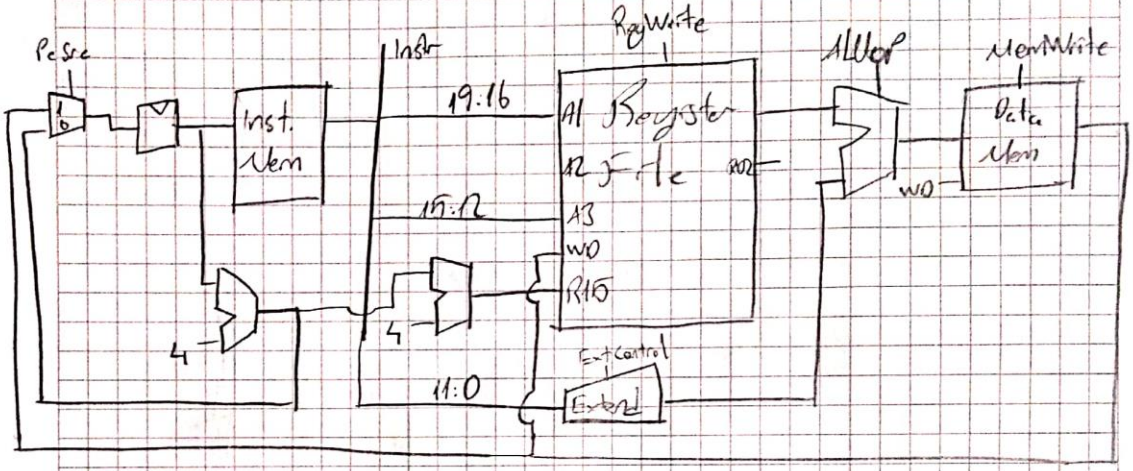
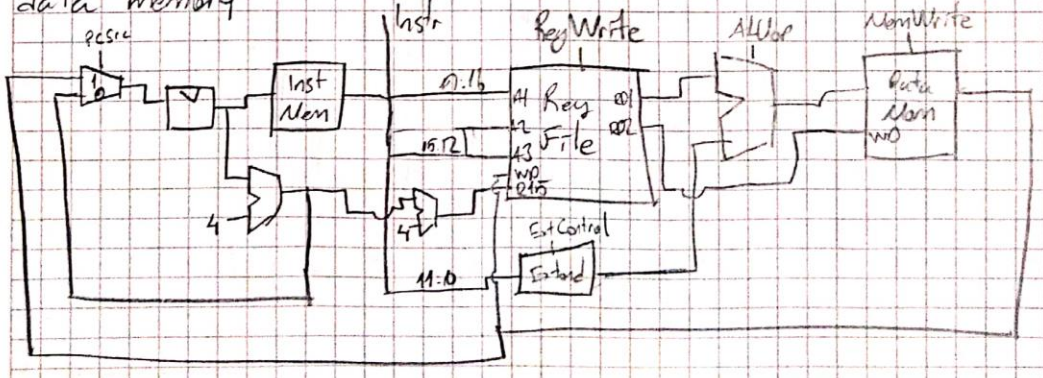
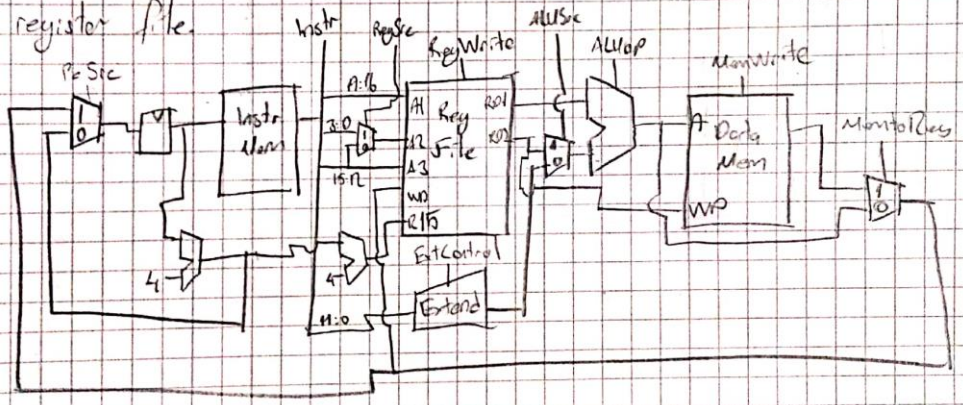
Initially, we are provided with the following datapath, which is extended to perform LDR operation:



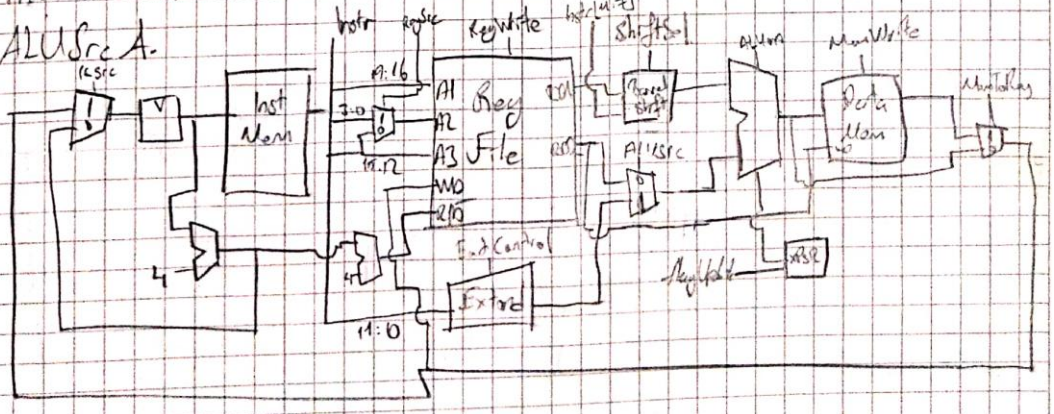
Extend this datapath to perform STR operation. For this, connect the destination register bits of the instruction to A2 to obtain its value in the output port of the register file. Then, connect RD2 to WD port of the data memory



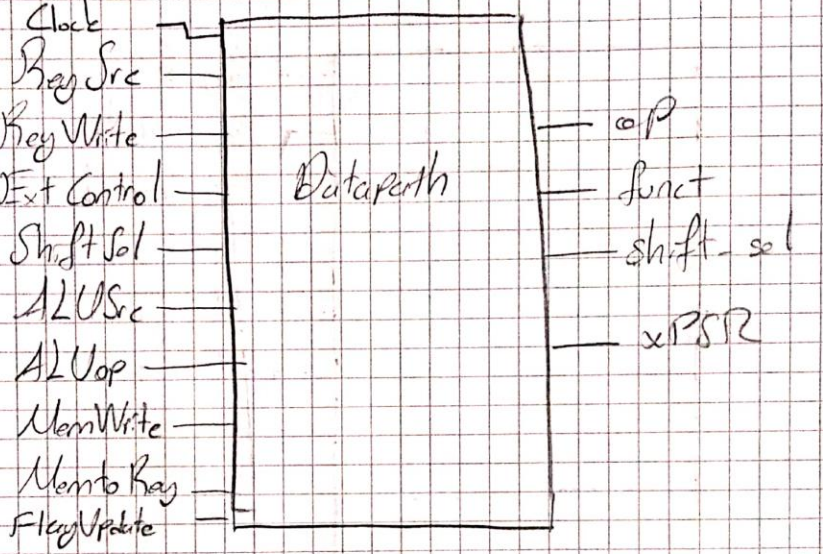
To be able to perform the ALU instructions, (ADD, SUB, AND, ORR, CMP), we need to be able to reach Rm. To do this, add a mux before RD2 input of the register file. The ALU result then is directly inputted to the register file. To achieve this, add another mux in front of WD3 input of the register file.



Lastly, to be able to perform LSL and LSR instructions, add a barrel shifter in front of the ALUSrcA.

