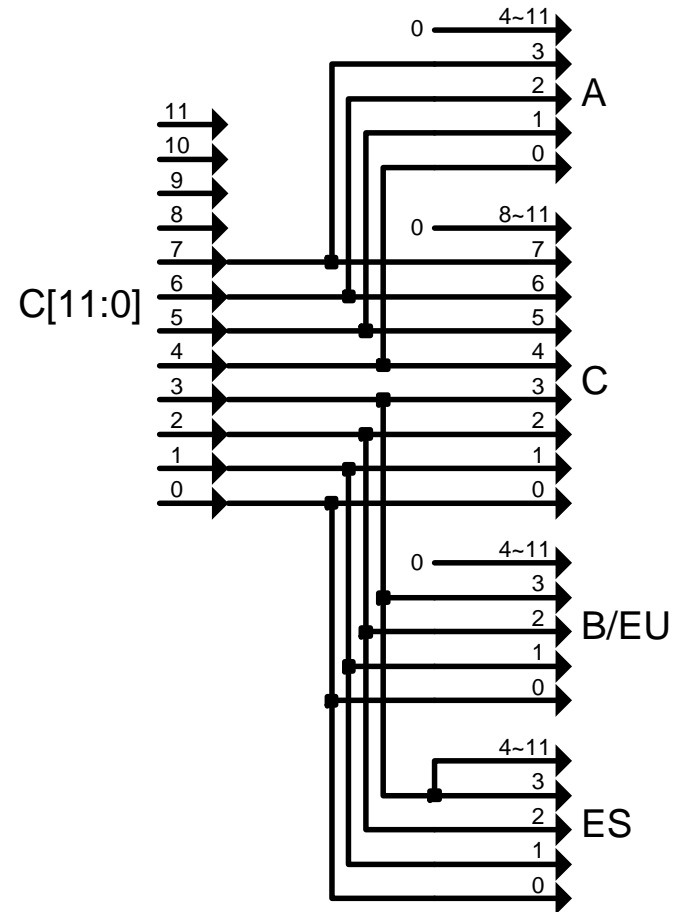
- Datapath
  - Arithmetic
  - Logic
  - Multiplexers
  - Registers
- Control Logic
  - Finite State Machine
  - Control Logic



# Command Decoding

command	encoding				function			
	11	8	7	4 3	0			
ADD	0	0	0	1	A	B	$Y = A+B$	
MOV	0	0	1	0	C		$Y = C$	
ADU	0	1	0		D	0	E	$Y = D+E$ //unsigned
ADS	0	1	0		D	1	E	$Y = D+E$ //signed
ADL	1	F				G		$Y = F+G$

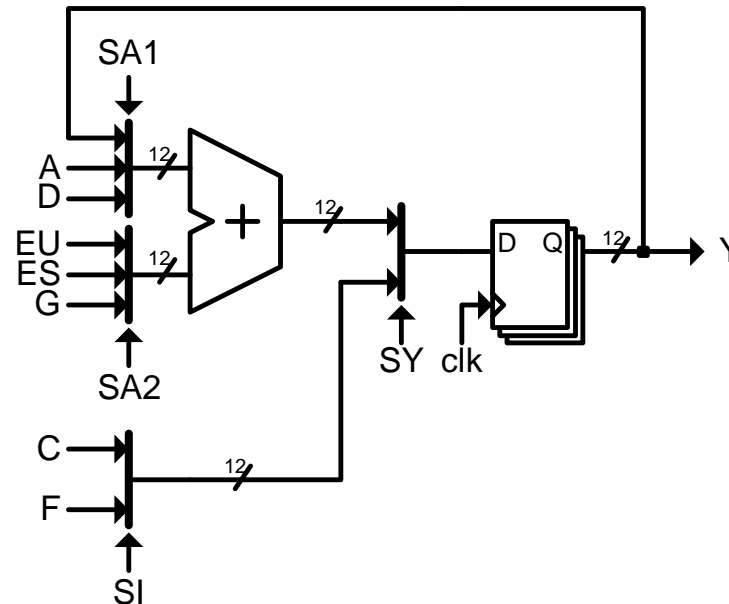
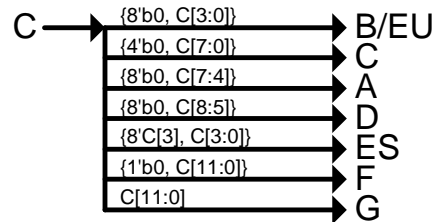
- Command (instruction) is compressed form of various data
- First, split it into original data



