Group 18: 312 Project — Milestone 1

Lucian Chauvin
133003371

Joshua Lass
531009387

Bjorn Quarfordt
230003985

1 Transformation Tables

Instruction	Fetch	Decode	Execute	Memory	Write Back	PC Update

`rrmovq rA, rB`
`irmovq V, rB`
	`icode:ifun← M_1[PC]`
	`rA:rB ← M_1[PC+1]`
	`valP← PC+2`
		`valA← R[rA]`
		`valB← R[rB]`
					`R[rB]← valA`	`PC← valP`

`rmmovq rA, D(rB)`	`icode:ifun ← M_1[PC]`
	`rA:rB ← M_1[PC+1]`
	`valC ← M_8[PC+2]`
	`valP ← PC+10`
		`valA← R[rA]`
		`valB← R[rB]`
			`valE←valB+valC`	`M_8[valE]← valA`		`PC ← valP`

`mrmovq D(rB), rA`	`icode:ifun ← M_1[PC]`
	`rA:rB ← M_1[PC+1]`
	`valC ← M_8[PC+2]`
	`valP ← PC+10`
		`valB← R[rB]`	`valE←valB+valC`	`valM ← M_8[valE]`	`R[rA]← valM`	`PC ← valP`

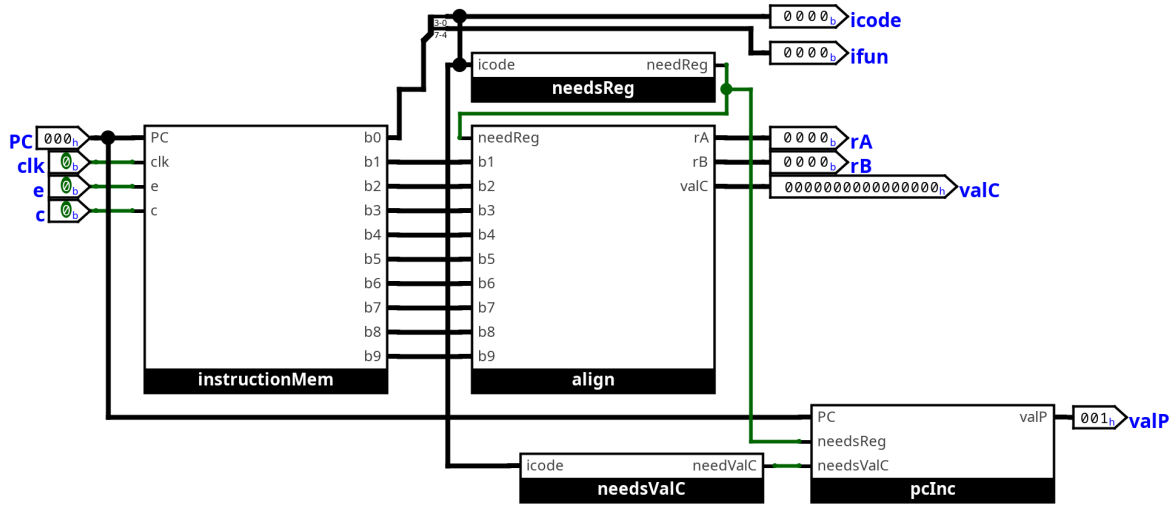
`OPq rA, rB`	`icode:ifun← M_1[PC]`
	`rA:rB ← M_1[PC+1]`
	`valP← PC+2`
		`valA← R[rA]`
		`valB← R[rB]`
			`valE← valB OP valA`		`R[rB]← valE`	`PC← valP`

