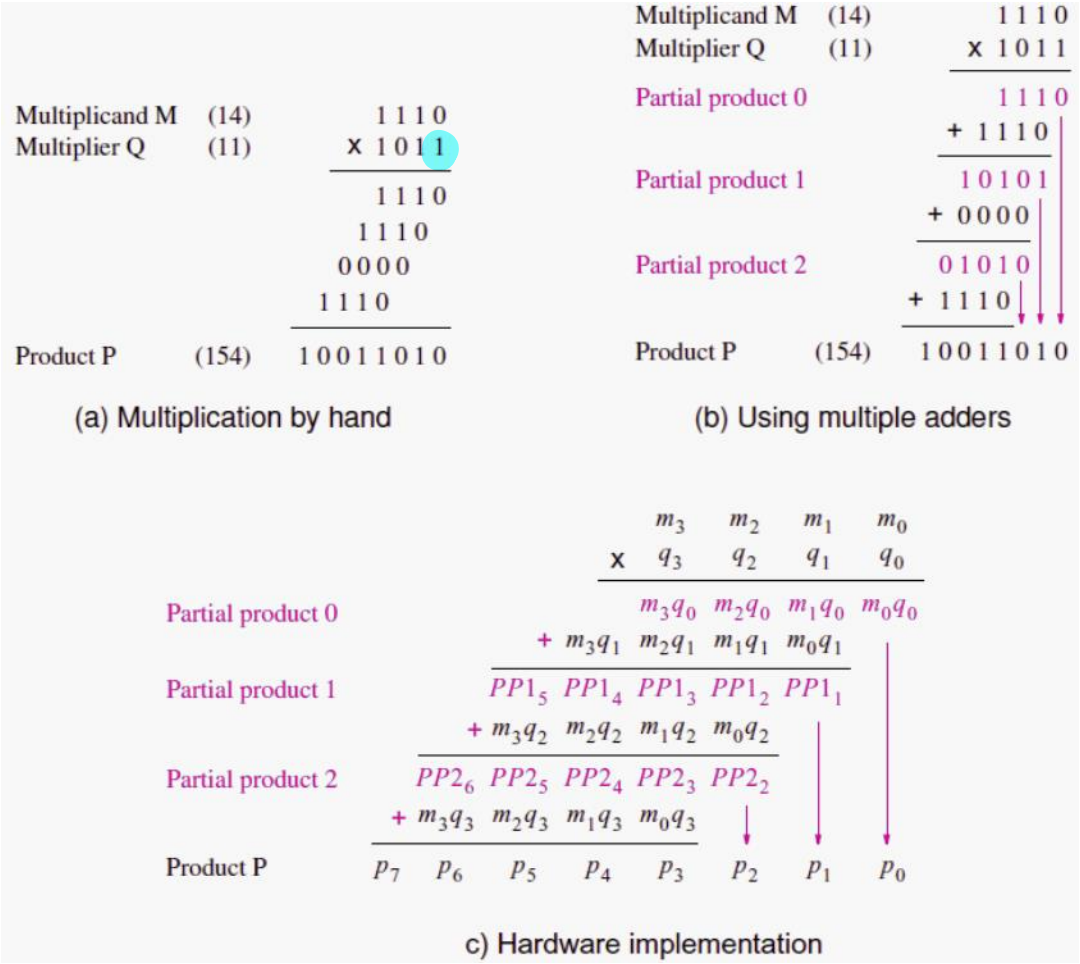
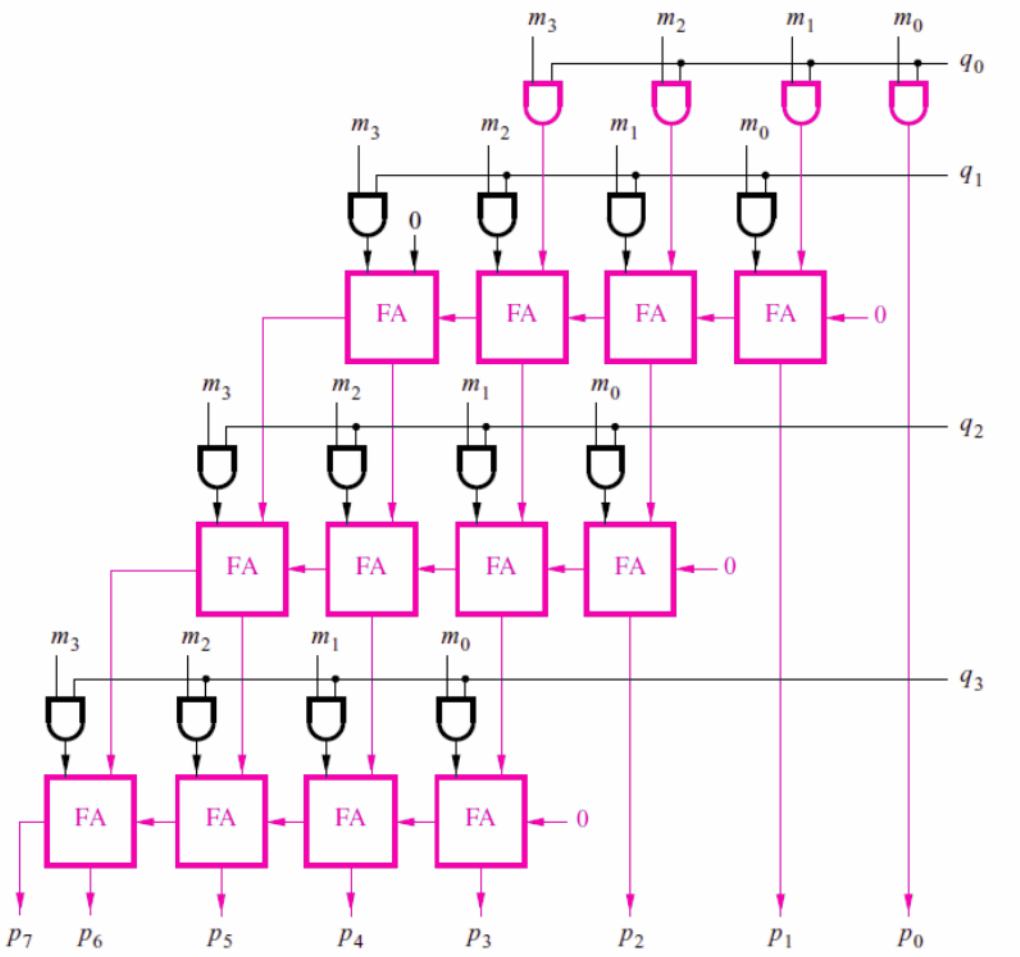
Binary Multiplication