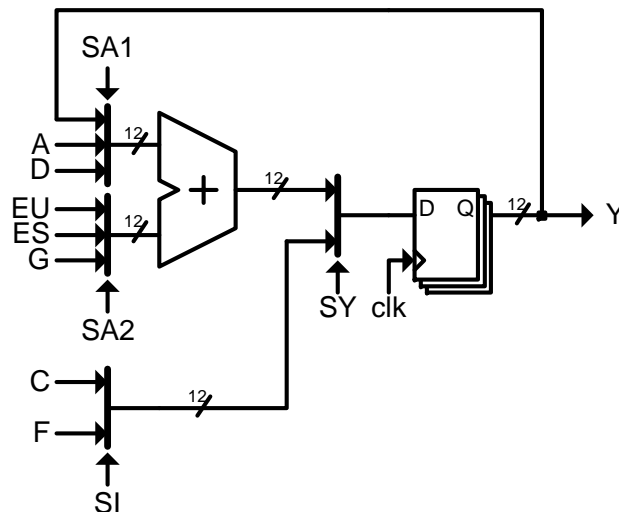
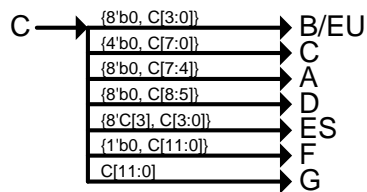
# Constructing Datapath

command	encoding					function	
	11	8	7	4	3	0	
ADD	0	0	0	1	A	B	$Y = A+B$
MOV	0	0	1	0	C		$Y = C$
ADU	0	1	0	D		0	$Y = D+E$ //unsigned
ADS	0	1	0	D		1	$Y = D+E$ //signed
ADL	1	F					$Y = F+G$
	G						

- Construct a unified path of dataflow
- Define Control Points
  - SA1, SA2, SI, SY



# Implementing Datapath



```
wire [11:0] A, C, D, F, G, EU, ES;
wire [11:0] opA, opB, add, imm, Ytmp;
```

```
assign A = {8'b0, C[7:4]};
assign EU = {8'b0, C[3:0]};
assign ES = {{8{C[3]}}, C[3:0]};
```

... **sign extension**

```
always@(SA1 or A or D or Y)
```

```
casex(SA1)
  2'b00: opA <= Y;
  2'b01: opA <= A;
  2'b1x: opA <= D;
endcase
```

**MUX**

...

```
add = opA + opB;
Ytmp = SY ? imm : add;
```

**ADDER**

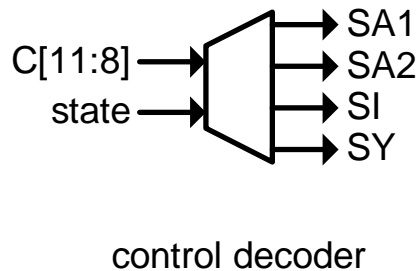
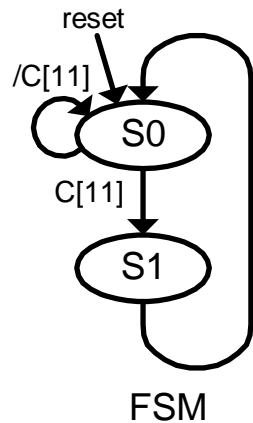
```
always@(posedge clk)
```

```
Y <= Ytmp;
```

