Mustafa Sefer Destegul

2109973

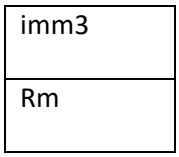
EE446 LABWORK #4

ISA & Datapath Design for Multi-Cycle CPU

1). ISA CONFIGURATION

a) Data Processing

15:14	13	12:10	9	8:6	5:3	2:0
Op=00	I	Cmd	S	Rd	Rn	Src2



0p: 00 for arithmetic and logic Operations

S : S=0 => no update on flags, S=1 => update flags.

Rd: Destination register

Rn: Source Register 1

Rm: Source Register 2

I	**Src2**
0	Rm
1	imm3

Cmd	**Operation**
000	Addition
001	Addition indirect
010	subtraction
011	Subtraction indirect
100	AND
101	OR
110	XOR
111	Clear

b) Shift Operations

15:14	13	12:10	9	8:6	5:0
Op=01	0	Cmd	0	Rd	Not used

0p: 01 for shift Operations

Rd: Destination register

Since we are asked that only 1 register will be used for shift operations, I didnt add immediate bit and Rn registers. Therefore, 1 register will be shifted by 1 in such a way one of the Cmd operation is chosen.

Cmd	Operation
000	Rotate Left
001	Rotate Right
010	Shift Left
011	Arith. Shift right
100	Logical shift right

c) Branch Operations

15:14	13	12:10	9	8:0
Op=10	I	Cmd	L	Src2

0	İmm8
Rd	000000

0p: 10 for brach Operations

L : L=1 => Branch with link , L=0 => Branch without link.

Rd: Destination register

I	Src2
0	Rm|00000
1	İmm8

Cmd	Operation
000	Branch unconditional
001	Branch with link
010	Branch indirect with link
011	Subtraction indirect
100	Branch if zero
101	Branch if not zero
110	Branch if Carry set
111	Branch if Carry not set

d) Memory Operations

15:14	13	12:10	9	8:6	5:0
Op=11	I	Cmd	0	Rd	Src2

İmm6	
Rn	İmm3

0p: 10 for brach Operations

Rd: Destination register

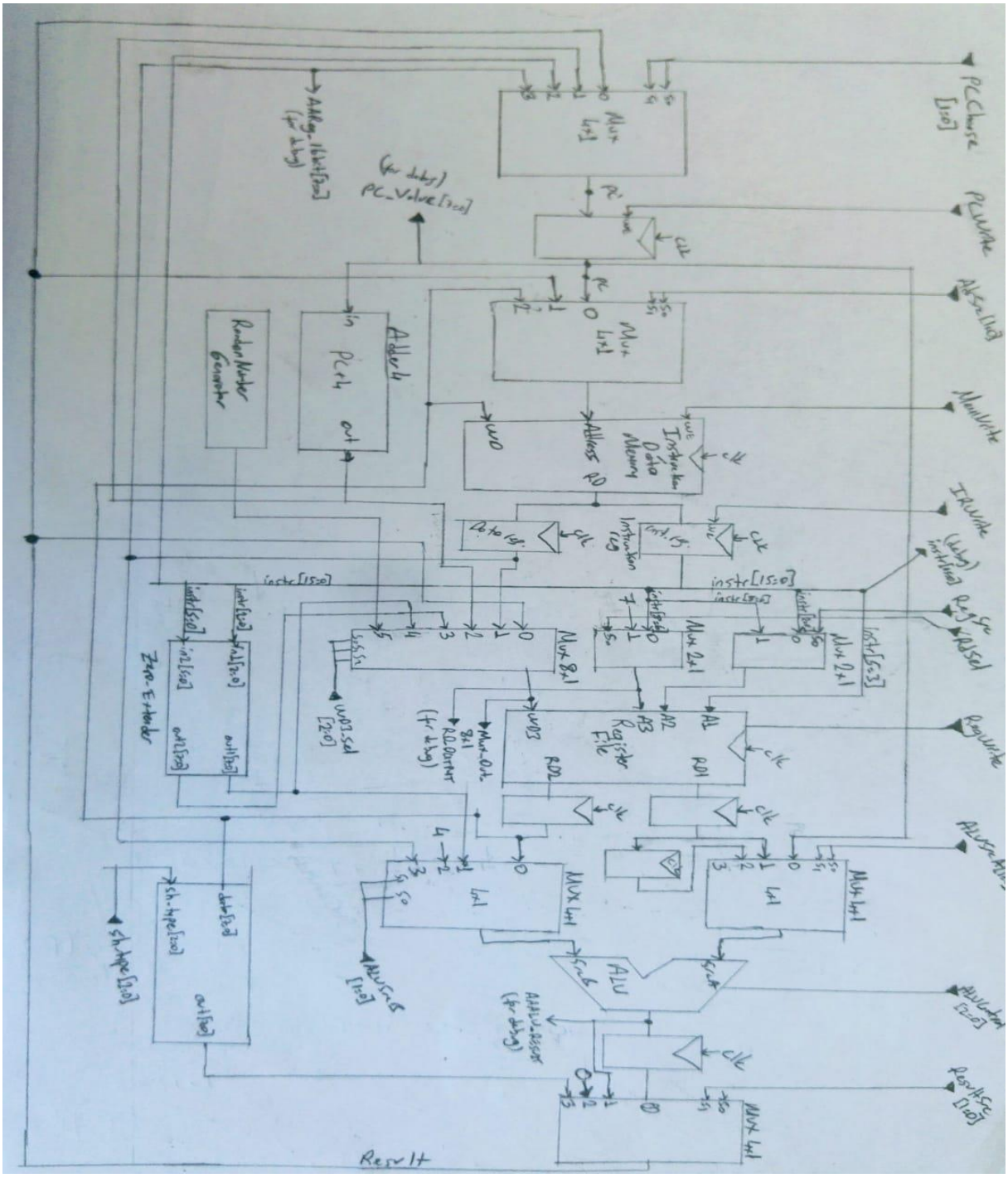
Rn: Source register

I	Src2
0	Rn|000
1	İmm6

Cmd	Operation
000	Load to reg. From memory
001	Load immediate to reg.
010	Store from reg. to memory

2) Multi-Cycle CPU Datapath Design

The whole datapath is as following.



2.1) Memory Operations(LDR,MOV,STR)

a) load to register from memory(LDR)

For daha path part, firstly I construct the memory operations since they cover most items in the datapath. The LDR(load to register from memory) instruction datapath is the same as in the lectures' multi-cycle CPU datapath. As can be seen from figure 1, The brown stars shows which component is necessary for loading to register from the memory. Since there can be both register to register operation or immediate operations (LDR,R0,R1,R1- LDR R0,R1,#5), a zero extender is used and a 4x1 multiplexer is added to the datapath to choose srcB input of the ALU.

1.cycle = Red , 2.cycle = green , 3.cycle =yellow , 4.cycle =purple, 5.cycle = blue

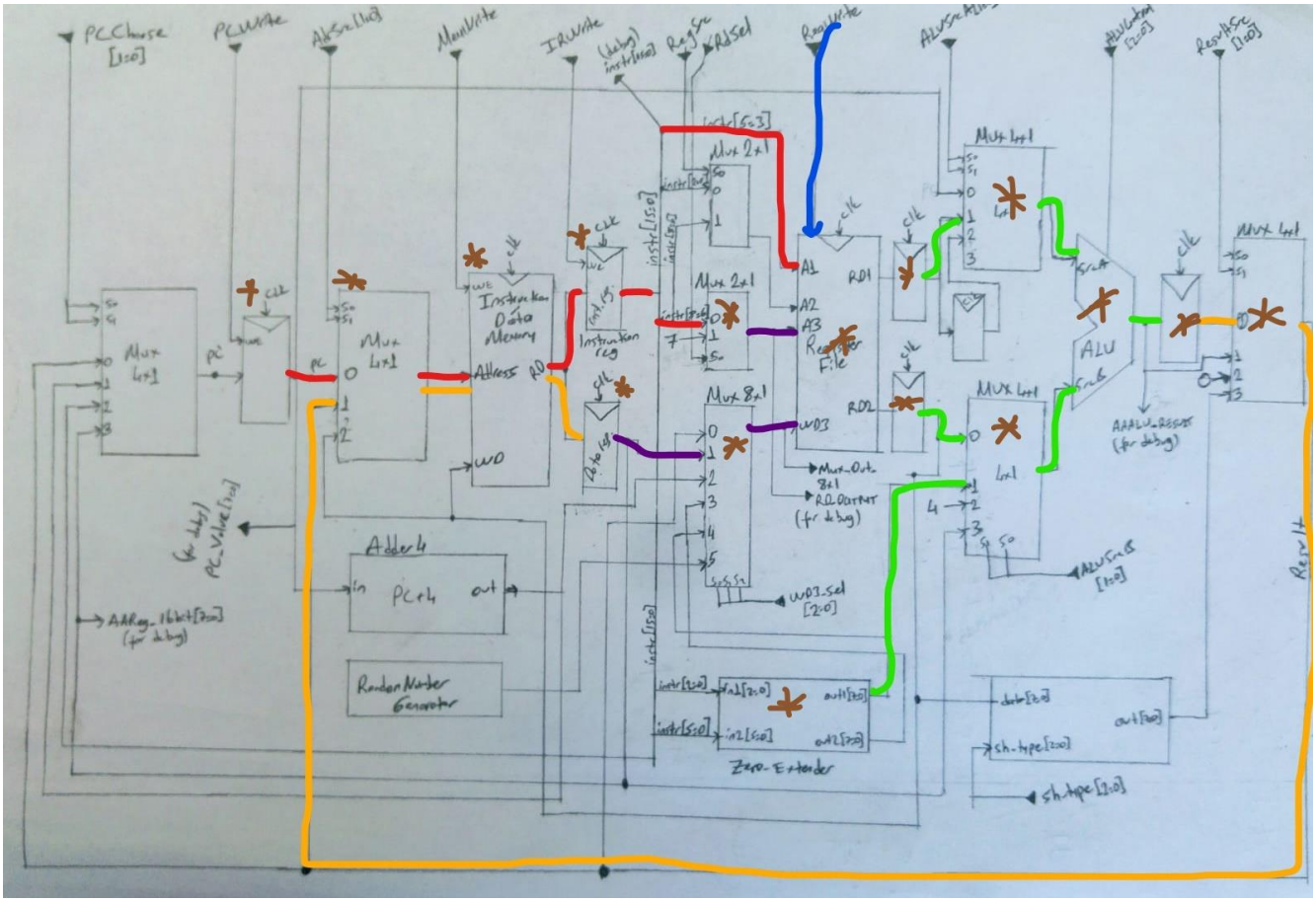


Figure 1: LDR instruction datapath for multi-cycle CPU

Control signals for LDR instruction

PCChoose[1:0] : to choose the next program counter value.

PCWrite : to write the data chosen by PCChoose control bir

AdrSrc[1:0] :to choose what the Address will be in IDM(instruction/Data Memory)

MemWrite :to write the WD data specified by Address.

IRWrite :to write the instruction to the path.

RegSrc :to choose the Rm signal to be sent to Register File A2 bits..

RdSel :to choose the destination register(Rd).

WD3_Sel[2:0] :to choose the WD3 input of the Register File.

ALUSrcA[1:0] :to choose the srcA input of the ALU.

ALUSrcB[1:0] :to choose the srcB input of the ALU.

ALUControl[2:0] : to decide the operation that will be taken place in the ALU(Add,Sub,Or…)

ResultSrc[1:0] to choose what the result will be.

Debuggin probes on the datapath:

AAReg_16bit[7:0] : to see the current value of the 16 bit Data register that is conected in front of the IDM.

PC_Value[7:0] :to see the current value of the PC.

İnstr[15:0] : to see the instruction fetched from the IDM.

Mux_Out_8x1[7:0] : to see the current value of the WD3 input of the Register Fle.

RD_OUTPUD[7:0] to observe the current value of the destination register that is connected to the A3 input of the Register File.

AAALU_RESULT[7:0] : to observe the changes in the ALU result.

REG0_OUT[7:0] : to see the value loaded to the register 0.

REG1_OUT[7:0] : to see the value loaded to the register 1.

REG2_OUT[7:0] : to see the value loaded to the register 2.

REG3_OUT[7:0] : to see the value loaded to the register 3.

REG4_OUT[7:0] : to see the value loaded to the register 4.

LR_OUT : to see the value loaded to the link register (Register 7).

b) load immediate to register(MOV)

For this operation, I didn't use execute and write-back stages. After the instruction is fetched in the very first cycle , I directly write the 6 bits immediate value to the register file since we know the Rd value and immediate value from the instruction fetched from the IDM(instruction/ Data memory).Since this instruction(MOV) uses 6 bits of the instruction fetched from the memory, Zero

extender is added to concanete the 6 bits to 8 bits to write to the register file. Therefore, no changes in the datapath is made as can be seen from figure 2.

1.cycle = Red , 2.cycle = blue

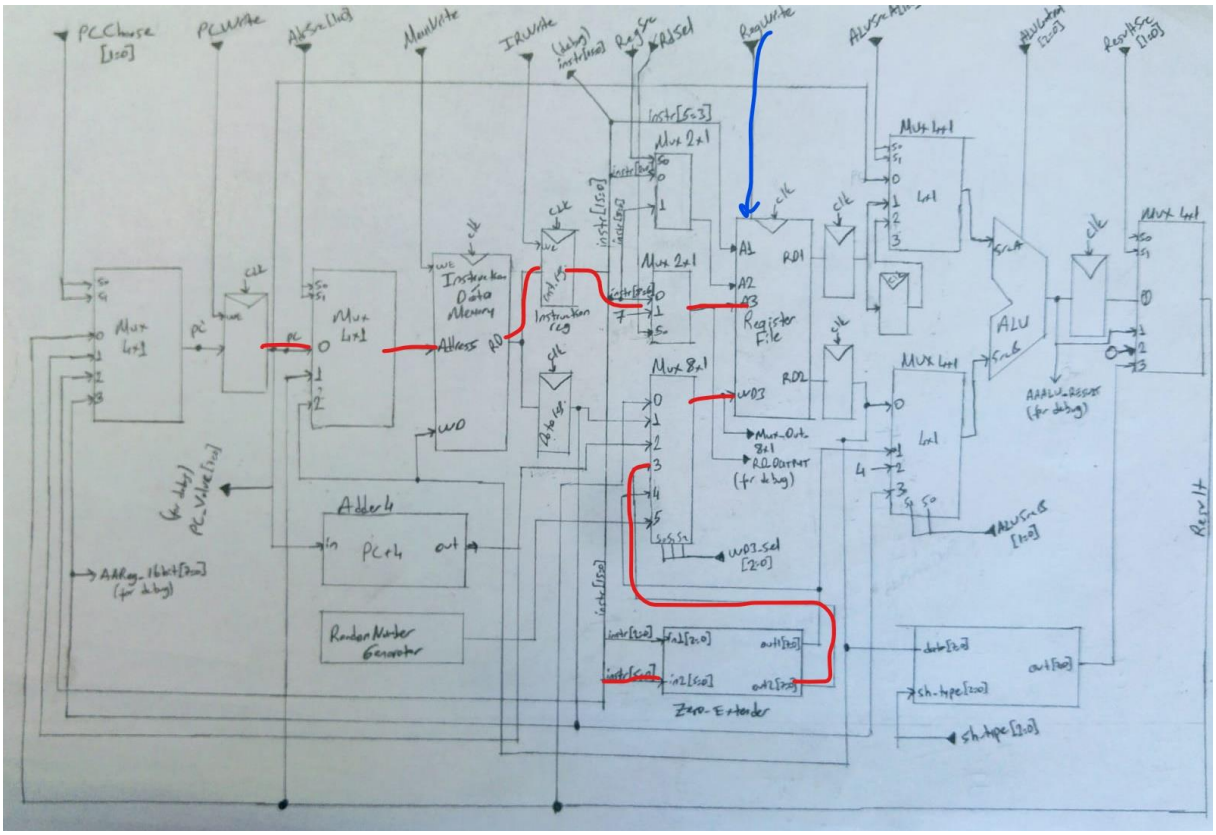


Figure 2: MOV instruction datapath for multi-cycle CPU

c) store from register to memory(STR)

To store from register to memory ,some changes need to be made in the datapath to forward the data in a correct way. Firstly, The the second input of the 2x1 Mux is connected to the Rd input (instr[8:6])since we want Rd to be forwarded to the IDM. For this purpose, the output of the register that is connected to the RD2 output of the Register File is forwarded to the WD input of the IDM. This datapath supports both STR Rd,Rn and STR Rd,Rn,#imm3 operations as can be seen from figure 3. Since we store some value to a address, firstly, a register should be filled with a value. For this purpose, as can be seen from the figure, a random number generator is added to the 8x1 MUX that is connected to the WD3 in of the Register File.

