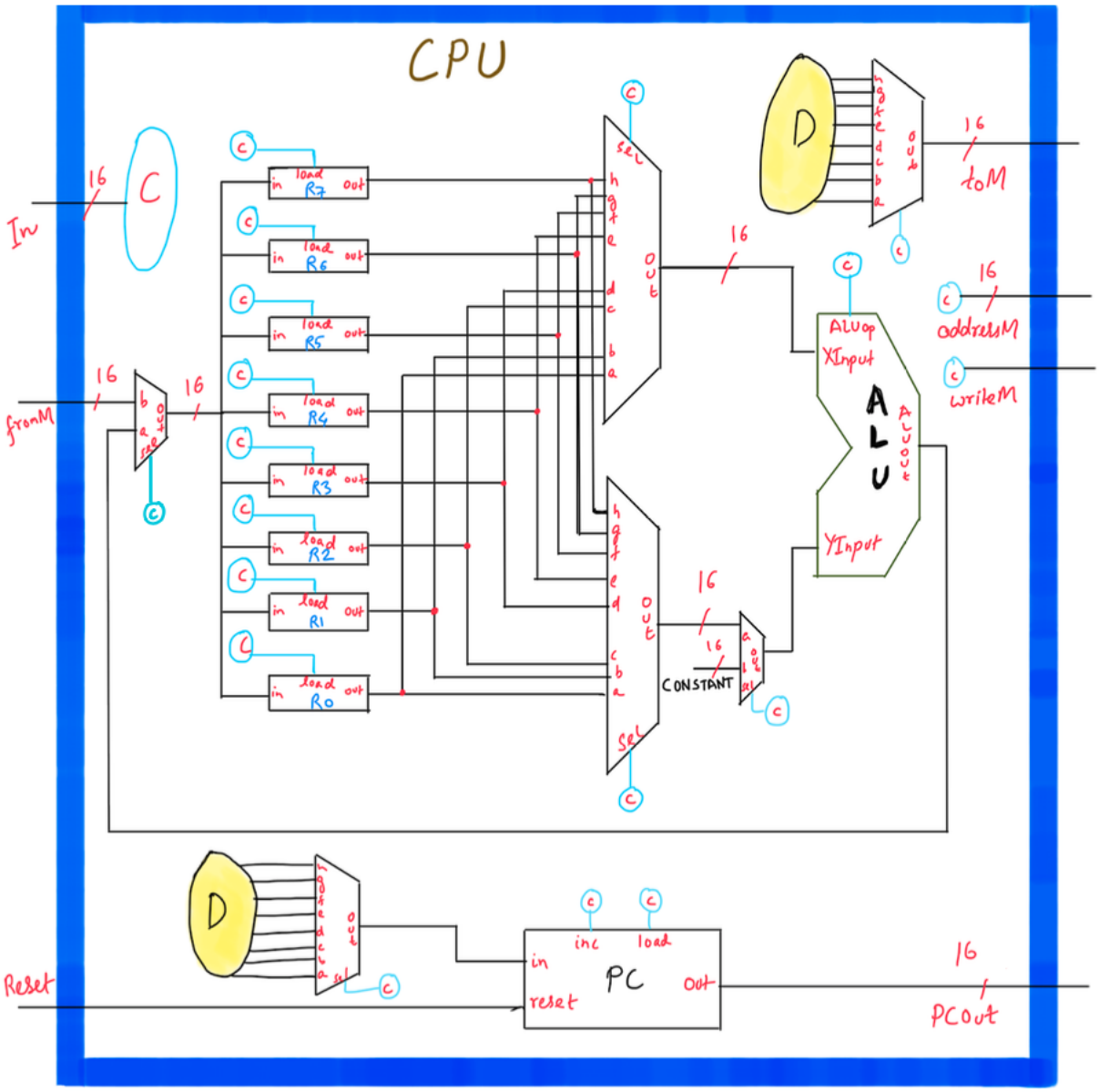
TIPS: BITBOT CPU LOGIC DESIGN



D = Datapath Source from Registers

c = Control signal generated using In and other logic.