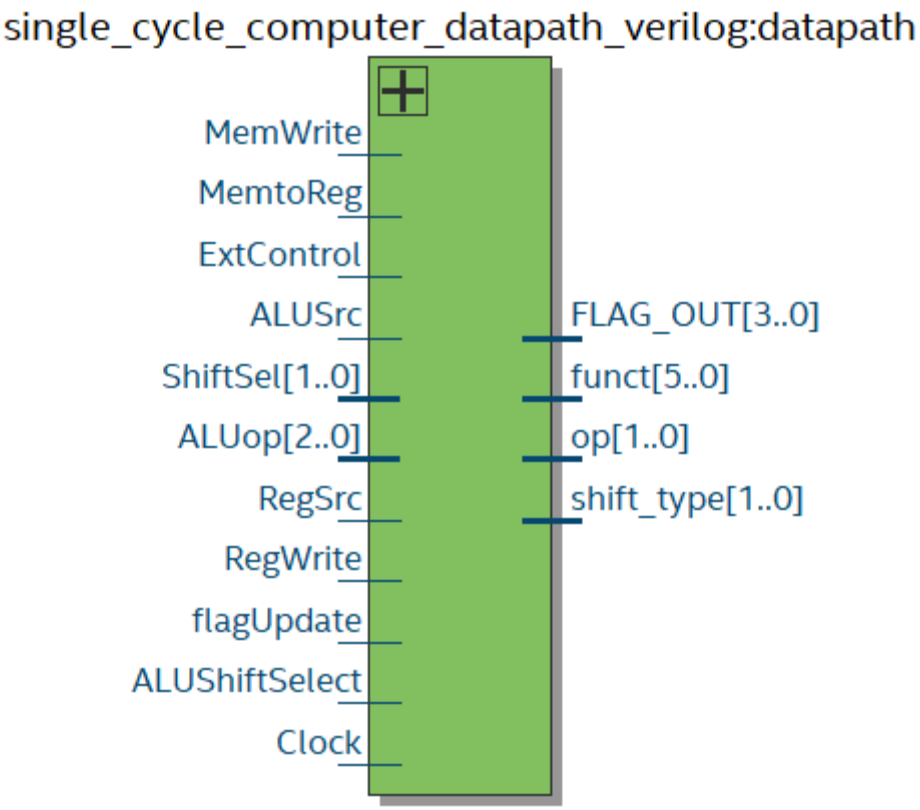


Black-box description of the datapath can be drawn as following



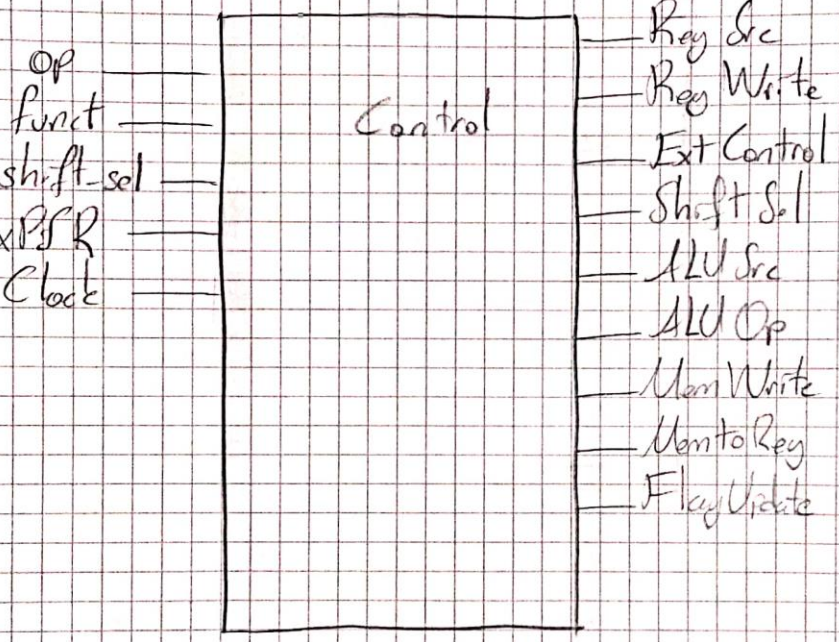
PCSrc is omitted since there is no branch instruction The controller is fed with the all instruction.

RTL view of the compiled black-box description of the datapath is as following:



Controller Design

The black-box diagram of the controller is as following.



The controller takes the parts of the instruction as input and produces the control signals as outputs.