1.cycle = Red , 2.cycle = green , 3.cycle =yellow , 4.cycle = blue

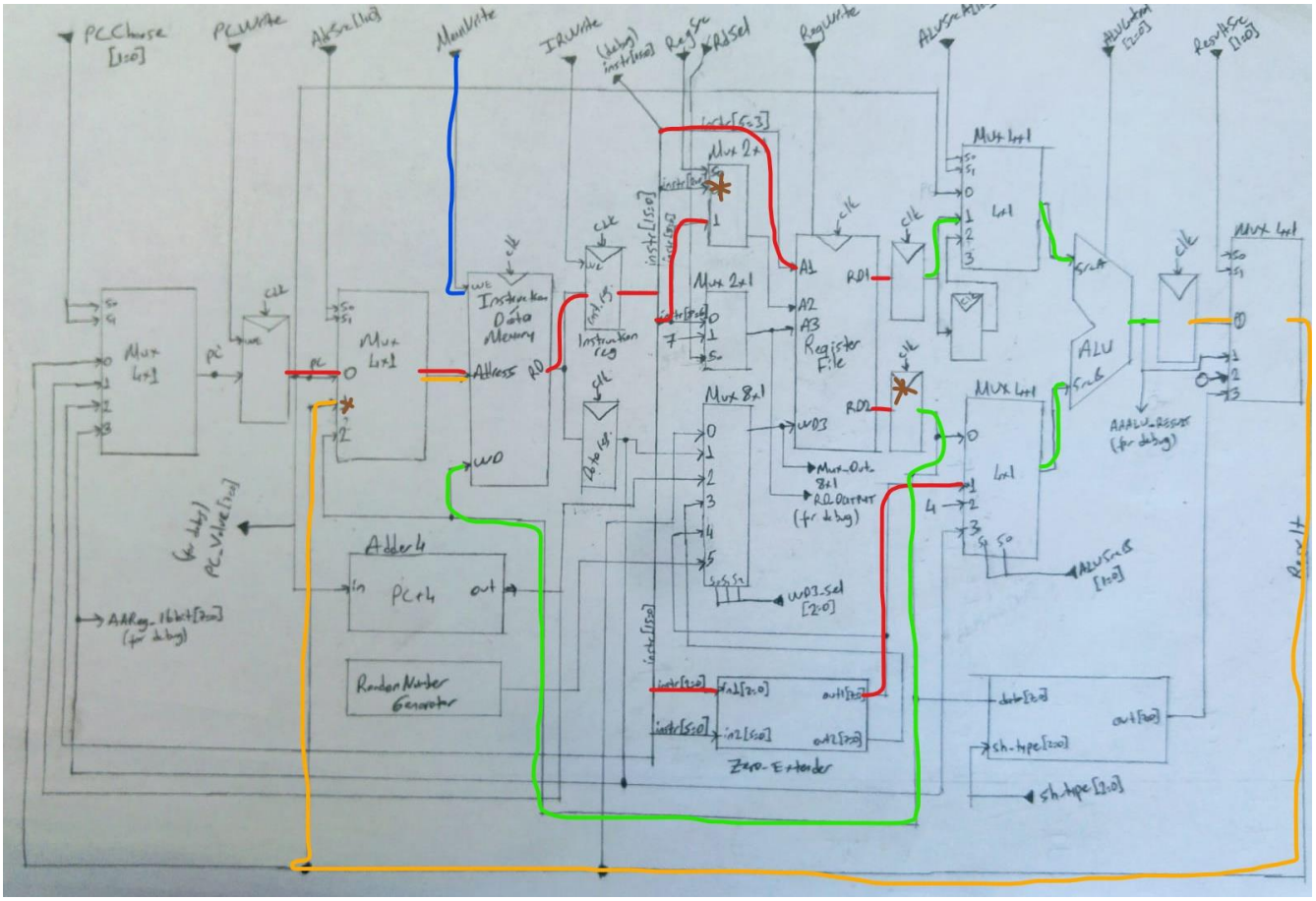


Figure 3: STR instruction datapath for multi-cycle CPU

2.2) Data Processing operations(ADD,ADDI,SUB,SUBI,AND,ORR,XOR,CLR)

For non-indirect operations (ADD,SUB,AND,ORR,XOR,CLR) our datapath already supports all feature to make this operations. Therefore, no changes are made on the datapath as can be seen from the figure 4. However, for indirect operations 1 cycle latency should be made in order to fetch indirect data from the IDM and forward to the ALU. Therefore, some changes are made on the datapath. Firstly, if we think ADDI R0,R1, [MEM], to add R1 and [MEM] we need to fetch the data addressed by MEM. For this purpose, 1 register is added in front of the register that is connected to the RD1 output of the Register file to make a 1 cycle latency. Furthermore, since we fetch data from the IDM, we need to forward the data registers output to the 4x1 MUX that is connected to the srcB input of the ALU. After making 1 cycle latency, we fetch data from memory and by giving the right control signals we can Add indirectly and subtruct indirecly.The necessary changes are shown in the figure 5. In addition to this operations, to gain from the cycle, the CLR operation is not made in the ALU ,but by connecting a constant 0 to the 4x1 MUX at the right most side. This will frovide to have 0, and this 0 will be written to the destination register directly to clear it as can be seen from the figure 4.

1.cycle = Red , 2.cycle = green , 3.cycle =yellow , 4.cycle = blue

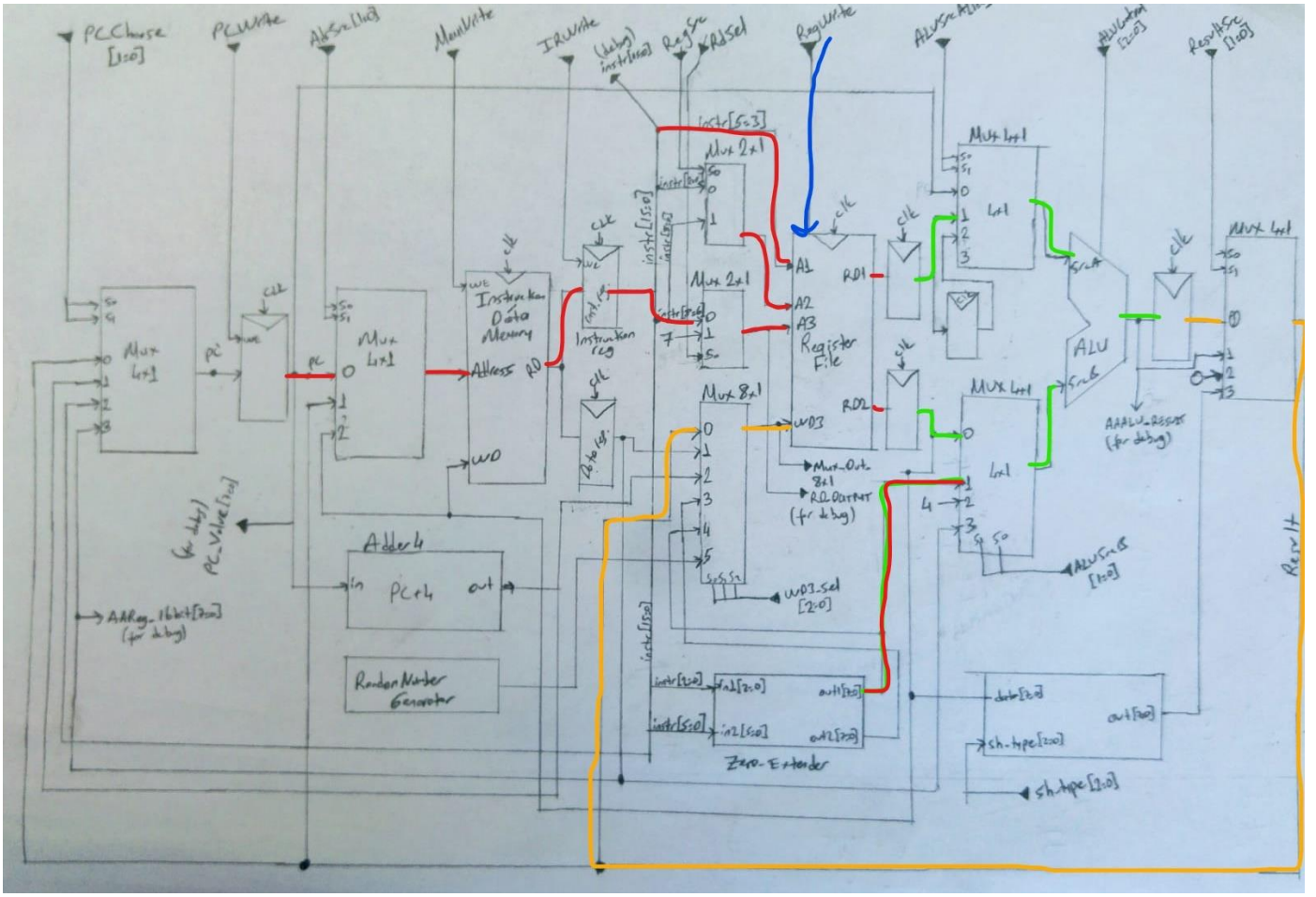


Figure 4: The datapath for the non-indirect instructions for multi-cycle CPU

1.cycle = Red , 2.cycle = green , 3.cycle =yellow , 4.cycle =purple, 5.cycle = blue

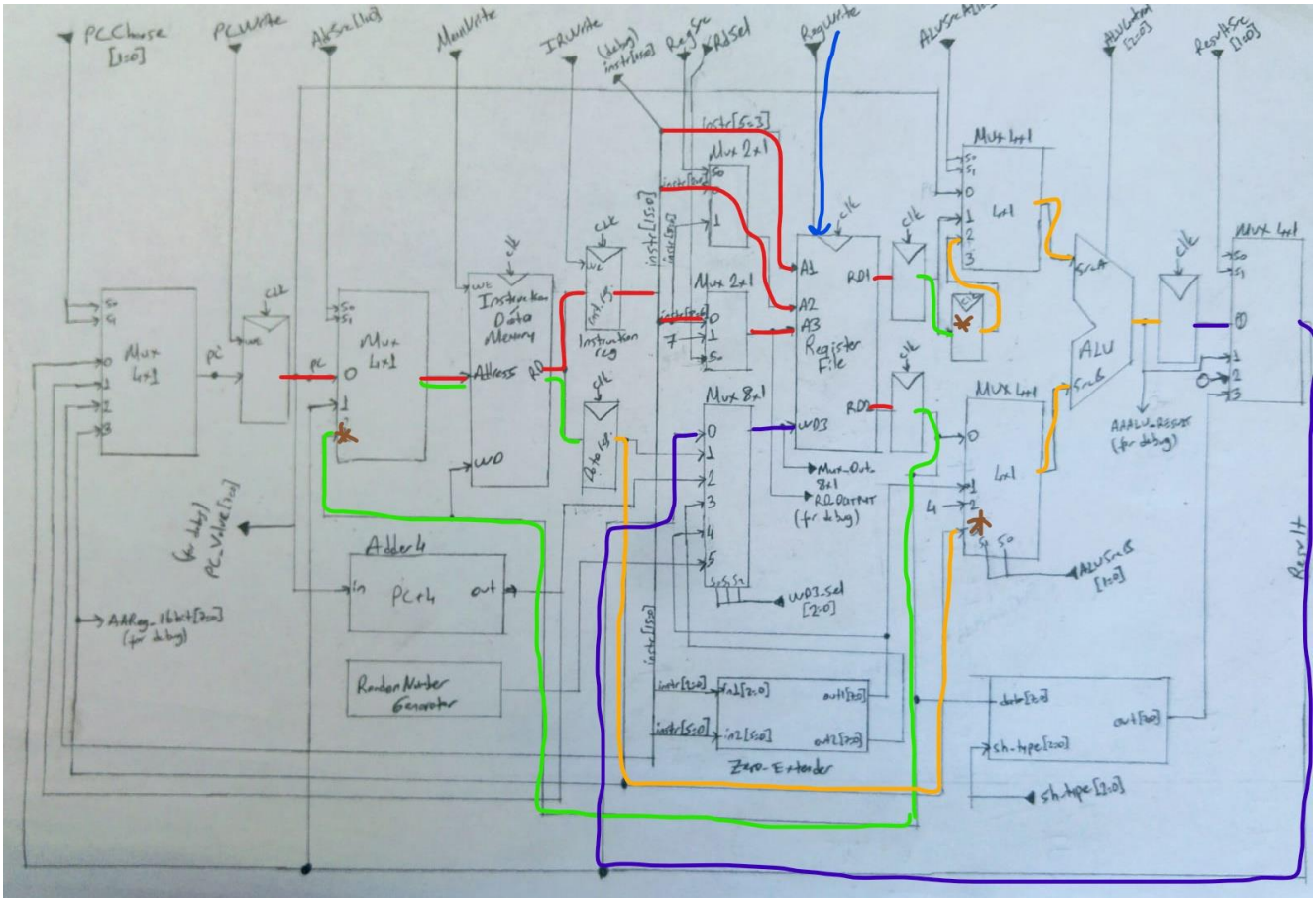


Figure 5: The datapath for the indirect instructions(ADDI,SUBI) for multi-cycle CPU

2.3) Shift Operations(RTL,RTR,SHL,ASHR,LSHR)

RTL = Rotate left ,RTR= Rotate right ,SHL= shift left

ASHR= Arithmetic shift right ,LSHR= Logical shift right

For shift operations, some changes on the datapath need to be made. Firstly, instead of making shift operations on the controller unit, I decided to make a combinational circuit that will be responsible for shift operation according to the shift type sending by the controller unit. For this purpose I added a shifter to the datapath that can be seen in bottom right in figure 6. Since we should just handle with only 1 register, the Rm registers output is

directly given to the shifter to provide data. Additionally, sh_type signal is added to the shifter. After the shifter takes the data and shift type, it directly forward the result to the Result Mux(4x1 at the right most side). After the result is forwarded to the WD3 input of the register file, in the next clock cycle the data is written to the destination register (Rd). The necessary changes can be seen on the figure 6.

