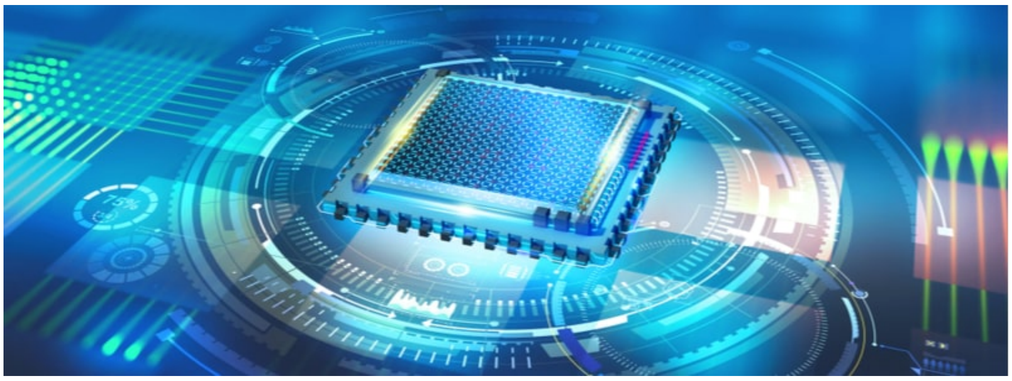
CA-ES Assignment

Group 6



Date 05 May 2023

Authors	Daniel Liashenko	2653117	TCS
	Daan Schram	2692759	TCS
	Guido Baels	2676478	TCS
	Valerijs Farbtuhs	2794764	TCS

Table of Contents

README

We actually implemented the CPU of the described custom architecture. We used the open-source program “logisim-evolution” for the circuit. We also prepared a short video presentation where it (CPU) executes the written program below (section 1).

- https://github.com/logisim-evolution/logisim-evolution
- https://youtu.be/dsNq0e-tasY (Demo)
- https://youtu.be/cHAd88OPcFk (Datapath in details)
- https://youtu.be/825Y6GAKMAM (Controller in details)

The source files could be found here: https://github.com/ShubaShaba/Custom_9-bit_CPU

1. Assembly Code

Below, assembly code is given for the application. It shows all required instructions, with an explanation next to it.

```
.org 11                        // special number indicates start of program

sethi %r1 [P]                  // write 5-bit number to first 5 bits of %r1
shift %r1 3                    // shift content of %r1 by three bits
sethi %r2 [W]                  // write 5-bit number to first 5 bits of %r2
shift %r2 3                    // shift content of %r2 by three bits

scalvs %r1 %r2 %r3             // takes memory location in %r1 and %r2
                               // which points to the start of vectors v1
                               // and v2, calls accelerator to access
                               // vectors directly from memory and
                               // calculate their scalar product, which
                               // is returned to %r3

sethi %r2 [b]                  // write 5-bit number (address of <b>) to
                               // first 5 bits of %r2
shift %r2 3                    // shift content of %r2 by three bits (now
                               // %r2 contains the m. location of <b>)
ld %r2 %r1                     // load data stored at m. location specified
                               // in %r2 onto %r1
add %r1 %r3 %r3                // add the content of %r1 and %r3, (store
                               // at %r3)
sethi %r2 [res]
shift %r2 3
st %r3 %r2                     // store the final result at reserved
                               // memory location for it

Content table: ---------------------------------------------------------------

P: 21, 3, 0, 2                 // vector P
W: 1, 9, 255, 17               // vector W
b: 141                         // b
res:                           // m. location reserved for final result
                               // should be 223 in the end
```

