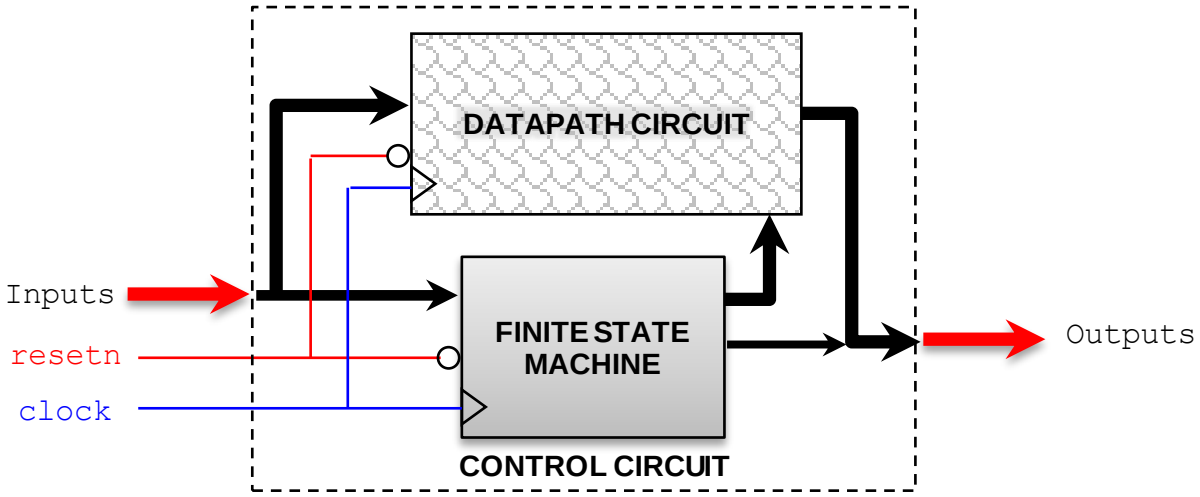
Digital System Design

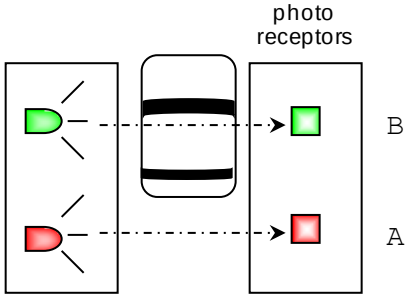
Digital System Model

FSM (Control) + Datapath Circuit



Examples

Car Lot Counter



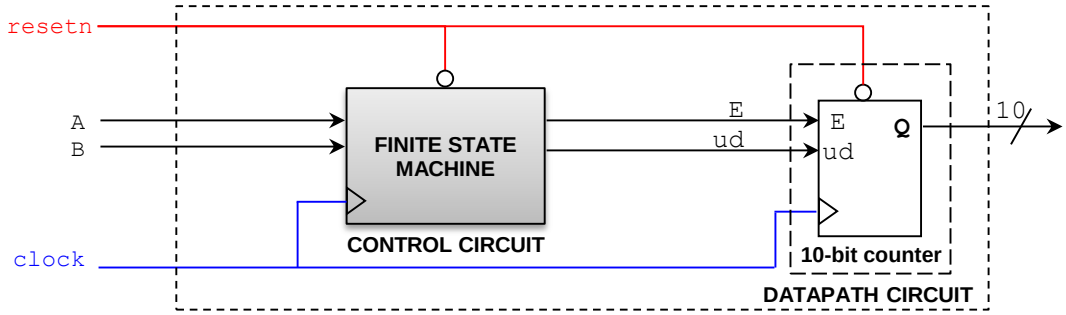
If A = 1 → No light received (car obstructing LED A)
If B = 1 → No light received (car obstructing LED B)

If car enters the lot, the following sequence (A|B) must be followed:
00 → 10 → 11 → 01 → 00
If car leaves the lot, the following sequence (A|B) must be followed:
00 → 01 → 11 → 10 → 00