1.cycle = Red , 2.cycle = green , 3.cycle = blue

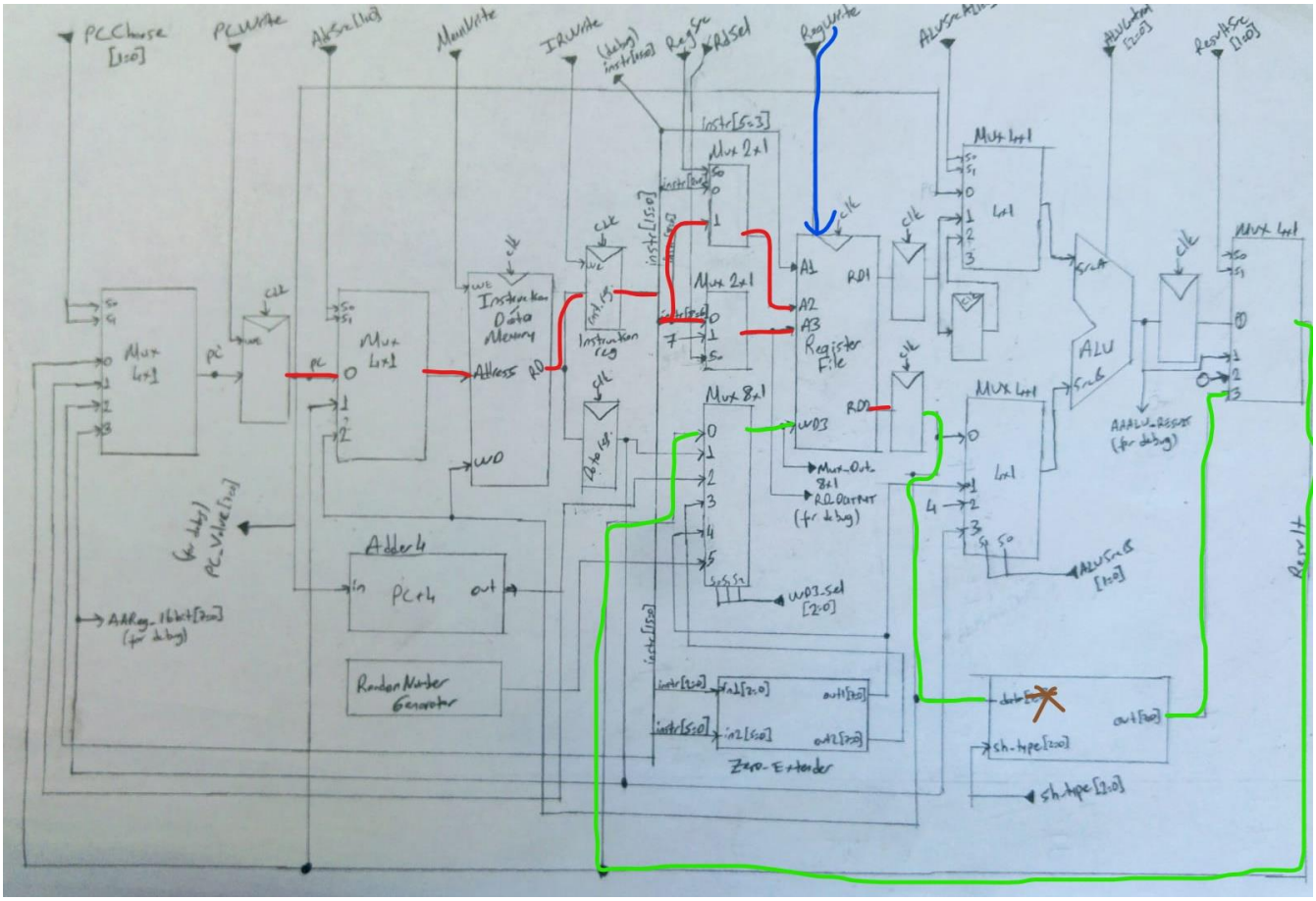


Figure 6: The datapath for shift operations for multi-cycle CPU

2.4) Branch Operations(BUN,BL,BX,BIL,BEQ,BNE,BCS,BCC)

BUN = Branch unconditionally BL= Branch with link BX=Exchange instruction

BIL=Branch indirect with link BEQ=Branch if zero BNE= Branch if not zero

BCS= Branch if carry set BCC= Branch if carry is not set.

Since this is a hypotethical Multi-Cycle CPU, I didn't calculate the Branch Target Address(BTA) with the given formula as BTA= imm12<<2+ (PC+8). The datapath is designed so that it will go directly immediate value specified in the instruction. However, in order to accomplish this, a sign extenders should be added to the datapath.

2.4.1) BUN:

For BUN operations, instr[7:0] is used for the BTA. Therefore, only changes on the datapath is adding one 4x1 MUX to the input of the PC register. This will also will help to distinguish the other branch operations. After the MUX chooses the right BTA, in the next clock cycle PC is written to the register as can be seen from figure 7.Bun operations can be thought of as BUN #19

1.cycle = Red , 2.cycle = green , 3.cycle = blue

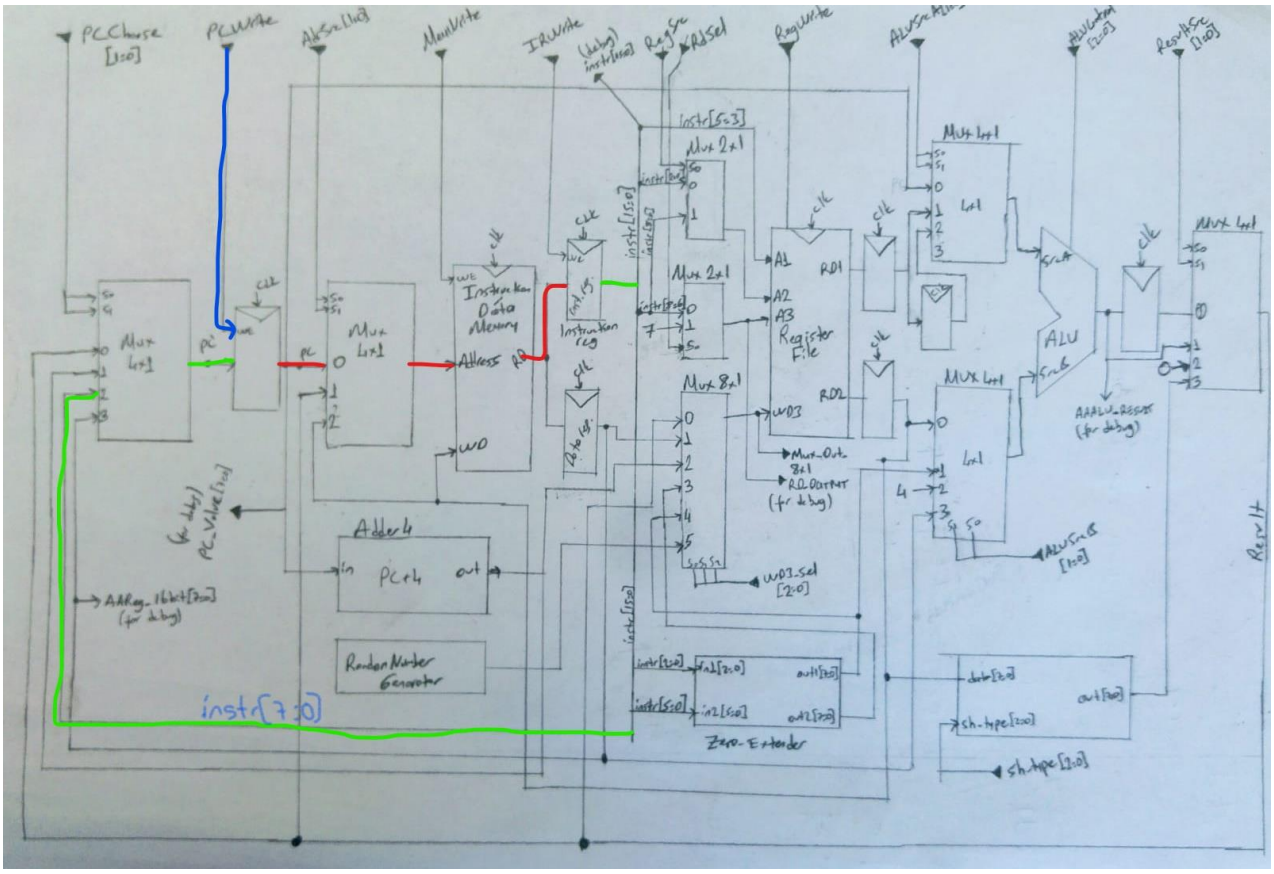


Figure 7: The datapath for Branch operation(BUN) for multi-cycle CPU

2.4.2) BL:

For BL(branch with link) operations, we need to save the PC+4 which shows the next instruction address fetched. For this purpose a link register(LR) is added in the register file to keep the address. I save the LR in the register 8. Indeed, I dont keep the address of the PC since I dont need in this design. To save the next instruction address(PC+4), I added a adder that adds 4 to the PC and gives the result to the 8x1 Mux in front of the Register File. Since I want to use all 8 bits in the ISA, I didnt add a destination register to the ISA, but an option that makes the datapath giving 7 to the Register file A3 in for the destination address(7) for LR. That gives the data path the ability of writing instructions as BL #19. The necessary changes are shown in the figure 8.

1.cycle = Red , 2.cycle = green , 3.cycle = blue

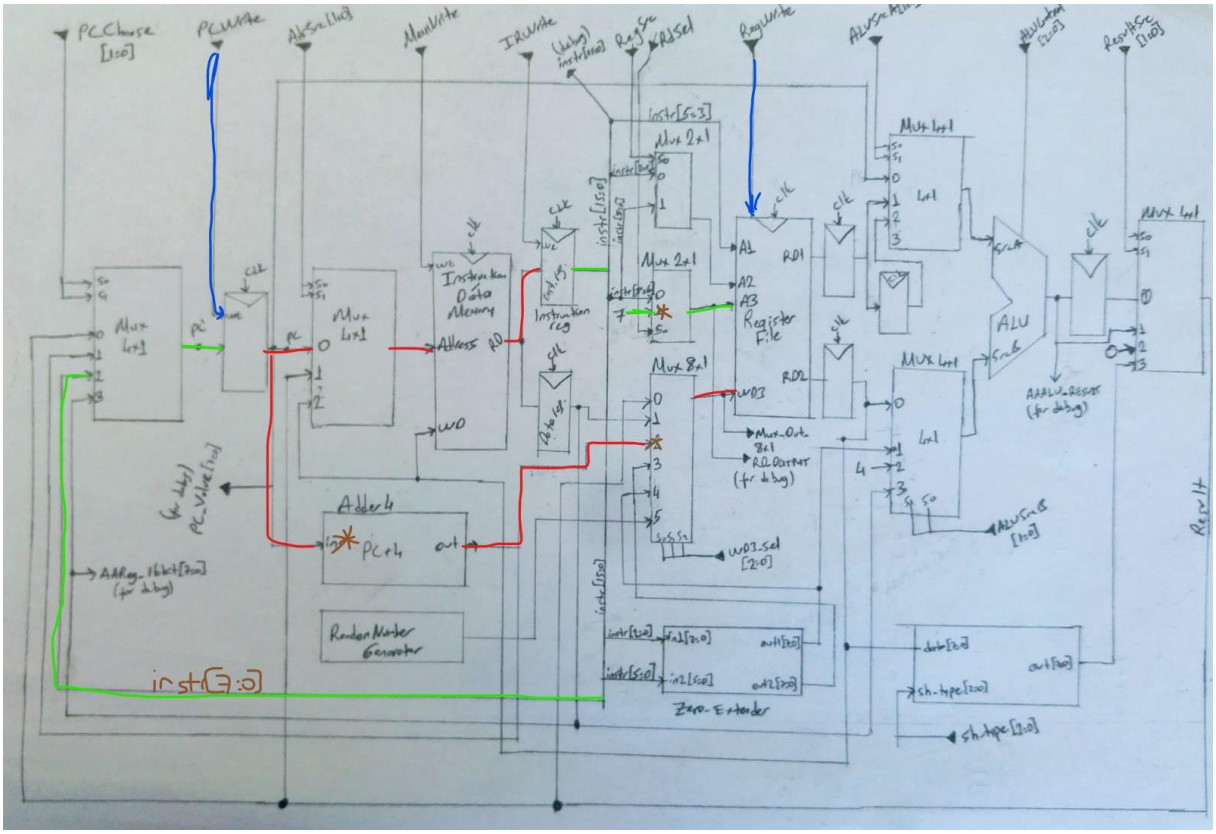


Figure 8: The datapath for Branch operation(BUN,BL) for multi-cycle CPU

2.4.3) BX:

BX operation is in order to come back from a subroutine by loading the Link register to the PC. Therefore, the only change on the datapath is forwarding the output of the RD2 register to the input of the Mux that is connected to the PC register.

2.4.4) BIL:

BIL(Branch indirect with link) will load data in the IDM that is addressed by the destination register to PC.Also, while doing that, it will save the PC+4 to the link register. To accomplish this, since we need to give the destination address value to the IDM to take the next program counter value, RD2 registers output is given to the input of the 4x1 MUX that is connected to the IDM to point the next instruction address. Additionally, after pointing the next PC address, we need to forward the address to the PC register by forwarding the output of the Data register to the 4x1 MUX that is connected to the PC register. The changes are shown with the brown stars again as can be seen from the figure 9.

1.cycle = Red , 2.cycle = green , 3.cycle =yellow , 4.cycle =blue

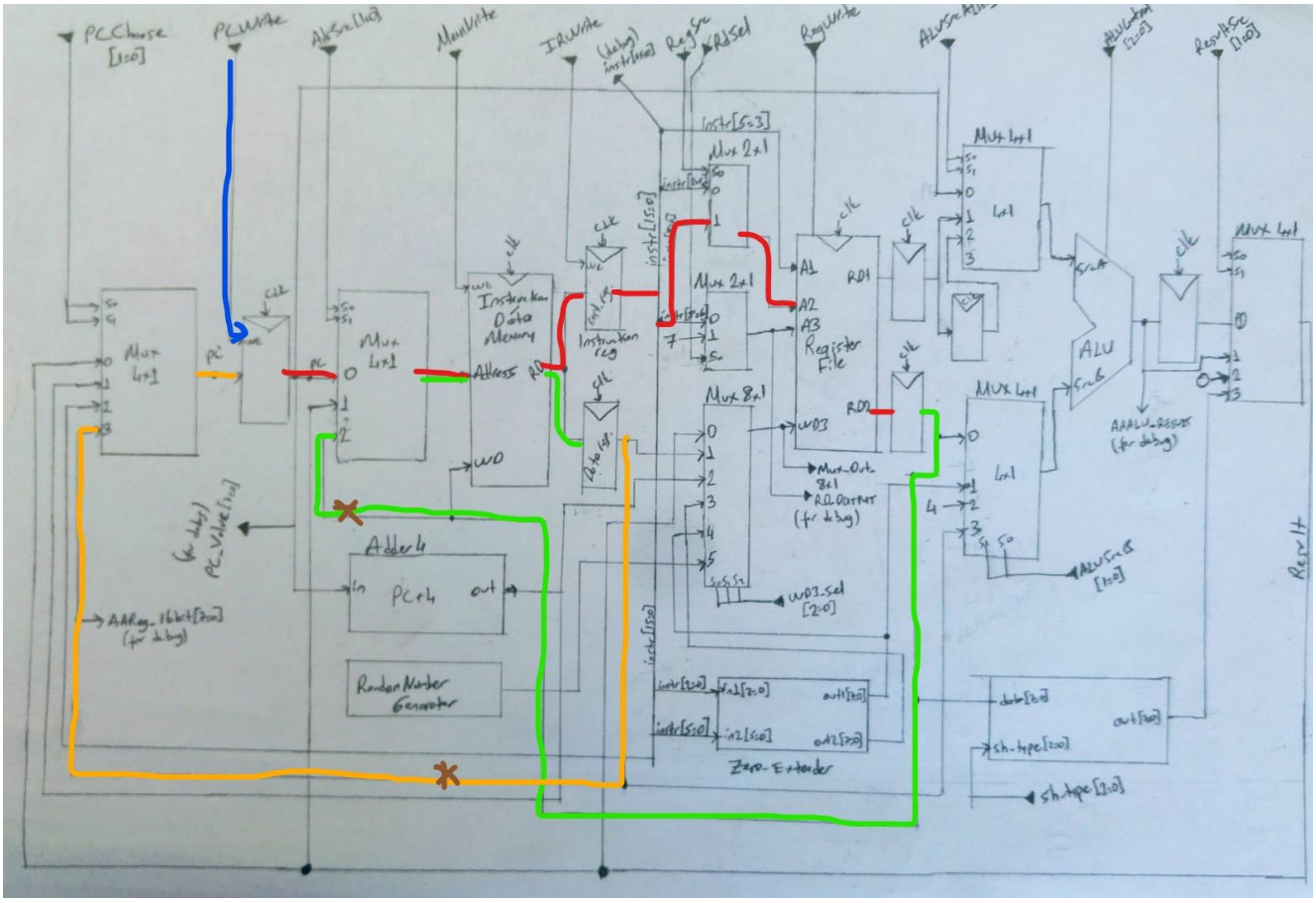


Figure 9: The datapath for Branch operation(BUN,BL,BIL) for multi-cycle CPU

2.4.5) BEQ,BNE,BCS,BCC:

The data path for this instructions is already constructed in the BUN instruction. The only difference is that the contoller unit will decide the branch will be taken or not. However, when the branch is not taken, we need to increment the PC by 4 to point the address to the next instruction in the IDM. Therefore, the Adder4 output is connected to the input of the 4x1 MUX that is connected to the PC register.The added input is in the figure 10. By the way, there are two way to increment program counter by 4. One of them is by ALU and the other one is with adder4 component. I use adder4 compenent in branch operations and ALU for other operations to gain from cycle.