- Multiplication of unsigned numbers



Parallel Binary Multiplier

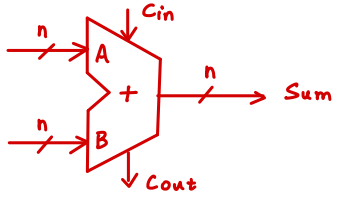
- *Parallel* multipliers require a lot of hardware



Sequential Binary Multiplier

- 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? 2n-bits for product.
  - 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? n-times.

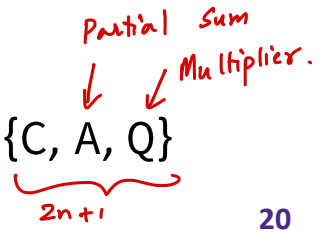


Counter
initialize to n
count down.

Sequential Binary Multiplier

- 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}

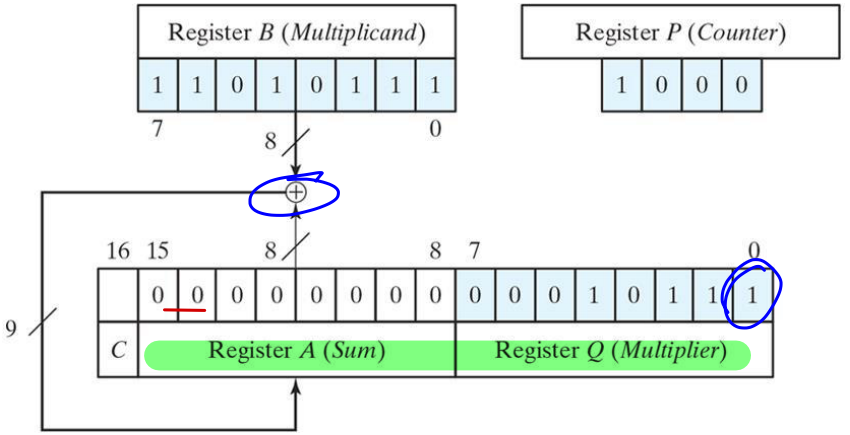


Sequential Binary Multiplier Operation

- A few steps of:

B→ 11010111

Q→x 00010111



Operation (completed)	C	A	Q	P
Initialize computation	0	00000000	00010111	1000
Add (Q[0] = 1)	0	11010111	00010111	0111
shift	0	01101011	10001011	0111
Add (Q[0] = 1)	1	01000010	10001011	0110
shift	0	10100001	01000101	0110

IEEE – 754

64 bits.

$10^{\pm 308}$

1	11	52
sign	exponent	fraction.

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: $Q[0]$, P_zero
  - Control signals: Add_regs, $shift\_regs$, $load\_regs$, $Decr\_P$

external inputs

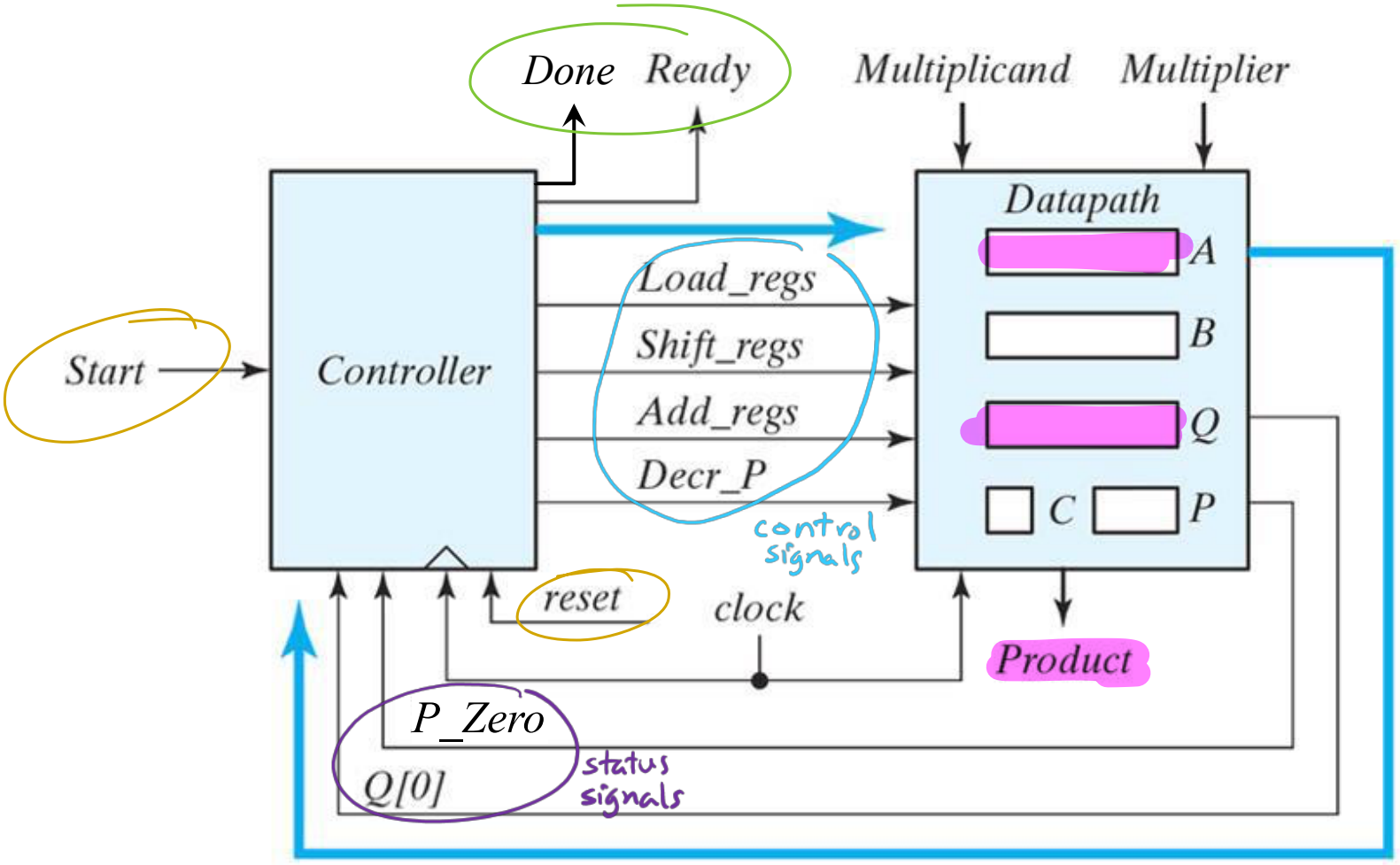
status indicators

Decision

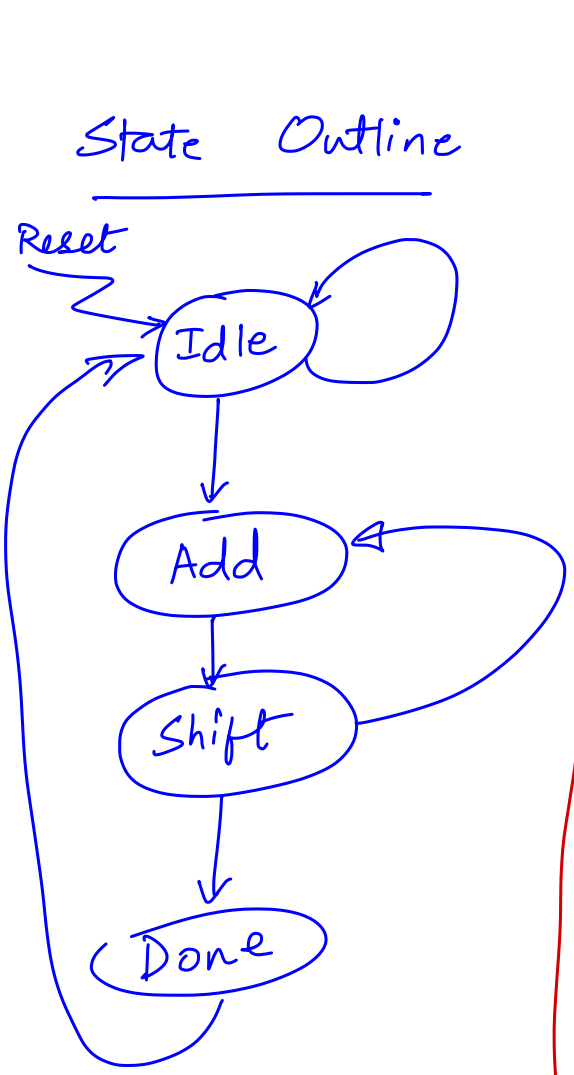
Action

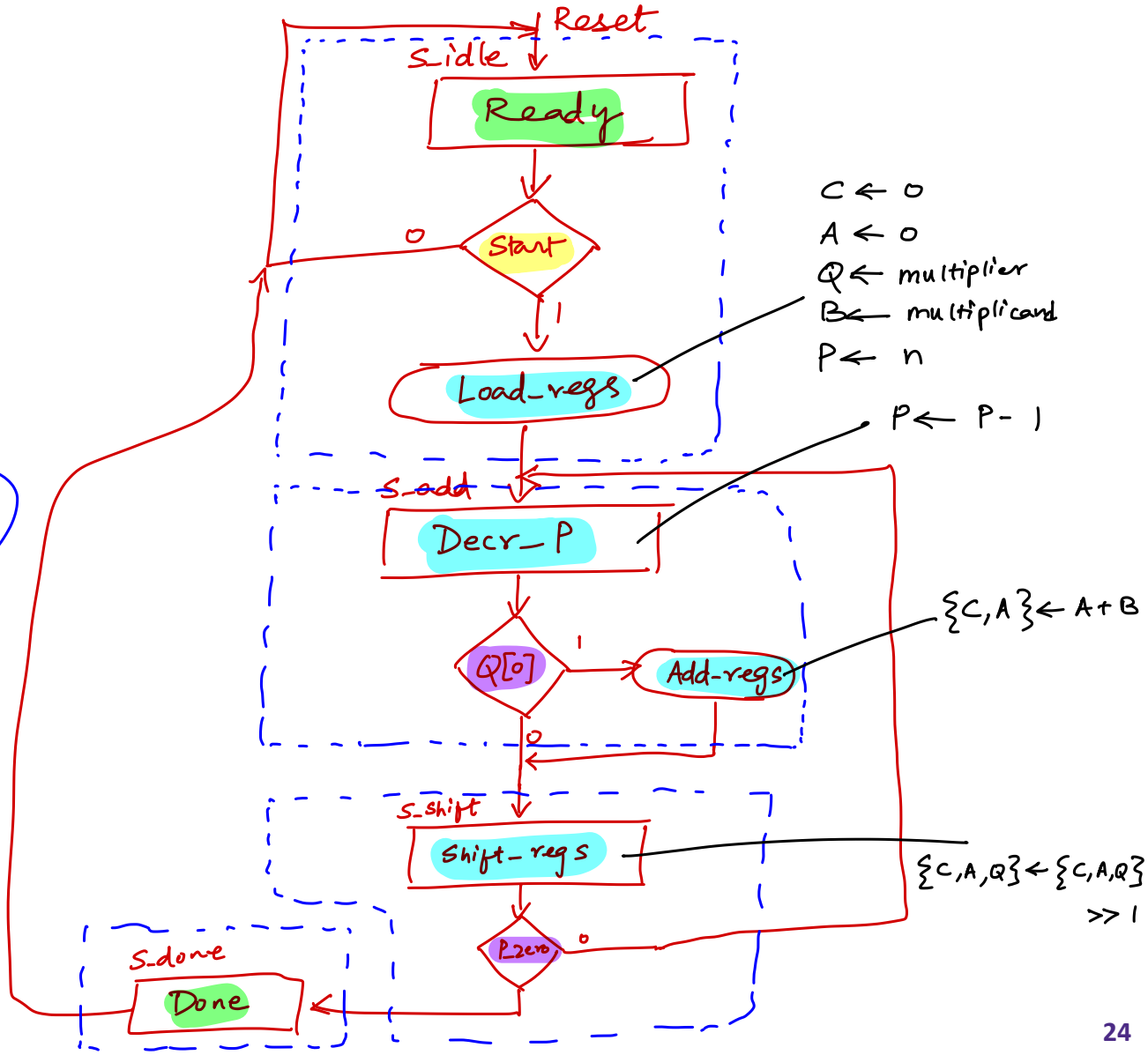
init

Binary Multiplier Block Diagram



Binary Multiplier (ASMD Chart)





Binary Multiplier Implementation

- Controller Logic

$$Load\_regs = S\_idle \,\&\, Start$$

$$Shift\_regs = S\_shift$$

$$Add\_regs = S\_add \,\&\, Q[0]$$

$$Decr\_P = S\_add$$

$$Ready = S\_idle$$

$$Done = S\_done.$$

Binary Multiplier (SV, Datapath)

```
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
```

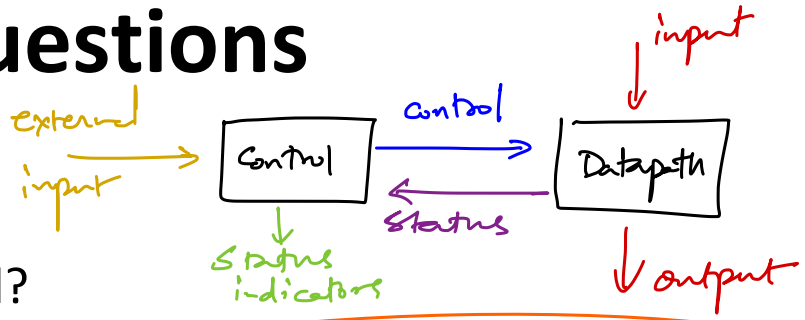
Binary Multiplier (SV, Datapath)

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

   // port definitions
   ...

   // internal logic
   ...