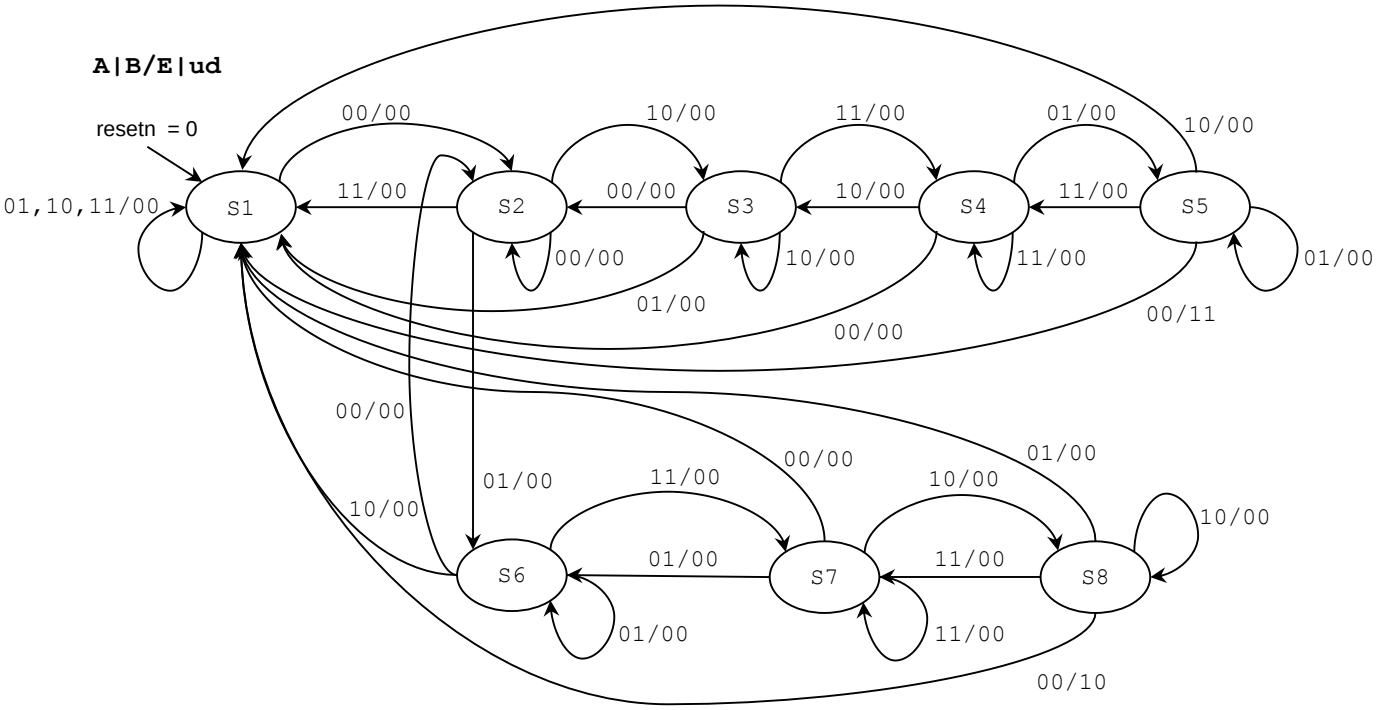
A car might stay in a state for many cycles since the car speed is very large compared to that of the clock frequency.

DIGITAL SYSTEM (FSM + Datapath circuit)

