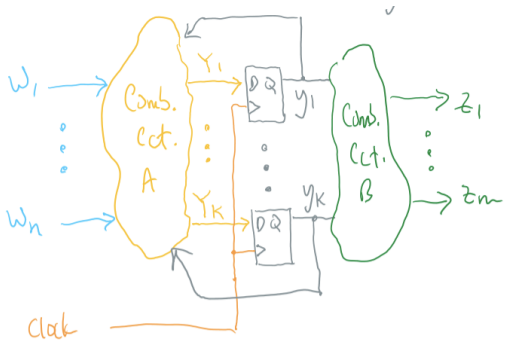
241 Notes (post-midterm)

Finite State Machines (FSMs)

> 💡 An FSM is a sequential circuit that has inputs w, outputs z, and flip-flops y

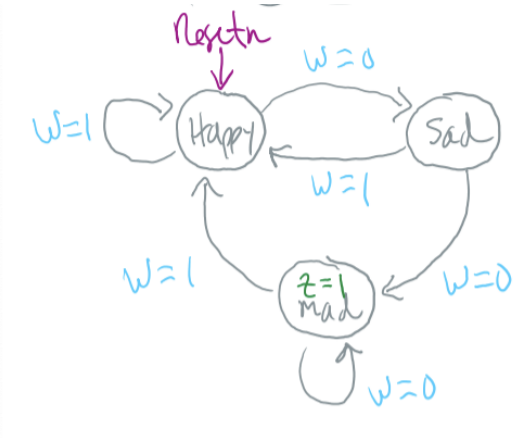


The FSM has a state at any given clock cycle, which is represented by $y_1, \ldots, y_k$. The next state $Y_1, \ldots, Y_K$ when a clock cycle occurs depends on the values of inputs w, and the present state $y_1, \ldots, y_k$

Example: 3 States of Copper — happy, sad, and mad

Changes in state depend on w which represents Copper getting a treat.

- On a clock edge, if $w = 1$, copper is happy
  - Happy
- If Happy and $w = 0$, then
  - Sad
- If Sad and $w = 0$, then
  - Mad
- If Mad
  - Output $z = 1$ (barking)



Present State	Next State		Output
	w=0	w=1	
Happy	Sad	Happy	0
Sad	Mad	Happy	0
Mad	Mad	Happy	1

In this example, we have used two flip flops (the minimum possible).

General Procedure to Build an FSM

1. Draw the state diagram
2. State table
3. State assignment
4. State-assigned table
5. Synthesize the circuit

We don't always need the state-assigned table, as the next state expressions can always be synthesized by inspection by looking at the arrows in the state diagram.

Using one FF/state is known as a **one-hot coding**

FSMs in Verilog

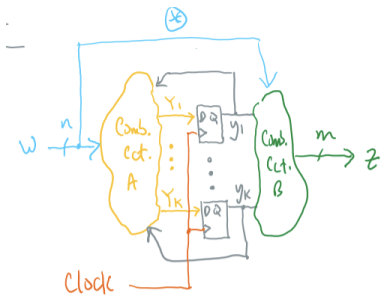
Always 3 sections of code (state table, FFs, output)

```
module FSM(w, Clock, Resetn, z);
  input w, Clock, Resetn;
  output z;
  reg [2:1] y,Y;
  parameter A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;

  // State table
  always @ (*)
```

```
      case (y)
        A: if (w) Y = B; else Y = A;
        B: if (w) Y = B; else Y = C;
        C: if (w) Y = D; else Y = A;
        D: if (w) Y = B; else Y = C;
      endcase

    // State FFs
    always @ (posedge Clock)
      if (!Resetn) y <= A;
      else y <= Y;

    // FSM outputs
    assign z = (y == D);
endmodule
```

> 💡 **Regarding Case Statements**
>
> For some valuations, if we don't specify a default value, then the compiler may create unwanted latches

Mealy FSMs



If the connection denoted by ⭐ exists, then the outputs depend on both the present state $y_1, \ldots, y_k$ and the inputs. This is called a Mealy FSM. When the connection doesn't exist, it is called a Moore FSM.