2. Instruction Set Architecture

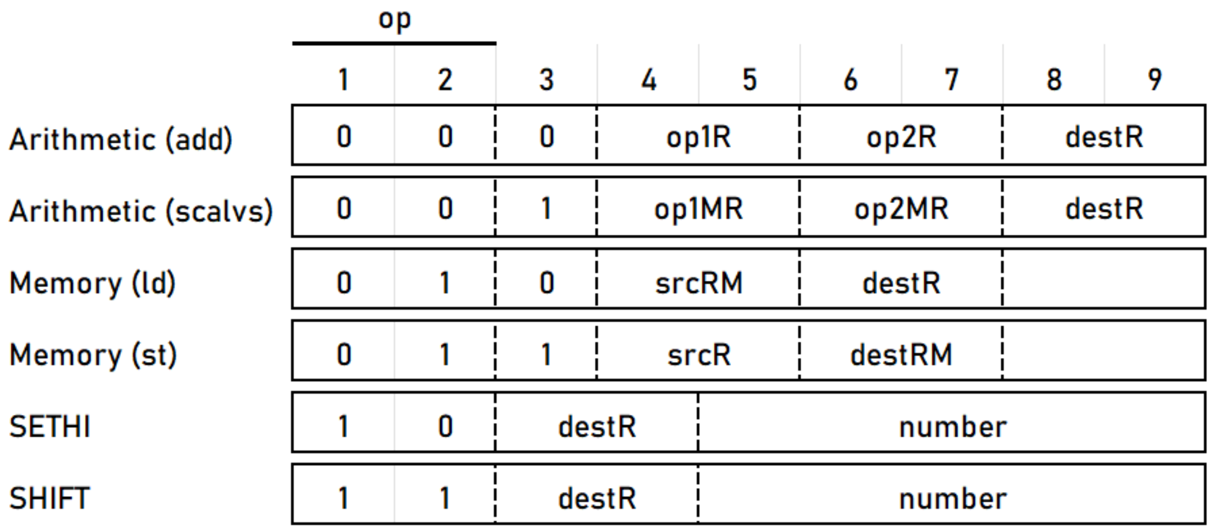
In order to achieve the desired application, we designed a custom 9-bit architecture. This was necessary to be able to execute all needed operations. A brief description for each operation is given below. Table 1 is a systematic schema of all possible operations in our ISA.

Arithmetic
An arithmetic instruction starts with bits 00. After that, there is one bit specifying whether we want to add (bit 0) or call the vector scalar multiplication accelerator (bit 1). After this bit, there are two bits for the register of the first operand. The next two bits are for the register of the second operand. The last two bits show what will be the destination register.

Memory
Instructions regarding memory start with bits 01. The next bit shows whether the instruction is used to load (bit 0) or store (bit 1). Bits 4 and 5 are used to show the source register, bits 6 and 7 are used to show the destination register.

SETHI
The SETHI instruction stores user-entered 5-bit numbers in a designated register, bits 3-4 set the destination register and bits 5-9 for the number to be stored. The stored number is written to the first 5 most significant bits of the 8-bit register (00000001 + 00101 => 00101001). The instruction was designed this way to allow flexibility for user-entered numbers, which can be shifted later with the SHIFT instruction, enabling users to store any 8-bit number with only 5 bits.

SHIFT
SHIFT instruction performs a logical shift to the left on the content of specified register. It starts with bits 11. Bits 3-4 specifies the register which contains the number to be shifted and bits 5-9 represent a number of shifting positions.