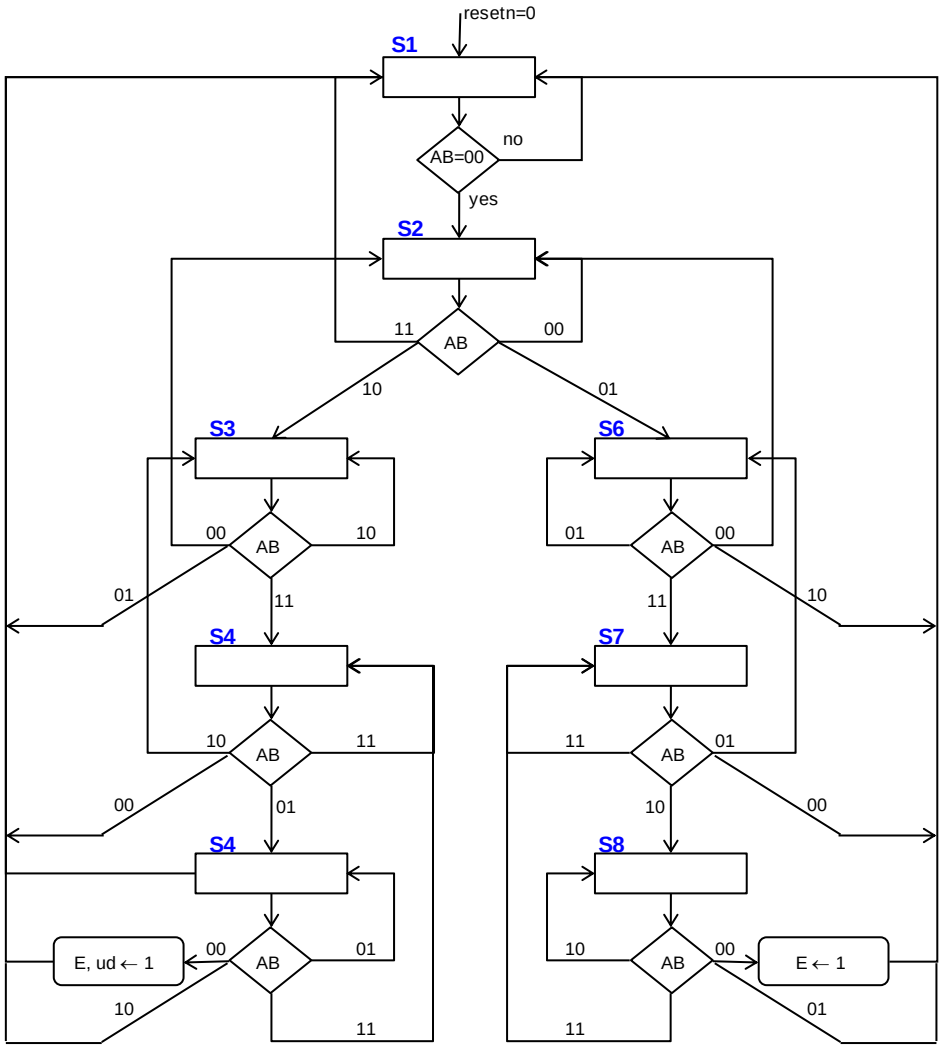
- Usually, when *resetn* (asynchronous clear) and *clock* are not drawn, they are implied.



- **Finite State Machine (FSM)**:



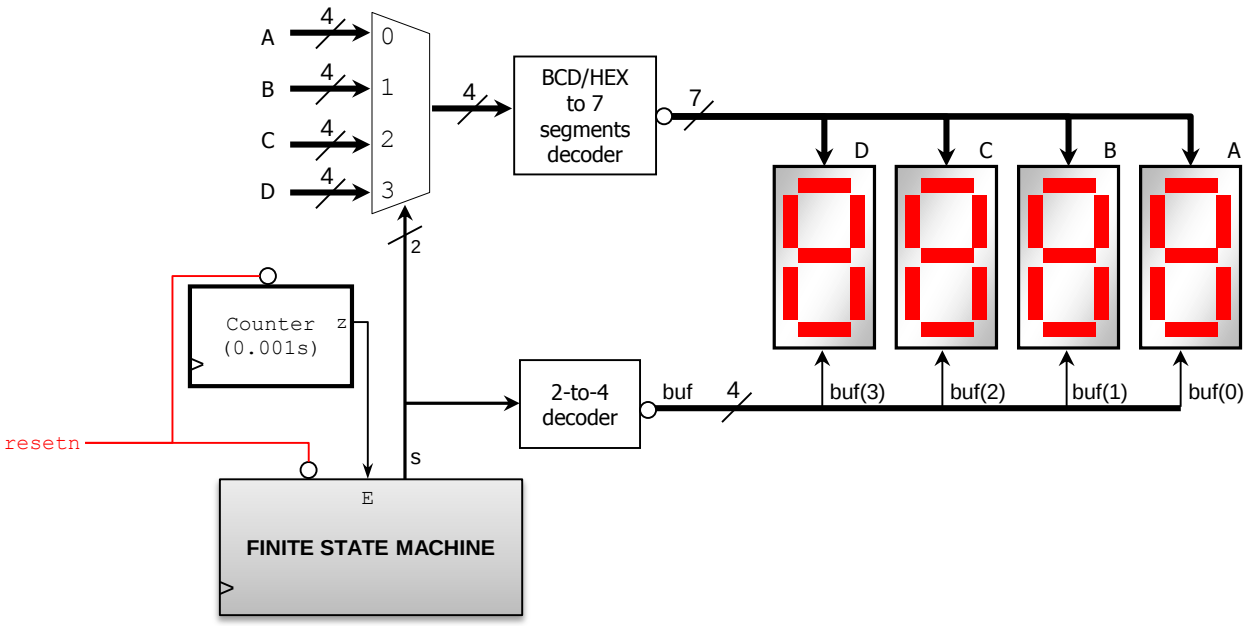
- **Algorithmic State Machine (ASM) chart:**



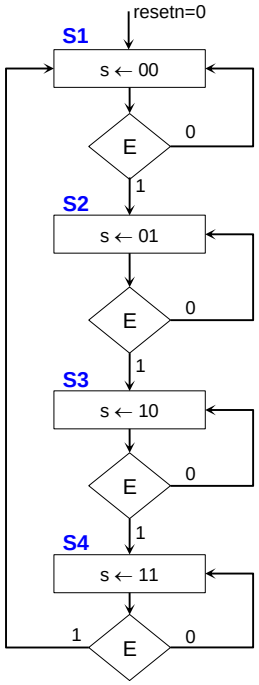
7-SEGMENT SERIALIZER

DIGITAL SYSTEM (FSM + Datapath circuit)

- Most FPGA Development boards have a number of 7-segment displays (e.g., 4, 8). However, only one can be used at a time.
- If we want to display four digits (inputs A, B, C, D), we can design a serializer that will only show one digit at a time on the 7-segment displays.
- Since only one 7-segment display can be used at a time, we need to serialize the four BCD outputs. In order for each digit to appear bright and continuously illuminated, each digit is illuminated for 1 ms every 4 ms (i.e. a digit is un-illuminated for 3 ms and illuminated for 1 ms). This is taken care of by feeding the output z of the 'counter to 0.001s' to the enable input of the FSM. This way, state transitions only occur each 0.001 s.
- Nexys-4/Nexys-4 DDR Board: For each display, we control the individual cathodes (7) of each LED: these active-low signals. The anode is common: his is the enable signal (active-low). The board has eight 7-segment displays; we are only using four displays in this circuit: thus, we need to control 4 enable signals and disable the remaining 4 (`buf(7..4) = 0`).



- **Algorithmic State Machine (ASM) chart:** This is a Moore-type FSM. The output s only depends on the present state. Note that this is actually a counter from 0 to 3 with enable.



