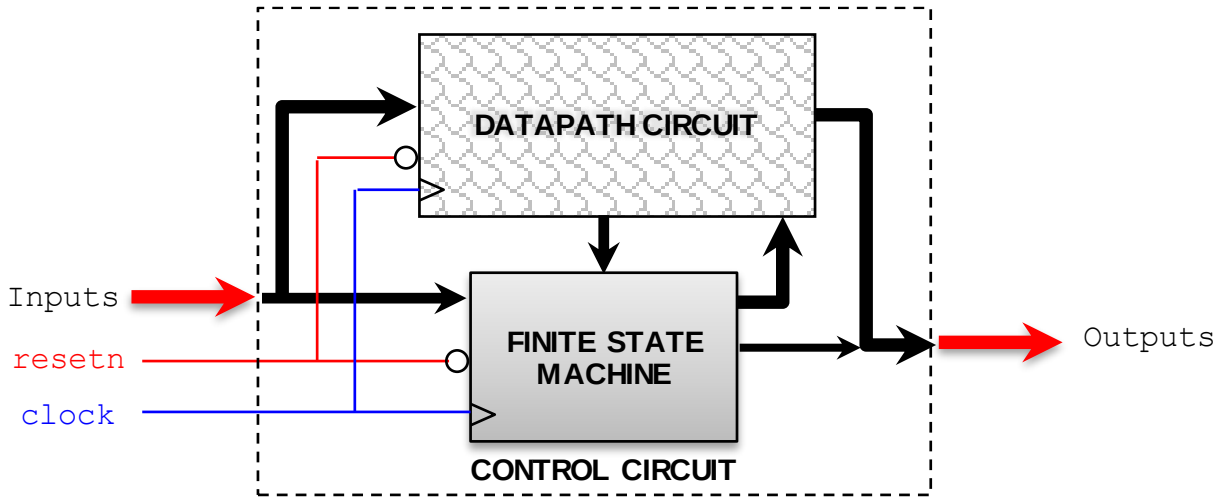
Unit 1 - Digital System Design

Digital System Model

FSM (Control) + Datapath Circuit

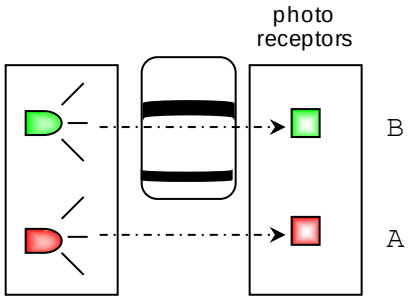


Use of Generic components

- Among the most common components used in the development of digital systems we have registers, shift registers, and counters. To optimize design time, it is recommended to use parameterized components (VHDL for FPGA Tutorial – Unit 5):
  - ✓ n-bit register with enable and synchronous clear: my_rege
  - ✓ Counter modulo-N with enable and synchronous clear: my_genpulse_sclr
  - ✓ n-bit parallel access (right/left) register with enable and synchronous clear: my_pashiftreg_sclr

Examples

Car Lot Counter



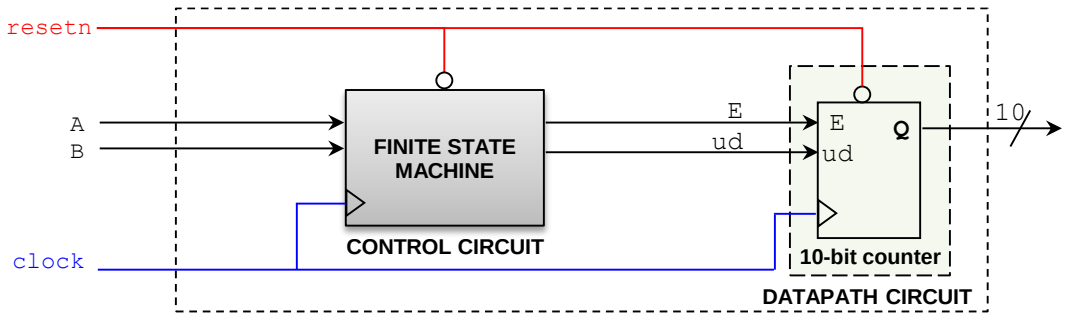
If A = 1 → No light received (car obstructing LED A)
If B = 1 → No light received (car obstructing LED B)

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

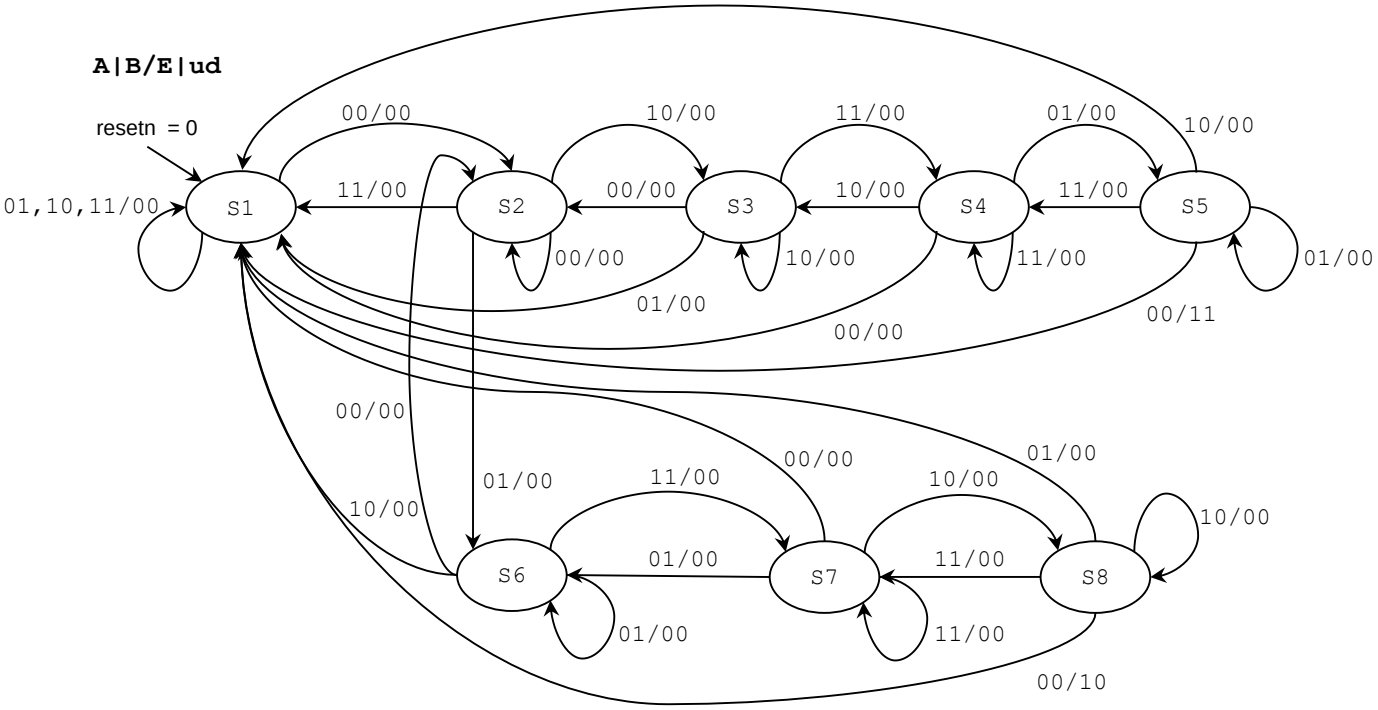
A car might stay in a state for many cycles since the car speed is very large compared to that of the clock frequency.

DIGITAL SYSTEM (FSM + Datapath circuit)

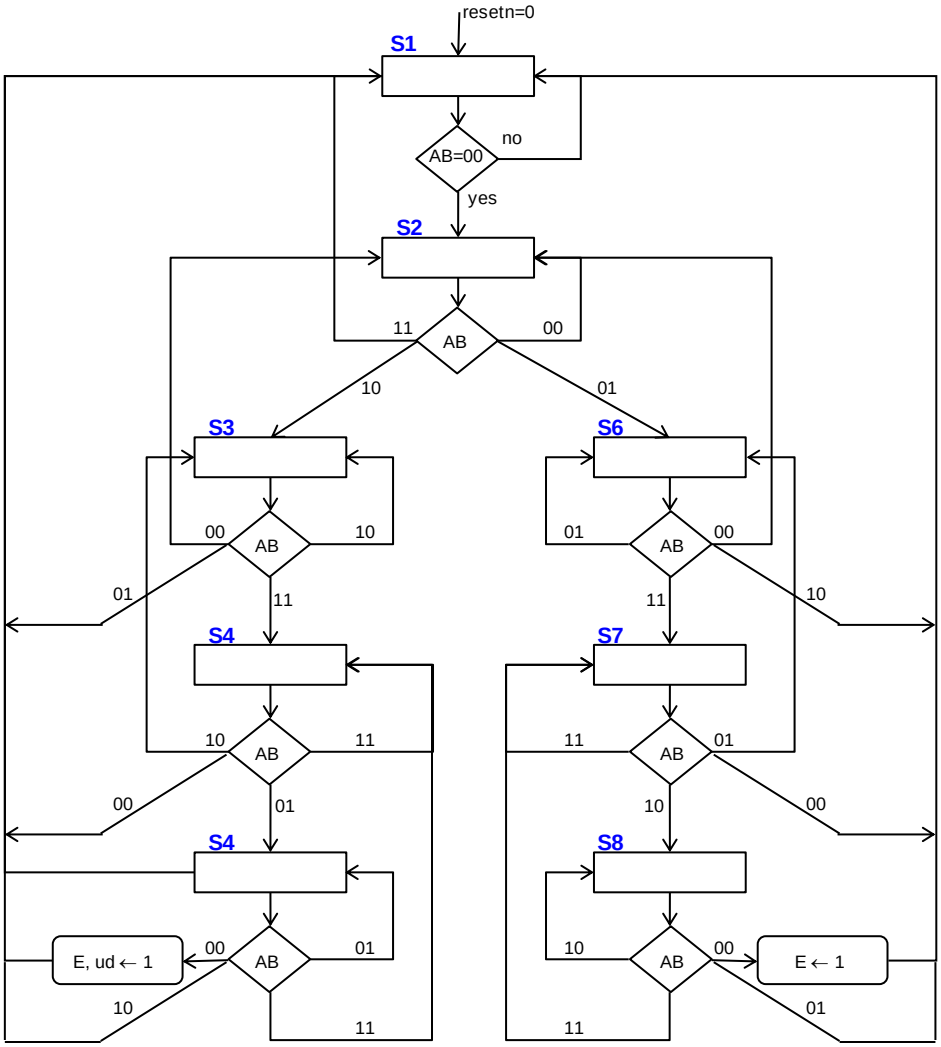
- Usually, when $resetn$ (asynchronous clear) and $clock$ are not drawn, they are implied.