Example FSM Design (Arbiter)

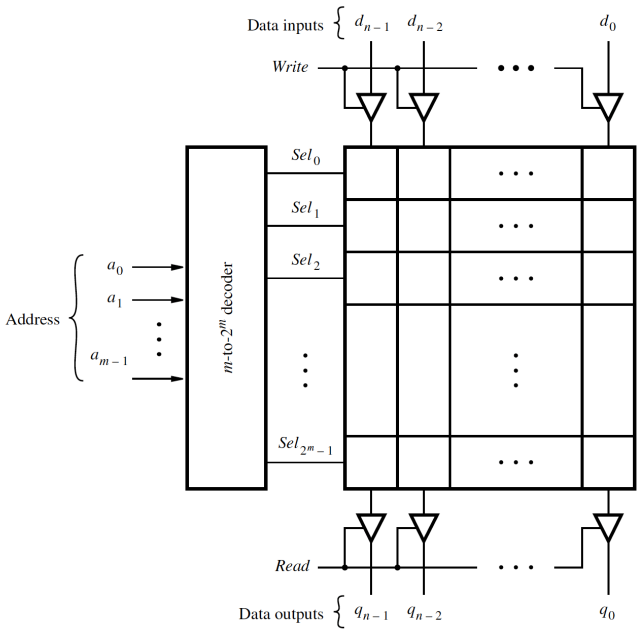
❓ Design an FSM that controls access to a shared resource, between three devices. Each device requests access to the resource by asserting a request signal r_1, r_2, r_3. The arbiter chooses which device gets the resource in each clock cycle by setting a grant output g_1, g_2, g_3. There is a priority scheme $r_1 > r_2 > r_3$.

Memory Technology

Every computer system needs memory to hold data and programs. Also, other logic circuits might need to store some data.

In every memory there are two main parts:

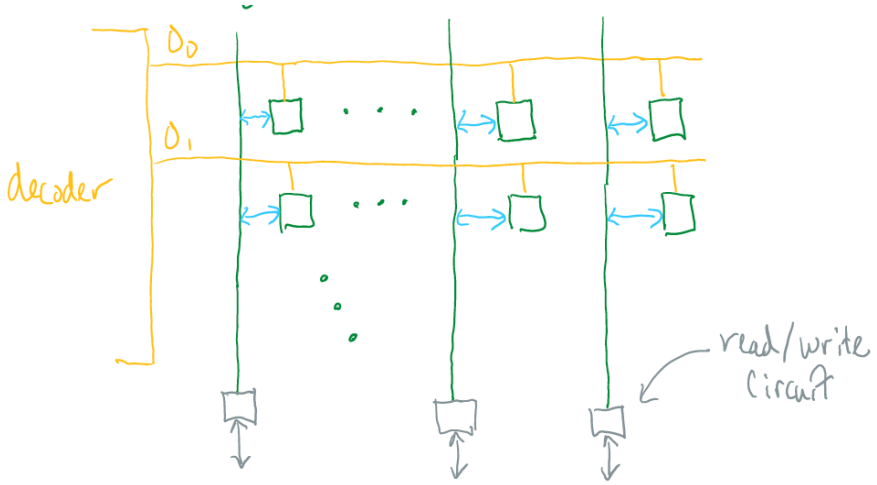
1. Decoder
2. Array of storage cells



- The number of address inputs depends on the depth (number of words) of the memory
- The number of data input/output bits depends on the width (number of bits per word) of the memory

Storage Cells

- Each storage cell in a memory holds one bit (0, or 1)



There are two main memory technologies:

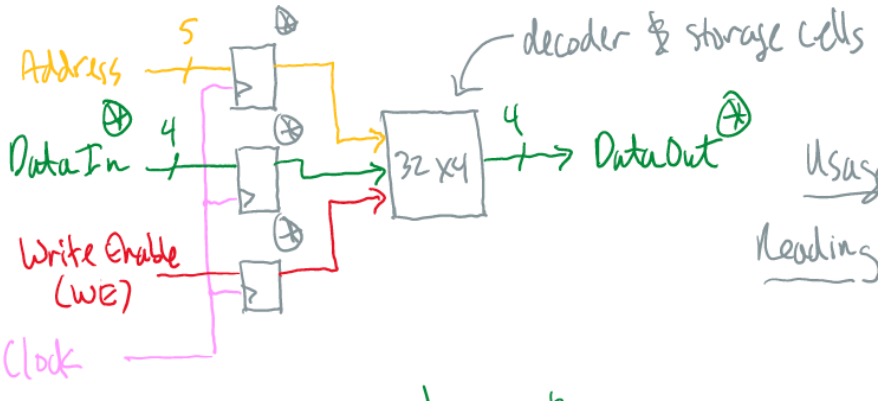
1. Static
    a. Easy to use, but capacity is small due to “large” storage cell.
    b. Example: FPGA on chip memory in the DE1-SoC board is SRAM (static random access memory)
2. Dynamic
    a. Hard to use but capacity is large (due to “small” storage cell)
    b. Example: a computer using >1GB is always using DRAM

FPGA On Chip Memory

The FPGA contains programmable logic blocks (for logic functions and FFs) but it also contains SRAM memory. Each block of memory has 10240 storage cells and 446 such blocks. These blocks can be figured to make memories of various aspect ratios (depth by width)

E.g. 19200 X 3 (VGA)

Architecture Example for (32 X 4)



🟩⭐ Data in/out are separate ports

⬛⭐ These registers are always present

Reading:

1. Apply address of the desired word, and set $we = 0$
2. wait for posedge clock
3. The desired word appears on DataOut

Writing:

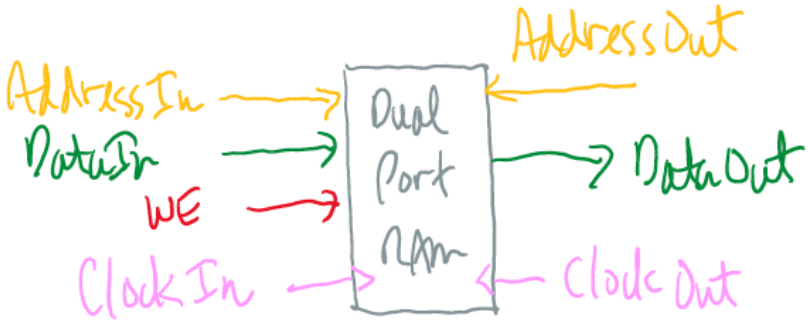
1. Apply address, apply DataIn, set $we = 1$
2. Wait for posedge clock
3. Write is complete

On Chip Memory Variants

RAM — random access to memory

ROM — read-only memory

Dual-Port RAM



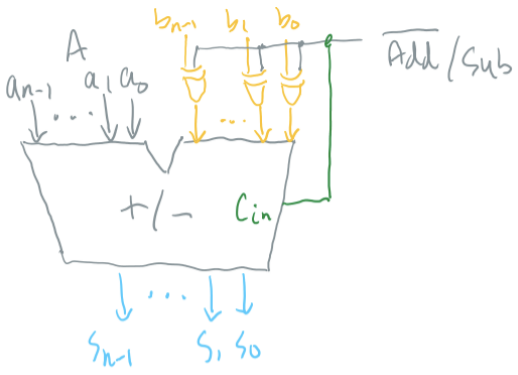
Signed Numbers (2's complement)

In general, $-k = 2^n - k$

We can use 2's complement to partition the numbers with n bits into 1/2 positive and 1/2 negative

