

DESIGN OF DIGITAL CIRCUITS AND SYSTEMS

ASM with Datapath II

Instructor: Justin Hsia

Teaching Assistants:

Colton Carroll

Hemil Patel

Rasya Fawwaz

Grace Zhou

Quinlyn Donohue

Rose Maresh

Relevant Course Information

- ❖ Homework 3 due Friday (4/24)
- ❖ Homework 4 released Thursday

- ❖ Quiz 2 (ROM, RAM, Reg files) this Thursday at **11:50 AM**
 - Based heavily on Homework 2
 - Memory sizing, addressing, initialization, and implementation (*i.e.*, circuit diagram)

- ❖ Lab 3 reports due next Friday (5/1)
 - Ideally finish by early next week so you can start Lab 4, which will be released this Thursday

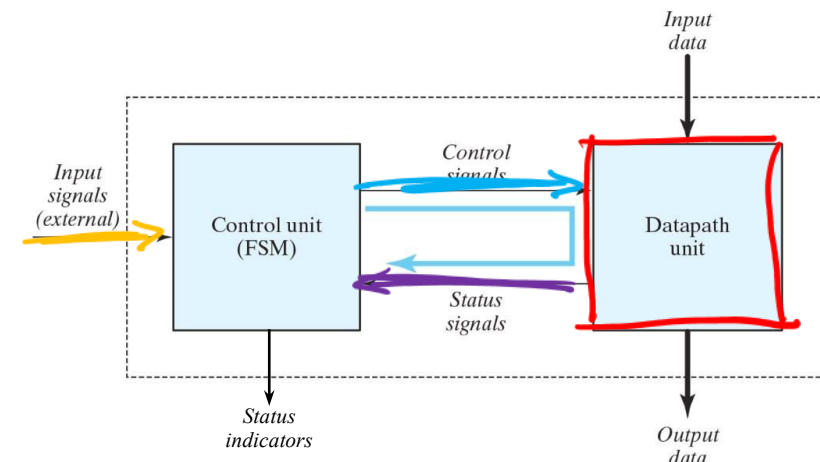
ASMD Design Procedure

- ❖ From problem description or algorithm pseudocode:
 - 1) Identify necessary datapath components and operations**
 - 2) Identify states and signals that cause state transitions** (external inputs and status signals), based on the necessary sequencing of operations
 - 3) Name the control signals** that are generated by the controller that cause the indicated operations in the datapath unit
 - 4) Form an ASM chart for your controller**, using states, decision boxes, and signals determined above
 - 5) Add the datapath RTL operations** associated with each control signal

Design Example #1

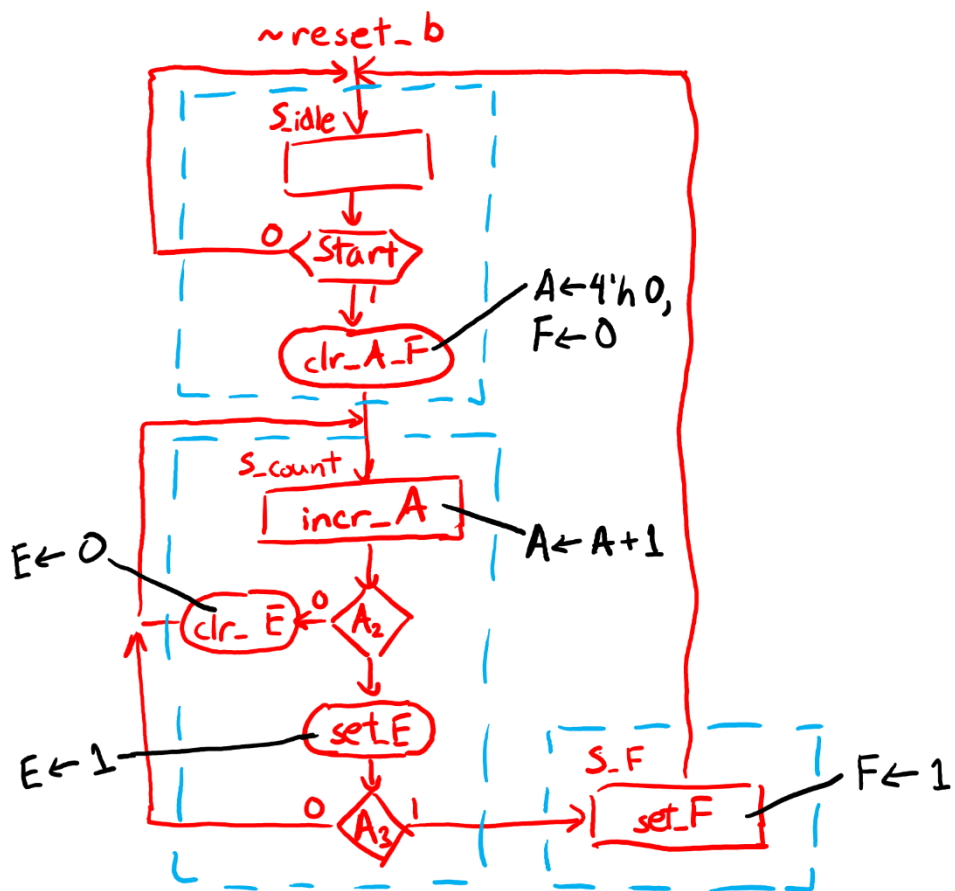
❖ System specification:

- datapath ■ Flip-flops E and F
- datapath ■ 4-bit binary up-counter A = 0bA₃A₂A₁A₀
- inputs to control ■ Active-low reset signal reset_b puts us in state S_idle, where we remain while signal Start = 0
- control signals ■ Start = 1 initiates the system's operation by clearing A and F. At each subsequent clock pulse, the counter is incremented by 1 until the operations stop.
- status signals ■ Bits A₂ and A₃ determine the sequence of operations:
 - control signals • If A₂ = 0, set E to 0 and the count continues
 - If A₂ = 1, set E to 1; additionally, if A₃ = 0, the count continues, otherwise, wait one clock pulse to set F to 1 and stop counting (*i.e.*, back to S_idle)

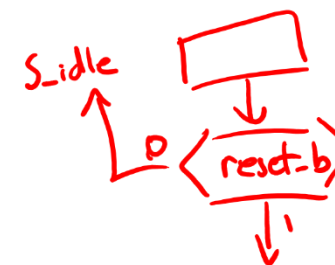


Design Example #1: ASMD Chart

- ❖ Synchronous or asynchronous reset?



for synchronous reset, add decision box on reset-b out of every state box:



Design Example #1: Controller Code

```
module controller (set_E, clr_E, set_F, clr_A_F,
                  incr_A, A2, A3, Start, clk,
                  reset_b);

    // port definitions
    input logic Start, clk, reset_b, A2, A3;
    output logic set_E, clr_E, set_F, clr_A_F, incr_A;

    // define state names and variables
    enum logic [1:0] {S_idle, S_1, S_2 = 3} ps, ns;

    // controller logic w/synchronous reset
    always_ff @(posedge clk)
        if (~reset_b)
            ps <= S_idle;
        else
            ps <= ns;
```

```
    // next state logic
    always_comb
        case (ps)
            S_idle: ns = Start ? S_1 : S_idle;
            S_1:    ns = (A2 & A3) ? S_2 : S_1;
            S_2:    ns = S_idle;
        endcase

    // output assignments
    assign set_E = (ps == S_1) & A2;
    assign clr_E = (ps == S_1) & ~A2;
    assign set_F = (ps == S_2);
    assign clr_A_F = (ps == S_idle) & Start;
    assign incr_A = (ps == S_1);

endmodule // controller
```

Design Example #1: Datapath Code

```
module datapath (A, E, F, clk, set_E, clr_E, set_F, clr_A_F,
                incr_A);

    // port definitions
    output logic [3:0] A;
    output logic E, F;
    input  logic clk, set_E, clr_E, set_F, clr_A_F, incr_A;

    // datapath logic
    always_ff @(posedge clk) begin
        if (clr_E)      E <= 1'b0;
        else if (set_E) E <= 1'b1;
        if (clr_A_F)
            begin
                A <= 4'b0;
                F <= 1'b0;
            end
        else if (set_F) F <= 1'b1;
        else if (incr_A) A <= A + 4'h1;
    end // always_ff

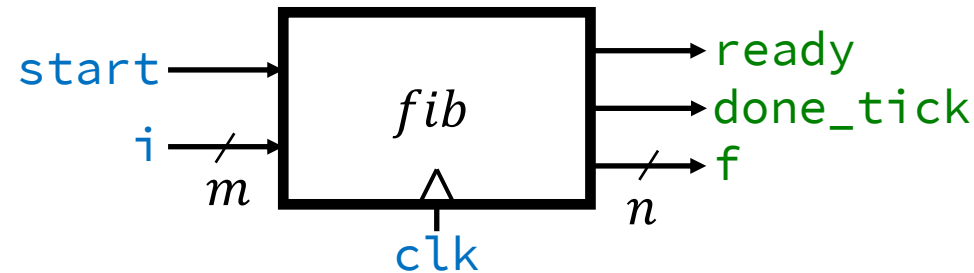
endmodule // datapath
```