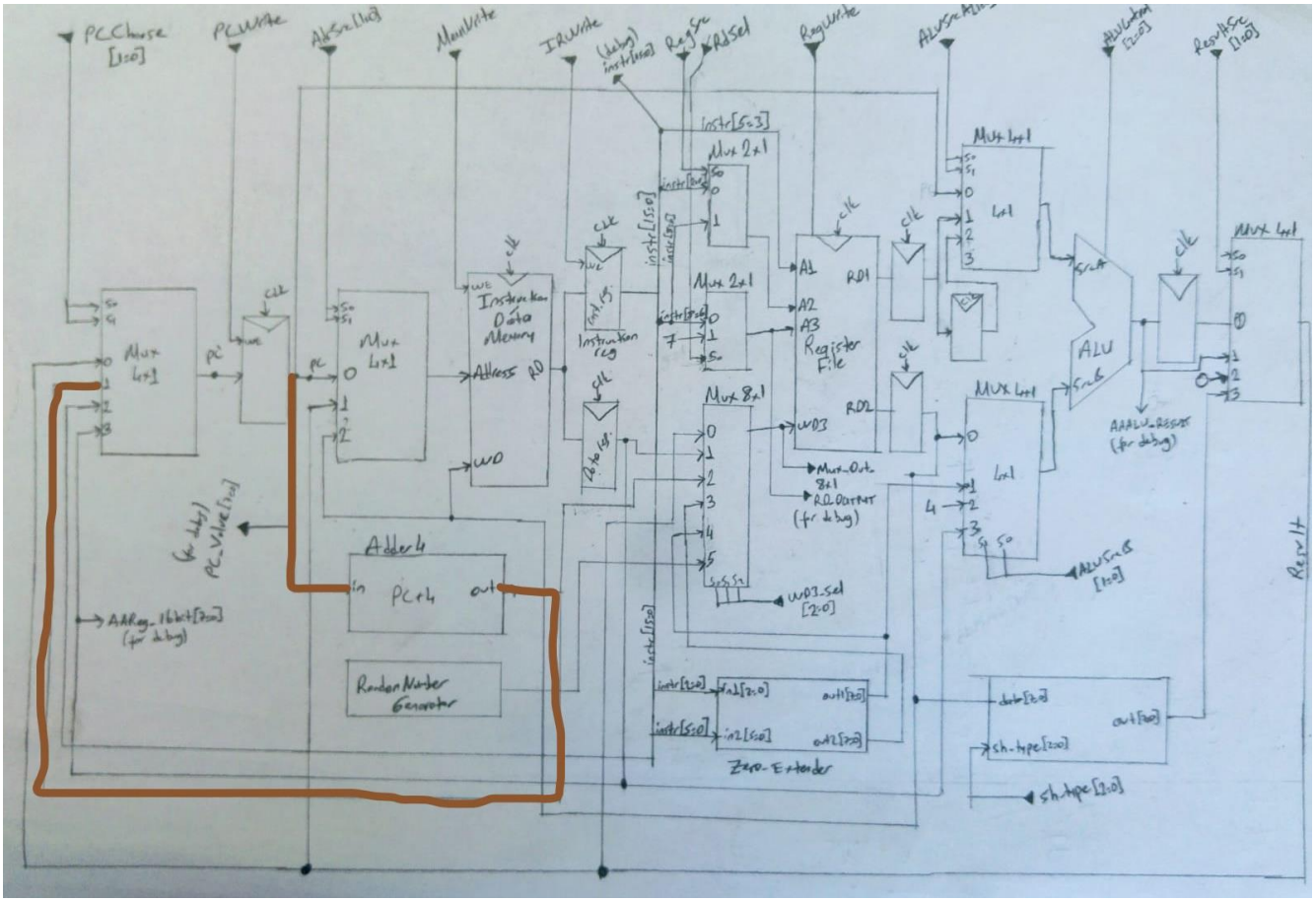


Figure 10: The datapath for Branch operation(BUN,BL,BIL, BEQ,BNE,BCS,BCC) for multi-cycle CPU

3) Validation of Operations

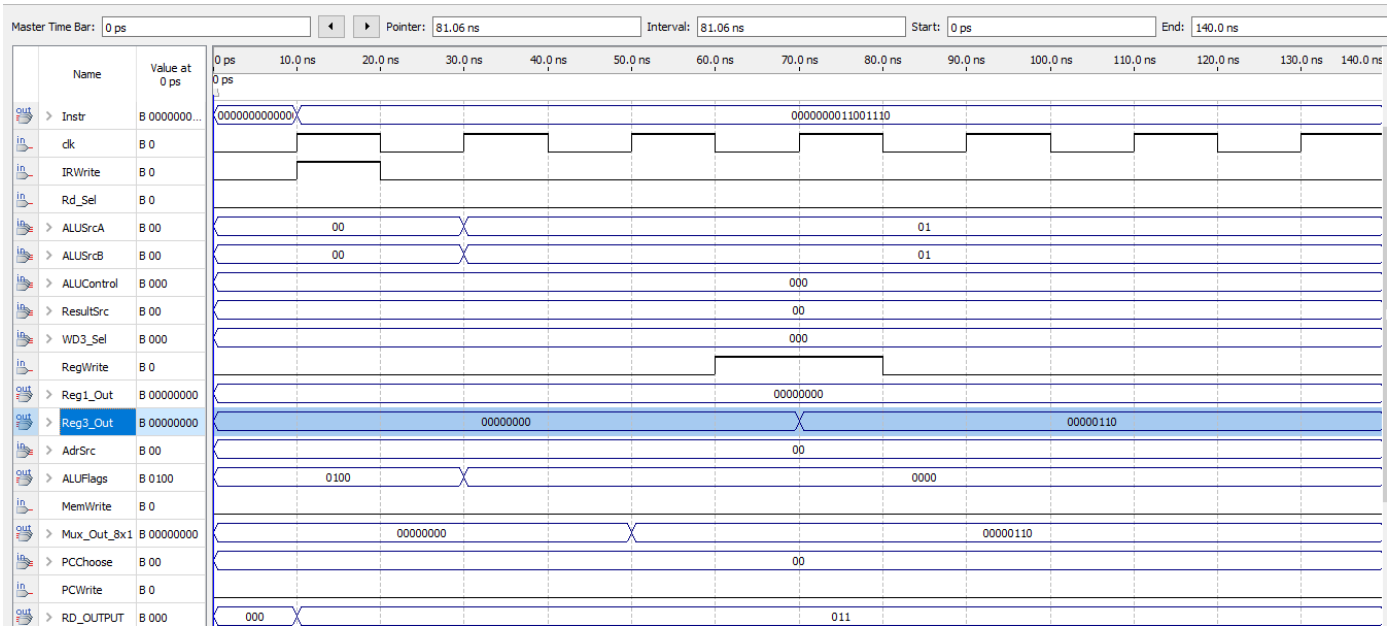
To validate my design, I used university program VWF properties of the Quartus 2 for debugging the datapath. For this purpose I create some clock signals and control signals. Later, I give them to the datapath to see the results. I will explain the results for each operation one by one.

3.1) Data Processing

The cycles taken place in order are numbered starting from 1 below. For example, in the first cycle IRWrite=1 and RDSel=0 signals are given to the datapath.

3.1.1) ADD R3,R1,#6 = (00_0_000_0_011_001_110)

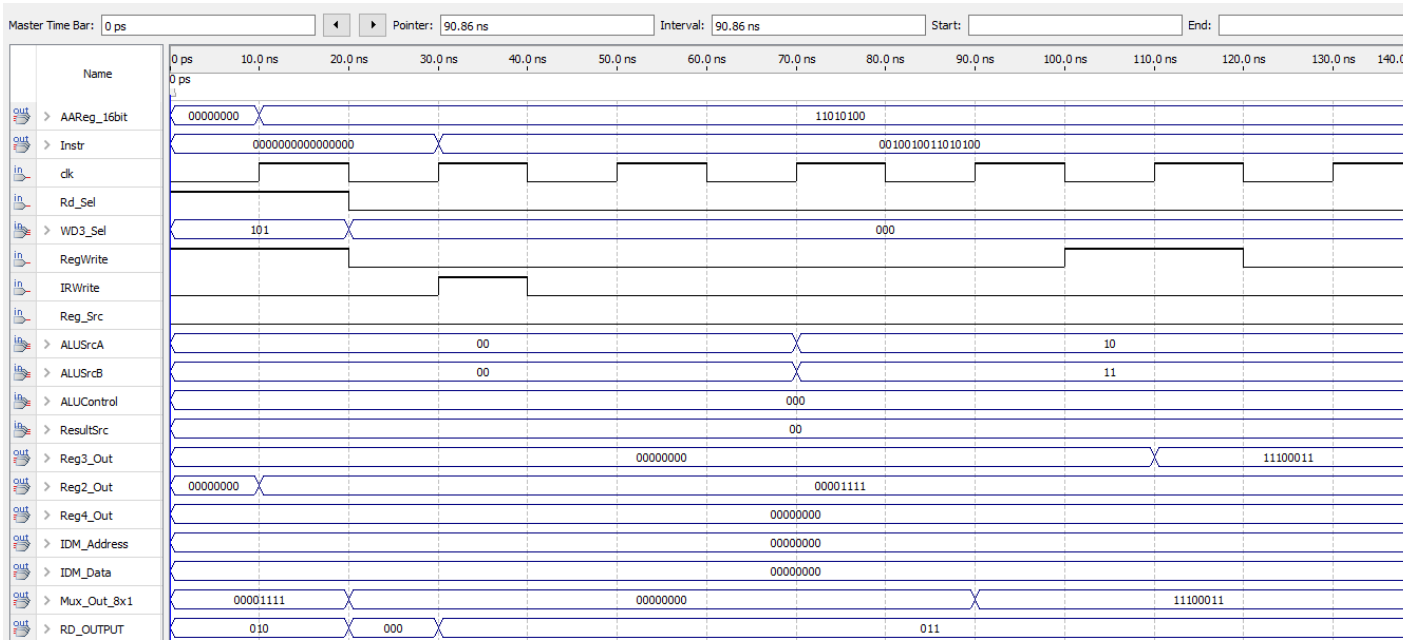
1) IRWrite = 1, Rdsel=0, RegSrc=0 (First Cycle)
2) ALUSrcA=1, ALUSrcB=1, ALUControl=000, (Second Cycle)
3) ResultSrc=0, WD3_Sel=000 (Third Cycle)
4) RegWrite=1 (Fourth Cycle)



3.1.2) ADDI R3,R2,R4 = (00_1_001_0_011_010_100)

In this instruction since at the beginning two register is loaded with 0, I decided to load 15 to R2 in the very first cycle as can be seen from the following figure. R4 points to the mem[0]. The mem[0] value can be seen as the top variable in the simulation result that is AAReg_16bit[7:0].

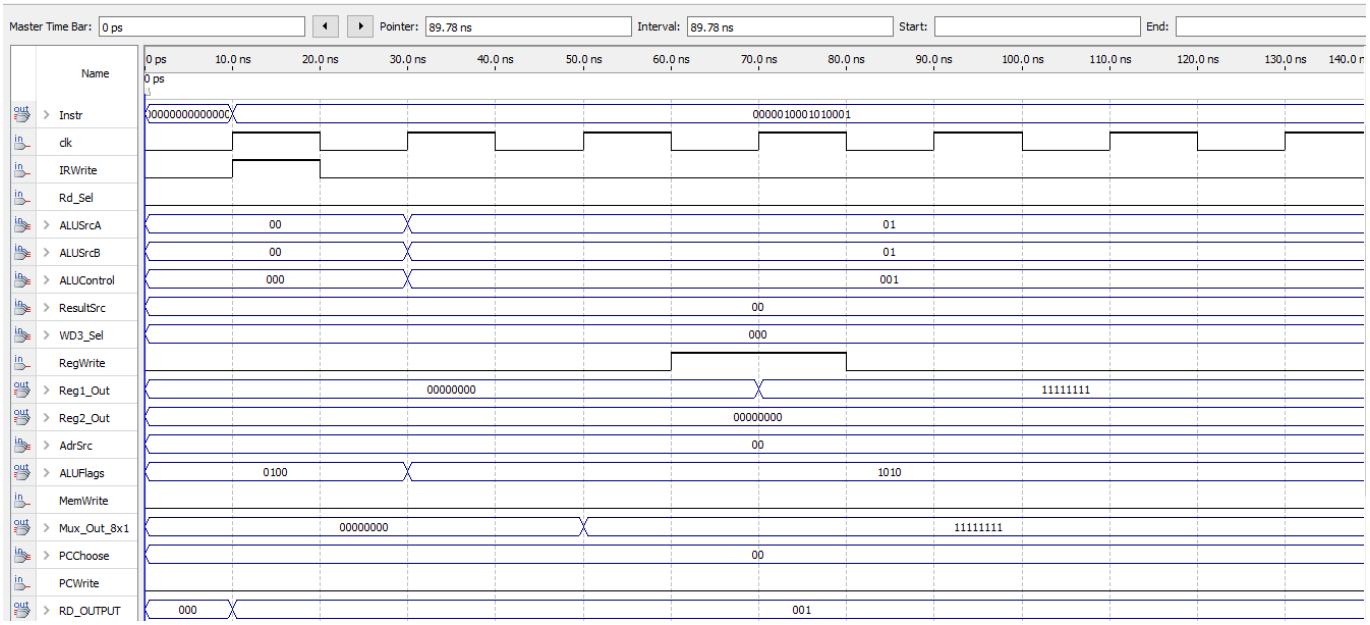
1) IRWrite=1, Rdsel=0, RegSrc=0 (First Cycle)
2) AdrSrc=2 (Second Cycle)
3) ALUSrcA=2, ALUSrcB=3, ALUControl=000, (Third Cycle)
4) ResultSrc=0, WD3_Sel=000 (Fourth Cycle)
5) RegWrite=1 (Fifth Cycle)



3.1.3) SUB R1,R2,#1 = (00_0_010_0_001_010_001)

The same procedure applies with the ADD instruction. Only ALUControl=001 changes.

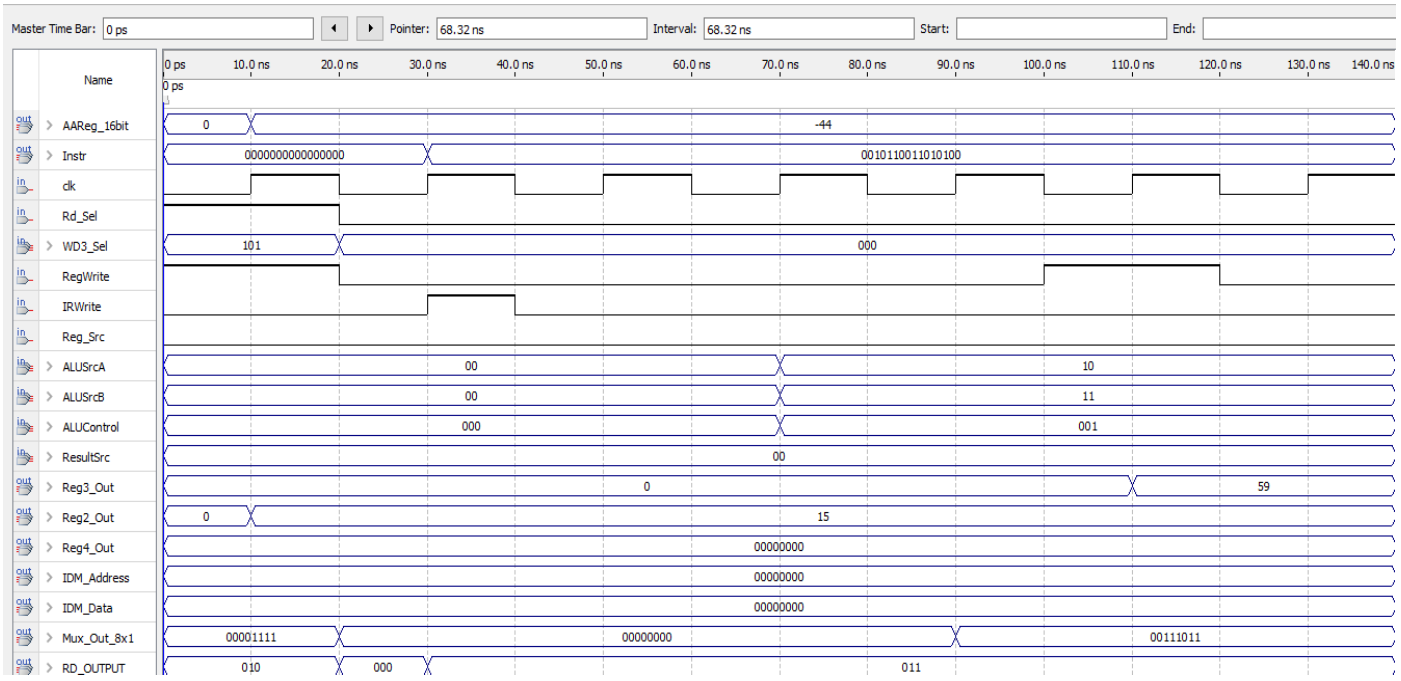
1) IRWrite = 1, Rdsel=0, RegSrc=0 (First Cycle)
2) ALUSrcA=1, ALUSrcB=1, ALUControl=001, (Second Cycle)
3) ResultSrc=0, WD3_Sel=000 (Third Cycle)
4) RegWrite=1 (Fourth Cycle)



3.1.4) SUBI R3,R2,R4 = (00_1_011_0_011_010_100)

The same procedure and control signals are applied with the ADDI instruction. Only ALUControl=001 changes.Again first cycle is for saving data to R2.