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



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



DEBOUNCING CIRCUIT (VHDL CODE)

- 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)

- Generic component (my_genpulse_sclr): Behavior on the clock tick:

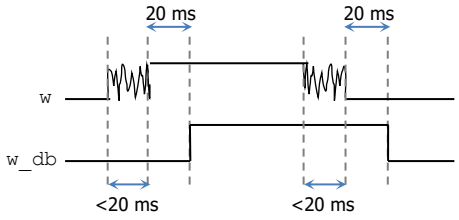
Counter modulo-N (0 to N-1): If E=0, the count stays

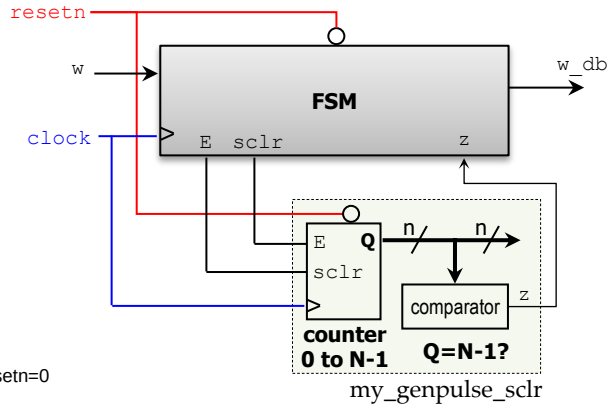
```
if E = 1 then
   if sclr = 1 then
      Q ← 0
   elseif Q = N-1 then
      Q ← 0
   else
      Q ← Q+1
  end if;
end if;
* z = 1 if Q = N-1 (max. count)
```

Getting the value of N for a given time (e.g. 20 ms.)

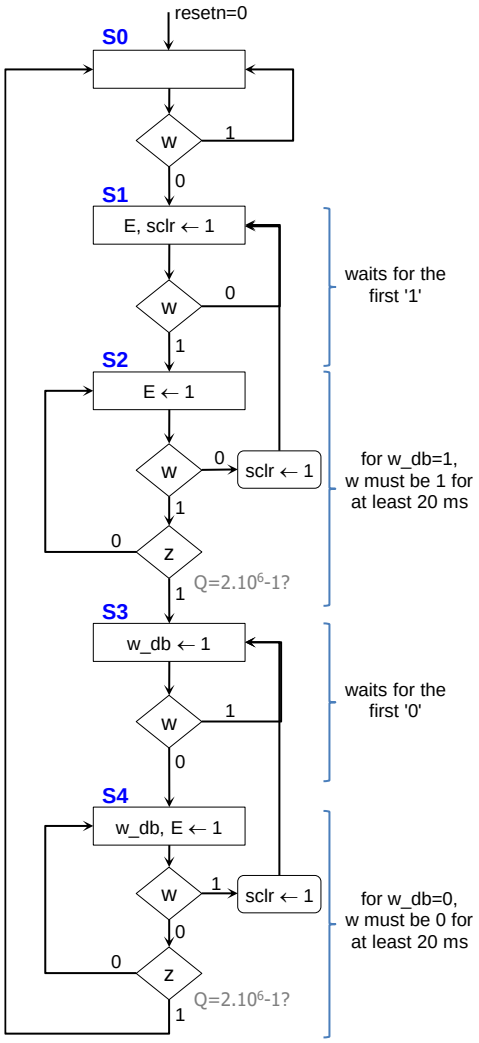
If T is the period of the clock signal, then $N = \frac{20ms}{T}$

For 100 MHz input clock, T = 10 ns.
Then $N = \frac{20ms}{10ns} = 2 \times 10^6$





- **Algorithmic State Machine:**



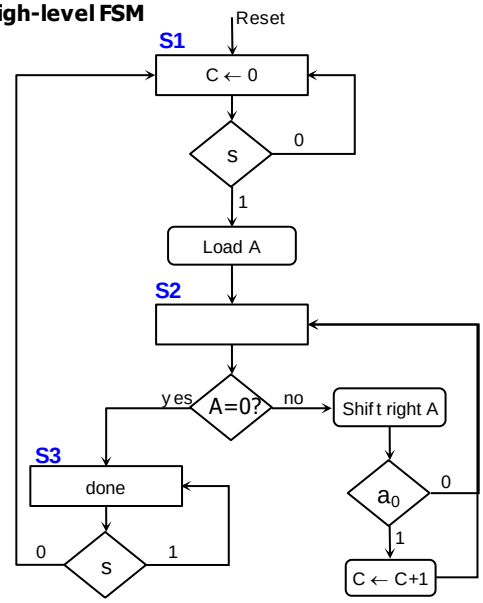
BIT-COUNTING CIRCUIT (COUNTING 1'S) (VHDL CODE)

- This circuit counts the number of bits in register A whose value is '1'. Example: $A = 00110111 \rightarrow C = 0101$.
- The sequential algorithm (pseudo-code) is available. In this case, we can follow this procedure to design a digital system:
  - ✓ Sketch the block diagram: We need start and done signals to indicate when the process starts and finishes. We also include input (Data for the Register A) and output data (Count).
  - ✓ Sketch the high-level control mechanism (state machine) for the Bit-counting circuit. Here, you can include combinational blocks as well as common synchronous blocks (registers, shift registers, counters).
  - ✓ With the block diagram and high-level control mechanism, we can start including the components and their signals in the datapath. Based on the high-level state machine, we can design the actual state machine that includes specific signals controlling the components.

Sequential Algorithm

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

High-level FSM



DIGITAL SYSTEM (FSM + Datapath circuit)

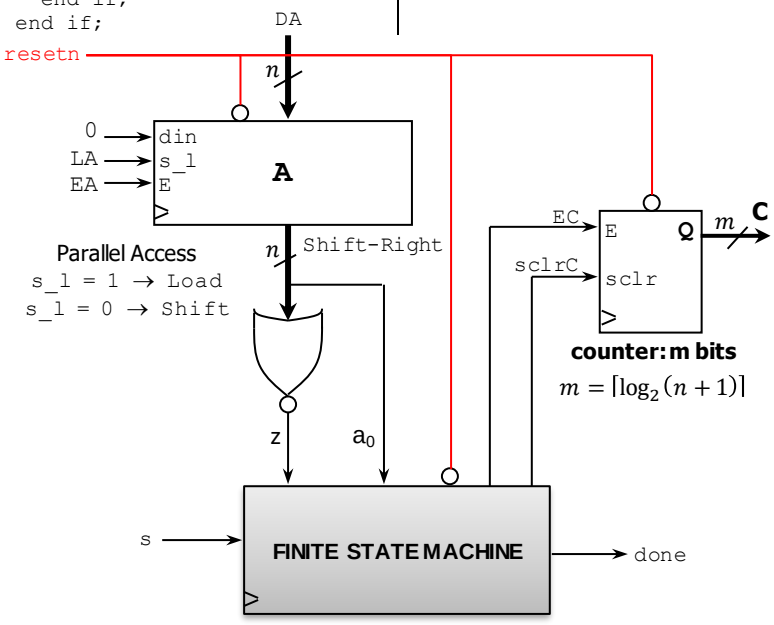
- FSM: as a ASM chart. Example: For $n = 8$: if $A = 00110110$, then $C = 0100$.
- Generic components: Behavior on the clock tick:

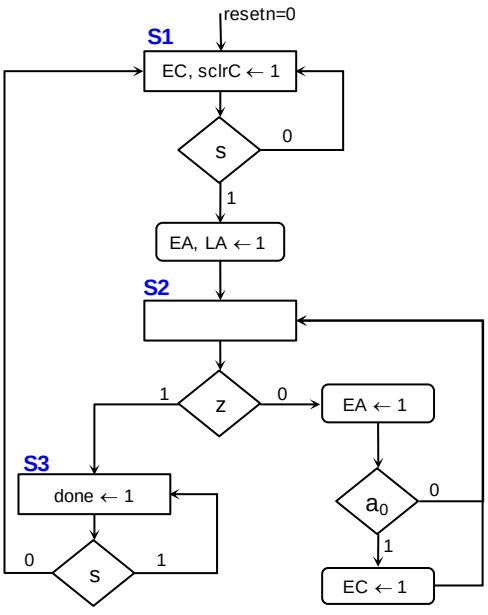
Counter modulo-n+1 (0 to n):
If E=0, the count stays.

```
if E = 1 then
   if sclr = 1 then
      Q ← 0
   elseif Q = n then
      Q ← 0
   else
      Q ← Q+1
  end if;
end if;
```