Design Example #1: Top-Level Code

```
module top_level (A, E, F, clk, Start, reset_b);  
  
    // port definitions  
    output logic [3:0] A;  
    output logic E, F;  
    input  logic clk, Start, reset_b;  
  
    // internal signals  
    logic set_E, clr_E, set_F, clr_A_F, incr_A;  
  
    // instantiate controller and datapath  
    controller c_unit (.set_E, .clr_E, .set_F,  
                      .clr_A_F, .incr_A, .A2(A[2]),  
                      .A3(A[3]), .Start, .clk,  
                      .reset_b);  
  
    datapath d_unit (.*);  
  
endmodule // top_level
```

Design Example #2: Fibonacci

- ❖ Design a sequential Fibonacci number circuit with the following properties:



- **i** is the desired sequence number
- **f** is the computed Fibonacci number:
$$fib(i) = \begin{cases} 0, & i = 0 \\ 1, & i = 1 \\ fib(i - 1) + fib(i - 2), & i > 1 \end{cases}$$
- **ready** means the circuit is idle and ready for new input
- **start** signals the beginning of a new computation
- **done_tick** is asserted for 1 cycle when the computation is complete

Design Example #2: Pseudocode

- ❖ Pseudocode analysis:
 - Variables are part of **datapath**; assignments become RTL operations
 - Chunks of related actions should be triggered by **control signals**
 - Decision points become **status signals**

Design Example #2: Block Diagram (Control-Datapath)

Design Example #2: ASMD Chart

Design Example #2: Code Outlines

```
fib_control:  
    // port definitions  
    // define state names and variables  
    // controller logic w/synchronous reset  
    // next state logic  
    // output assignments  
  
fib_datapath:  
    // port definitions  
    // datapath logic  
  
fib:  
    // port definitions  
    // define status and control signals  
    // instantiate control and datapath
```

Other Hardware Algorithms We Will Look At

- ❖ Sequential binary multiplier or divider
- ❖ Arithmetic mean

- ❖ Lab 4: Bit counting
- ❖ Lab 4: Binary search
- ❖ Lab 5: Bresenham's line

TECHNOLOGY

BREAK

Hardware Acceleration

- ❖ ASMD as a design process can be used to implement software algorithms
- ❖ Custom hardware can accelerate operation:
 - Hardware can better exploit parallelism
 - Hardware can implement more specialized operations
 - Hardware can reduce “processor overhead” (*e.g.*, instruction fetch, decoding)
- ❖ “Hardware accelerators” are frequently used to complement processors to speed up common, computationally-intensive tasks
 - *e.g.*, encryption, machine vision, cryptocurrency mining

Binary Multiplication

- ❖ Multiplication of unsigned numbers:

	Multiplicand M (14)		1 1 1 0		Multiplicand M (14)		1 1 1 0
	Multiplier Q (11)	x	1 0 1 1		Multiplier Q (11)	x	1 0 1 1
			1 1 1 0		Partial product 0		1 1 1 0
			1 1 1 0			+	1 1 1 0
			0 0 0 0		Partial product 1		1 0 1 0 1
			1 1 1 0			+	0 0 0 0
			1 0 0 1 1 0 1 0		Partial product 2		0 1 0 1 0
						+	1 1 1 0
Product P (154)					Product P (154)		1 0 0 1 1 0 1 0