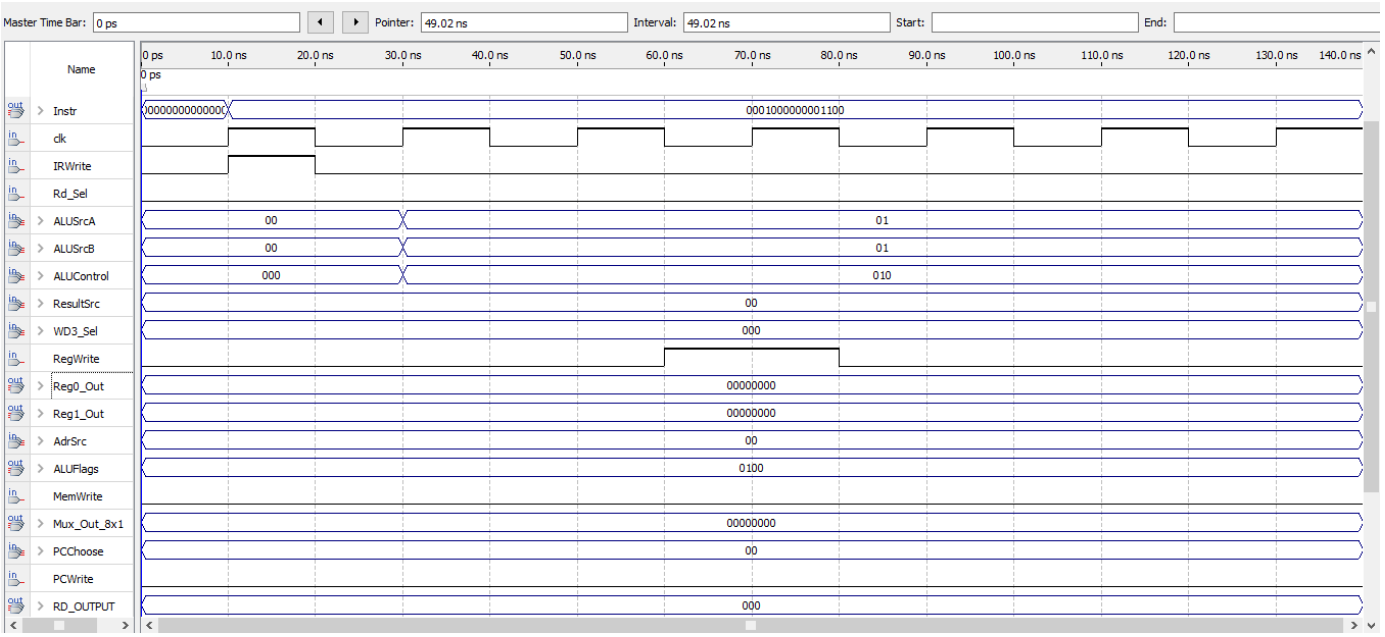
1) IRWrite=1, Rdsel=0, RegSrc=0 (First Cycle)
2) AdrSrc=2 (Second Cycle)
3) ALUSrcA=2, ALUSrcB=3, ALUControl=001, (Third Cycle)
4) ResultSrc=0, WD3_Sel=000 (Fourth Cycle)
5) RegWrite=1 (Fifth Cycle)



3.1.5) AND R0,R1,#4 = (00_0_100_0_000_001_100)

The same procedure applies with the ADD and SUB instructions. Only ALUControl=010 changes. By knowingly give 0 to registers to see if there any wrong operation on the datapath.

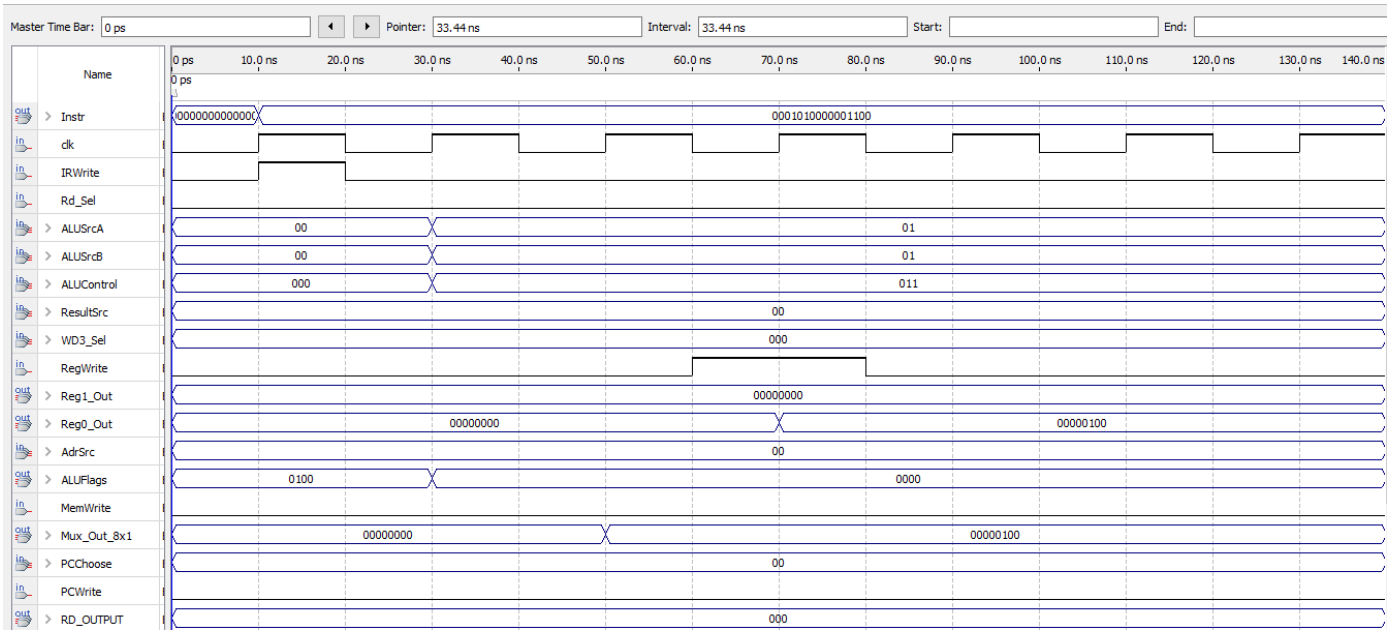
1) IRWrite = 1, Rdsel=0, RegSrc=0 (First Cycle)
2) ALUSrcA=1, ALUSrcB=1, ALUControl=010, (Second Cycle)
3) ResultSrc=0, WD3_Sel=000 (Third Cycle)
4) RegWrite=1 (Fourth Cycle)



3.1.6) ORR R0,R1,#4 = (00_0_101_0_000_001_100)

The same procedure applies with the ADD and SUB instructions. Only ALUControl=011 changes.

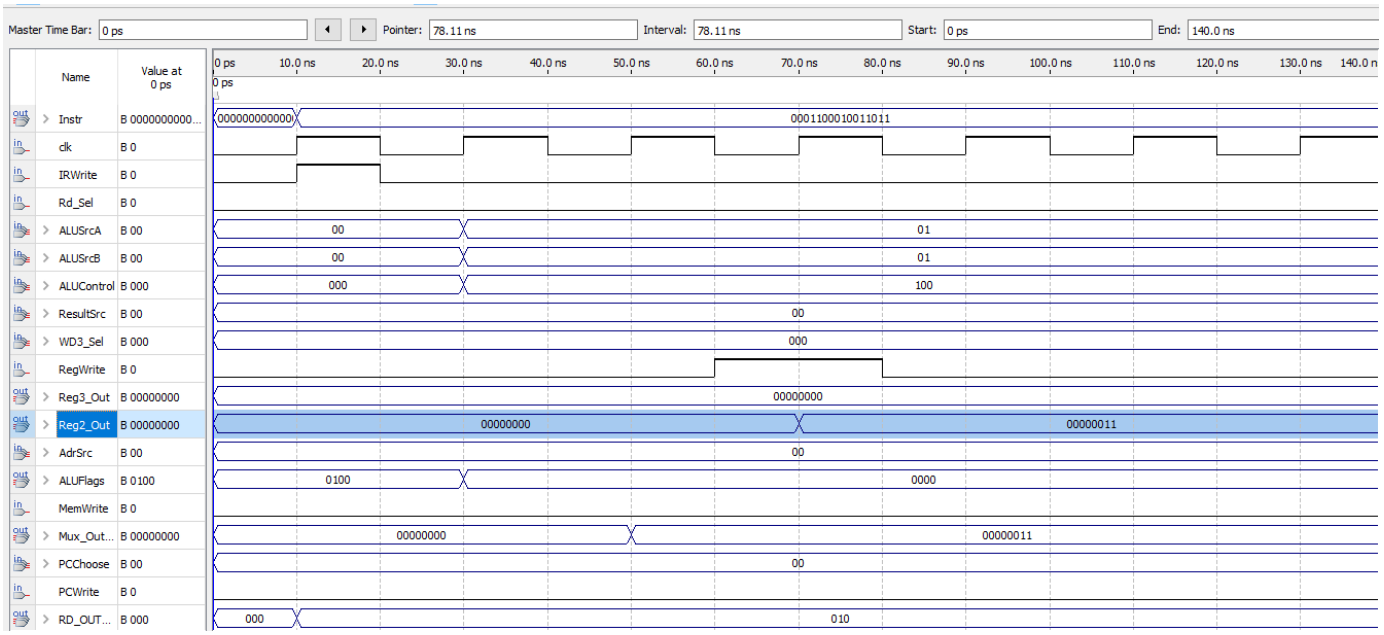
1) IRWrite = 1, Rdsel=0, RegSrc=0 (First Cycle)
2) ALUSrcA=1, ALUSrcB=1, ALUControl=011, (Second Cycle)
3) ResultSrc=0, WD3_Sel=000 (Third Cycle)
4) RegWrite=1 (Fourth Cycle)



3.1.7) XOR R2,R3,#3 = (00_0_110_0_010_011_011)

The same procedure applies with the ADD and SUB instructions. Only ALUControl=100 changes.

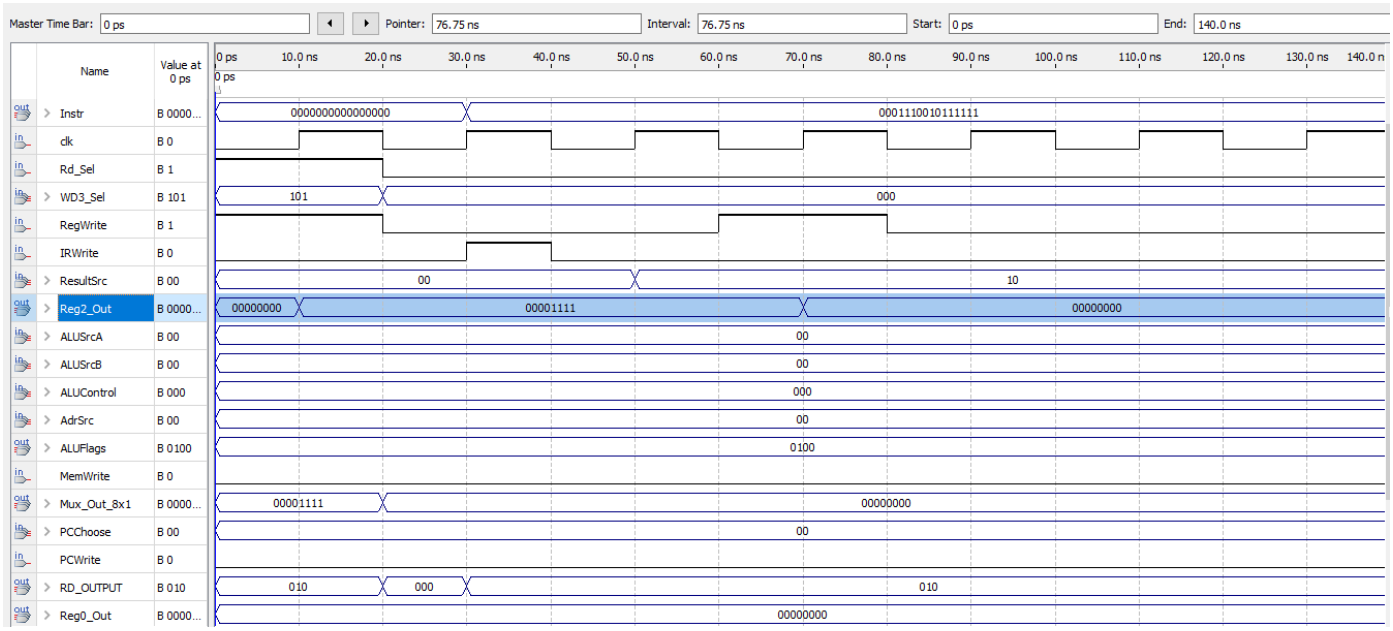
1) IRWrite = 1, Rdsel=0, RegSrc=0 (First Cycle)
2) ALUSrcA=1, ALUSrcB=1, ALUControl=100, (Second Cycle)
3) ResultSrc=0, WD3_Sel=000 (Third Cycle)
4) RegWrite=1 (Fourth Cycle)



3.1.8) CLR R2 = (00_0_111_0_010_111_111)

The same procedure but this function doesn't process in the execute stage. Directly goes to the write-back stage since we give 0 directly by the MUX.