💡 Shortcut: we can use $((2^n - 1) - k) + 1$

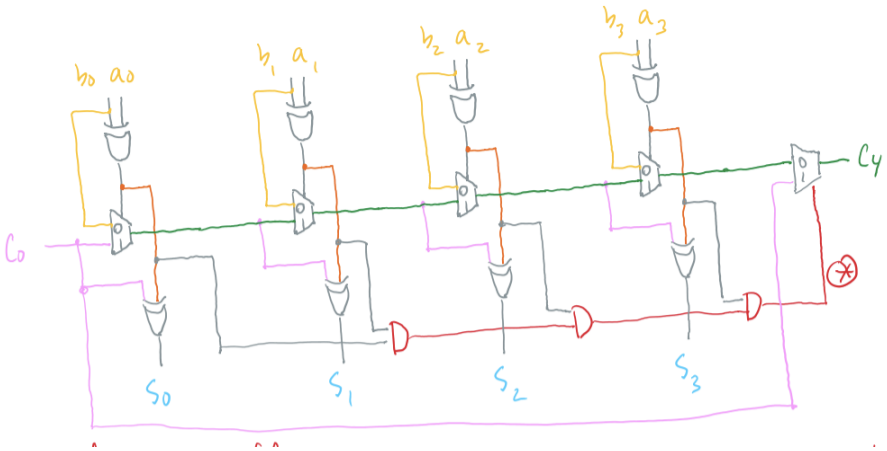
We can use this shortcut to make an adder/subtractor A+/-B



The Shortest Cut

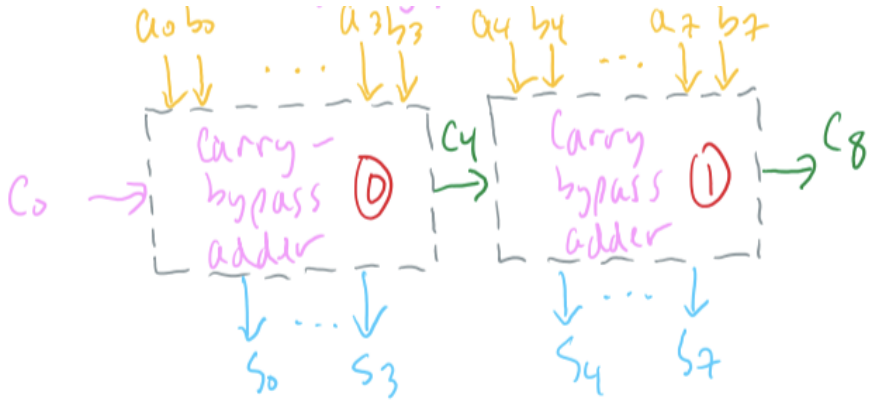
Keep all digits of k up to and including the first 1, then complement the rest

Carry-Bypass Adder

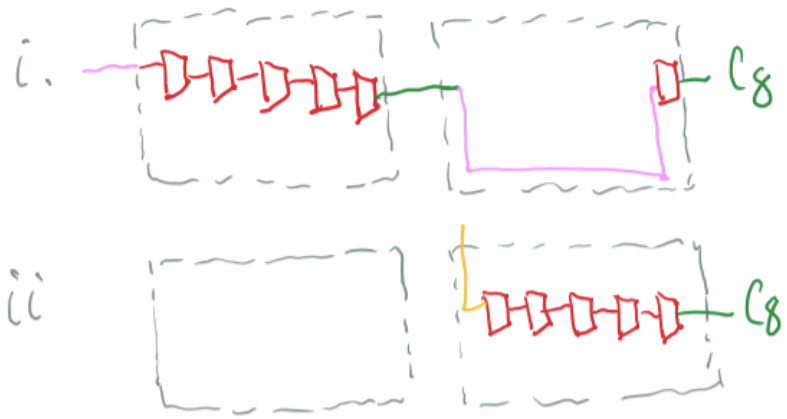


🟥⭐ The reg signal is 1 iff all four pairs of a_i, b_i are different. This is how we use the carry-bypass

For larger adders, we can use multiple instances of this



- What is the worst delay to get to c_8?
  - Looking at adder 1, there are two cases:
    - If all pair of $a_4b_4, \ldots, a_7b_7$ are different, then $c_8 = c_4$ (see the red note above)
    - Any pair of $a_4b_4, \ldots, a_7b_7$ are the same. In that case, c_8 is independent of c_4