(a) Multiplication by hand

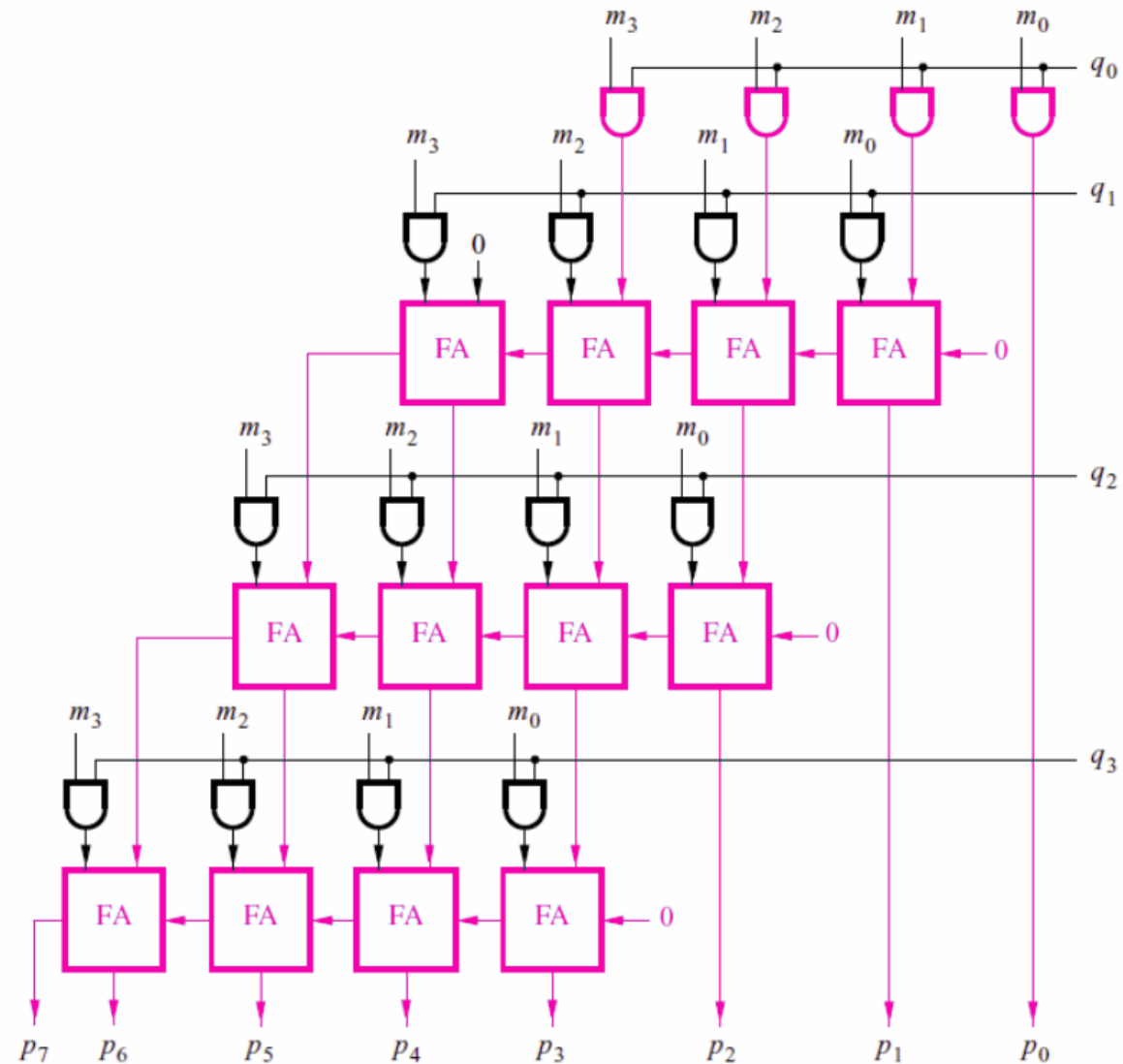
(b) Using multiple adders

		m_3	m_2	m_1	m_0		
	x	q_3	q_2	q_1	q_0		
		m_3q_0	m_2q_0	m_1q_0	m_0q_0		
Partial product 0		+	m_3q_1	m_2q_1	m_1q_1	m_0q_1	
		$PP1_5$	$PP1_4$	$PP1_3$	$PP1_2$	$PP1_1$	
Partial product 1		+	m_3q_2	m_2q_2	m_1q_2	m_0q_2	
		$PP2_6$	$PP2_5$	$PP2_4$	$PP2_3$	$PP2_2$	
Partial product 2		+	m_3q_3	m_2q_3	m_1q_3	m_0q_3	
		P_7	P_6	P_5	P_4	P_3	P_2
Product P							P_1
							P_0