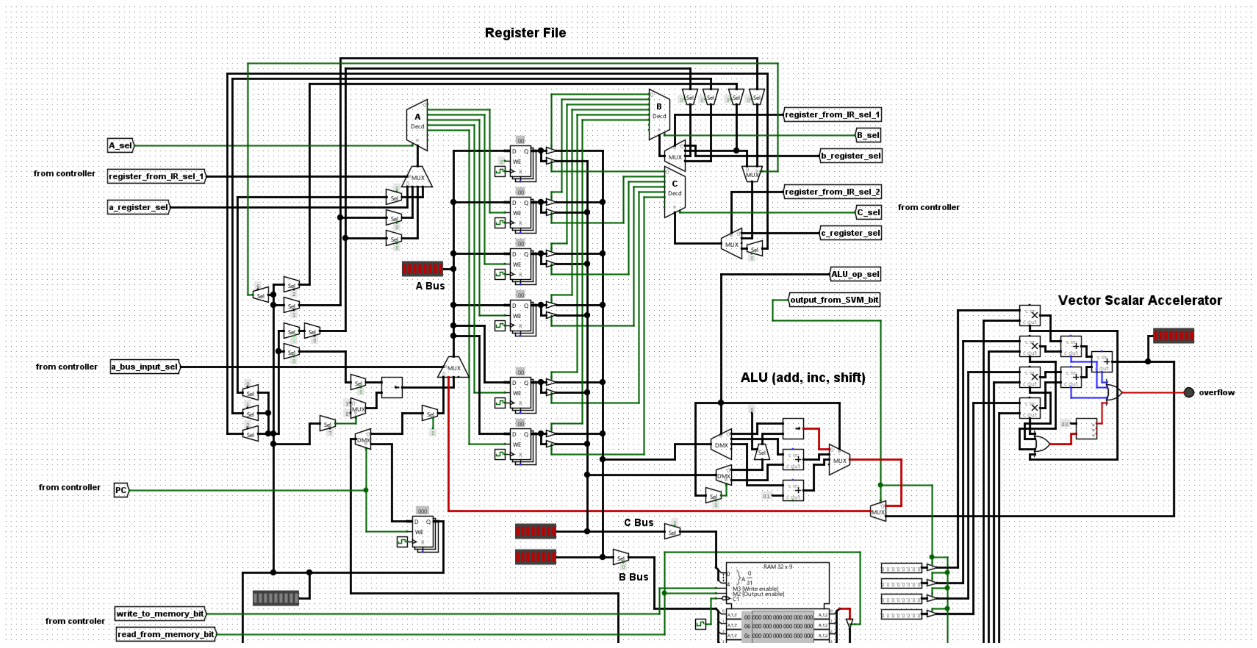


	op								
	1	2	3	4	5	6	7	8	9
Arithmetic (add)	0	0	0	op1R		op2R		destR	
Arithmetic (scalvs)	0	0	1	op1MR		op2MR		destR	
Memory (ld)	0	1	0	srcRM		destR			
Memory (st)	0	1	1	srcR		destRM			
SETHI	1	0	destR		number				
SHIFT	1	1	destR		number				

Table 1: Systematic schema of all possible operation in our ISA

3. Data-Path, Resources, Memory and Control Path

Data-Path



Pic. 1

The datapath for our system resembles the general architecture discussed in M5, which is used in the ARC processor. The register file is connected to memory and a custom ALU / Vector Scalar Accelerator through A, B, and C buses. A bus is used for storing numerical value to register and B and C allows those to propagate through the rest of DP. Decoders are connected to these buses to determine which register can use the bus at a given moment. Other wires are used to directly connect the content of the instruction register (IR) to the rest of the datapath, specifically the decoders. This is necessary because the IR contains information needed to execute instructions at any given time.

*Note that a more detailed explanation of the datapath structure, including examples of instruction execution, can be found in Section 4.

Resources

GENERAL PURPOSE REGISTERS	
%r0	8 bits, const value 0
%r1	8 bits
%r2	8 bits
%r3	8 bits

SPECIAL REGISTERS	
%temp0	8 bits, storing temporary values
%PC (program counter)	8 bits
%IR (instruction register)	9 bits

ALU	
+	8-bit unsigned addition
->	logical right shift (bitwise)
+=1	increment 8 bit number