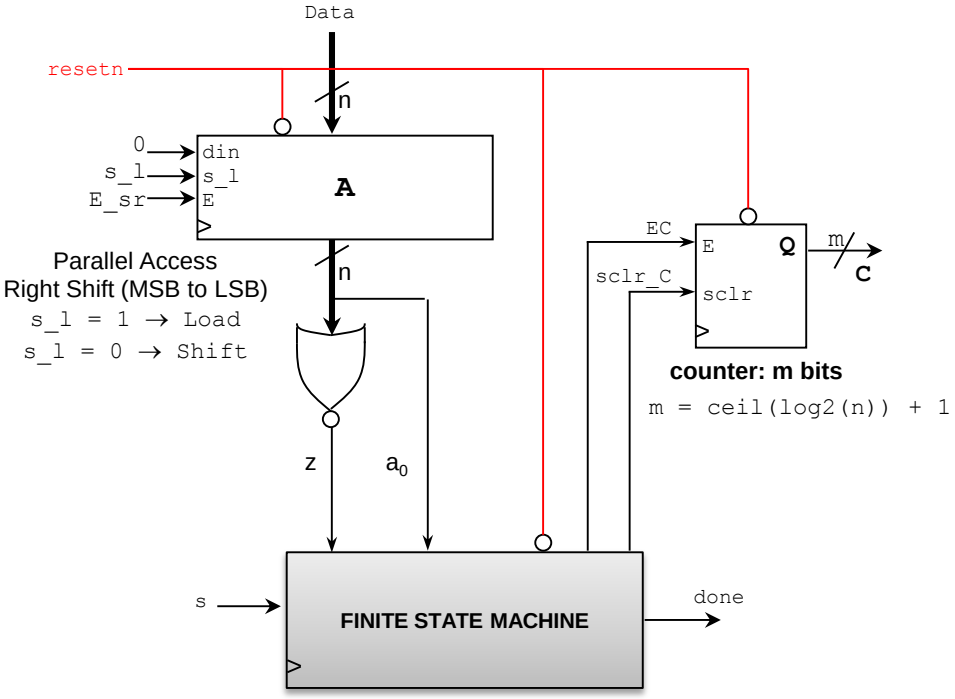
BIT-COUNTING CIRCUIT

SEQUENTIAL ALGORITHM

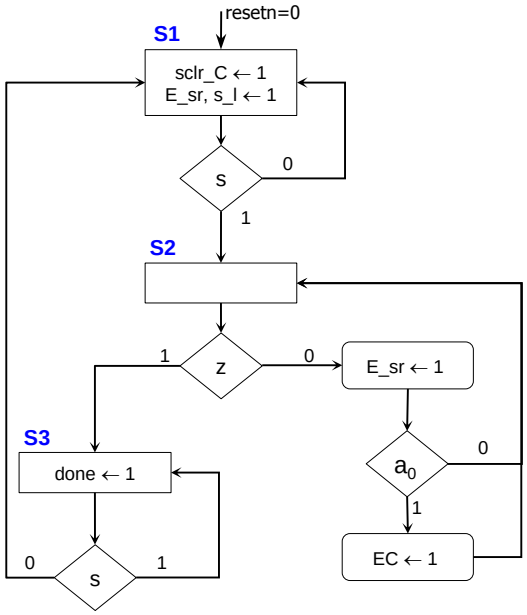
```
C ← 0
while A ≠ 0
   if a0 = 1 then
      C ← C + 1
   end if
   right shift A
end while
```

DIGITAL SYSTEM (FSM + Datapath circuit)

- **Counter Design:** EC=1 increases the count. `sclr`: Synchronous clear. The way this is designed, if sclr = '1', the count is initialized to zero (here, we do not need EC to be 1).