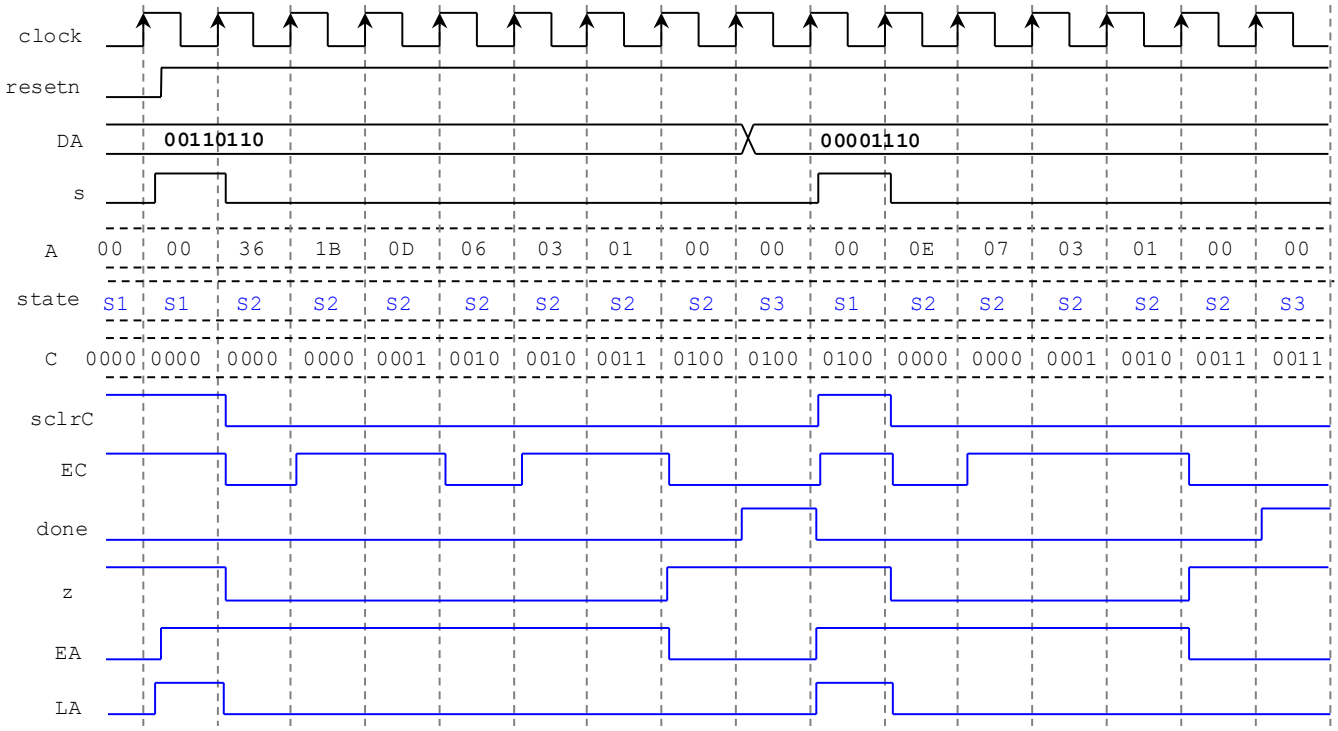
n-bit Parallel access shift register:
If E=0, the output is kept

```
if E = 1 then
   if s_l = '1' then
      Q ← D
   else
      Q ← shift in 'din' (to the right)
  end if;
end if;
```





Timing diagram $(n = 8, m = 4)$:



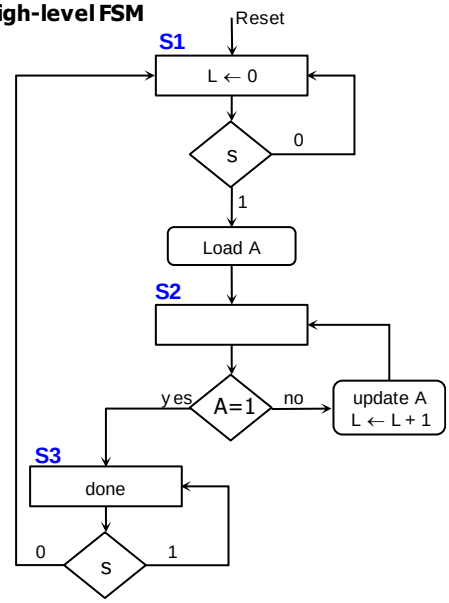
INTEGER BASE-2 LOGARITHM (VHDL CODE)

- This circuit computes $\lceil \log_2 A \rceil$. Data (unsigned integer) is contained in register A (n bits)
- The sequential algorithm (pseudo-code) is available. We follow the procedure specified in the previous example.
- Note that instead of performing a right shift to A to get $\lfloor A/2 \rfloor$, we will get $\lfloor A/2 \rfloor$ by created a shifted version of A. Example (n=4): $A = a_3a_2a_1a_0 \rightarrow \lfloor A/2 \rfloor = 0a_3a_2a_1$.
- Also asking $A = 1$ is the same as asking whether $\lfloor A/2 \rfloor = 0$.

Sequential Algorithm

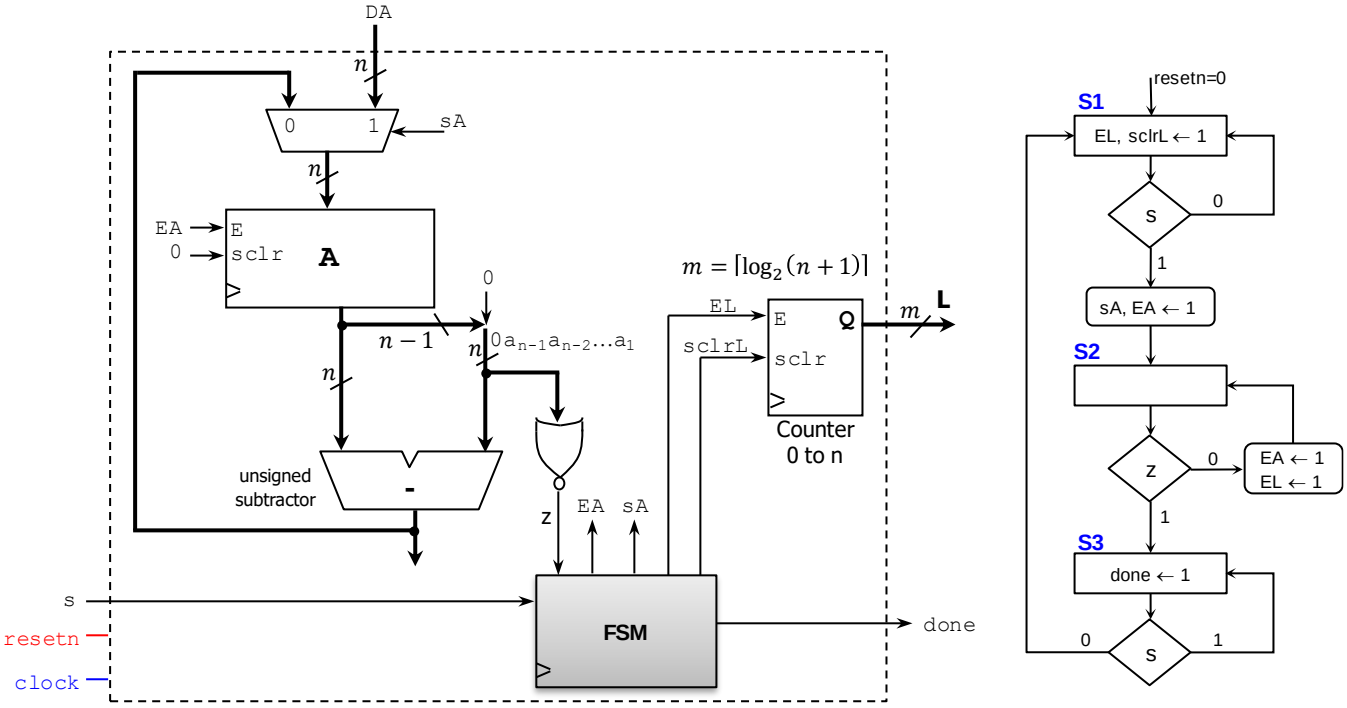
```
L ← 0
while A ≠ 1
    A ← A - ⌊A/2⌋
    L ← L + 1
end while
```

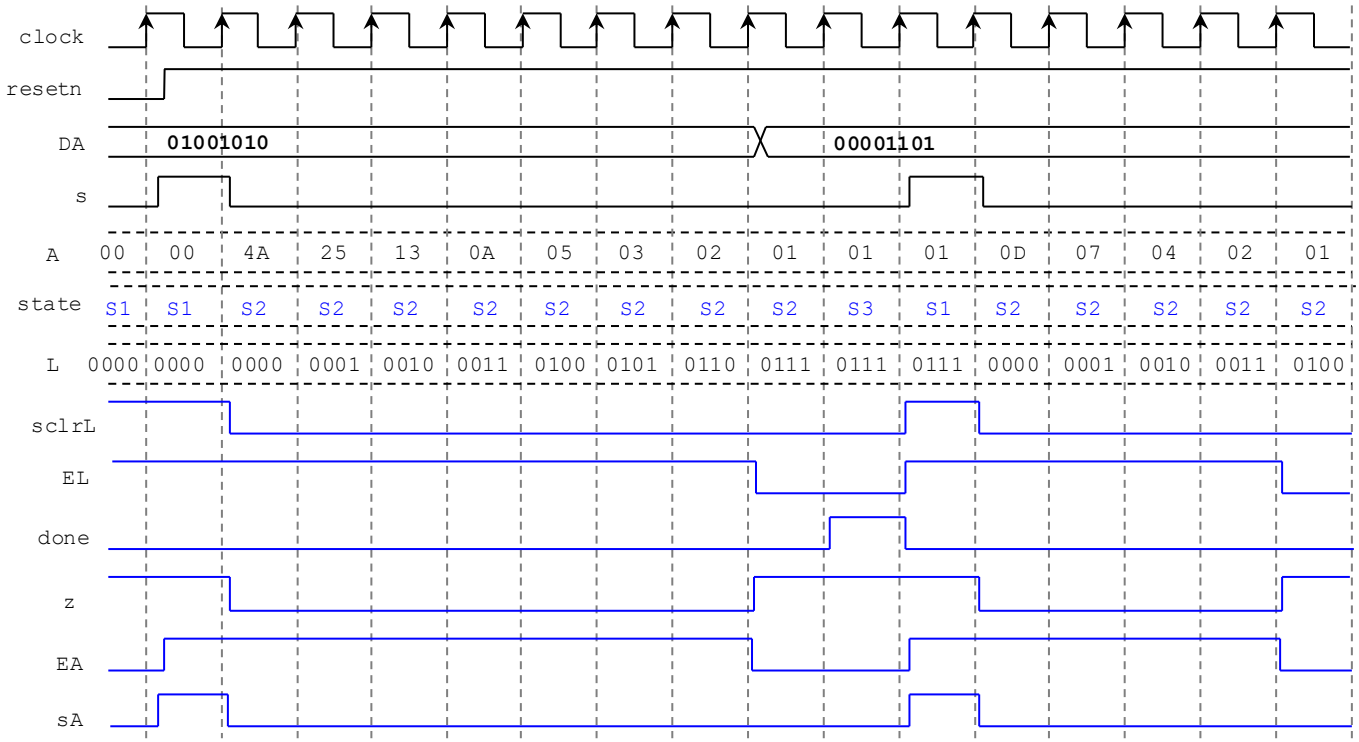
High-level FSM



DIGITAL SYSTEM (FSM + Datapath circuit)

- The digital system is depicted below. For n bits, $\lceil \log_2 A \rceil \in [0, n]$. Thus, the result needs $\lceil \log_2 (n+1) \rceil$ bits.
  - ✓ m-bit counter: If $E = sclr = 1$, the count is initialized to zero. If $E = 1, sclr = 0$, the count is increased by 1.
  - ✓ Register: If $E = 1$: $sclr = 1 \rightarrow$ Clear, $sclr = 0 \rightarrow$ Load.
  - ✓ Note that $z = 1$ when $\lfloor A/2 \rfloor = 0$ (this includes the case $A = 0$).