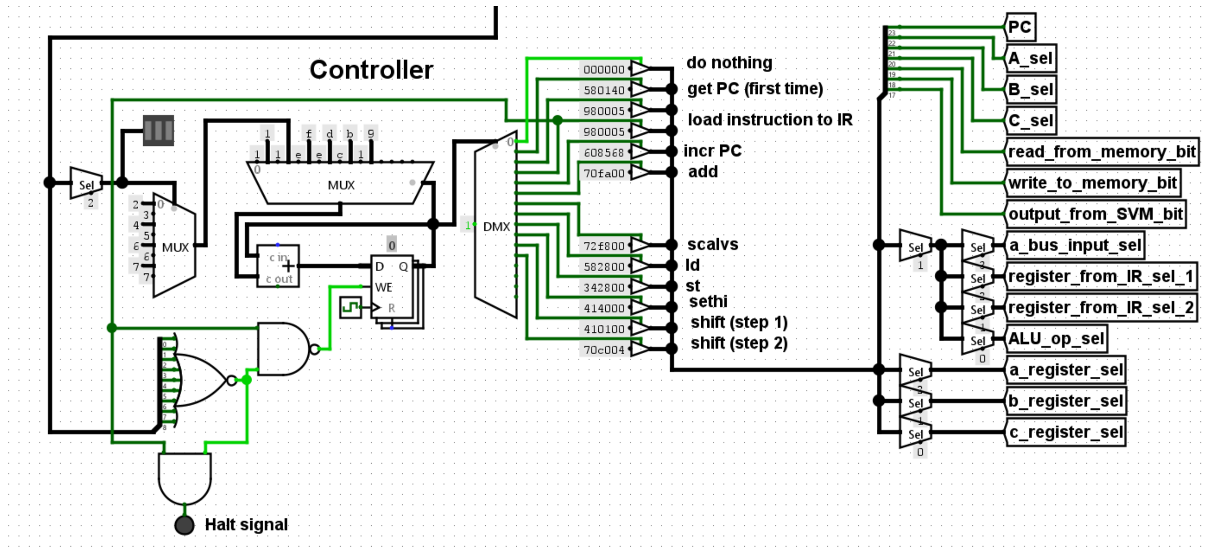
Memory	32 x 9 RAM

Memory (stored program from section 1)

Memory location	**Memory content**
0	Program data (initial instruction)
1	Final result
2	P1
3	P2
4	P3
5	P4
6	W1
7	W2
8	W3
9	W4
10	B
11	instruction 1
12	instruction 2
13	instruction 3
14	instruction 4
15	instruction 5
16	instruction 6
17	instruction 7
18	instruction 8
19	instruction 9
20	instruction 10
21	instruction 11
22	instruction 12

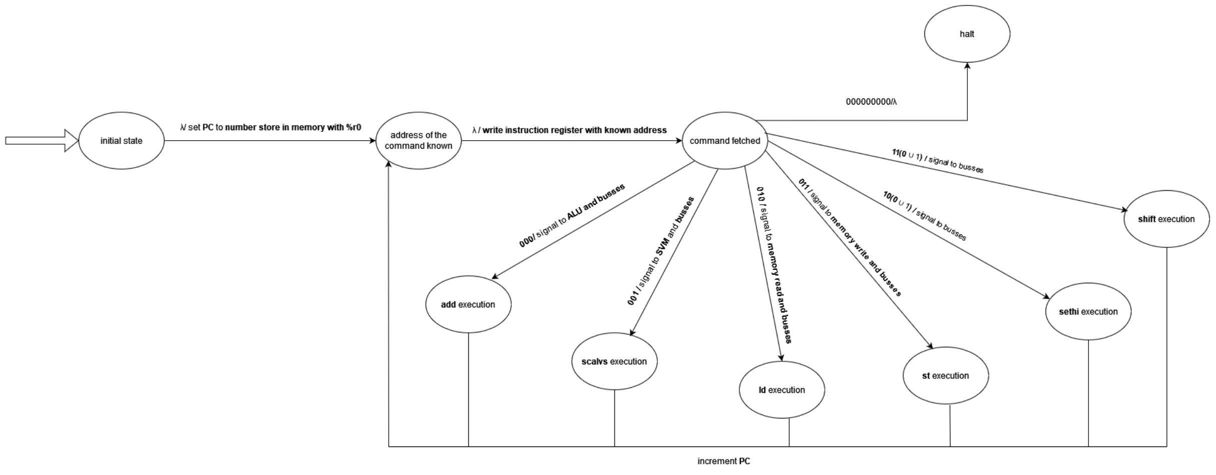
*Note: memory locations 23 up until 31 are not used by current algorithm

Control Path and FSM



Pic.2

Our controller is a classic Moore's machine that employs a 4-bit numerical value to represent the current state. A transition to the next state is achieved via an adder that is connected to the input of the register. The two multiplexers incorporated in the design allow for flexible, non-linear state transitions, resulting in control signals for the datapath.


halt
000000000/λ
initial state
λ/ set PC to number store in memory with %r0
address of the command known
λ / write instruction register with known address
command fetched
11(0 ∪ 1) / signal to busses
shift execution
10(0 ∪ 1) / signal to busses
sethi execution
011 / signal to memory write and busses
st execution
010 / signal to memory read and busses
ld execution
001 / signal to SVM and busses
scalvs execution
000/ signal to ALU and busses
add execution
increment PC

Pic.3

The image above (pic.3) describes a high level FSM of our controller, with the purpose of giving a clear overview of the input/output given for each instruction. First of all, the controller will always send a signal to fetch the correct number to set the program counter to. After that, the right value will be written to the instruction register. Now the controller will keep on executing instruction by sending the correct signals based on input from IR and then increment the program counter. However, if the input is all 0’s, the controller will halt.