(c) Hardware implementation

Parallel Binary Multiplier

- ❖ *Parallel* multipliers require a lot of hardware:



Sequential Binary Multiplier (1/2)

- ❖ Design a *sequential* multiplier that uses **only one adder and a shift register**
 - Assume one clock cycle to shift and one clock cycle to add
 - More efficient in hardware, less efficient in time
- ❖ Considerations:
 - n -bit **multiplicand** and **multiplier** yield a product at most how wide?
 - What are the ports for an n -bit adder?
 - How many shift-and-adds do we do and how do we know when to stop?

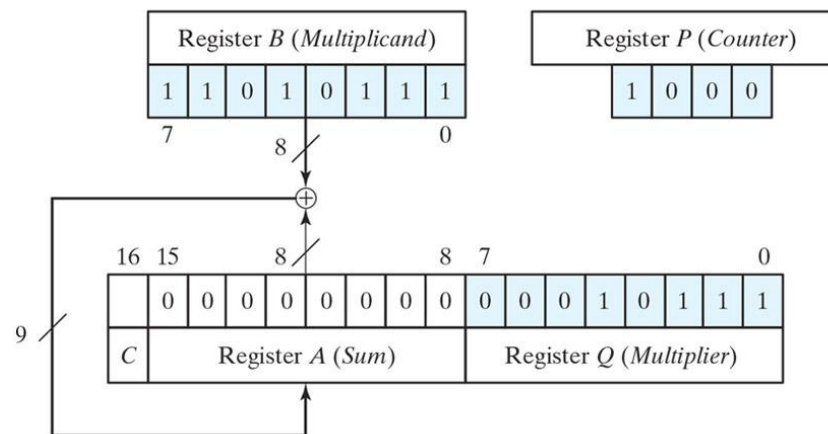
Sequential Binary Multiplier (2/2)

- ❖ Design a *sequential* multiplier that uses **only one adder and a shift register**
 - Assume one clock cycle to shift and one clock cycle to add
 - More efficient in hardware, less efficient in time
- ❖ Implementation Notes:
 - If current bit of **multiplier** is 0, then skip the adding step
 - Instead of shifting **multiplicand** to the left, we will shift the **partial sum** (and the **multiplier**) *to the right*
 - We will re-use the **multiplier** register for the lower half of the product
 - Treat **carry**, **partial sum**, and **multiplier** as one shift register {C, A, Q}

Sequential Binary Multiplier Operation

❖ A few steps of:

$$\begin{array}{r} 11010111 \\ \times 00010111 \\ \hline \end{array}$$



Operation (completed)	C	A	Q	P
Initialize computation	0	00000000	00010111	1000

Binary Multiplier: Specification

❖ Datapath

- $(2n+1)$ -bit *shift register* with bits split into 1-bit C , n -bit A , and n -bit Q
- Multiplicand stored in register B , multiplier stored in Q
- An n -bit *parallel adder* adds the contents of B to A and outputs to $\{C, A\}$
- A $\lceil \log_2(n + 1) \rceil$ -bit *counter* P

❖ Control

- Inputs *Start* and *Reset*, outputs *Ready* and *Done*
- Status signals:
- Control signals:

Binary Multiplier: ASMD Chart

Binary Multiplier Implementation

❖ Controller Logic

Load_regs =

Shift_regs =

Add_regs =

Decr_P =

Ready =

Done =

Binary Multiplier: Datapath Outline

```
module datapath #(parameter WIDTH=8)
    (product, Q, P, multiplicand, multiplier, clk,
     Load_regs, Shift_regs, Add_regs, Decr_P);

    // port definitions
    output logic [2*WIDTH-1:0] product;
    output logic [WIDTH-1:0] Q, P; // note: unnecessary bits for P
    input  logic [WIDTH-1:0] multiplicand, multiplier;
    input  logic clk, Load_regs, Shift_regs, Add_regs, Decr_P;

    // internal logic
    logic C;
    logic [WIDTH-1:0] A, B;

    // datapath logic

endmodule
```