GREATEST COMMON DIVISOR (GCD)

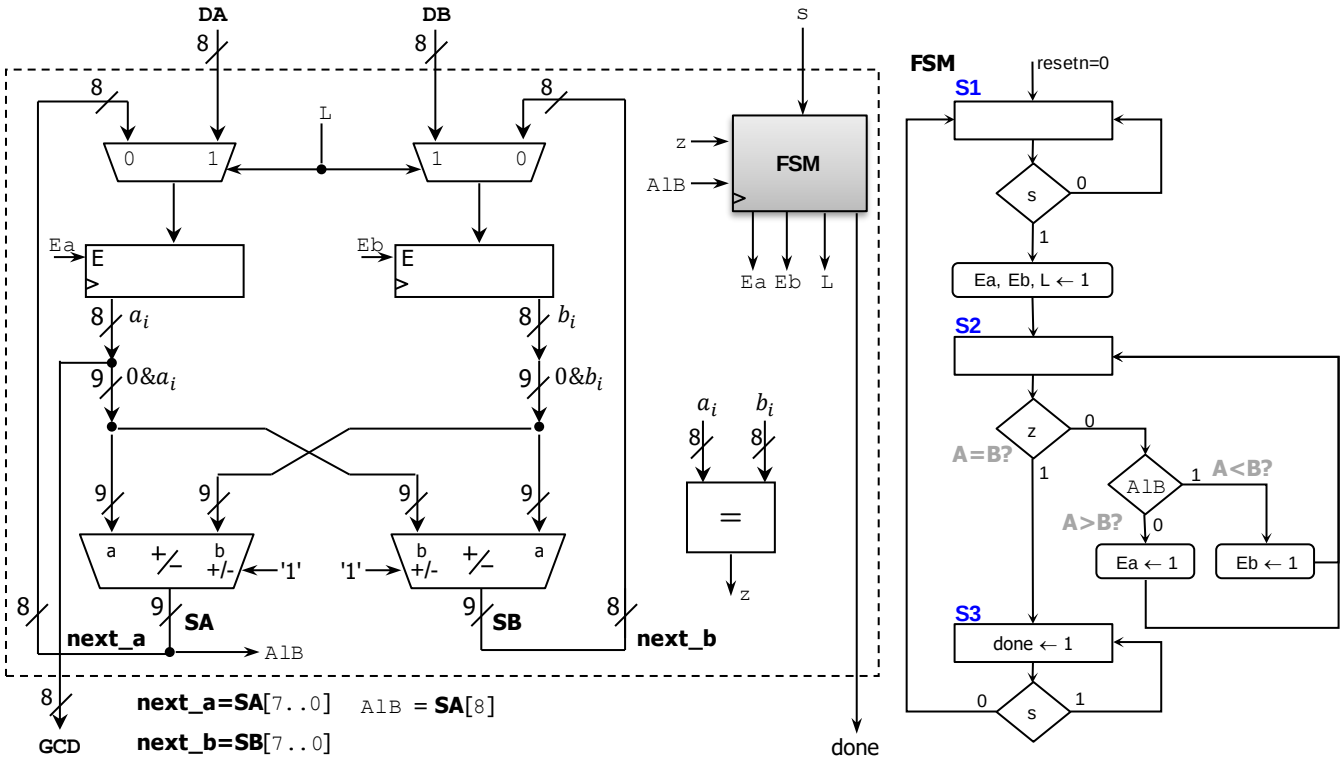
- This circuit processes two n-bit unsigned numbers (A, B) and generates the GCD of A and B. For example:
  - ✓ If A = 216, B = 192 → GCD = 24.
  - ✓ If A = 132, B = 72 → GCD = 12.
  - ✓ If A = 169, B = 63 → GCD = 1.
- Sequential algorithm: we will use the Euclid's GCD Algorithm:
  - ✓ Based on this algorithm, we design an iterative circuit.

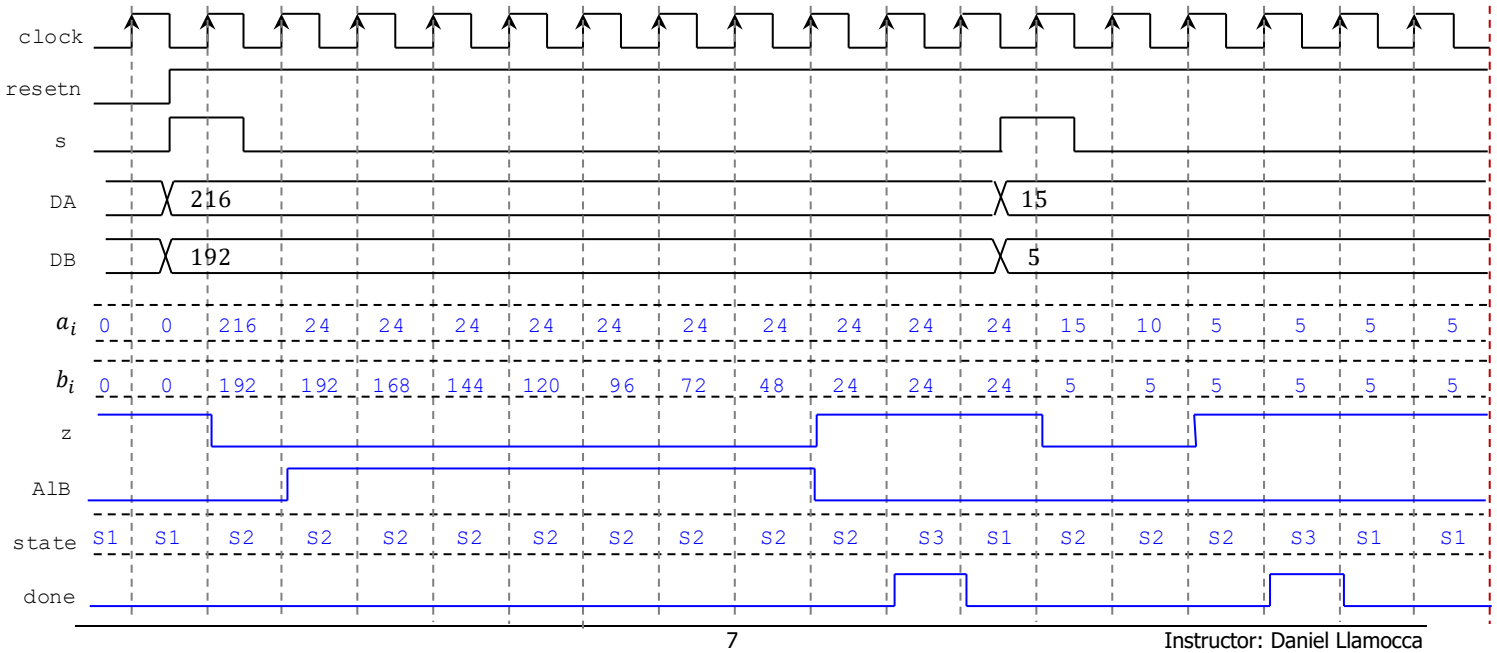
Sequential Algorithm

```
a,b: unsigned integers
while a ≠ b
    if a > b
        a ← a-b
    else
        b ← b-a
    end
end while
return a
```

DIGITAL SYSTEM (FSM + Datapath circuit)

- The figure depicts the FSM (in ASM form) and a datapath circuit for $n = 8$.
  Input data: DA, DB. Output data: GCD.

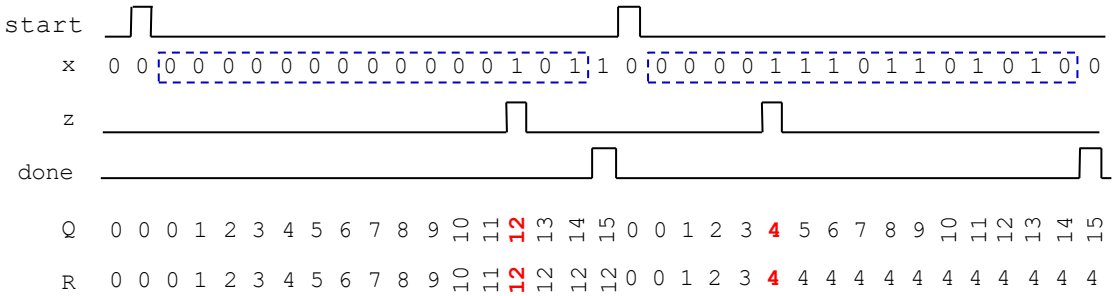




LEADING ZERO DETECTOR (ITERATIVE)

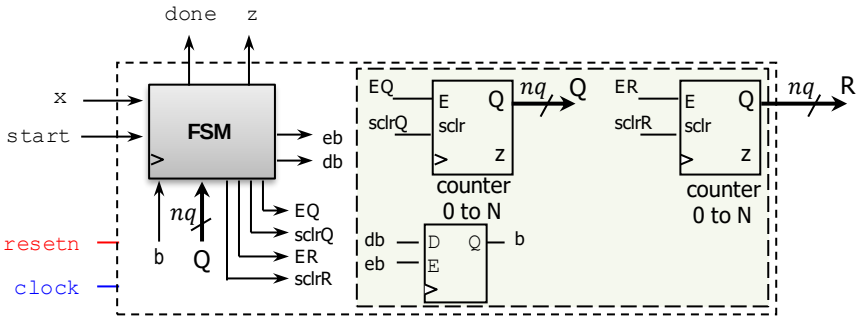
- A sequence of N bits is fed to the circuit serially. The circuit generates the number of 0's that exist before the first 1.
- Example (N=15):
  - ✓ If the number is: 000000000000101 → Output: 12
  - ✓ If the number is: 000000000000000 → Output: 15
  - ✓ If the number is: 000010001000001 → Output: 4

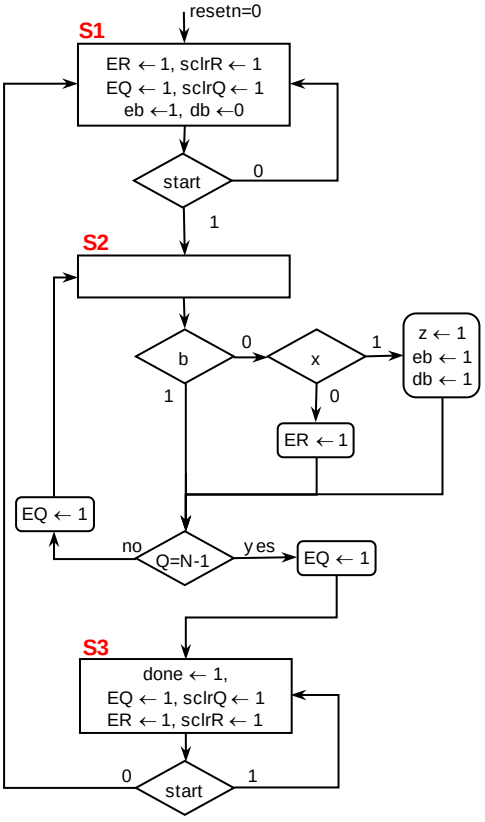
$$nq = \lceil \log_2(N+1) \rceil$$



DIGITAL SYSTEM (FSM + Datapath circuit)

- Inputs: X (serial data, MSB first), start.
- Outputs: z (leading 1 detected), done (N bits processed), and R (number of 0's before the first 1).
- The process beings with the assertion of start. When the first 1 is detected, then z=1 (for 1 clock cycle) and R is frozen. When all the bits of the sequence have been processed, done=1. We can re-start the process after this. Note that if X is just 0's, z is never '1'.
- Counter R and Q: modulo-(N+1): count from 0 to N.

