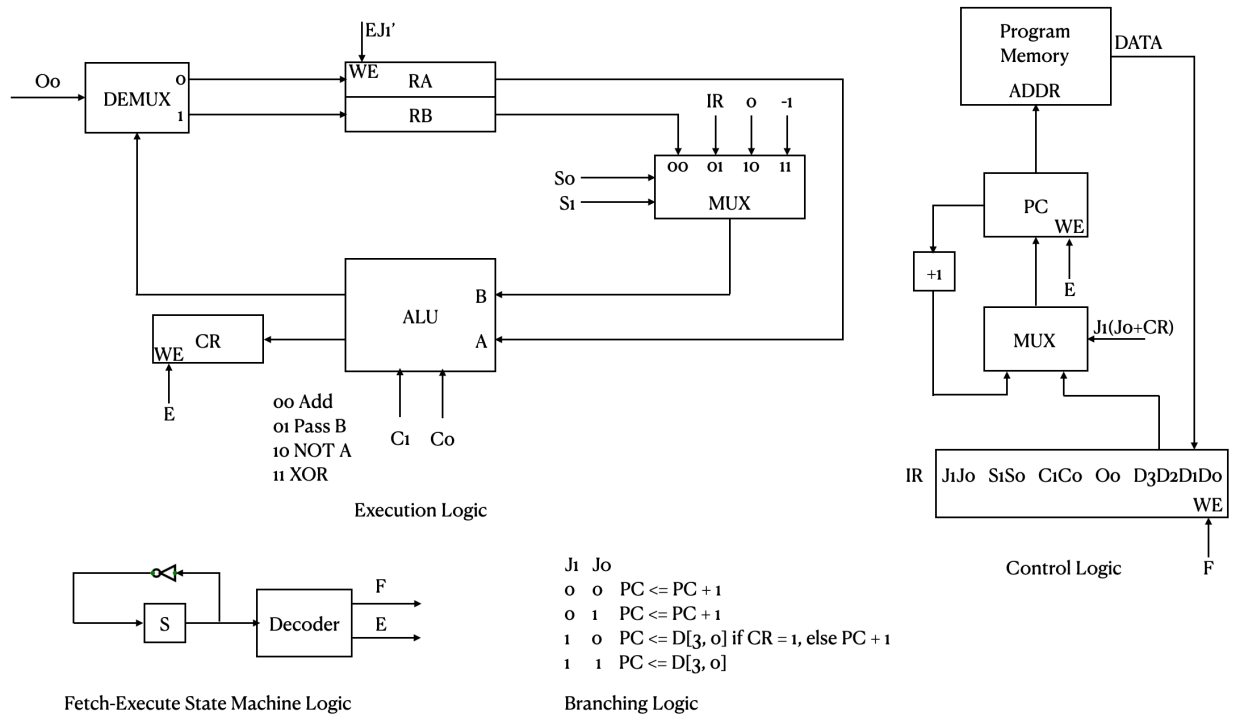


Programmable Circuits (II)

Extending The Count



- To make the machine be able to adjust control flow based on data requires the [ability to modify the PC](#).
 - So, we need a new MUX for the inputs to the PC, and the sources are $D_3 \sim D_0$ in IR or $PC + 1$.
 - Also needs some condition to test to know whether the PC needs to load a new value. Here, we use $J_1(J_0 + CR)$ as an example.
- We need a **Condition Register (CR)** to hold the condition of the ALU.
 - CR is also named status register. It holds a set of status flags for ALU. Common flags include Zero flag (result of ALU was zero), Carry flag (an arithmetic carry or borrow has been generated out of the most significant ALU bit position), Sign flag (result of ALU was negative), and Overflow flag (the result is too large to fit in the register)
 - In this example, we assume CR only has Zero flag. CR is 1 when the result of ALU was zero.
- Need a [branching logic](#) for unconditional and conditional jumps to implement a for loop.

- Since the control logic now has the ability to modify the PC's value based on some condition, we need an additional clock cycle to load the new PC value. So, we assign a **Writing Enable (WE)** bit to PC.
- Since $D_3 \sim D_0$ in IR can be written in the PC based on some condition, loading an instruction into IR should be in a cycle different from the one the PC's value can be updated. So, we assign a WE bit to IR and let it be able to fetch a new instruction in the F cycles. PC and other registers (RA, RB, and CR) can be written in E cycles when the count is executing an instruction.
- We also introduce a Fetch-Execute State Machine Logic to implement the F/E cycles.
- We assume that RA and RB each is 4 bits.

A Loop for the Count

- This circuit gives us a set of operations to use. If we use mnemonics to represent those operations, they are
 - **LOAD X Y**: transfer Y's value to X
 - **ADD X Y Z**: calculate the sum of the values of X and Y, and store the result at Z.
 - **NOT X Y**: negate X's value, and store the result at Y
 - **XOR X Y Z**: xor the values of X and Y, and store the result at Z
 - **BRZ X**: conditional branch, branch to line X if the ALU result is zero
 - **BRA X**: unconditional branch, branch to line X
- We can then leverage the mnemonics to write an assembly program to get the sum of 5, 4, 3, 2, 1, and store the sum in RA.
 - Initialize RB = 0 and RA = 5. Have a loop, in each iteration, the result of the sum of the values stored at RA and RB is assigned to RB, and then decrement RA's value by 1. As long as the value in RA is not zero, the loop continues. Put the result to RA after the loop terminates.
 - If using pseudocode, we can have something like

```

0    RB <= 0
1    RA <= 5
2    while (RA != 0) {
3        RB <= RA + RB
4        RA <= RA - 1
5    }
6    RA <= RB

```