- **Algorithmic State Machine (ASM) chart:** Mealy FSM

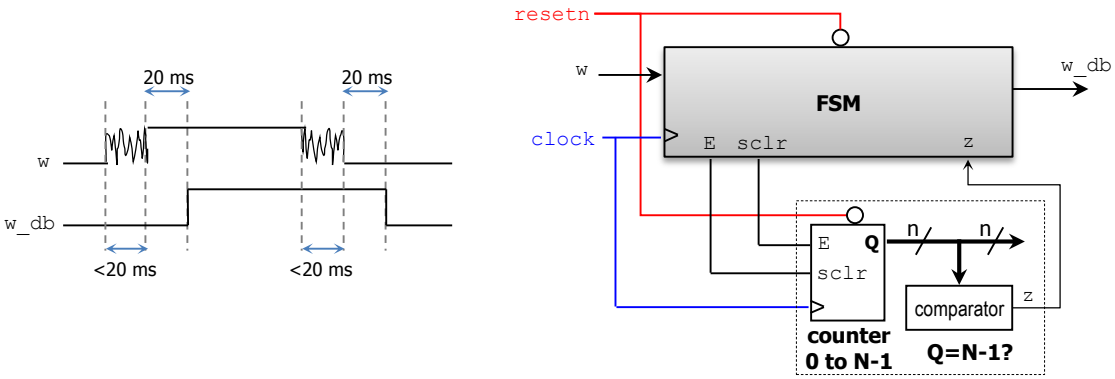


DEBOUNCING CIRCUIT

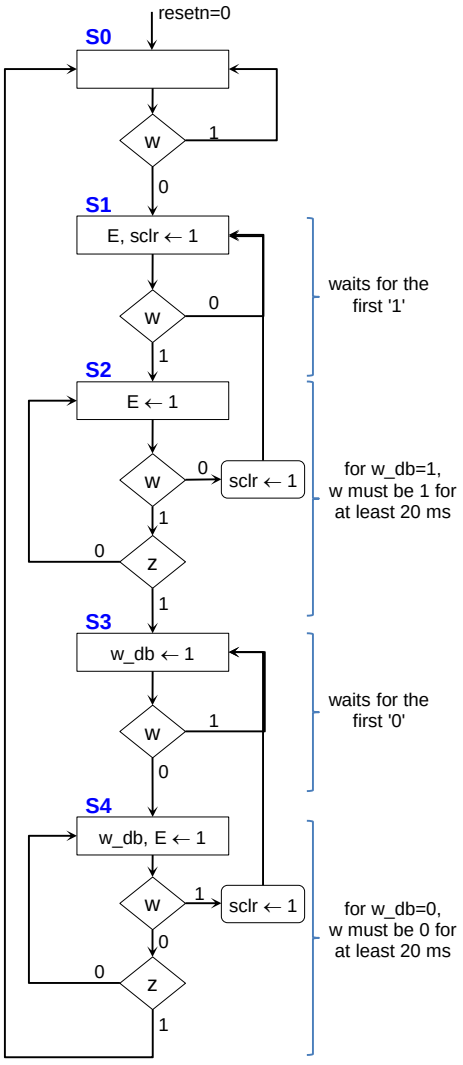
- Mechanical bouncing lasts approximately 20 ms. The, we have to make sure that the input signal w is stable ('1') for at least 20 ms before we assert w_db. Then, to deassert w_db, we have to make that the w is stable ('0') for at least 20 ms.

DIGITAL SYSTEM (FSM + Datapath circuit)

- **Counter 0 to N-1:** `E=1 → Q = Q+1`. `sclr`: Synchronous clear. The way it is designed, if sclr = '1' and E='1', then `Q=0`. If T is the period of the clock signal, then $N = \frac{20ms}{T}$.
  For example, for 100 MHz input clock, T = 10 ns. Then $N = \frac{20ms}{10ns} = 2 \times 10^6$