`jXX Dest`
`cmovXX rA, rB`
`call Dest`
`ret`
`pushq rA`
`pop rA`						

2 Fetch Implementation

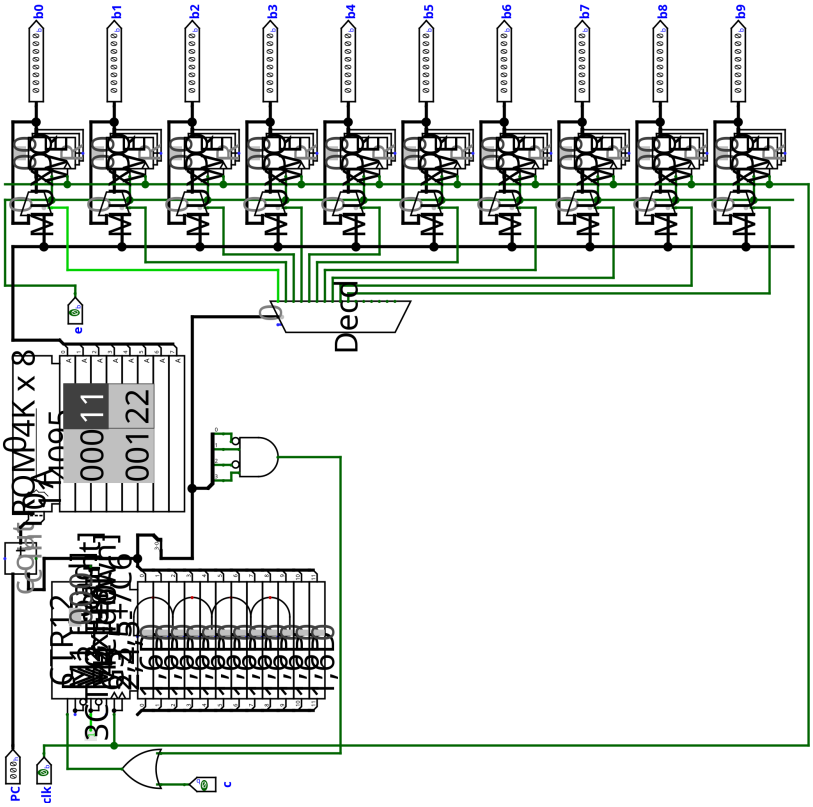
Our design is basically a one-to-one implementation of what is shown on the slides. Our instruction memory module stores the program in ROM and reads 10 bytes in from the current `PC`. We then pass these 10 bytes to our align module which — based on if we need registers or not — sets out `rA`, `rB`, and `valC` correctly. We determine whether we need registers based on the value of `icode`. Then based on if we read in registers or if we read in a `valC` we increment our `PC` using our PC increment module.

2.1 Fetch



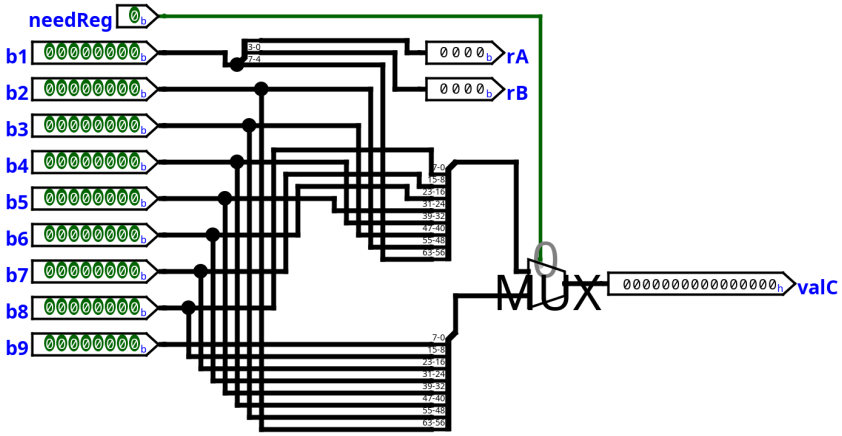
2.2 Instruction Memory

Our instruction memory module consists of a ROM module that stores the program along with a counter and 10 registers to store each byte of our program. Based on the counter we use a decoder to set the value of the corresponding register the count points to. We also utilize a simple 4-way `AND` gate to determine when to reset our counter. This module takes 10 cycles to read in all 10 bytes (one for each byte).



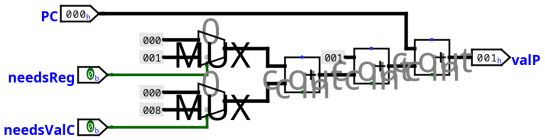
2.3 Align

The align module determines the values of `rA`, `rB`, and `valC` based on whether our instruction needs registers or not. When the instruction doesn't need registers we simply construct `valC` based on the first byte being the most significant to the eighth being the least. We just let our registers still be the first byte in this case as it does not matter what is in them. When we do have registers used in our instructions we start with our most significant byte in `valC` being the second byte to the least significant being the ninth.



2.4 PC Increment

PC increment simply increments the `PC` based on if our instruction read in registers and/or a `valC`. If we read in registers we add 1 to our `PC` and if we read in a `valC` we add 8 to our `PC`. We then also just add 1 for our first byte containing `icode:ifun`.



2.5 Needs ValC and Registers

We simply encode which instructions need a `valC` and registers based on their `icode`.