4. Explanation of Design Choices

Design of Data-Path

The register file is required to have exactly 6 registers to ensure etch, decode, execute, and memory access. Out of these, the first 4 registers (%r0-%r3) are reserved for user programs and can be used for both reading and writing. The first register (%r0) has a unique property in that its value is always zero, which is necessary for accessing the first memory cell at the start of program execution.

The fifth register (%temp0) is necessary for executing shift operations. As users specify the number of positions to shift in the instruction, this information must be stored in the register file before proceeding to the Arithmetic Logic Unit (ALU).

The sixth register is the Program Counter (PC). The PC holds the memory address of the instruction that the system is currently executing. By our design the first memory cell contains a number indicating where the program is stored in memory, and its value is loaded into the PC (with the use of %r0, which always points to the first memory cell if its signal is sent onto memory) at the start of program execution.

Example of instruction execution:

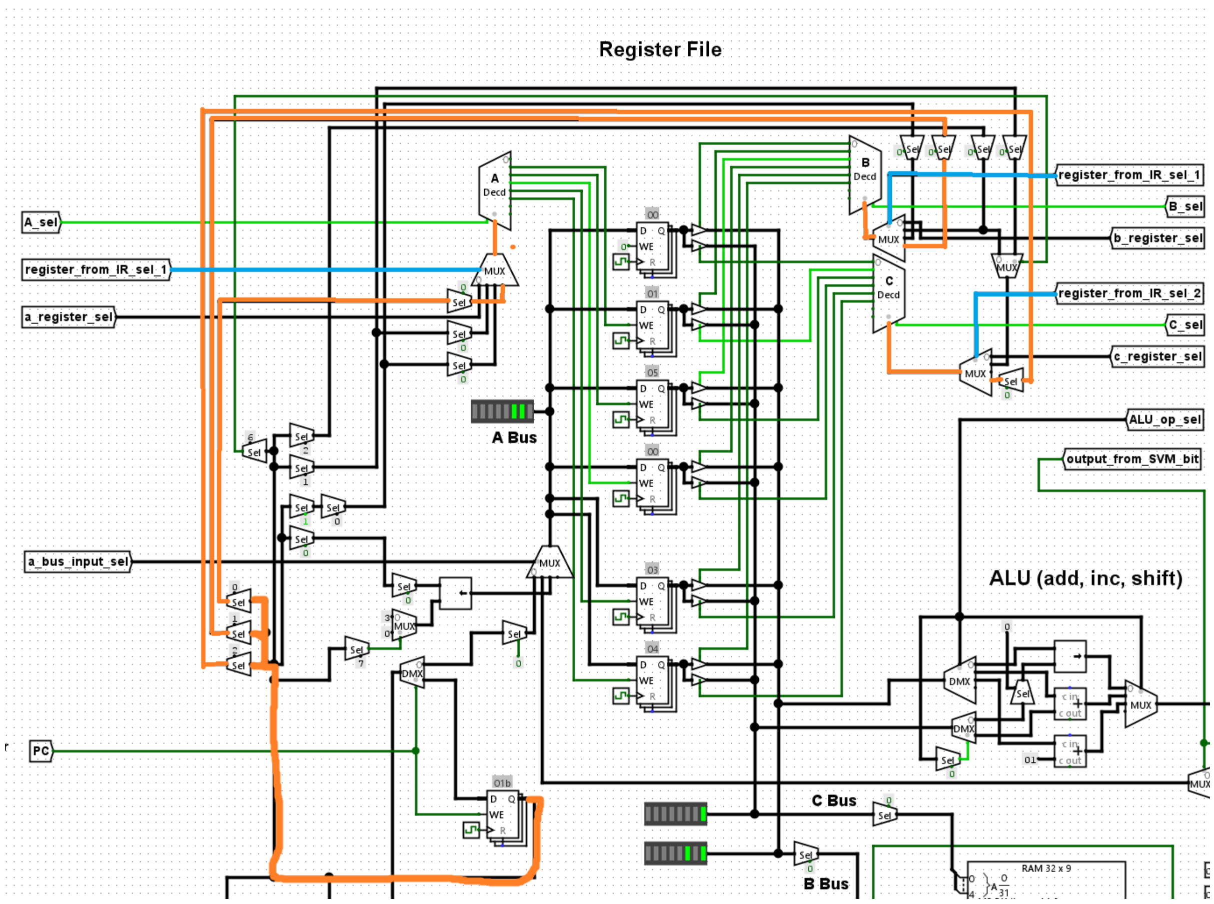
The following section provides a closer look on the datapath actions during instruction. The similar procedure was done for the rest of the instructions in one of earlier presented videos: https://youtu.be/cHAd88OPcFk.