1) IRWrite = 1, Rdsel=0, RegSrc=0 (First Cycle)
2) ResultSrc=2, WD3_Sel=000 (Second Cycle)
3) RegWrite=1 (Third Cycle)

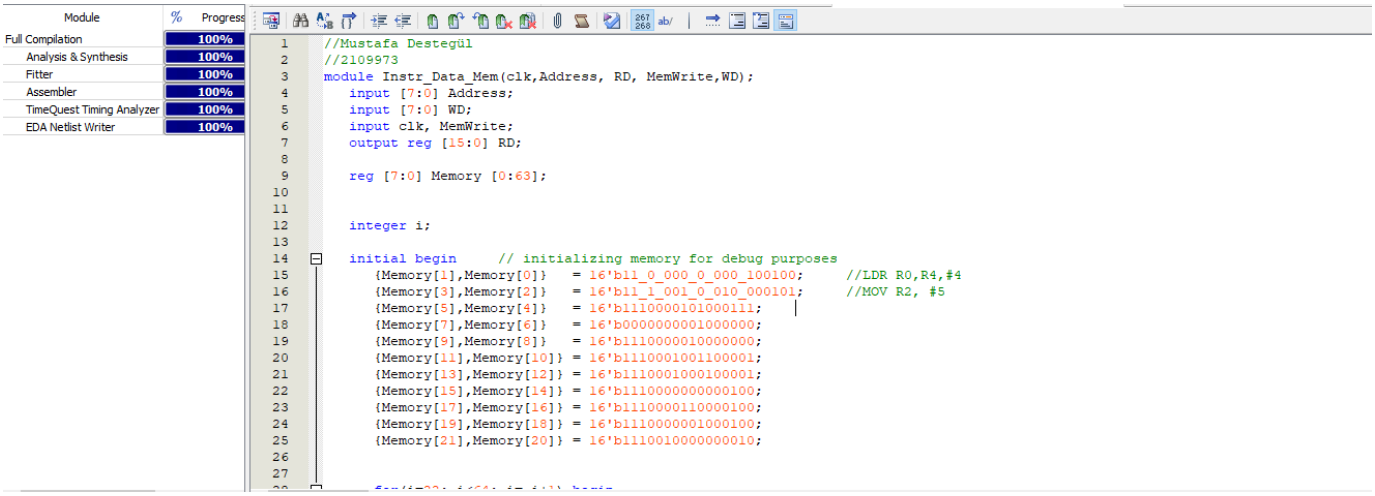


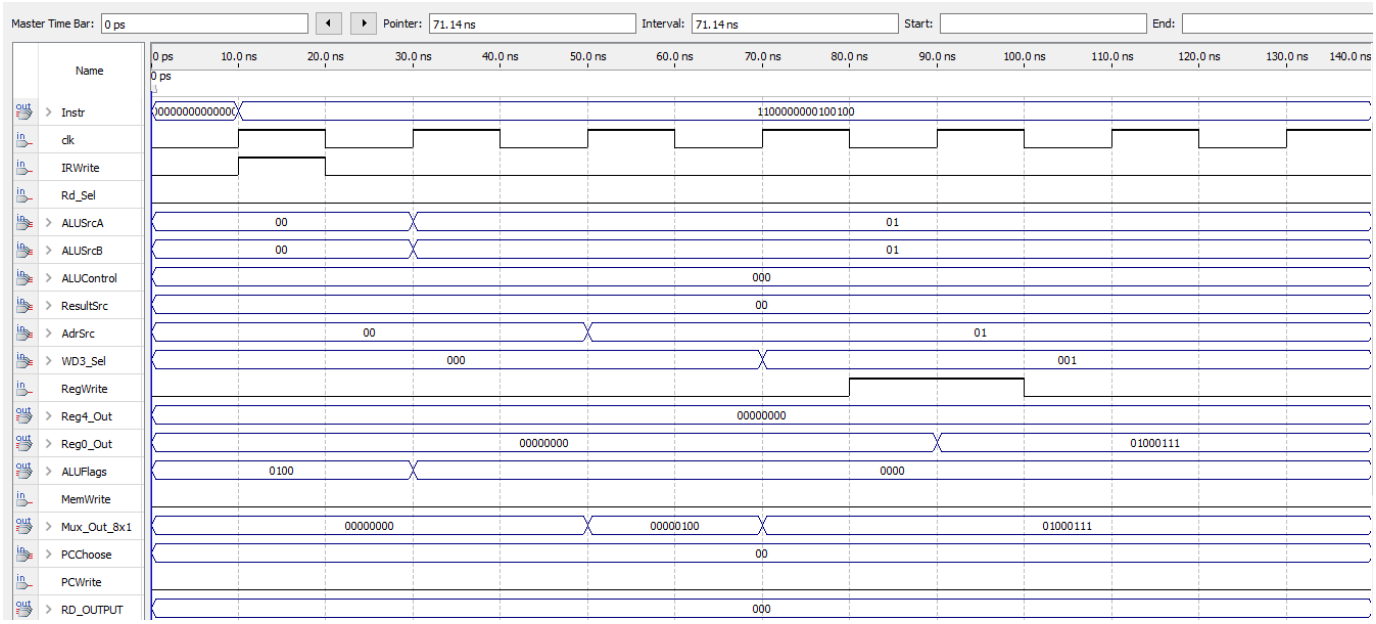
3.2) Memory Operations

3.2.1) LDR RO,R4,#4 =11_0_000_0_000_100_100

Since we take data from memory the first image shows the memory inside. The second image is the simulation result after the operation. R4=0 => R4+4= 01000111 in the memory. You can observe the output from Reg0_Out probe.

1) IRWrite = 1, Rdsel=0, RegSrc=0 (First Cycle)
2) ALUSrcA=1, ALUSrcB=1, ALUControl=000, (Second Cycle)
3) ResultSrc=0, AdrSrc=1 (Third Cycle)
4) WD3_Sel=001 (Fourth Cycle)
5) RegWrite=1 (Fifth Cycle)

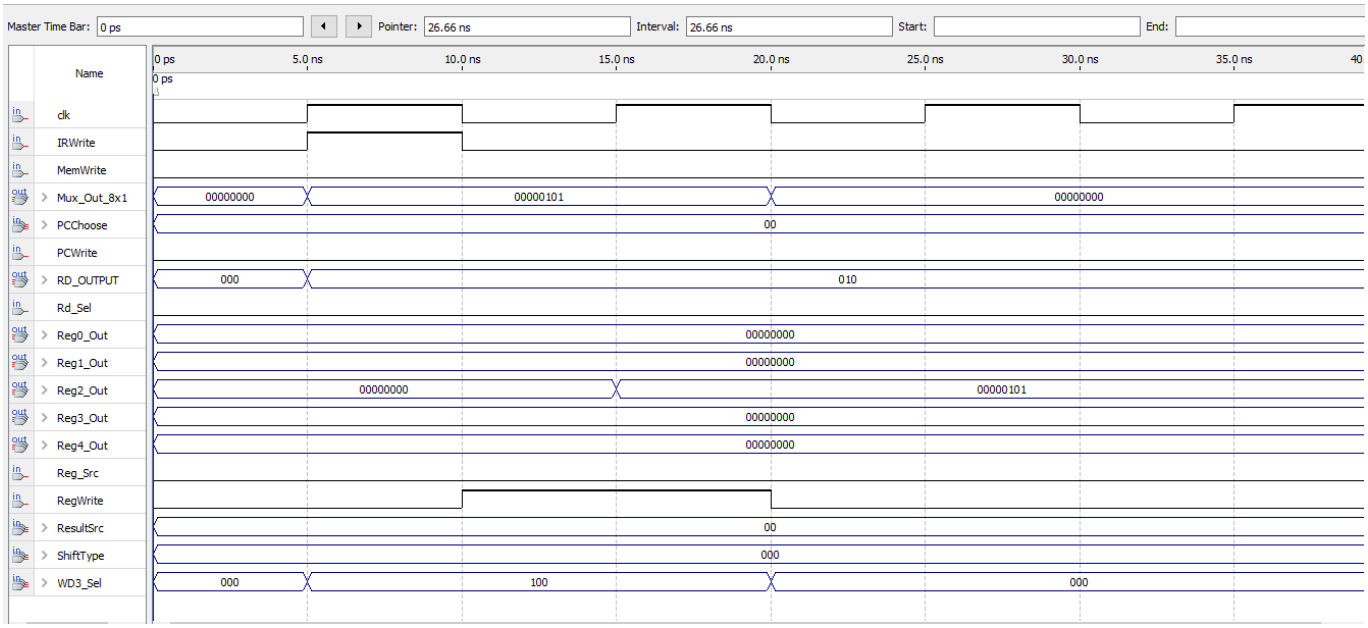




3.2.2) MOV R2,#5 =11_1_001_0_010_000101

The MOV instruction doesnt proceed execute and write-back stages. It directly saves the immediate value to the destination register. You can observe the result from Reg2_0ut

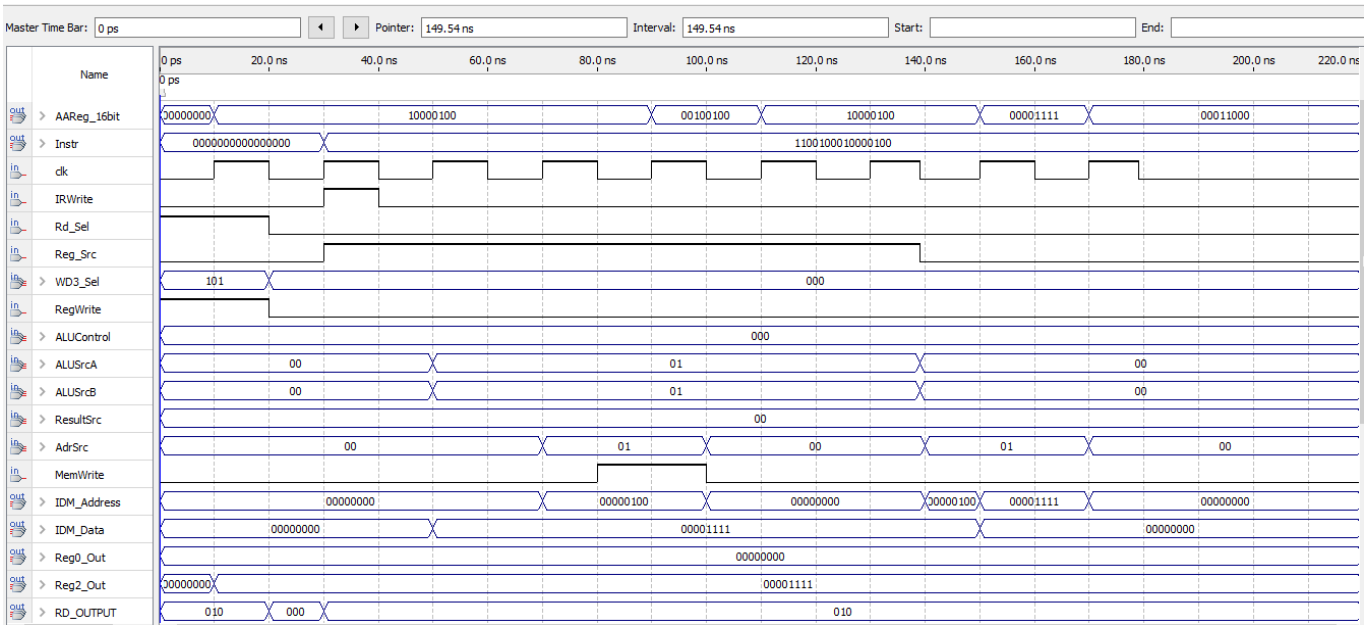
1) IRWrite = 1, Rdsel=0, RegSrc=0 (First Cycle)
2) WD3_Sel=011, RegWrite=1 (Second Cycle)



3.2.3) STR R2,[R0,#4] =11_0_010_0_010_000_100

In the very first cycle R2 is loaded by 15 as you can see to store to the memory. Additionally ,AAReg_16 bit shows the current value written to the IDM. However, since the IDM register reading happens when the address changes 2 cycle is needed to see the output. Instead, one can observe the correct operation by IDM_Data[7:0] probe. When MemWrite signal is applied, IDM_Address= 4 and data is 15.

1) IRWrite = 1, Rdsel=0, RegSrc=1 (First Cycle)
2) ALUSrcA=1, ALUSrcB=1, ALUControl=000, (Second Cycle)
3) ResultSrc=0, AdrSrc=1 (Third Cycle)
4) MemWrite=1 (Fourth Cycle)



3.3) Shift Operations

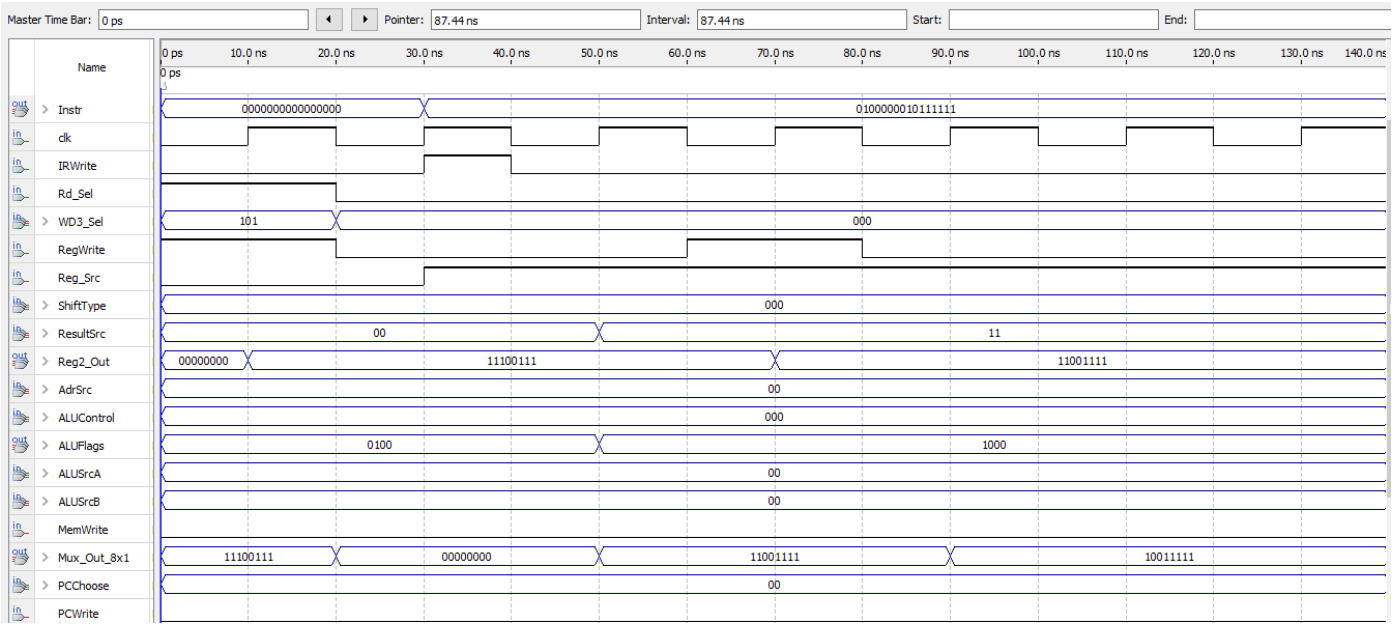
All shift operations are made in the same manner.After the fetch cycle, shift type and data is sent to the combinational shifter module. Only sh_type changes.For the shift operations, since we deal with only 1 register, 1 cycle at the beginning is used to load a number to the R2 register by the help of random number generator module.Additionally, since only instr[12:10] changes between the instructions, to gain from the time, for 1 instruction that is saved to the memary is used for all other shift operations.However, you can celary see the changes at the results. Additionally, R2_Out[7:0] probe shows the changes on the register 2.

Sh_type	value
RTL	000
RTR	001
0SHL	010
ASHR	011
LSHR	100

3.3.1) RTL R2 =01_0_000_0_010_000_000

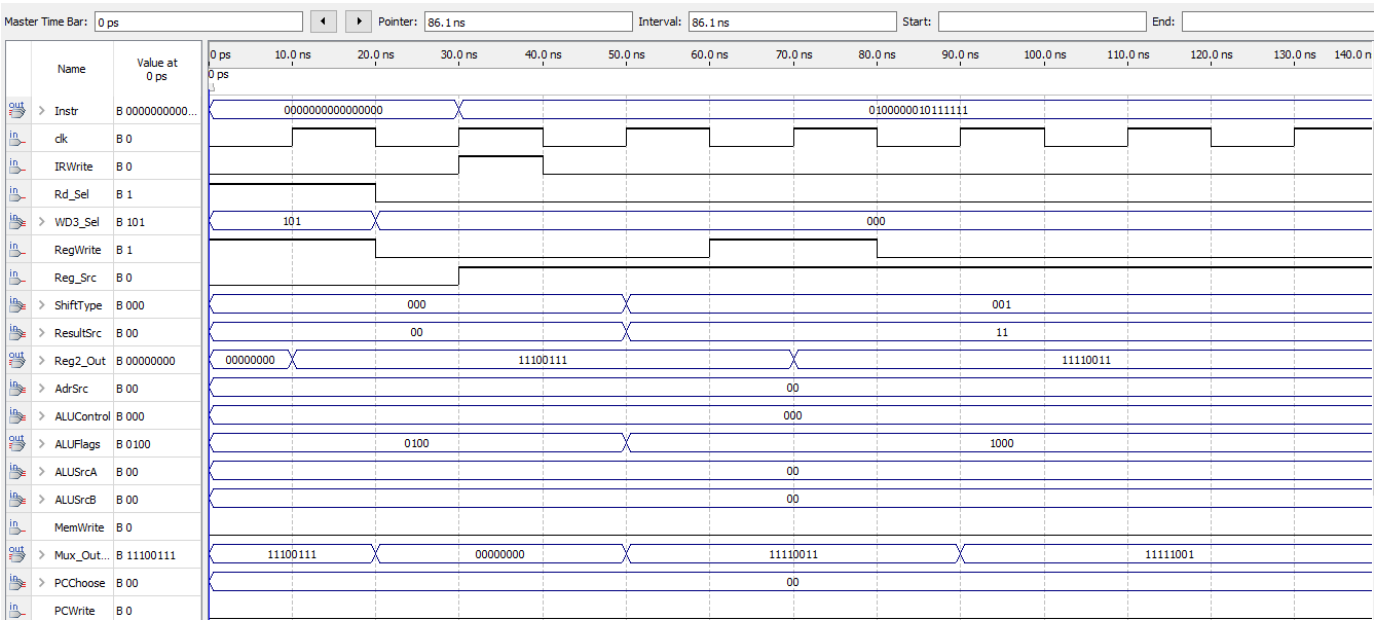
Again, First cycle is for saving data to R2 register. For all shift operations.