NOTATION FOR SYMBOLS

Instruction Set

Arithmetic	EXAMPLE	In[15]	In[14]	In[13]	In[12]	In[11]	In[10]	In[9]	In[8]	In[7]	In[6]	In[5]	In[4]	In[3]	In[2]	In[1]	In[0]
		OPCODE		OPTYPE		DEST REGISTER			SRC1 REGISTER			SRC2 REGISTER					
ADD	ADD R0, R1, R2 (i.e. R0 = R1 + R2)	0	0	0	0	Any Reg: 000 to 111			Any Reg: 000 to 111			Any Reg: 000-111					
ADDI	ADDI R0, R1, 8 (i.e. R0 = R1 + 8)			0	1							Six Bit Immediate Value (0-63)					
SUB	SUB R0, R1, R2 (i.e. R0 =R1 - R2)			1	0							Any Reg: 000-111					
SUBI	SUBI R0, R1, 8 (i.e. R0 = R1 - 8)			1	1							Six Bit Immediate Value (0-63)					
Logical	EXAMPLE	In[15]	In[14]	In[13]	In[12]	In[11]	In[10]	In[9]	In[8]	In[7]	In[6]	In[5]	In[4]	In[3]	In[2]	In[1]	In[0]
		OPCODE		OPTYPE		DEST REGISTER			SRC1 REGISTER			SRC2 REGISTER			UNUSED		
NAND	NAND R0, R1, R2 (i.e. R0 = R1 NAND R2)	0	1	0		Any Reg: 000 to 111			Any Reg: 000 to 111			Any Reg: 000 to 111					
NOR	NOR R0, R1, R2 (i.e. R0 = R1 NAND R2)			1													
Memory	EXAMPLE	In[15]	In[14]	In[13]	In[12]	In[11]	In[10]	In[9]	In[8]	In[7]	In[6]	In[5]	In[4]	In[3]	In[2]	In[1]	In[0]
		OPCODE		OPTYPE		LEFT SIDE			RIGHT SIDE								
READ	READ R0, R1 (i.e. R0 = MEM[R1])	1	0	0	0	DEST REGISTER			SRC PTR REGISTER								
WRITE	WRITE R0, R1 (i.e. MEM[R0] = R1)			1	0	DEST PTR REGISTER			SRC REGISTER								
Branch	EXAMPLE	In[15]	In[14]	In[13]	In[12]	In[11]	In[10]	In[9]	In[8]	In[7]	In[6]	In[5]	In[4]	In[3]	In[2]	In[1]	In[0]
		OPCODE		OPTYPE		TARGET ADDRESS			SRC REGISTER								
JMP	JMP R0 (i.e. PCOut = R0)	1	0	1	1	Any Reg: 000 to 111											
BEQ	BEQ R0, R1 (i.e. PCOut = R0 if R1==0)			0	1				Any Reg: 000 to 111								
INPUT/OUTPUT	EXAMPLE	In[15]	In[14]	In[13]	In[12]	In[11]	In[10]	In[9]	In[8]	In[7]	In[6]	In[5]	In[4]	In[3]	In[2]	In[1]	In[0]
		OPCODE		OPTYPE		TARGET ADDRESS			SRC REGISTER								
INP	INP R0 (i.e. R0 = MEM[KEYBOARD])	1	1	1	0	Any Reg: 000 to 111											
OUT	OUT R0, R1 (i.e. SCREEN[R0] = R1)			0	0				Any Reg: 000 to 111								

Resources

- Compute Element: ALU
- Memory Element: Registers R0..R7
- Instruction Address Generator: Program Counter

External Interactions

- Read/Write/Input/Output interactions with the Data Memory
- Next Instruction Address to the Instruction ROM
- External Reset Input

ALU Logic

- Examine the ALU instructions: ADD/SUB, NAND/NOR.
- Look at the format of these 4 instructions:
    - They all have TWO source operands and ONE destination
        - One source operand is always a register
        - The other source operand can be a register or an immediate constant embedded in the instruction
    - The destination is always a register
- Connecting the ALU
    - Data Inputs
        - XInput can come from one of registers R0..R7
        - YInput can come from one of registers R0..R7 **or** from Immediate Constant
    - Control Inputs