   // datapath logic
   always_ff @(posedge clk) begin
      if (Load_regs) begin
         A <= 0;  C <= 0;  P <= WIDTH;
         B <= multiplicand;
         Q <= multiplier;
      end
      if (Decr_P)       P <= P - 1;
      if (Add_regs)     {C, A} <= A + B;
      if (Shift_regs)   {C, A, Q} <= {C, A, Q} >> 1;
   end  // always_ff

   assign product = {A, Q};

endmodule
```

EE/CSE 371
Design of Digital Circuits and Systems

Lecture 8: Algorithms to Hardware I

ASMD Chart Review Questions



- Circle all that apply:

1. Where can **control signals** be found? (outputs)
   a State boxes b Decision boxes c Conditional output boxes
2. Where can **status signals** be found? (inputs)
   a State boxes b Decision boxes c Conditional output boxes
3. Where can **external input signals** be found?
   a State boxes b Decision boxes c Conditional output boxes
4. What is the *first* thing a path should encounter in an **ASM block**?
   a State boxes b Decision boxes c Conditional output boxes
5. What can be found outside of **ASM blocks**? None!
   a State boxes b Decision boxes c Conditional output boxes
6. What can **RTL operations** be attached to?
   a Control signals b Status signals c External output signals

ASMD Process Review

1) Identify datapath components, control signals, and status signals from description or pseudocode.
2) [*optional*] Create control-datapath circuit diagram.
3) [*optional*] Create state outline to plan out states and transitions between them.
4) Draw out ASM state boxes, decision boxes, and paths between them.
5) Augment state boxes with Moore-type outputs and add conditional output boxes with Mealy-type outputs.
6) Add ASM blocks to organize states.
7) Add RTL operations to control signals.
8) Double-check decision box edge cases and timing of operations (*i.e.*, debug).

Division Circuit

- Design a circuit that implements the long-division algorithm:

```
      15
   ------
9 ) 140
     9
    ---
     50
     45
    ---
      5
```

(a) An example using decimal numbers

```
                 00001111   <- quotient
              ----------
divisor -> 1001 ) 10001100   <- dividend
                  1001
                 -----
                  10001
                   1001
                  -----
                   10000
                    1001
                   -----
                    1110
                    1001
                   -----
                     101   <- remainder
```

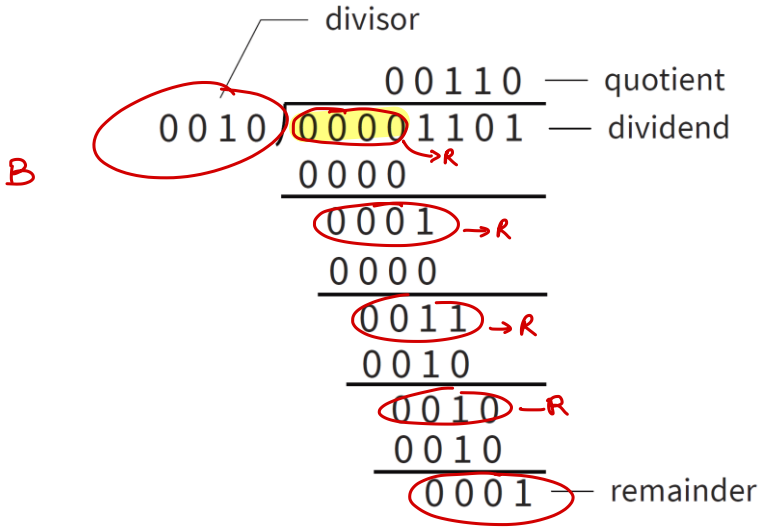
(b) Using binary numbers

- Considerations:
  - Main operations? Subtract, shift, compare
  - Stop condition? n iterations if both divisor & dividend are n-bits.

Division Circuit

- Design a circuit that implements the long-division algorithm:
  1) Double the dividend width by appending 0's in front and align the divisor to the leftmost bit of the *extended dividend*.
  2) If the corresponding dividend bits are $\geq$ the divisor, subtract the divisor from the dividend bits and make the corresponding quotient bit 1. Otherwise, keep the original dividend bits and make the quotient bit 0.
  3) Append one additional dividend bit to the previous result and shift the divisor to the right one position.
  4) Repeat steps 2 and 3 until all dividend bits are used.

Division Circuit



- Implementation Notes:
  - If current dividend window is smaller than the divisor, skip subtraction
  - Instead of shifting divisor to the right, we will shift the dividend (and the quotient) *to the left*
  - We will re-use the lower half of the dividend register to store the quotient