1) IRWrite = 1, Rdsel=0, RegSrc=1 (First Cycle)
2) Sh_type=000, ResultSrc=11, Rdsel=0,WD3_Sel=0 (Second Cycle)
3) RegWrite=1 (third Cycle)



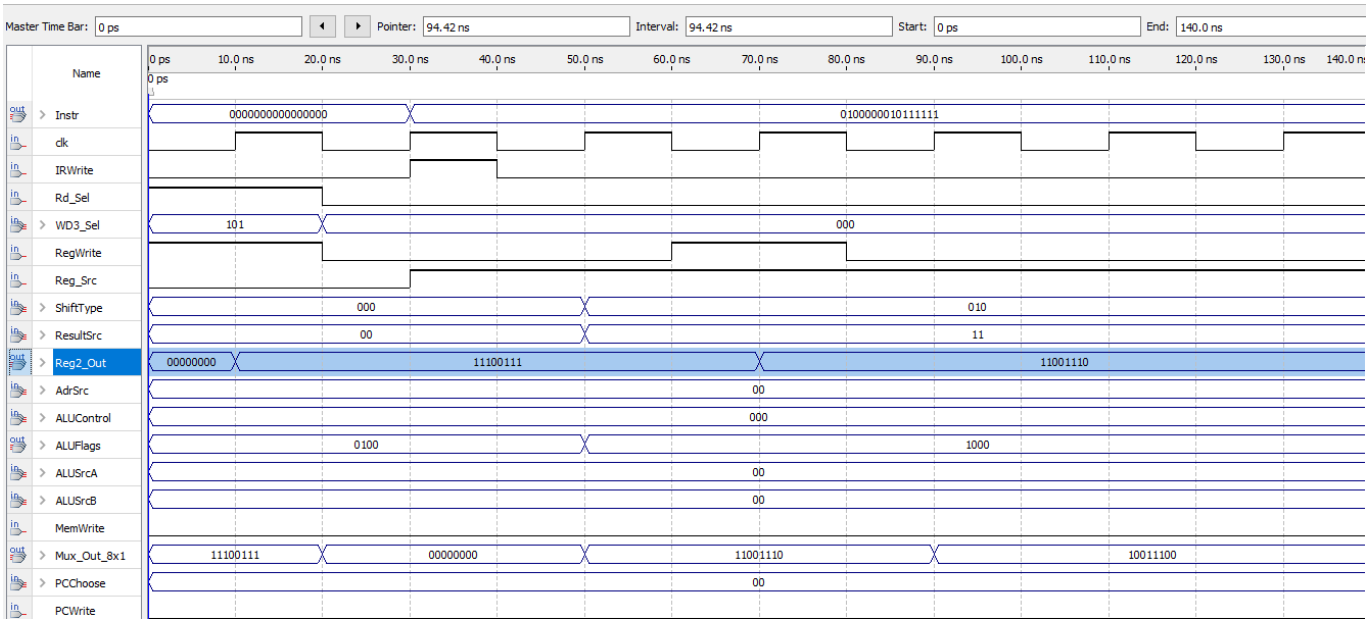
3.3.2) RTR R2 =01_0_001_0_010_000_000

1) IRWrite = 1, Rdsel=0, RegSrc=1 (First Cycle)
2) Sh_type=001, ResultSrc=11, Rdsel=0,WD3_Sel=0 (Second Cycle)
3) RegWrite=1 (third Cycle)



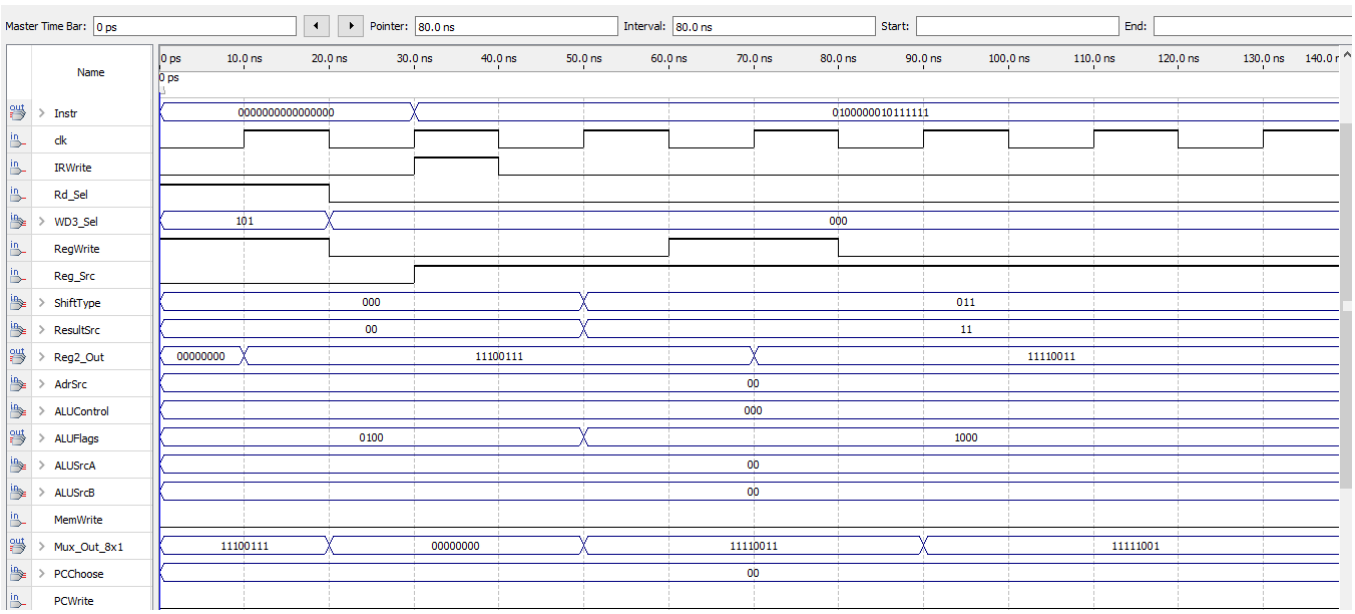
3.3.3) SHL R2 =01_0_010_0_010_000_000

1) IRWrite = 1, Rdsel=0, RegSrc=1 (First Cycle)
2) Sh_type=010, ResultSrc=11, Rdsel=0,WD3_Sel=0 (Second Cycle)
3) RegWrite=1 (third Cycle)



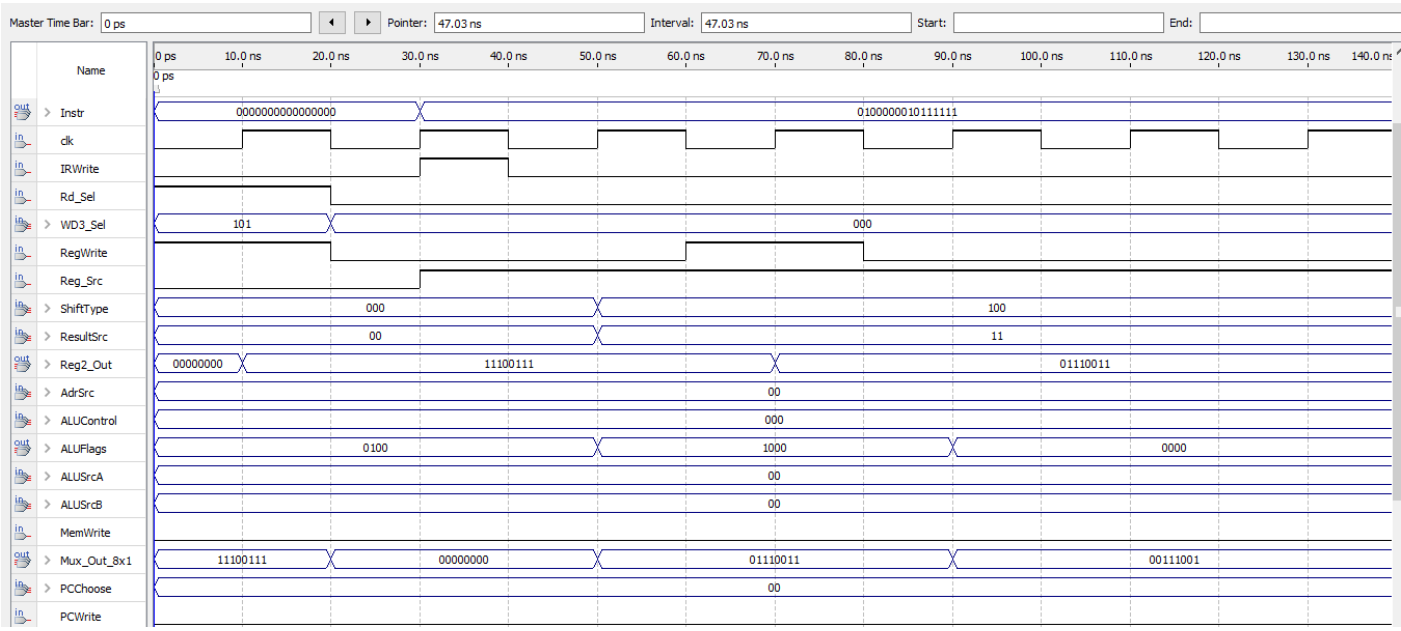
3.3.4) ASHR R2 =01_0_011_0_010_000_000

1) IRWrite = 1, Rdsel=0, RegSrc=1 (First Cycle)
2) Sh_type=001, ResultSrc=11, Rdsel=0,WD3_Sel=0 (Second Cycle)
3) RegWrite=1 (third Cycle)



3.3.5) LSHR R2 =01_0_100_0_010_000_000

1) IRWrite = 1, Rdsel=0, RegSrc=1 (First Cycle)
2) Sh_type=100, ResultSrc=11, Rdsel=0,WD3_Sel=0 (Second Cycle)
3) RegWrite=1 (third Cycle)

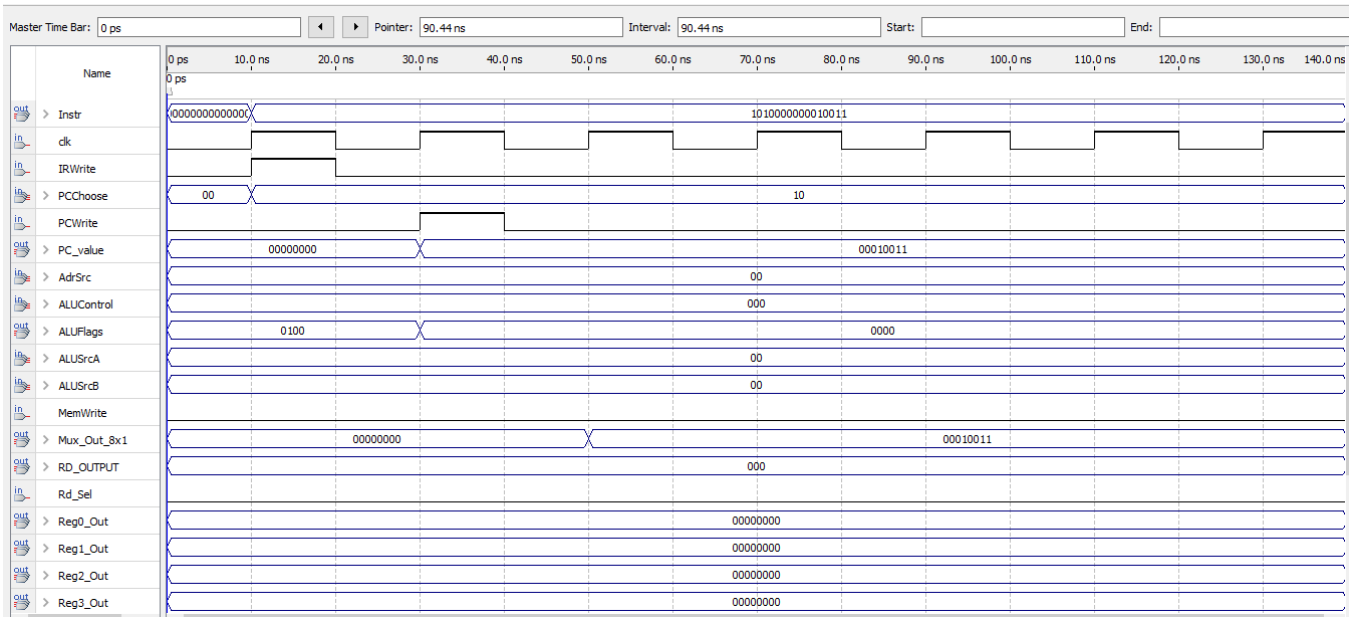


3.4 Branch Operations

3.4.1) BUN #19 =10_1_000_0_000010011

For BUN operation the program counter is directly loaded with 19. You can observe the changes in the PC value by observing PC_Value[7:0] probe.

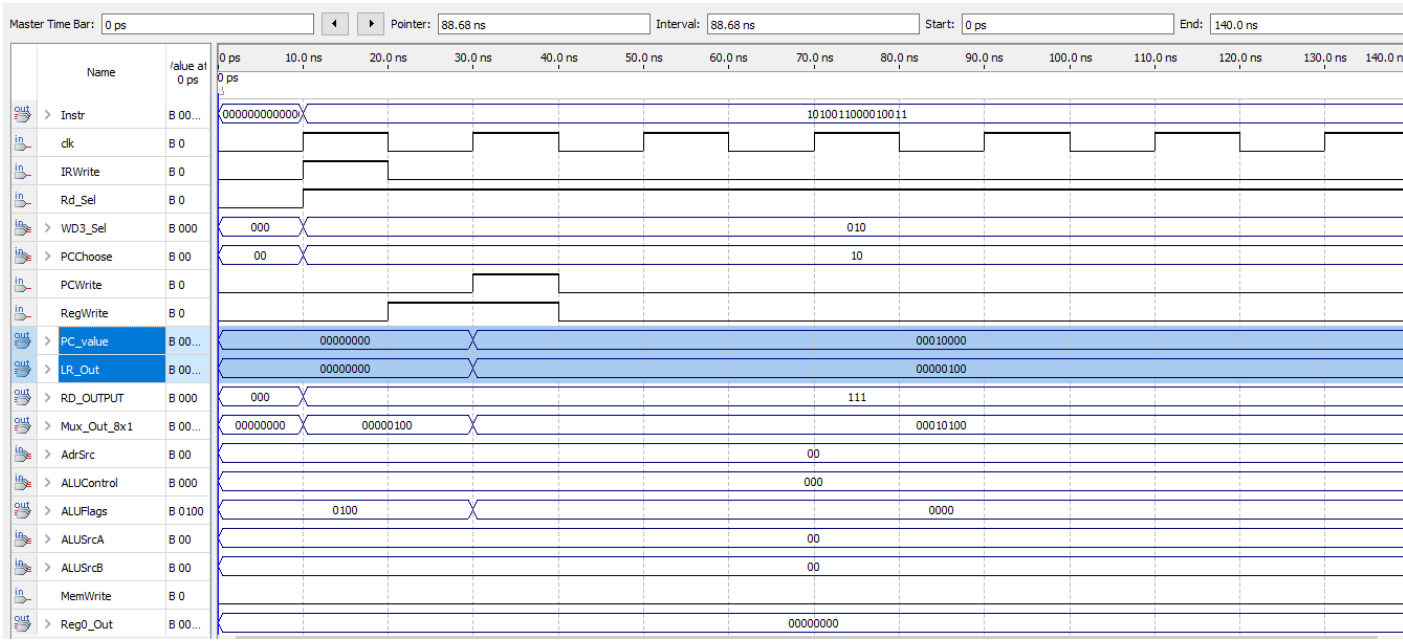
1) IRWrite = 1, (First Cycle)
2) PC_Choose=2, PCWrite=1 (Second Cycle)



3.4.2) BL #19 =10_1_001_1_000010011

For BUN operation the program counter is directly loaded with 19. You can observe the changes in the PC value by observing PC_Value[7:0] probe. The Link Register changes can be observed on the LR_Out[7:0] probe. Since at the beginning PC=0, 4 is loaded to the LR.