Verilog Code

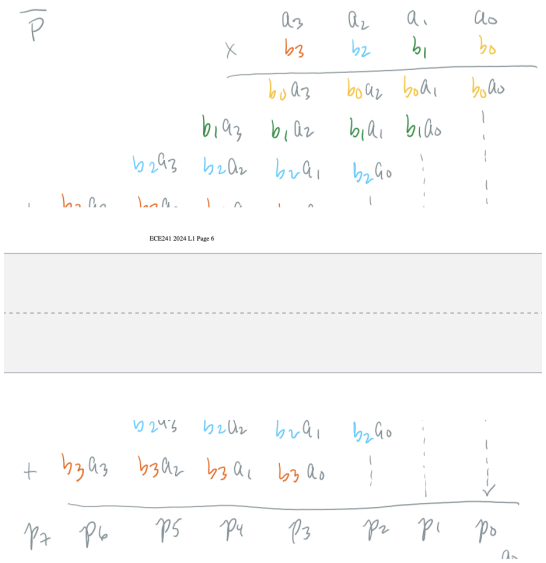
```
module addn (Cin, A, B, Cout, S);
  parameter n = 4;
  input Cin;
  input [n-1:0] A, B;
  output reg Cout;
  output reg [n-1:0] S;
  reg [n:0] C;
  always @ (*) begin
    C[0] = Cin;
    for (integer k = 0; k < n; k = k + 1) begin
      S[k] = A[k]^B[k]^C[k];
      C[k+1] = (A[k] == B[k]) ? B[k] : C[k];
```

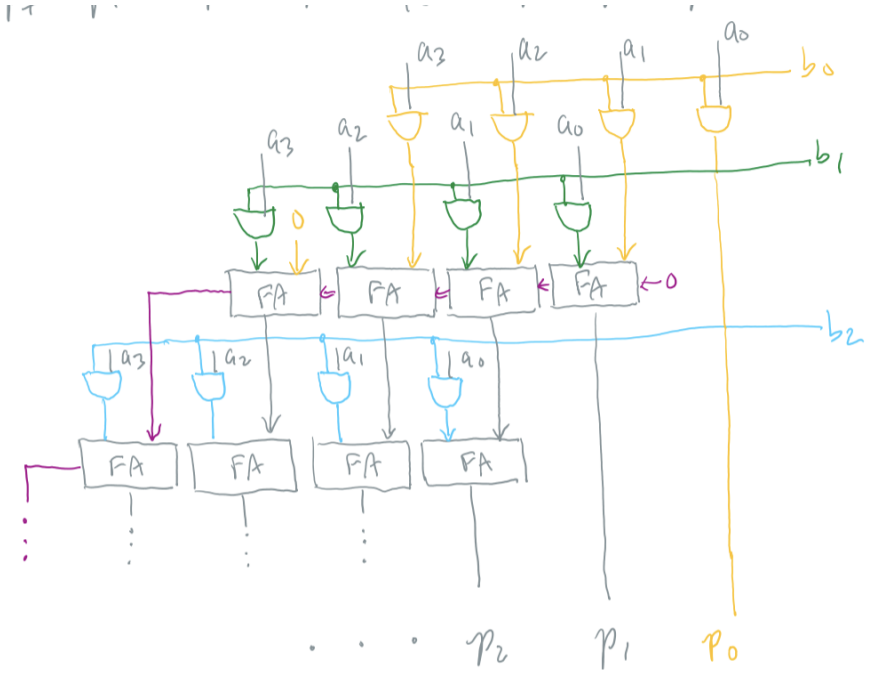
```
    end
    Cout = C[k];
  end
endmodule
```

- The statements in the for loop are simply repeated for each value of k. This only saves typing.

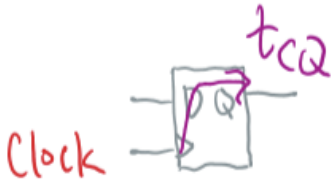
```
module addn (...);
  // port declarations
  parameter n = 4;
  always @ (*)
    {Cout, s} = A + B + Cin;
endmodule
```

Multiplication (positives only)





Timing Analysis



t_{su} — FF setup time: This is a window of time before the active clock edge during which the D input of the flip flop should not change

t_h — FF hold time: This is a window of time after the active clock edge for which the input D should remain unchanged

t_{cQ} — FF clock to output delay: This is a propogation delay for a new value of Q to appear on after a clock edge

$$T_{min} = \frac{1}{F_{max}}$$

- In most examples we assume no clock skew, meaning that the clock signal arrives at all FFs at exactly the same time.