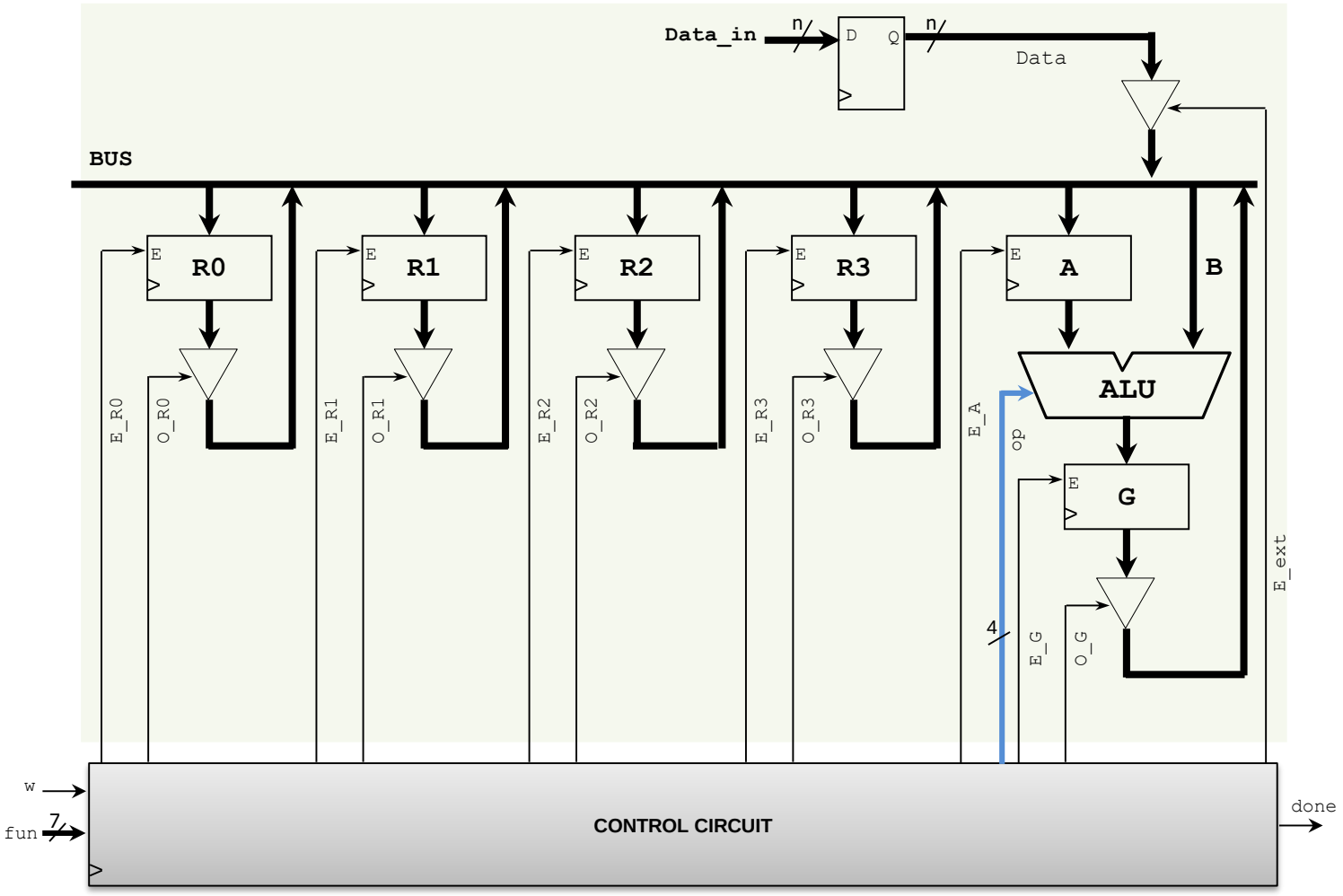




SIMPLE PROCESSOR

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.

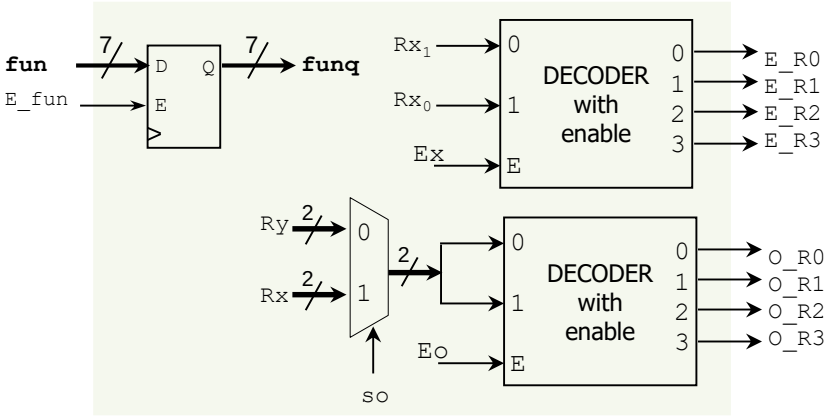


OPERATION

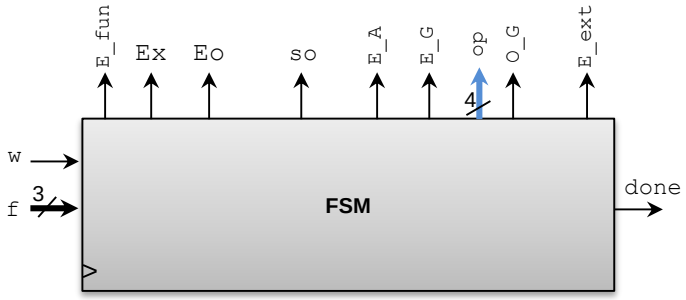
- Every time w = '1', we grab the instruction from fun and execute it.
- $Instruction = |f_2|f_1|f_0|Ry_1|Ry_0|Rx_1|Rx_0|$. This is called 'machine language instruction' or Assembly instruction:
  - ✓ $f_2f_1f_0$: Opcode (operation code). This is the portion that specifies the operation to be performed.
  - ✓ 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.



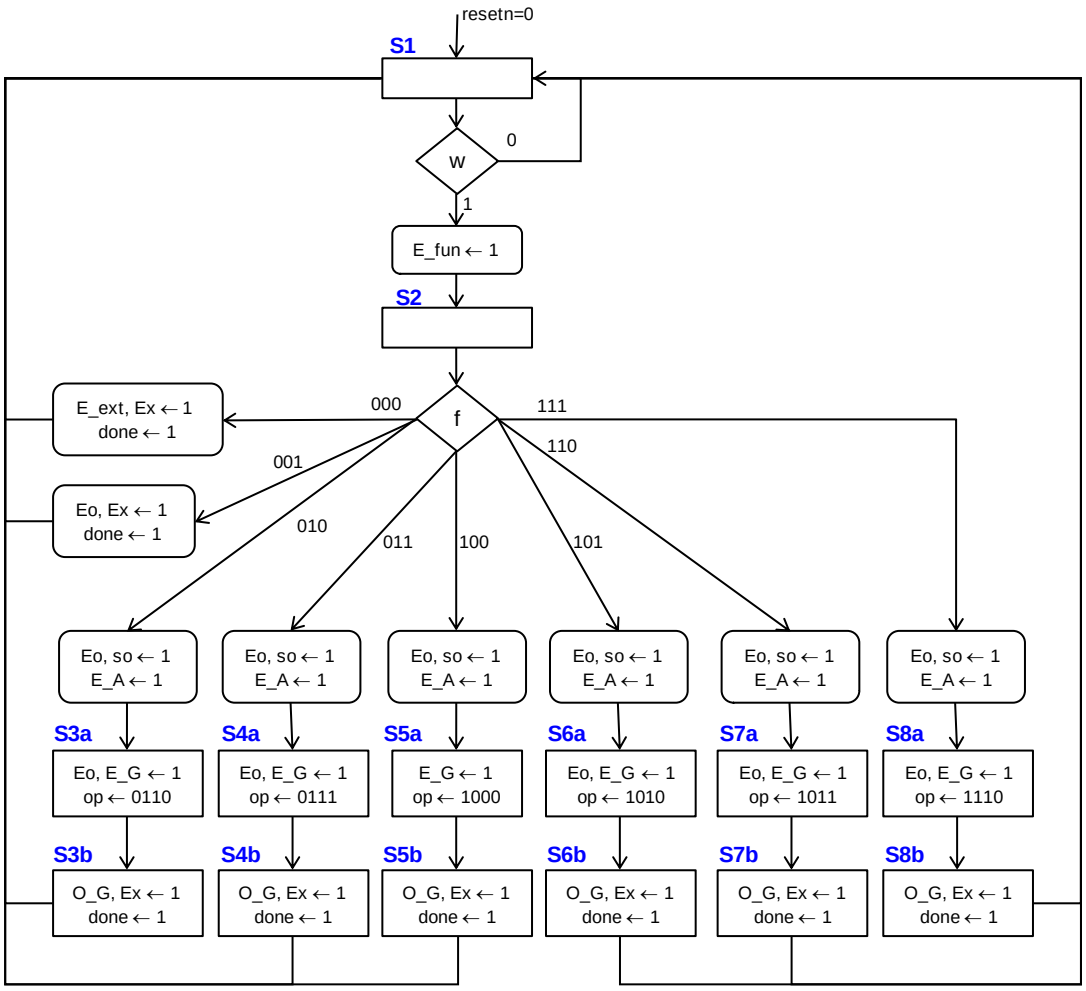
`funq = |f2|f1|f0|Ry1|Ry0|Rx1|Rx0|`



- **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.