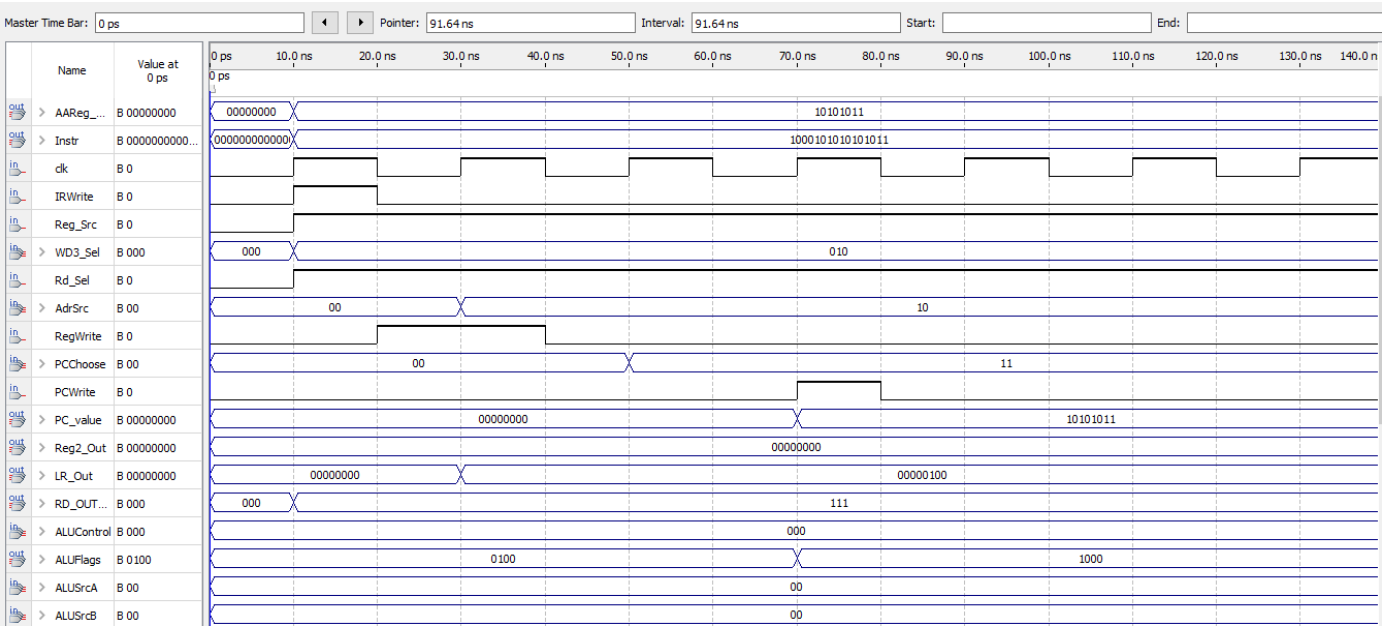
1) IRWrite = 1, RdSel=1, WD3_Sel=2 (First Cycle)
2) RegWrite=1, PC_Choose=2,PCWrite=1 (Second Cycle)



3.4.3) BIL R2 =10_0_010_1_010_101011

For BIL operation the program counter is loaded with the address in the IDM that is pointed by the R2. You can observe the changes in the PC value by observing PC_Value[7:0] probe. The Link Register changes can be observed on the LR_Out[7:0] probe. Since at the beginning PC=0, 4 is loaded to the LR. Additionally, since R2=0 at the beginning , the Link register is loaded with the instr[7:0]. Mem[0]= instr[7:0]

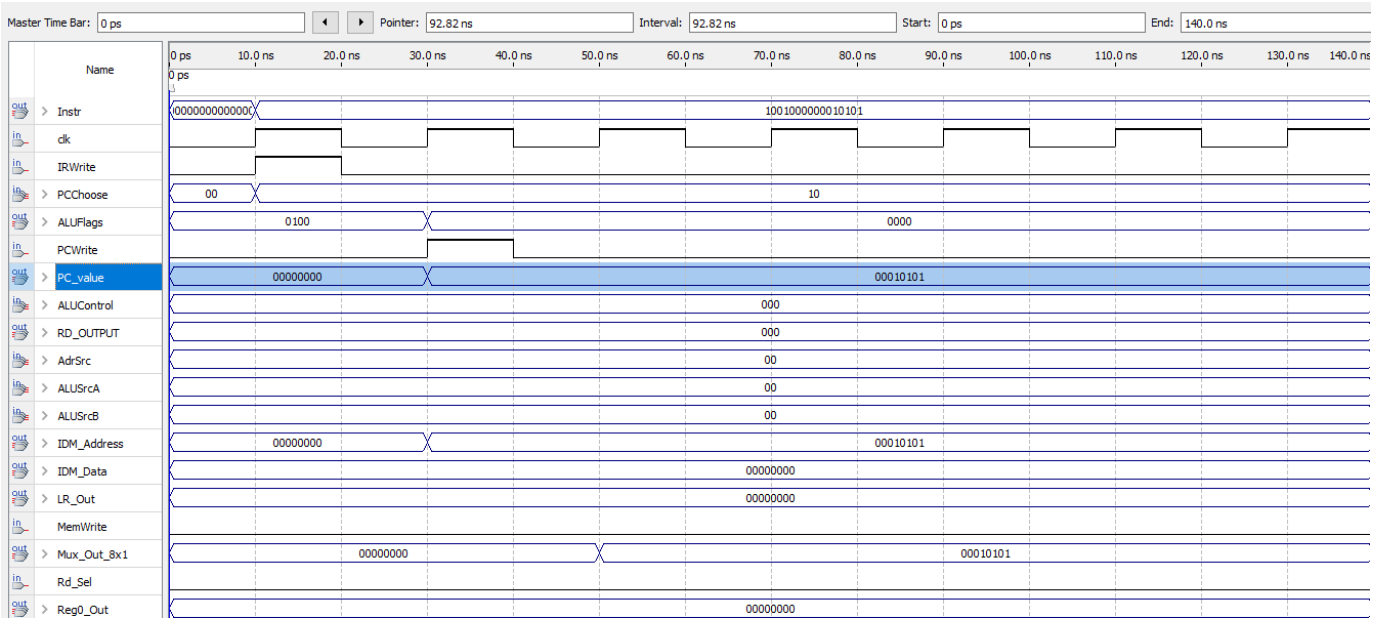
1) IRWrite = 1, Rdsel=1, RegSrc=1, WD3_Sel=2 (First Cycle)
2) AdrSrc=2 , RegWrite=1 (Second Cycle)
3) PCChoose=3 (third Cycle)
4) PCWrite=1 (Fourth Cycle)



3.4.4) BEQ #21 =10_0_100_0_0__00010101

Since ALU flags are updated by the controller unit, the brach is taken according to the ALUFlags decided by the ALU unit. Since zero is set in our case, the branch will be taken. You can observe the changes in PC_Value and LR_Out probes.

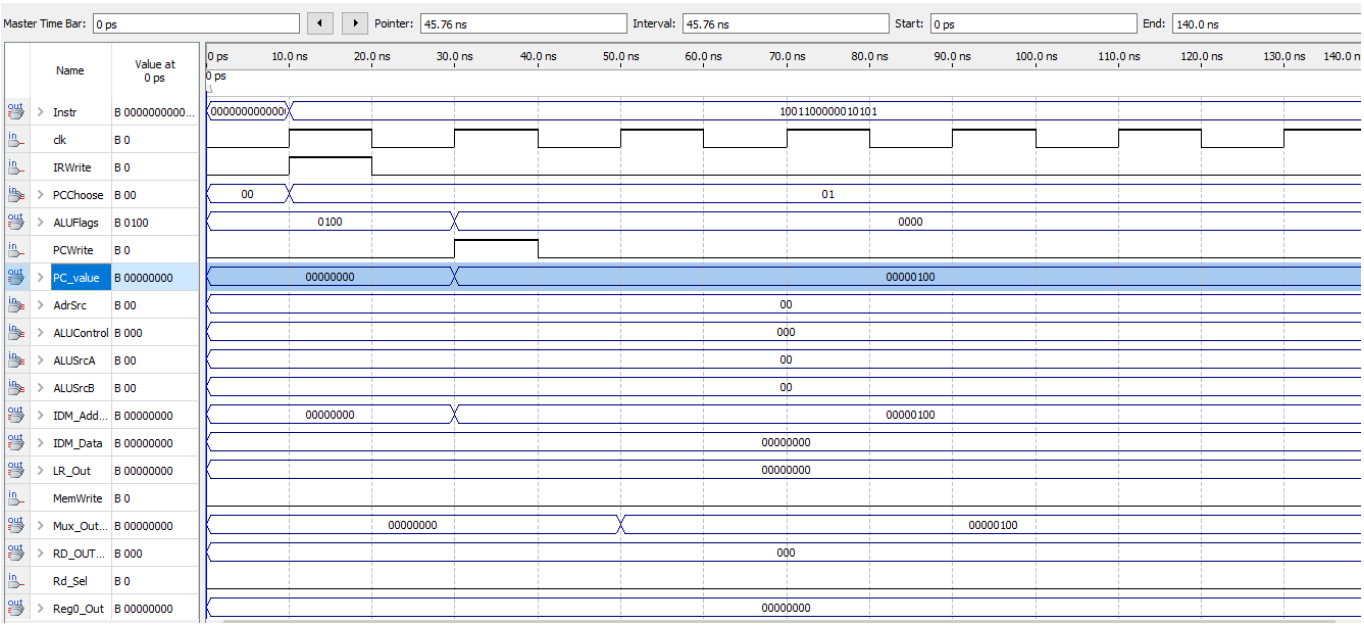
1) IRWrite = 1, (First Cycle)
2) PC_Choose=2, PCWrite=1 (Second Cycle)



3.4.5) BNE #21 =10_0_101_0_0__00010101

Since ALU flags are updated by the controller unit, the brach is taken according to the ALUFlags decided by the ALU unit. Since zero is not set in our case, the branch will not be taken. You can observe the changes in PC_Value and LR_Out probes. PC_Value will be PC+4.

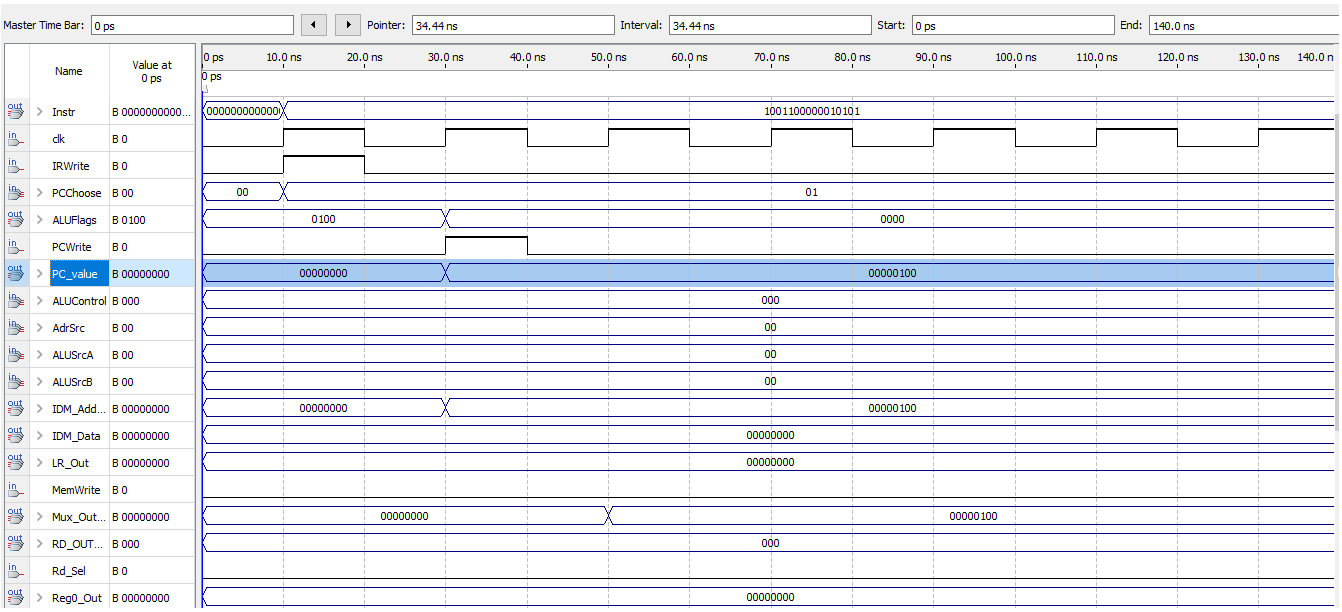
3) IRWrite = 1, (First Cycle)
4) PC_Choose=2, PCWrite=1 (Second Cycle)



3.4.6) BCS #21 =10_0_110_0_0__00010101

Since ALU flags are updated by the controller unit, the brach is taken according to the ALUFlags decided by the ALU unit. Since Carry is not set in our case, the branch will not be taken. You can observe the changes in PC_Value and LR_Out probes. PC_Value will be PC+4.

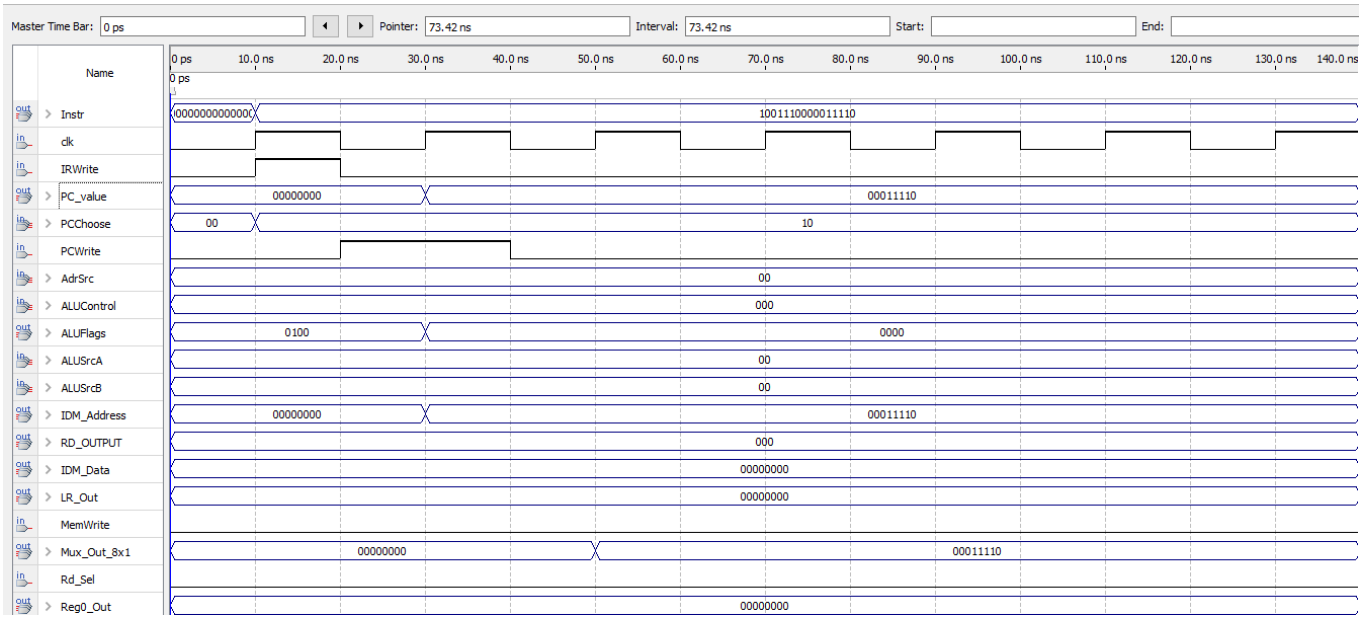
5) IRWrite = 1, (First Cycle)
6) PC_Choose=2, PCWrite=1 (Second Cycle)



3.4.7) BCC #30 =10_0_111_0_0__00011110

Since ALU flags are updated by the controller unit, the brach is taken according to the ALUFlags decided by the ALU unit. Since carry is not set in our case, the branch will be taken. You can observe the changes in PC_Value and LR_Out probes.

7) IRWrite = 1, (First Cycle)
8) PC_Choose=2, PCWrite=1 (Second Cycle)



NOT: BX LR function is added but, since I need to increase the Mux at the left most side from 4x1 to 8x1, I will implement it in the next lab.