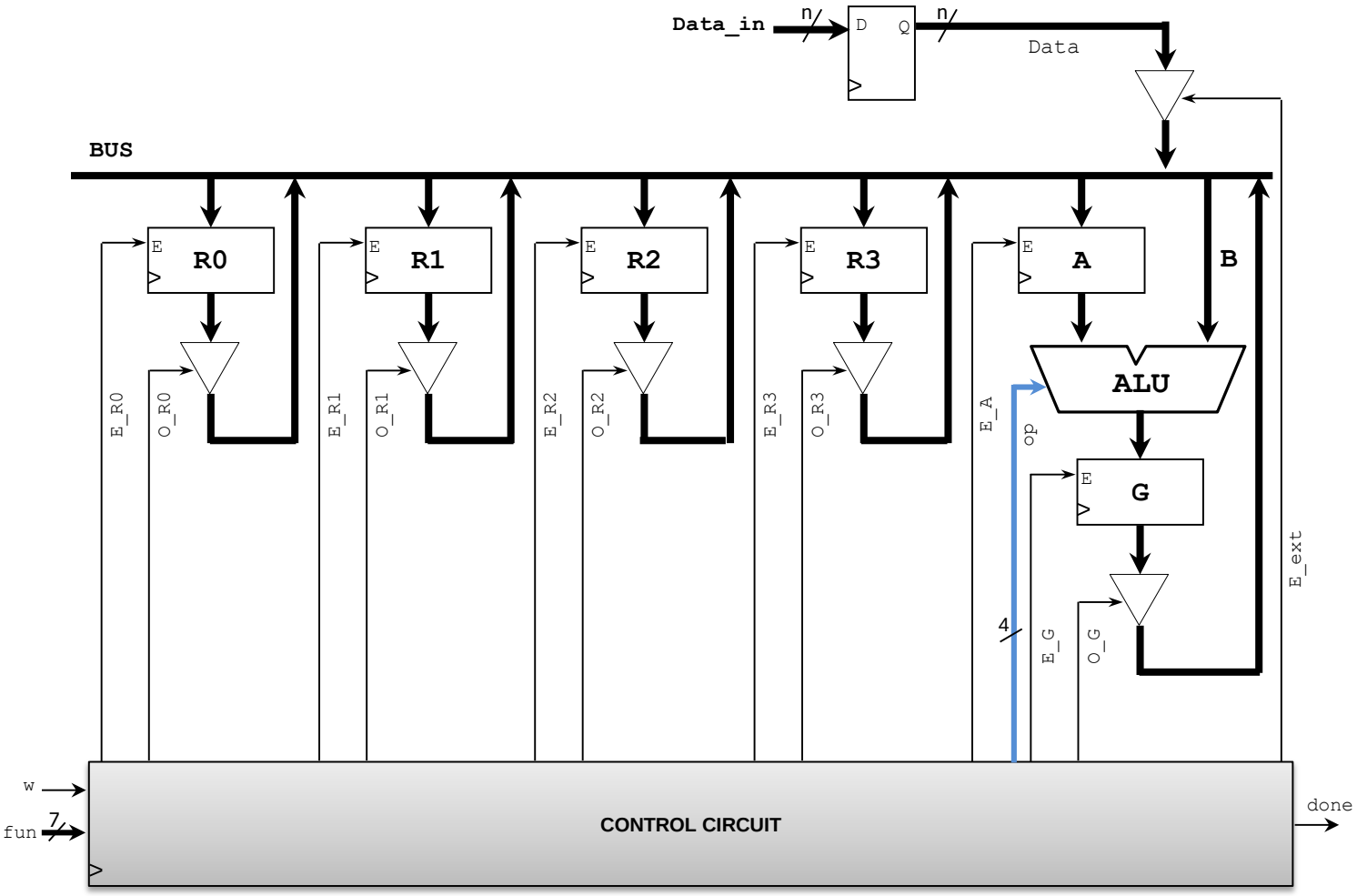
- **Algorithmic State Machine:**



SIMPLE PROCESSOR

DIGITAL SYSTEM (FSM + Datapath circuit)

- This system is a basic Central Processing Unit (CPU). For completeness, a memory would need to be included.
- Here, the Control Circuit could be implemented as a State Machine. However, in order to simplify the State Machine design, the Control Circuit is partitioned into a datapath circuit and a FSM.