**Output Reg.**

# Implementing Control Logic

command	encoding										
	11	10	9	8	7	6	5	4	3	2	0
ADD	0	0	0	1	A			B			
MOV	0	0	1	0	C						
ADU	0	1	0	D			0	E			
ADS	0	1	0	D			1	E			
ADL	1	F									
	G										



```
reg state;
```

```
always@(posedge clk)
  if(reset) state = 0;
  else if(C[11]) state = 1;
  else state = 0;
```

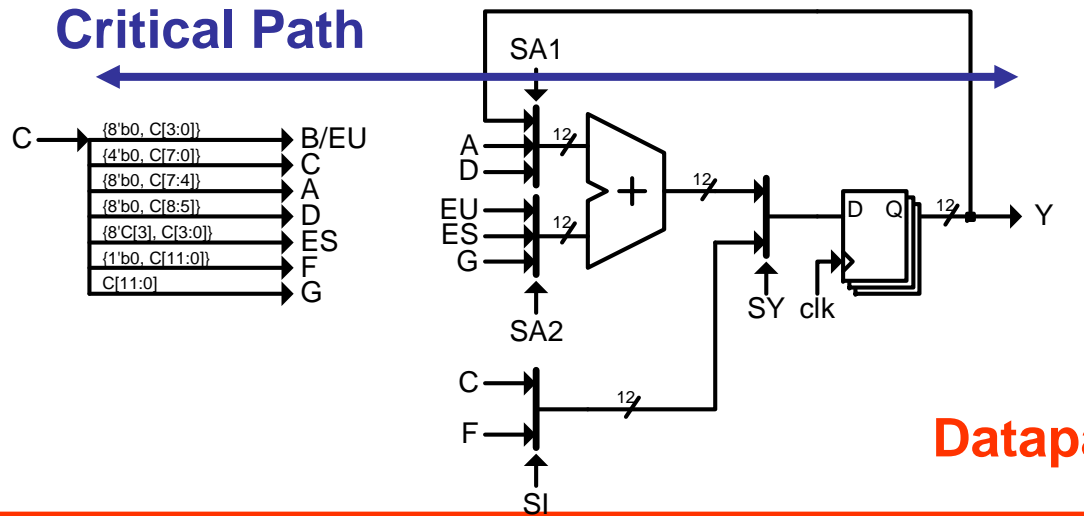
**FSM**

```
always@(C or state)
  casex({C[11:8], state})
    5'bxxxx_1: SA1 <= 0; SA2 <=2; SI <=0; SY <=0;
    5'b0001_0 : SA1 <= 1; SA2 <=0; SI <=0; SY <=0;
    5'b0010_0 : SA1 <= 0; SA2 <=0; SI <=0; SY <=1;
    5'b010x_0 : SA1 <= 2; SA2 <=1; SI <=0; SY <=0;
    5'b1xxx_0 : SA1 <= 0; SA2 <=0; SI <=1; SY <=1;
  endcase
```

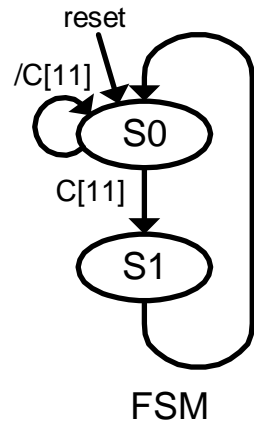
**Control Logic**

# Implementation Results

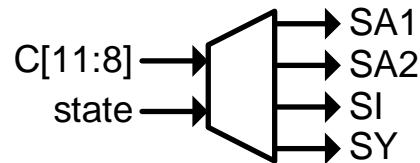
## Critical Path



**Datapath** → optimize for speed



FSM



control decoder

**Control Logic** → optimize for area

# HDL Design vs. Schematic Capture

---

- Manual ( = Schematic Capture)
  - **Architecture**
  - **Logic Partition**
  - **Instantiation and Connection of Logic Elements**
- Automatic (  $\leftrightarrow$  Schematic Capture)
  - **Synthesis**
    - Only for elements (ex, adder)
  - **Optimization**
  - **Technology Migration**