Due to the size of the circuit it would have been inconvenient to include it here, so we decided to put it in a video, we hope it is acceptable.

Consider the instruction **`add %r1 %r3 %r3:`**

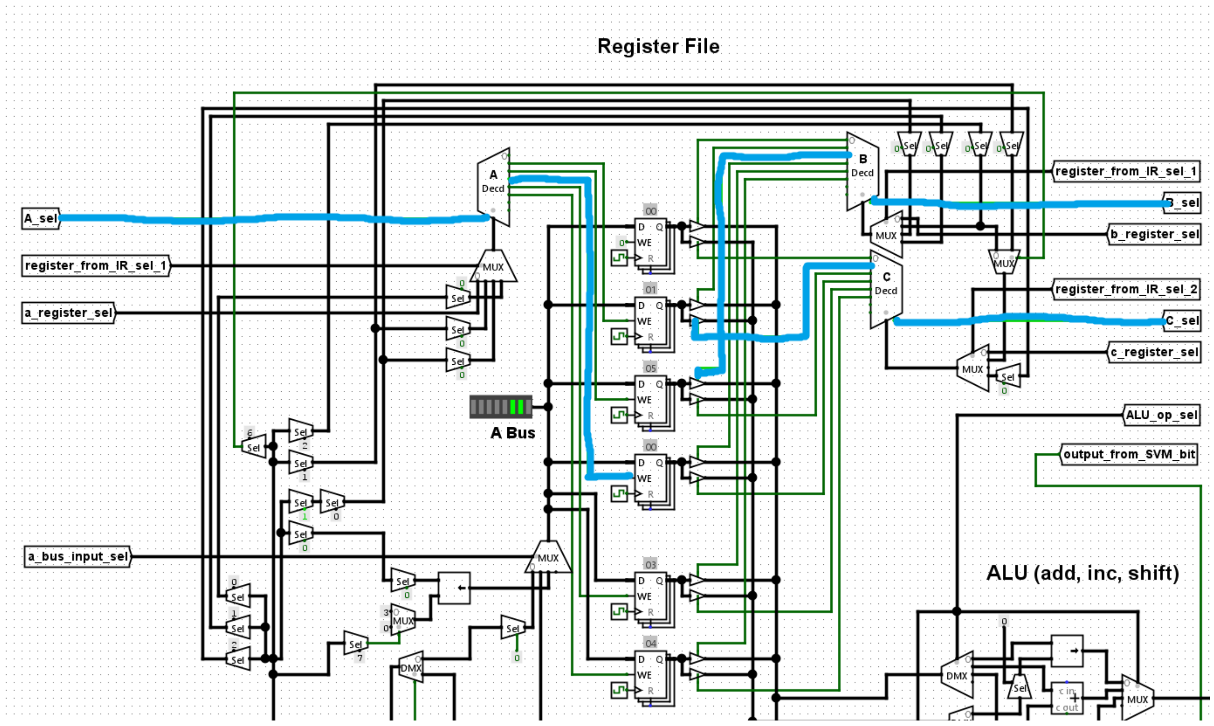
Step 1:

The controller sends signals to the multiplexers (blue) to propagate the value from the IR (orange), which should contain the arguments for registers selection as specified in the arithmetic(add) type of instruction (see section 2).