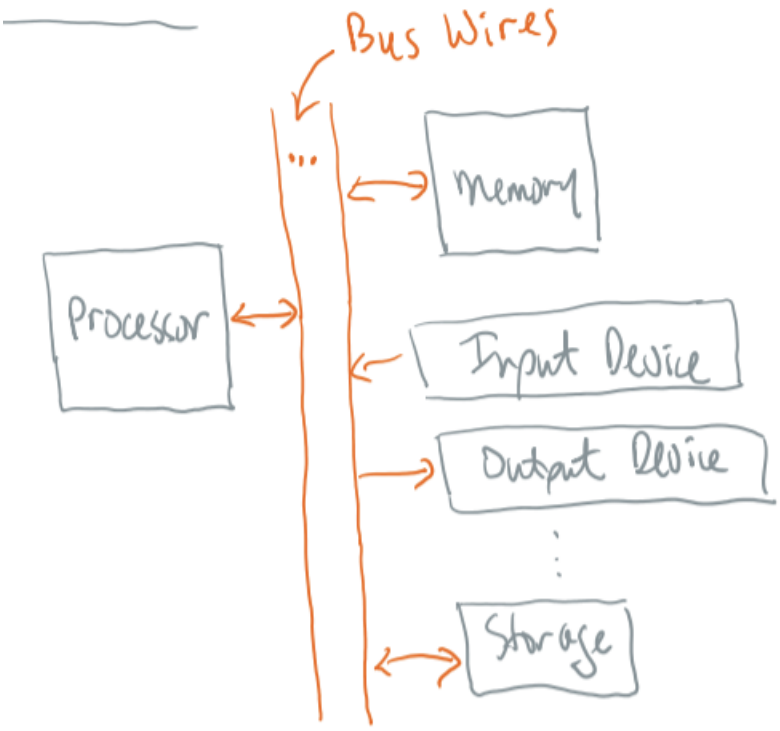
Moral of the story: Ensure that all FFs are connected directly to the same clock signal. Otherwise, we will be a bad engineer

What happens if t_{su} and/or t_h are violated?

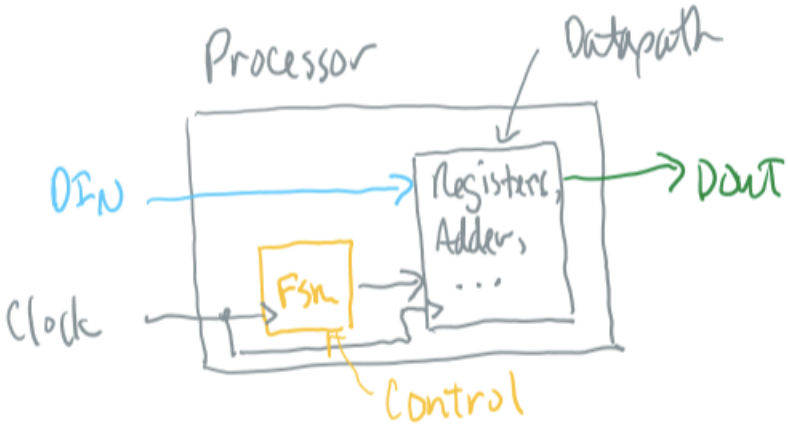
For FFs, the ball on top of the hill represents metastability, where Q is neither 0 nor 1, after a clock edge that violates either timing parameter. After some random time period (but within that clock cycle), Q will randomly settle to either 0 or 1.

If Q_1 becomes metastable, it will randomly settle to 0 or 1 before it gets stored into Q_2. Thus, Q_2 is never metastable. This is called a synchronizer.