BINARY TO BCD CONVERSION (VHDL CODE)

DOUBLE DABBLE ALGORITM

- Given an N-bit unsigned binary number, we left shift every one of the N bits into a new register (that holds the BCD digits). We thus have N iterations. If after shifting, any of the BCD digits (one or more) is greater than 4, we increment it by 3. The exception is on the last iteration, where after shifting, we do not increment any nibble.
- For N bits, we need $\lceil N/3 \rceil$ BCD digits. So the BCD register is $4 \times \lceil N/3 \rceil$ bits wide.

Number of bits	Binary range	Number of BCD digits
4	[0-15]	2
7	[0-127]	3
10	[0-1023]	4
14	[0-16383]	5
N	$[0, 2^N - 1]$	$\lceil N/3 \rceil$

- Why does it work? Think of BCD shifting: we want to shift the bits of a BCD number, but preserving the BCD representation. If a number lower or equal than 4 is shifted, we get at most 8, still representable in BCD. If a number greater than 4 is shifted, the minimum number we get is $10=1010_2$, which is not in BCD. If we add 3 to the number and then shift, we will get the proper BCD result with 2 BCD digits.
  - ✓ Example: $5=0101_2$. If we shift, we get 1010_2. 1010_2 would be 0001 0000 in BCD. To get this, we can add 3 to 0101_2, resulting in 1000_2. After shifting, we get 0001 0000. The same happens for numbers 6, 7, 8, and 9.

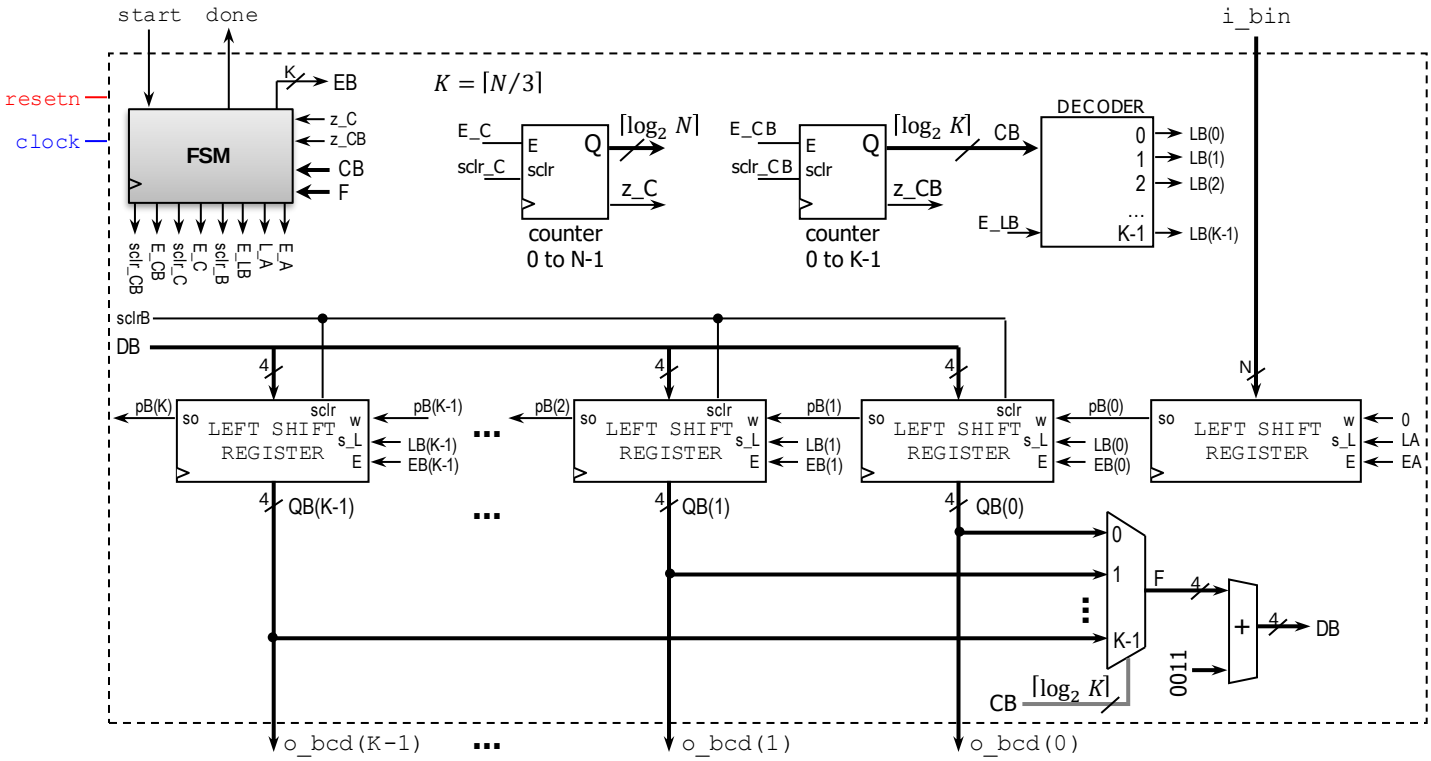
- **Example**: 255 = 11111111 (N=8). We need $\lceil N/3 \rceil = 3$ BCD digits. Note that there are $N = 8$ iterations.
  The table shows the state after an operation has been applied. In the example, only a nibble gets incremented at a time.

BCD number	Binary number	Procedure just applied	Procedure to apply
0000 0000 0000	1111 1111	Initialization	Left shift
0000 0000 0001	1111 1110	Left shift	Left shift
0000 0000 0011	1111 1100	Left shift	Left shift
0000 0000 0111	1111 1000	Left shift.	0111>4. Then we add 3 to 0111
0000 0000 1010	1111 1000	Add 3 to 0111	Left shift
0000 0001 0101	1111 0000	Left shift.	0101>4. Then we add 3 to 0101
0000 0001 1000	1111 0000	Add 3 to 0101	Left shift
0000 0011 0001	1110 0000	Left shift	Left shift
0000 0110 0011	1100 0000	Left shift.	0110>4. Then we add 3 to 0110
0000 1001 0011	1100 0000	Add 3 to 0110	Left shift
0001 0010 0111	1000 0000	Left shift	0111>4. Then we add 3 to 0111
0001 0010 1010	1000 0000	Add 3 to 0111	Left shift
0010 0101 0101	0000 0000	Left shift.	None. We do not add 3 in the last iteration

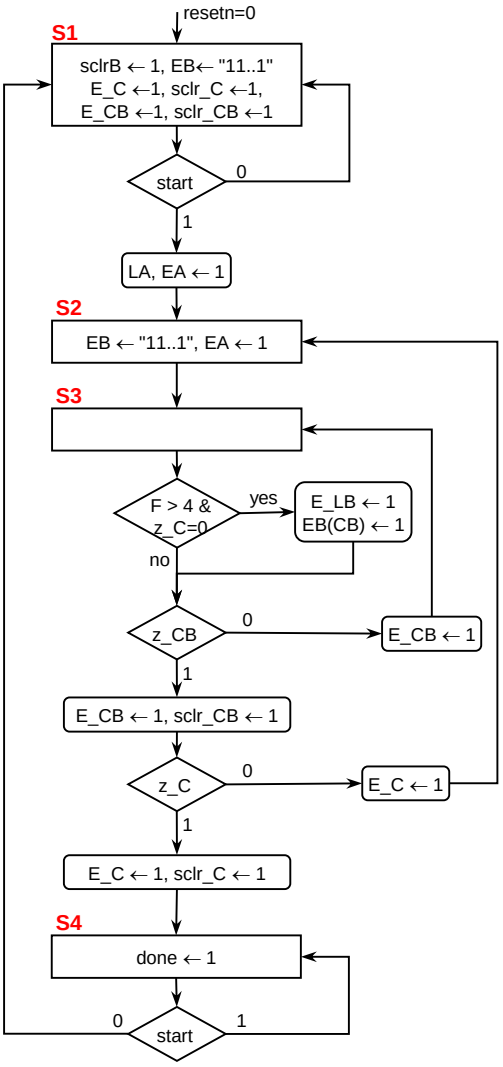
- **Example**: 172 = 10101100 (N=8). We need $\lceil N/3 \rceil = 3$ BCD digits. Note that there are $N = 8$ iterations.
  The table shows the state after an operation has been applied. In general, more than a nibble can be incremented at a time.

BCD number	Binary number	Procedure just applied	Procedure to apply
0000 0000 0000	1010 1100	Initialization	Left shift
0000 0000 0001	0101 1000	Left shift	Left shift
0000 0000 0010	1011 0000	Left shift	Left shift
0000 0000 0101	0110 0000	Left shift	0101>4. Then we add 3 to 0101
0000 0000 1000	0110 0000	Add 3 to 0101	Left shift
0000 0001 0000	1100 0000	Left shift	Left shift
0000 0010 0001	1000 0000	Left shift	Left shift
0000 0100 0011	0000 0000	Left shift	Left shift
0000 1000 0110	0000 0000	Left shift	1000>4, 0110>4. Then we add 3 to 1000 and 0110
0000 1011 1001	0000 0000	Add 3 to 1011 and 1001	Left shift
0001 0111 0010	0000 0000	Left shift	Note. We do not add 3 in the last iteration

DIGITAL SYSTEM (FSM + Datapath circuit)



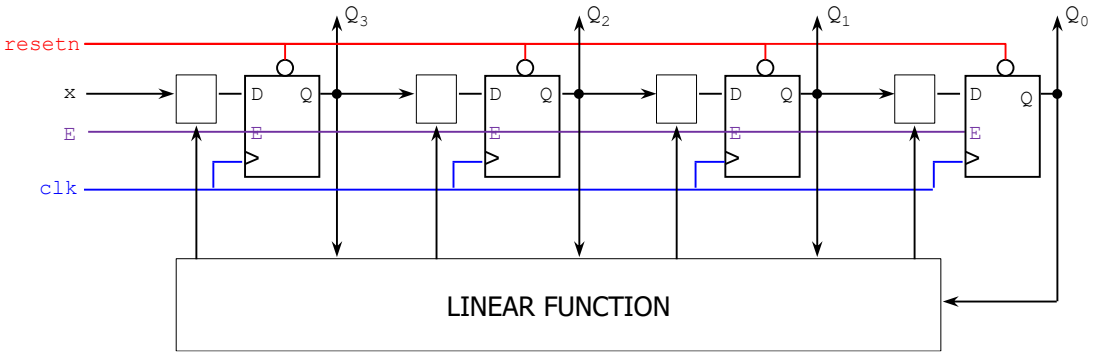
- **Algorithmic State Machine (ASM) Chart:**