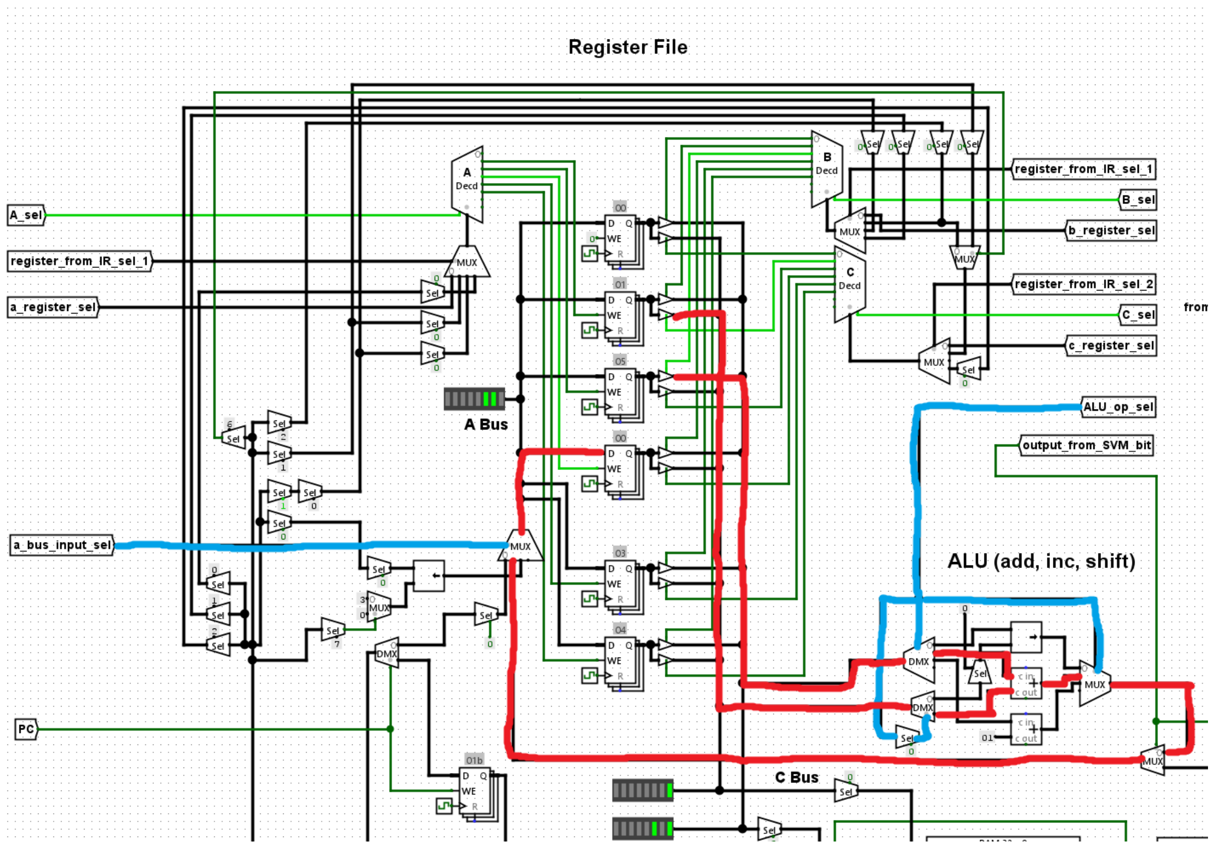
Step 2:

Controller sends signals (blue) to A, B and C decoders to allow early specified registers to use buses.



Step 3:

The data (red) is propagated through B, C buses to the ALU and back to RF with A bus. The ALU is set to the add operation by controller (blue signal).



5. Optimized Design for Performance

1) Since the designed architecture is multi-cycled (in case of **fetch** procedure and **shift instruction**), in theory, it is possible to add pipelining for those operations specifically.

2) The original plan for our project was to incorporate 8 additional 8-bit registers for storing two vectors, which would then be loaded onto the accelerator for computing their product. However, upon further analysis, we decided to propose a theoretical design for a custom memory unit instead. Although this design has not yet been implemented in a circuit-wiring application, it offers the potential to access 8 memory locations simultaneously **in a single clock cycle**, thereby greatly improving the overall efficiency of our system.

   The proposed memory unit will have an additional control bit that indicates the way the memory is extracted during the current clock cycle (traditional vs 8 locations simultaneously). Furthermore, it will include an extra 8-bit input for memory location. The procedure for accessing the memory will be as follows: First, the control bit will indicate that we need two 4-number vectors. Then, we will input the address of the first number of the vector P and the address of the first number of the vector W (which is indicated by the Y address). The memory output will then be the following: the four numbers stored at addresses X, X+1, X+2, and X+3 (which represent the first vector) and the four numbers stored at addresses Y, Y+1, Y+2, and Y+3 (which represent the second vector).

   In summary, our proposed design for a custom memory unit offers a significant improvement over the original plan of using additional registers. It allows us to access 8 memory locations simultaneously in a single clock cycle, thereby reducing the overall processing time required for our system.

   ***Note:** the issue with accessing vectors form memory was also addressed in demo video*