Computer Hardware

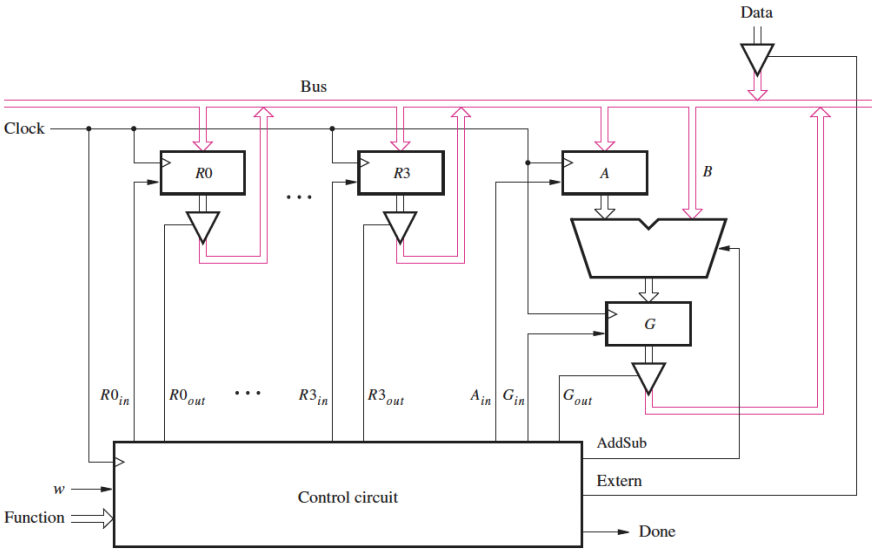


A processor consists of two parts: the datapath and the control.



Processor Design

Consider a set of n-bit registers $R_0, \ldots, R_7$. We wish to intialize a registers contents, transfet reg→reg, add the contents of registers, and subtract.



We need to provide an instruction that tells the processor (FSM) what operation should be executed at any given time. We have to choose an encoding and decide what the instructions look like.

```
mv Rx,Ry // copy Rx<-[Ry]
mvi Rx,#D // initialize Rx<-D
add Rx,Ry // Rx<-[Rx]+[Ry]
sub Rx,Ry // Rx<-[Rx]-[Ry]
```

`mv` , `mvi` , `add` , `sub` are all called mnemonics

- We assume after reading the `mvi` instruction from DIN into IR, then the processor can next read #0 from DIN and use `set` to put it on the "bus."

- Some terminology:
  - `mv` , `mvi` , `add` , `sub` are called the assembly language instructions.
  - Every processor has a unique assembly language.
  - The **encoding** of instructions are called the opcodes
  - An Assembler program produces the opcodes and data which is called machine code
- Execution of instructions:
  - Each instruction appears on DIN, and is stored into IR. Call this clock cycle T0. Then, in the following clock cycle, the FSM will set the control signal as needed in each clock cycle to complete the instruction.

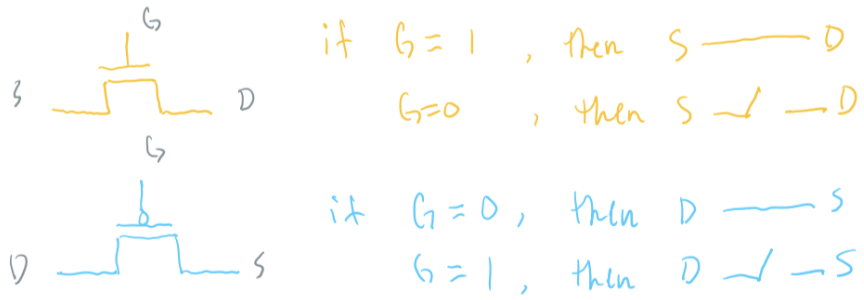
```
module proc (DIN, Resetn, Clock, Run, Done);
    input [7:0] DIN;
    input Resetn, Clock, Run;
    output reg Done;
    reg [2:1] T_D, t_Q;
    parameter TO = 2'b00, T1 = 2'b01,...,;
    parameter mv = 2'b00, mvi = 2'b01,...;
    ... other signals (yada-yada)
    // instantiate the components of the Dataport
    // FSM state table
    always @ (*)
        case (t_Q)
            T0: if (!Run) T_D = T0;
                    else T_D = T1;
            T1: if (Done) T_D = T0;
                    else T_D = T2;
            T2: T_D = T3;
            T3: T_D = T0;
        endcase
    // FSM outputs
    always @ (*) begin
        LR0 = 0; LR1 = 0;... LG = 0; Done = 0; LIR = 0;
        Sel = 3'bxx;

        case (y_Q)
            T0: LIR = 1;
            T1: case(II)
                        mv: begin
                            sel = YYY;
                            LR = "XXX"; (*)
                            Done = 1;
                        end
```

```
                    endcase
            endcase
        end
```