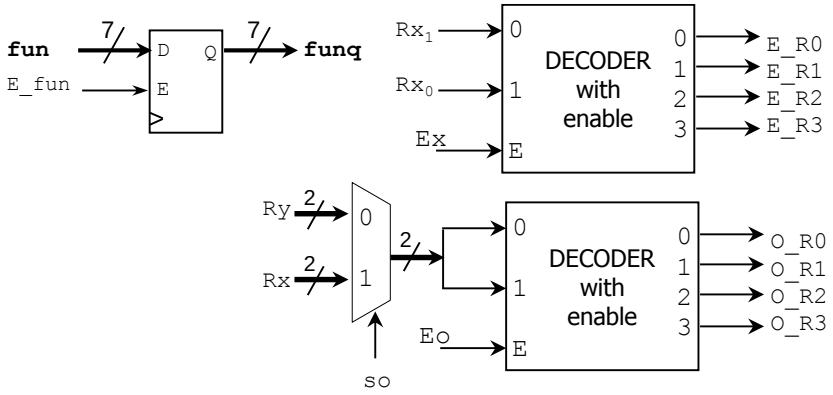


OPERATION

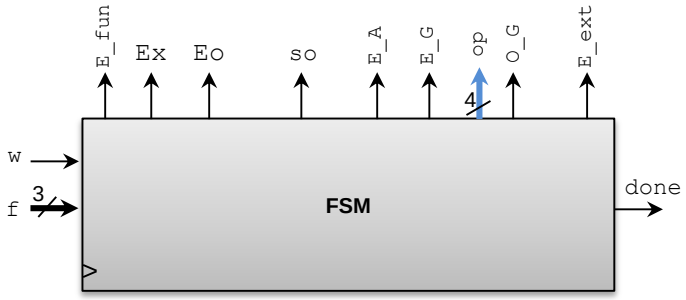
- Every time w = '1', we grab the instruction from fun and execute it.
- $Instruction = |f_2|f_1|f_0|Ry_1|Ry_0|Rx_1|Rx_0|$. This is called 'machine language instruction' or Assembly instruction:
  - ✓ $f_2f_1f_0$: Opcode (operation code). This is the portion that specifies the operation to be performed.
  - ✓ Rx: Register where the result of the operation is stored (we also read data from Rx). Rx can be R1, R2, R3, R4.
  - ✓ Ry: Register where we only read data from. Ry can be R1, R2, R3, R4.

$f = f_2f_1f_0$	Operation	Function
000	Load Rx, Data	Rx ← Data
001	Move Rx, Ry	Rx ← Ry
010	Add Rx, Ry	Rx ← Rx + Ry
011	Sub Rx, Ry	Rx ← Rx - Ry
100	Not Rx	Rx ← NOT (Rx)
101	And Rx, Ry	Rx ← Rx AND Ry
110	Or Rx, Ry	Rx ← Rx OR Ry
111	Xor Rx, Ry	Rx ← Rx XOR Ry

- **Control Circuit:**
  This is made out of some combinational units, a register, and a FSM:
  - ✓ `Ex`: Every time we want to enable register Rx, the FSM only asserts `Ex` (instead of controlling `E_R0`, `E_R1`, `E_R2`, `E_R3` directly). The decoder takes care of generating the enable signal for the corresponding register Rx.
  - ✓ `Eo`, `so`: Every time we want to read from register Ry (or Rx), the FSM only asserts `Eo` (instead of controlling `O_R0`, `O_R1`, `O_R2`, `O_R3` directly) and `so` (which signals whether to read from Rx or Ry). The decoder takes care of generating the enable signal for the corresponding register Rx or Ry.



`funq = |f2|f1|f0|Ry1|Ry0|Rx1|Rx0|`



- **Arithmetic-Logic Unit (ALU):**

op	Operation	Function	Unit
0000	y <= A	Transfer 'A'	Arithmetic
0001	y <= A + 1	Increment 'A'	
0010	y <= A - 1	Decrement 'A'	
0011	y <= B	Transfer 'B'	
0100	y <= B + 1	Increment 'B'	
0101	y <= B - 1	Decrement 'B'	
0110	y <= A + B	Add 'A' and 'B'	
0111	y <= A - B	Subtract 'B' from 'A'	
1000	y <= not A	Complement 'A'	Logic
1001	y <= not B	Complement 'B'	
1010	y <= A AND B	AND	
1011	y <= A OR B	OR	
1100	y <= A NAND B	NAND	
1101	y <= A NOR B	NOR	
1110	y <= A XOR B	XOR	
1111	y <= A XNOR B	XNOR	

- **Algorithmic State Machine (ASM)**:
  Every branch of the FSM implements an Assembly instruction.