Progression and Truth Table

We will follow the same progression of instructions in the controller as we did in datapath. Initially, we only have LDR instruction to perform this, we need control signals: RegWrite, ALUop, MemWrite, Ext Control. Their values should be as following:

RegWrite = 1
Ext Control = 0
ALUop = 3'b000
MemWrite = 0

Next, we will have STR operation To perform this instruction, we do not need to add a new control signal. For STR operation, control signals become as following

RegWrite = 0
Ext Control = 0
ALUOp = 3'b000
MemWrite = 1

As a next step, we add ADD, SUB, AND, ORR, CMP instructions. We require additional control signals for the multiplexers we added. Namely, we added RegSrc, and ALUSrc

For these instructions the control signals are as following:

Common for ADD, SUB, AND, ORR :

RegWrite = 1
Ext Control = X (Immediate is not used)
MemWrite = 0
RegSrc = 1
ALUSrc = 0

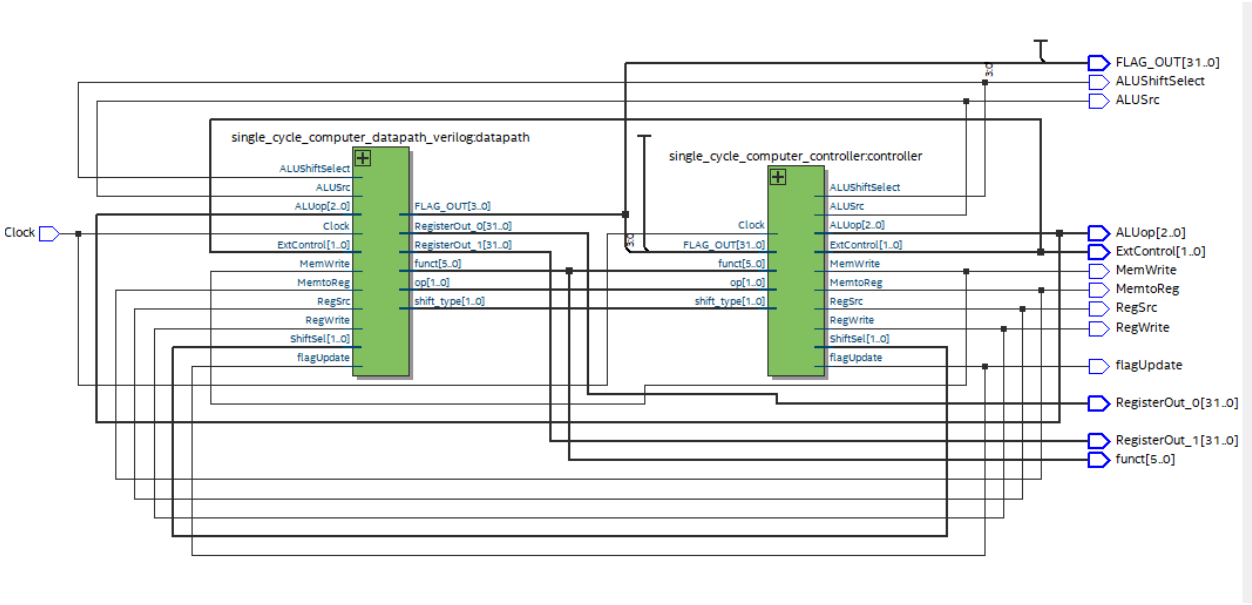
ALUop changes for instructions.

ALUop = 3'b000 (ADD)
= 3'b001 (SUB)
= 3'b101 (ORR)
= 3'b100 (AND)
= 3'b001 (CMP)

Lastly, we add additional multiplexers and control signals for the shift operations and the current program status register. Complete truth table for concluding datapath for all of the instructions is as following:

	Shifts	ALUop	RegSrc	ExtControl	RegWrite	MemWrite	MemtoReg	ALUSrc	FlagUpdate	ALUShifts
ADD	11	000	0	X	1	0	0	0	0	0
SUB	11	001	0	X	1	0	0	0	0	0
ORR	11	101	0	X	1	0	0	0	0	0
LSL	00	000	0	1	1	0	0	0	0	1
LSR	01	000	0	1	1	0	0	0	0	1
AND	11	100	0	X	1	0	0	0	0	0
CMP	11	010	0	X	0	0	0	1	1	0
STR	11	000	1	1	0	1	X	1	0	0
LDR	11	000	1	1	1	0	1	1	0	0

When the datapath and the controller is merged, the concluding connection diagram, which clearly shows inputs and outputs can be observed:



The instruction memory is initialize with following instructions to show that our single cycle computer is capable of executing all the instructions given.