EXTRA TOPICS

LINEAR FEEDBACK SHIFT REGISTERS (LFSRS)

- LFSRs have a simple and fairly regular structure. Typical components: D-type flip flops, and logic gates.
- Advantages: very little hardware and high speed operation.
- Inputs to flip flops are linear functions of the previous state.
- Despite the simple appearance, LFSRs are based on rather complex mathematical theory and have interesting applications in the area of digital system testing, cryptography, and fault-tolerant computing.
- Typical applications: Error correction/detection, pseudo random number generators, fast counters.
- A generic 4-bit LFSR is depicted below:

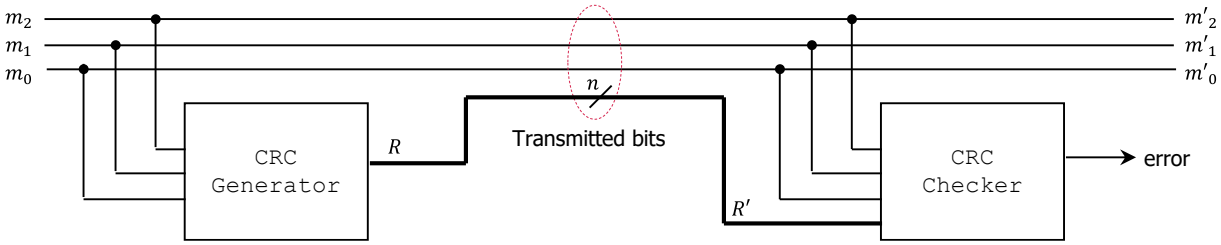


APPLICATION: CYCLIC REDUNDANCY CHECK (CRC)

- This error-detection code is used in digital communications (e.g.: Ethernet, CAN), and storage devices (e.g.: RAMs).

Communication system

- k-bit message: $M = m_{k-1}m_{k-2} \dots m_1 m_0$.
- CRC code: $R = r_{n-1}r_{n-2} \dots r_1 r_0$.
- Stream sent: $m_{k-1}m_{k-2} \dots m_1 m_0 r_{n-1}r_{n-2} \dots r_1 r_0$



- Associated polynomials:
  - ✓ $M(x) = m_{k-1}x^{k-1} + m_{k-2}x^{k-2} + \cdots + m_1 x^1 + m_0 x^0$. Order: $k-1$
  - ✓ $R(x) = r_{n-1}x^{n-1} + r_{n-2}x^{n-2} + \cdots + r_1 x^1 + r_0 x^0$. Order: $n-1$
- The CRC code is calculated as a remainder: $R(x) = remainder\left(\frac{x^n M(x)}{G(x)}\right)$
- $G(x) = g_n x^n + g_{n-1}x^{n-1} + \cdots + g_1 x^1 + g_0 x^0$. This is the generating polynomial of order n. $G = g_n g_{n-1} \dots g_1 g_0$

- Here, when dealing with polynomial operations, we use modulo-2 polynomial arithmetic (Galois field of two elements: GF(2)). The coefficients of the polynomials can only be 0 or 1. The multiplication operation can be implemented as a simple AND gate, and the addition (or subtraction) operation can be implemented by a XOR gate: $a \pm b = a \oplus b, a \in \{0,1\}$.
  - ✓ Example: $x^n + x^n = x^n - x^n = 0$.

- **CRC generator**: It computes $R(x)$. Its input is $x^n M(x) \equiv m_{k-1}m_{k-2} \dots m_1 m_0 00 \dots 0$.

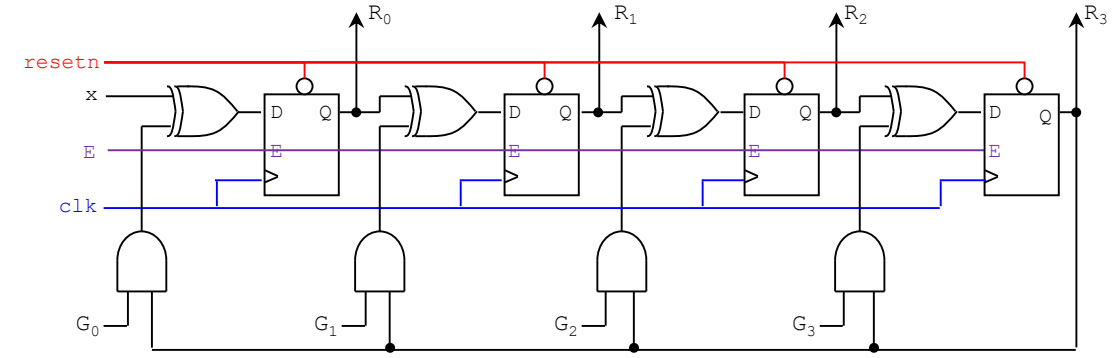
- **CRC checker**: On the receiver side, both the message and CRC code might be corrupted by the channel: $M'(x)$ and $R'(x)$.
  - ✓ So, we must check if $remainder\left(\frac{x^n M'(x)}{G(x)}\right) = R'(x)$. If so, we say that the system passed the CRC check.
  - ✓ Note that this implies: $x^n M'(x) = Q'(x)G(x) + R'(x) \rightarrow x^n M'(x) - R'(x) = x^n M'(x) + R'(x) = Q'(x)G(x)$. In modulo-2 arithmetic, $R'(x) = -R'(x)$. It follows that $x^n M'(x) + R'(x)$ should be divisible by $G(x)$.
  - ✓ The CRC checker tests if $remainder\left(\frac{x^n M'(x) + R'(x)}{G(x)}\right) = 0$. If the remainder is zero, we say that it passed the CRC check. It is still possible for $R'(x)$ and $M'(x)$ to make the remainder equal to zero but this is far less likely.
  - ✓ The input to the CRC checker is $x^n M'(x) + R'(x) \equiv m'_{k-1}m'_{k-2} \dots m'_0 r'_{n-1}r'_{n-2} \dots r'_0$.

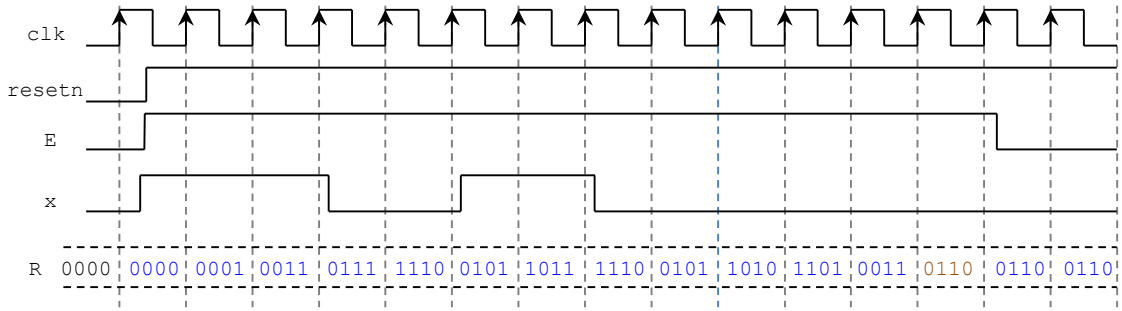
- **Example**: CRC-4 ($n = 4$): $M = 11100110$, $G = 11001$
  $M(x) = x^7 + x^6 + x^5 + x^2 + x$
  $\rightarrow x^n M(x) = x^4(x^7 + x^6 + x^5 + x^2 + x) = x^{11} + x^{10} + x^9 + x^6 + x^5$
  $G(x) = x^4 + x^3 + 1$
  $R(x) = x^2 + x$

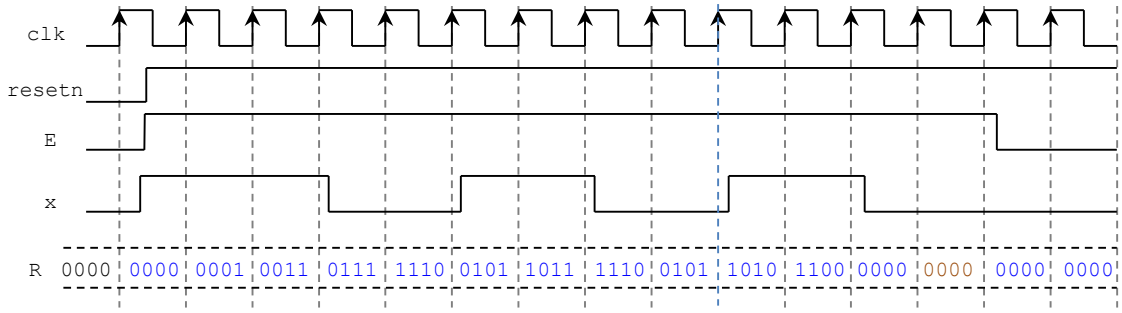
$$
\begin{array}{r|l}
 & x^7 + x^5 + x^4 + x^2 + x \\
\hline
x^4 + x^3 + 1 & x^{11} + x^{10} + x^9 + x^6 + x^5 \\
 & \underline{x^{11} + x^{10} + x^7} \\
 & x^9 + x^7 + x^6 + x^5 \\
 & \underline{x^9 + x^8 + x^5} \\
 & x^8 + x^7 + x^6 \\
 & \underline{x^8 + x^7 + x^4} \\
 & x^6 + x^4 \\
 & \underline{x^6 + x^5 + x^2} \\
 & x^5 + x^4 + x^2 \\
 & \underline{x^5 + x^4 + x} \\
 & x^2 + x
\end{array}
$$

CRC circuit

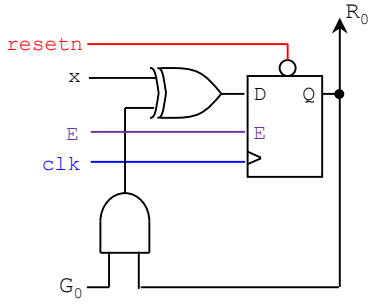
- The following LFSR implements 4-bit CRC. Components: flip flops, AND, XOR gates.
- Note that $g_4 = 1$ is not used. This is because it is common in CRC to have $g_n = 1$. This is already considered in the design of the circuit.
- Example with $M = 11100110$, $G = 11001$:
  - ✓ When integrated into the CRC generator, the input is given by: $x^n M(x) \equiv 11100110\mathbf{0000}$. The output result is $R = 0110$.
  - ✓ When integrated into the CRC checker, the input is given by: $x^n M'(x) + R'(x) \equiv 11100110\mathbf{0110}$. The result of the circuit is $remainder = 0000$, indicating a valid transmission.