`(*)` = we would actually use a 3-8 decoder to produce LR0,...,LR7 from XXX.

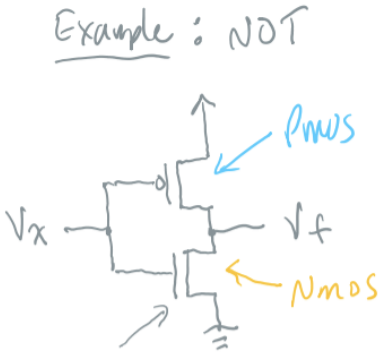
- This processor needs to be enhanced to have an introduce to a memory. Instructions and data would come from that memory. Instructions and data would come from that memory
- We would add instructions to the processor for reading or writing memory.
- Finally, we need to be able to know where in the memory the instructions are stored. A special register which is a loadable counter, called a program counter (pc) is used for this.
  - Normally, pc is used as an address (transferred through ADDR to memory) from which to read the next instruction from memory.
  - The FSM always increments pc by 1 so it has the address of the next instruction.
  - Finally, pc can be loaded with any number (address) to implement loops.

CMOS Technology



CMOS technology — complementary MOS

- In CMOS, every circuit has an equal number NMOS and PMOS transistors.



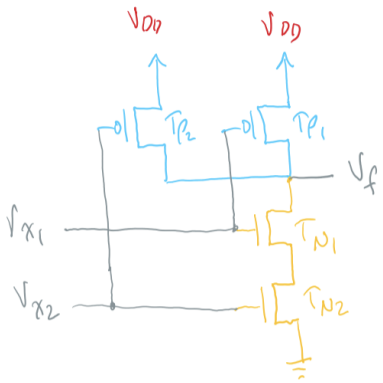
- Without any voltage applied, these transistors are OFF, because of the charge under the gate being the opposite of the source/drain.
- In CMOS, there is never a path from V_{DD} to Gnd in the steady state. Thus, current flow is much smaller.
-

CMOS NAND Gate

$f = \overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$ (P-part)

for N-part

$\bar{f} = \overline{\bar{x}_1 \bar{x}_2} = x_1 x_2$

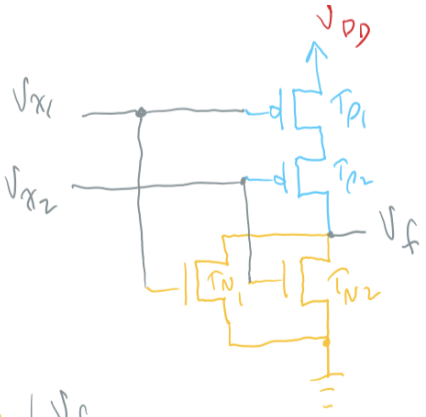


V_{x1}	V_{x2}	T_{P1}	T_{P2}	T_{N1}	T_{N2}	V_f
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	1
1	0	OFF	ON	ON	OFF	1
1	1	OFF	OFF	ON	ON	0

CMOS NOR Gate

$f = \overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

$\bar{f} = \overline{\bar{x}_1 + \bar{x}_2} = x_1 + x_2$



V_{x1}	V_{x2}	T_{P1}	T_{P2}	T_{N1}	T_{N2}	V_f
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	0
1	0	OFF	ON	ON	OFF	0
1	1	OFF	OFF	ON	ON	0