Instruction #	ARM Format	RTL Format	HEX Format	R0	R1
1	LDR R0, [R0, #0]	R0 <- MEM[0]	0x04100000	4	0
2	LDR R1, [R2, #2]	R1 <- MEM[2]	0x04181002	4	12
3	ADD R0, R0, R8	R0 <- R0 + R8	0x00800008	4	12
4	ADD R1, R1, R8	R1 <- R1 + R8	0x00811008	4	12
5	SUB R0, R1, R0	R0 <- R1 – R0	0x00410000	8	12
6	ADD R0, R0, R8	R0 <- R0 + R8	0x00800008	8	12
7	ORR R0, R0, R1	R0 <- R0 | R1	0x01810001	12	12
8	ADD R0, R0, R9	R0 <- R0 + R8	0x00800008	12	12
9	CMP R0, R1	SET THE FLAG	0x01410000	12	12
10	ADD R0, R0, R9	R0 <- R0 + R9	0x00800008	12	12
11	LSR R0, R0, #1	R0 <- R0>>1	0x01A000A0	6	12
12	AND R0, R0, R1	R0 <- R0 & R1	0x00000001	4	12
13	ADD R0, R0, R8	R0 <- R0 + R8	0x00800008	4	12
14	LSL R0, R0, #1	R0 <- R0<<1	0x01A00080	8	12
15	ADD R0, R0, R8	R0 <- R0 + R8	0x00800008	8	12
16	STR R0, [R8, #0]	MEM[0] <- R0	0x04080000	8	12
17	LSR R0, R0, #1	R0 <- R0>>1	0x01A000A0	4	12
18	ADD R0, R0, R8	R0 <- R0 + R8	0x00800008	4	12
19	LDR R0, [R8, #0]	R0 <- MEM[0]	0x04180000	8	12
20	ADD R0, R0, R1	R0 <- R0 + R8	0x00800008	20	12

See that we frequently add our sole destination register R0 with R8. This is to see the value of R0 in the output line which will be connected to the LEDs on the FPGA board. The addition does not change the value of R0 since R8 has the value 0.

According to the expected operation of these instructions, we created test vectors for the destination register, which is almost always R0 to be able to show all the operations with limited LEDs, Created test vectors are as following:

// R0

00000000000000000000000000000000

00000000000000000000000000000100

00000000000000000000000000001100

00000000000000000000000000001100

00000000000000000000000000001000

00000000000000000000000000001100

00000000000000000000000000001100

00000000000000000000000000001100

00000000000000000000000000001100

00000000000000000000000000001100

00000000000000000000000000001100

00000000000000000000000000000110

00000000000000000000000000000100

00000000000000000000000000000100

00000000000000000000000000001000

00000000000000000000000000000000

00000000000000000000000000001000

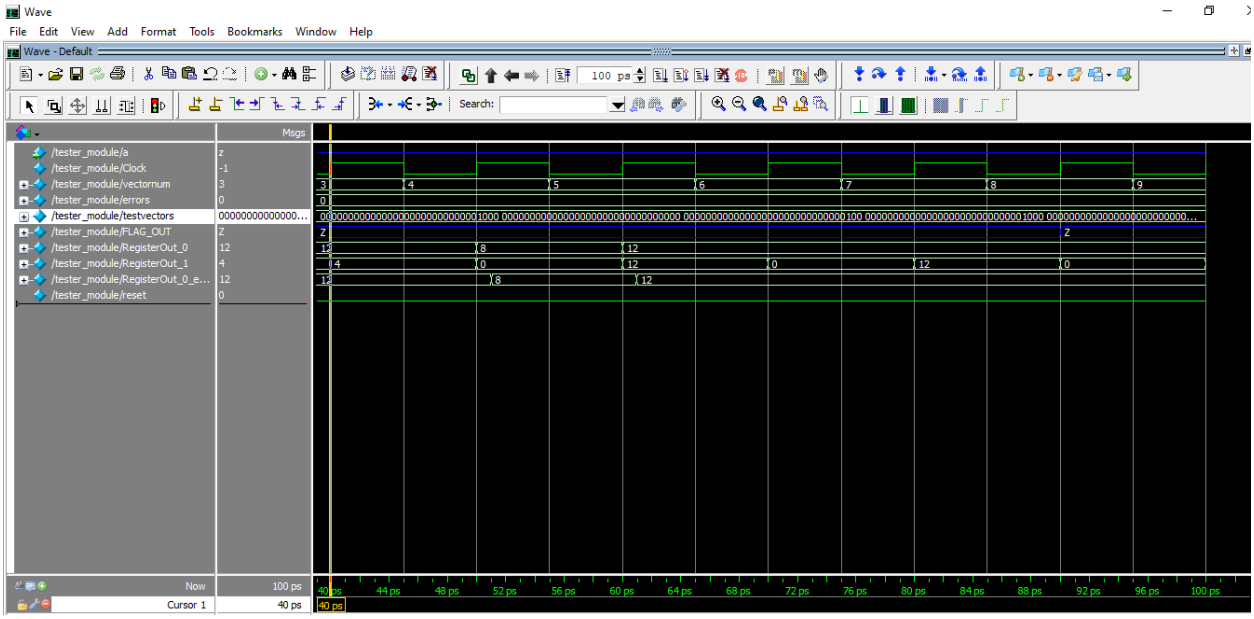
00000000000000000000000000000100

00000000000000000000000000000000

00000000000000000000000000001000

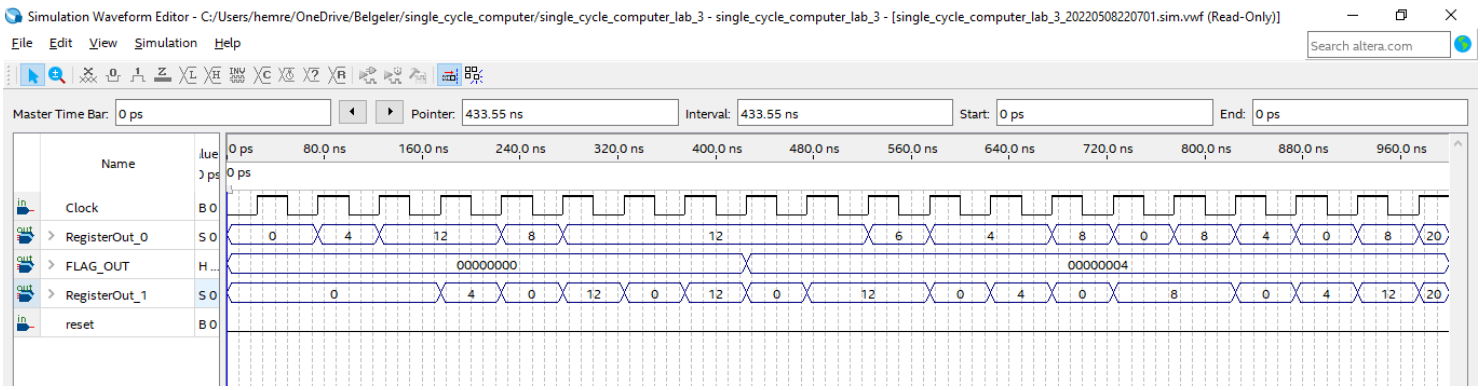
See that with correct progression of instructions it is not necessary to observe the output of all registers. For example, in the beginning, we load certain memory location to register R1, and then make an operation with R1 and R0, and write the result to R0. The correct result in R0 allows us to infer that the information is drawn from the memory correctly, and the operation is done correctly. In other words, by only seeing the partial view, we can infer the complete one.

Testbench results with this vector table is as following:

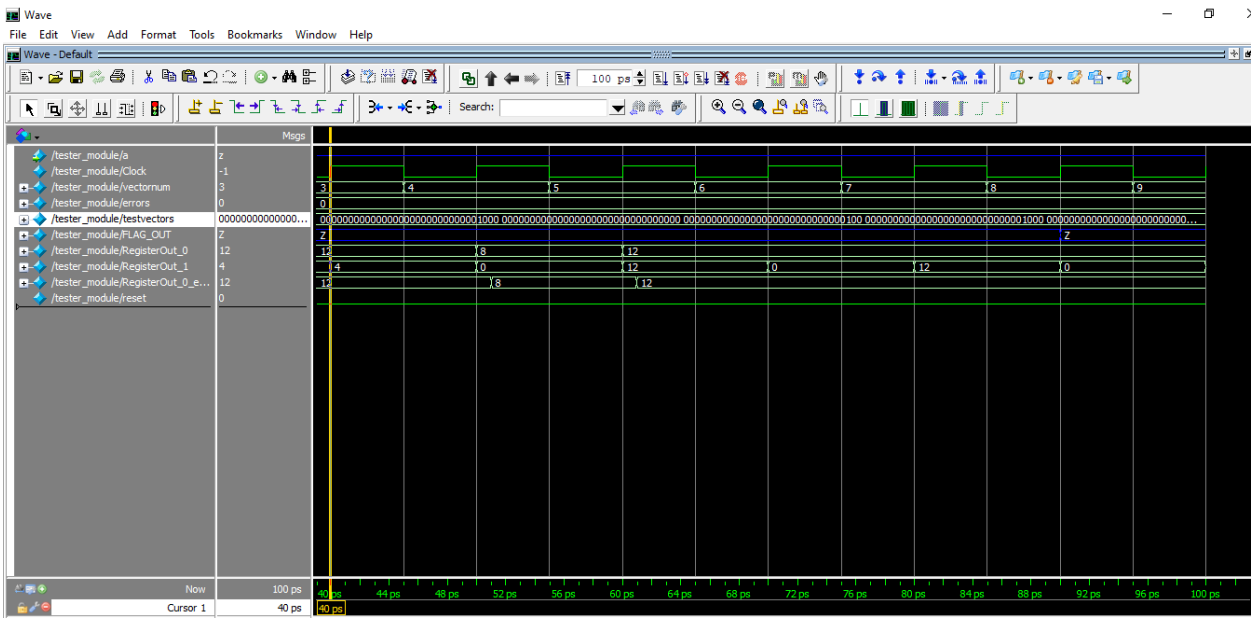


As can be seen, there is no error in the simulation, meaning our design works.

The detailed instruction progression table can also be observed with uwf, which is also added as follows for clarity:



Another testbench result without test vectors are also added for they are more reliable than the uwf program. Simulation results do not contain any mismatch.



Notes on Implementation

See that we separately created an instruction memory and a data memory, which is in real scenarios would not be quite desirable. Yet, for the single cycle computer, it is a must. Both memories work combinational, which is critical since the data or instruction must be present before the clock cycle to have a healthy operation of the computer. For both instructions and data memory, we initialized them to have certain values before any clock hits. See that this is realizable in FPGA environments whereas it is not in ASIC environments. On the other hand, this is not a big draw-back in terms of “right-way of doing things” since in real life putting instruction in instruction memory and initializing the data memory corresponds to programming a certain device, and programming a device almost always done with the help of an external device, which separately writes the instructions or data to correct memory locations. Instructions are also written to ROMs to be able not to lose it when the power is cut. In each initialization of the computer, these instructions are drawn from the ROM and executed.

Notes on Parametrization

Parametrization in general gives designer a significant elasticity with the modules he designs since it is not required for the designer to create the same module out of scratch just because width of certain port changes. This elasticity becomes very critical especially for modules such as registers and multiplexers. It is important to have a parametrized design for this kind of abundantly used components with variable input lengths. Parametrized design pushes the designer to one more abstract level, where instead of thinking with only concrete busses and wires, a comprehensive approach including all the cases of the parameter is essential.

Yet, for our design, parametrized design might cause certain complications. As an example, see that our instructions are assumed to be 32-bits. Whole ARM instruction set architecture is based on concluding the values of certain control signals depending on specific bits of the instruction being executed. We should not overlook things such as specific bits of the instruction go into register file to decided which register to be selected in the output, lower order bits goes into immediate extender to conclude the immediate value depending on the instruction, etc. Other problems might rise due to the memory organization of both data and instructions. Instructions and data are read from the memory in 4 bytes, and the memory is addressed and organized in that regard. Basically, to reach the next instruction, we increment the program counter by 4. Yet, if the data length in the memory would be changed, we should change this constant sum, meaning we should change parts of the architecture in correspondence with the new data length. Aside from that, changing any data length related to the memory units might cause a specific program written for the machine not to operate at all. Or, we would be required to alter certain parts of the architecture, which is also dependent on the new parameter we set. It is possible to see that making some modules' parameters, data-widths variables, we face with variable dependent architecture design problem, whereas the purpose of parametrization was to reduce this complexity, not to increase it. This is due to certain conventions such as instruction format, and memory organization of the computers we design.

In my implementation, I used parametrized modules as much as possible, especially where I believed later be altered. Yet, for modules that strictly dependent on the 32-bit ARM instruction set architecture, I followed a literal approach rather than parametrized to reduce the complexity and prevent potential misconceptions about the possibility of changing a set parameter, where in reality would not be possible without changing significant portion of the architecture.

Lastly, design would not work with arbitrary data-width since it requires the memory organization and addressing to change significantly.