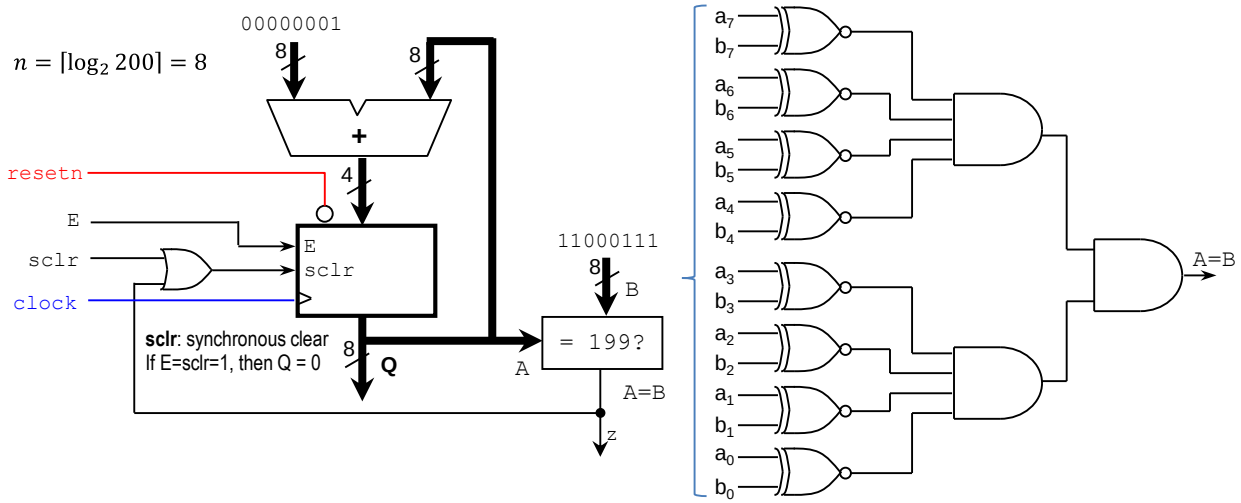


- **Applications**: The circuit above can be implemented for any number of bits n. For example: CRC-32 (Ethernet), CRC-15 (CAN), and CRC-1 (trivial even parity generator with $G(x) = x + 1$).
- This is a serial implementation, i.e., each bit is fed to the circuit at a time. Other implementations process some bits in parallel (for CRC-1, the parity generator in *ECE2700: Notes – Unit 1* processes all the bits at once).
- The design of the generating polynomial $G(x)$ is outside the scope of this class.
- These circuits are better implemented in hardware for high speed and low resource utilization.



MODULO-N COUNTER DESIGN

- A modulo-N counter could be designed by the State Machine method, but this can be very cumbersome if N is a large number. For example, if N=1000, we need 1000 states.
- The figure shows a counter modulo-200 that uses a register, and adder, a comparator, and logic gates. Q: count. z: output signal asserted only when the maximum count (199) is reached. This can be easily described in VHDL using the structural description. You can also use the behavioral description, where the count is increased by 1 every clock cycle and 'z' is asserted when the count reaches 'N-1'.

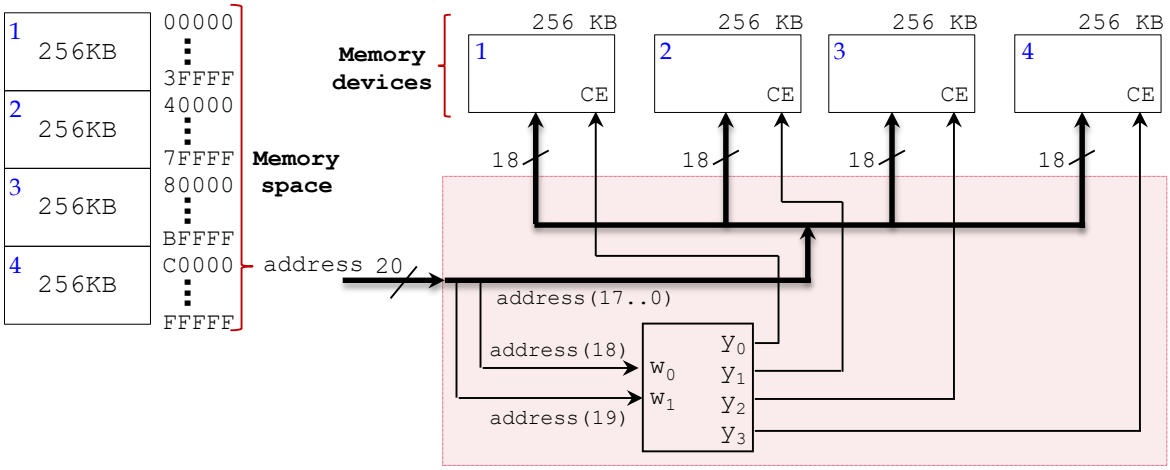


MEMORY DECODING

- Physical implementation of memory is not homogeneous:
  - ✓ Different portions of memory are used for different purposes: RAM, ROM, I/O devices.
  - ✓ A processor can usually address a memory space that is much larger than the memory space covered by an individual memory chip. Even if all the memory was of one type, we still have to implement it using multiple ICs.
  - ✓ For a given valid address, one and only one memory-mapped component must be accessed.
- Address Decoding: Process of generating chip select (CS, CE) signals from the address bus for each device in the system.
- The Address bus line (N+M bits) is split into two sections:
  - ✓ The N most significant bits are used to generate the CS signals for the different devices.
  - ✓ The M least significant bits are passed to the devices as addresses.

EXAMPLE

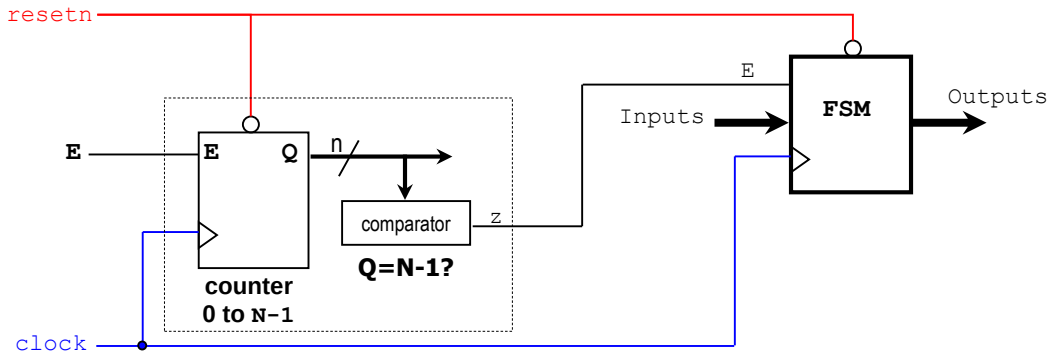
- A 20-bit address line in a μprocessor handles up to $2^{20} = 1\,MB$ of addresses, each address containing one-byte of information. We want to connect four 256KB memory chips to the μprocessor.
- The 2 MSBs of the address line are used to generate the CS signals. The 18 LSBs are passed to the devices as addresses.
- The pink-shaded circuit: i) addresses the memory chips, and ii) enables only one memory chip (via CE: chip enable) when the address falls in the corresponding range. Example: if $address = 0x5FFFF$, → only memory chip 2 is enabled (CE=1). If $address = 0xD0123$, → only memory chip 4 is enabled.



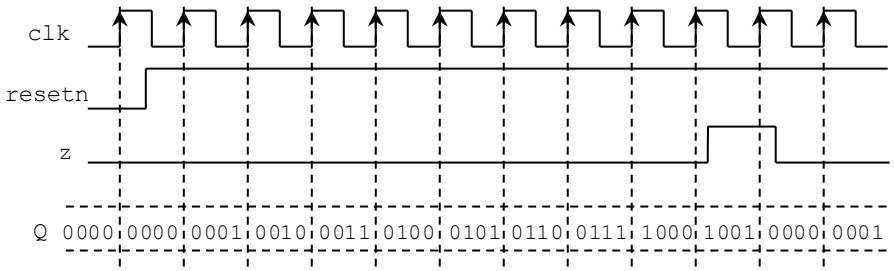
FLIP FLOPS: PRACTICAL ASPECTS

Reducing rate of change of a synchronous circuit:

- Here, we use FSM as an example of a synchronous circuit. But we can apply this technique to any synchronous circuit.
- We usually would like to reduce the rate at which FSM transitions occur. A straightforward option is to reduce the frequency of the input clock. But this is a very complicated problem when a high precision clock is required.
- Alternatively, we can reduce the rate at which FSM transitions occur by including an enable signal in our FSM: this means including an enable to every flip flop in the FSM. For any FSM transition to occur, the enable signal has to be '1'. Then we assert the enable signal only when we need it. The effect is the same as reducing the frequency of the input clock.
  - ✓ The figure below depicts a counter modulo-N (from 0 to N-1) connected to a comparator that generates a pulse (output signal z) of one clock period every time we hit the count 'N-1'. The number of bits the counter is given by $n = \lceil \log_2 N \rceil$. The effect is the same as reducing the frequency of the FSM to f/N, where f is the frequency of the clock.



  - ✓ Example: Timing diagram of a modulo-10 counter. Notice that 'z' is only asserted when the count reaches "1001". This 'z' signal controls the enable of a FSM, so that the FSM transitions only occur every 10 clock cycles, thereby having the same effect as reducing the frequency by 10.



- We can apply the same technique not only to FSMs, but also to any sequential circuit. This way, we can reduce the rate of any sequential circuit (e.g. another counter) by including an enable signal of every flip flop in the circuit.

Flip flop timing parameters:

- Propagation Delay.
- Setup Time: Interval of time before the clock edge where the data must be held stable.
- Hold Time: Interval of time after the clock where the data must be held stable.
- If setup time or hold time are violated, the output may become unpredictable, or even worse it might enter into metastability.