        - ALU controls are simply based on opcode and optype

Register Logic

- Examine the instructions that deal with registers – Highlight the ones where registers are destinations.
    - **ALU instructions**: 1-2 source registers and **1 destination register**
    - **Memory Instructions**
        - **READ**: 1 source register that contains address of memory location and **1 destination register**
        - **WRITE**: 1 source register that contains data to be written to memory, 1 register that contains address of memory location
    - **I/O instructions**
        - **INP**: **1 destination register** where the keyboard value will be 'inputted'
        - **OUT**: 1 source register that contains data to be 'outputted', 1 register that contains screen memory address
    - **Branch instructions**
        - **JMP**: 1 source register that contains the jump location in the ROM
        - **BEQ**: 1 source register that contains the jump location in the ROM and 1 condition check register whose content must be ZERO for the branch to occur
- **Connecting the Registers**
    - **Data Input**: Register input can either come from
        - ALU: ALU operation
        - Memory: READ or INP
    - **Control Input**: Register Load
        - Highlight the conditions where registers are updated (ALU, READ, or INP)
        - Generate a register load signal based on these conditions and propagate it to registers

Talking to the Data Memory

- **Examine the instructions that deal with the Data Memory**
    - **READ**: 1 source register that contains memory address and 1 destination register
    - **WRITE**: 1 source register that contains data to be written to memory, 1 register that contains memory address
    - **INP**: 1 destination register where the keyboard value will be 'read'
    - **OUT**: 1 source register that contains data to be 'written', 1 register that contains screen memory address

- **Now think about the interactions with the data memory**
    - **Data**: toM and fromM are the data buses into and out of the data memory.

- **Address**: addressM is the address sent to the data memory to identify the location for Read/Write/Input/Output
- **Load**: The writeM signal enables Write/Output to the data memory.

- **Let's talk about each of these interactions**
  - **Data (**aka **toM and fromM)**:
    - **fromM**: Data input to the CPU that forms one of the legs of the input MUX that feeds input to registers R0..R7
    - **toM**: Data Output to the CPU. Comes straight from the MUX that collates outputs from registers R0..R7
  - **Address (**aka **addressM)**
    - **addressM**: The reasons you want to send address to the Memory are to enable the following:
      - READ from RAM32K or
      - INP from KBRD or
      - Write to RAM32K or
      - OUT to SCREEN
    - **Example**: Let's focus on RAM32K as example for address generation:
      - The interaction with RAM32K is through the READ and WRITE commands. Let's take READ as an example: here the instruction format shows that the READ address can come from any one of the 8 registers (R0..R7). The source register address is given in In[6..8]. Similarly, for WRITE, the instruction format shows that the WRITE address can come from any one of the 8 registers (R0..R7).
      - How do we identify which register the instruction is referring to? Well, in case of a READ instruction, it is bits In[6..8]. In case of a WRITE instruction, it is bits In[9..11]. This would be a good time to look at the Instruction Code Table.
      - So in order to generate addressM (assuming for the moment we are only talking about RAM32K), you can get the contents of register id'd by In[6..8] in case the instruction was READ and (otherwise) the contents of register id'd by In[9..11] in case the instruction was WRITE. A MUX is really good to accomplish this objective.
    - Now, you can expand the logic for addressM generation by also thinking about the cases for IN (Keyboard) and OUT(Screen) instructions.
  - **Load to Memory (**aka **writeM)**
    - **writeM**: We need to ask the question – when will we want to write to the memory? There are only two situations: (a) Data Write to the RAM32K or (b) OUT to SCREEN in MMIO locations. Let's look at each:

        - **Write to Memory**: writeM must be active if OPcode==Memory (i.e. In[15..14]==10) AND Optype==Write (i.e. In[13..12]==10)
        - **OUT to Screen**: writeM must be active if Opcode==I/O (i.e. In[15..14]==11) AND Optype==OUT (i.e. In[13..12]==00)

Talking to the Instruction ROM

- CPU sends the address of the next instruction to be fetched from the Instruction ROM (aka ROM32K).
- This address is generated by the CPU and loaded into the Program Counter (aka PC).
- By default, the PC increments every clock cycle.
- The exception to this default can be caused by two possible instructions
    - **Unconditional Jump (aka JMP Rx):** In this case, we load the PC with the content of register Rx identified in the JMP instruction.
    - **Conditional Jump (aka BEQ Rx Ry):** In this case, we load the PC with the content of register Rx if content of register Ry==0
- So the logic for generating next instruction address comes down to the following two components:
    - **PC Load:** Logical OR of the two conditions:
        - Unconditional Jump: Simply recognize if instruction is JMP (i.e. In[15..12]==1011)
        - Conditional Jump: Recognize it is a BEQ instruction and the content of condition check register ==0 (i.e. In[15..12]==1001 && [In[8..6]]==0)
    - **PC Input:** This comes from the output of the selected register identified by In[9..11]