EE446 Laboratory 4&5
Preliminary Work

Hakan Emre Gedik 2304657

Contents

Introduction

In this study, we create an ISA that is capable of specifying all the instructions given in the manual for the multicycle computer and create a datapath for it to execute each instruction. Multicycle computer Is advantageous in terms of maximum clock frequency since instructions use fewer components back-to-back. That is to say, depending on the instruction, hardware is used partially, which allows us to increase the clock frequency beyond the allowable border for the single cycle computer, which is constrained by the gate capacitances ever-present in each hardware block.

In this design, we endeavored to stick to ARM instruction set as much as possible for we see that it simplifies the design. Overall, the ISA is designed to simplify the datapath and the controller unit as much as possible.

1 - ISA

As expressed before, we endeavored to stick to the ARM ISA as much as possible. Yet, to be able to handle all the required instructions, we need to implement indirect addressing mode for the branch and some arithmetic operations. Taking the ARM ISA as baseline, we can implement indirect branch by using 7th bit of the instruction as the indicator, which is always 1 for branch instructions in ARM ISA. For indirect arithmetic operations, we intend to use the MSB of the Rn field in ARM ISA for it is not necessary to specify the register in our implementation since we have 8 registers in the register file, and each one of them can be addressed using 3 bits in total. For the shift operations, we will extend the barrel shifter we created in the first laboratory work.

Instruction length is chosen to be 32-bit even though it is a sub-optimum solution for the design we are required to implement. Additional functionalities such as shifting a register before an arithmetic or memory operation, choosing an immediate, which is specified by the I bit, instead of a register are desired to be kept, and they will be implemented for the sake of completeness even though they are not given as a must in the manual.

1.1 - Arithmetic, Logic, and Shift Instructions

All these instructions can be described by a single instruction by choosing the appropriate values for the opcode, indirect bit, shift type, and shift amount. Since in total 8 registers exist in the register file, 3 bits for specifying Rd, Rn, and Rm are enough. 15th bit, which would correspond to MSB of Rd in ARM ISA is set to 0, where the 19th bit is used for indicating whether an additional memory read must be done for the execution of the instruction.

Shift instructions can be included as well by selecting the opcode 1101, which is valid for all shift instructions. In the datapath, a barrel shifter is employed, for which one of its inputs is the shift type. Logic instructions can also be specified by the opcode. For example, opcode for EXOR operation is 0001. In summary, all arithmetic, logic, and shift operations can be identified with a single format of instruction, and they can be classified under the title of ‘data processing instructions’.

31 28	27 26	25	24 21	20	19	18 16	15	14 12	11 7	6 4	3	2 0
Cond	OP(00)	Im	OpCode	S	Ind	Rn	0	Rd	Shamt5	Shift Type	0	Rm

Figure 1 Data Processing Instruction Format

In figure 1, data processing instruction format is highlighted bit-by-bit. See that the datapath that will be designed for this instruction format will allows us to shift the second register before any kind of data processing instructions.

Since our final datapath will be capable of handling arithmetic instructions with an immediate value and register output, we include the following instruction format for choosing the second ALU operand as immediate. This inclusion is necessary for completeness since when the computer is initialized, if the there is no field for data memory, we can create the required data with these instructions, and store them to the memory for later use. See that the field that corresponds to the rotation value in ARM ISA is set to 0 since our register length is 8-bit and 8-bit immediate value is enough to describe all possible values of operation.

31 28	27 26	25	24 21	20	19	18 16	15	14 12	11 8	7 0
Cond	OP(00)	Im	OpCode	S	Ind	Rn	0	Rd	0000	8-bit Immediate

Figure 2 Data Processing with Immediate Value Instruction Format

Data processing operations also includes indirect operations. For that case, the last 8 bits of the instruction becomes the memory address of the second operand of the ALU. The format is as following:

31 28	27 26	25	24 21	20	19	18 16	15	14 12	11 8	7 0
Cond	OP(00)	Im	OpCode	S	Ind	Rn	0	Rd	0000	8-bit Indirect Address

Figure 3 Data Processing with Indirect Memory Access Instruction Format

1.2 - Branch Instructions

For branch instructions, the OP field must be specified as 10. In ARM ISA, 25th bit is always 1. For our implementation, we made it an indicator bit for the indirect branch instruction, which retrieves the memory address specified in offset and the branch is completed by taking the data retrieved from the memory as the offset value. Cond field is enough to specify all the conditional executions. It is important to note that we will design the controller and the datapath such that if the branch is decided not to be taken and indirect mode of addressing is to be used, the computer will directly go to the fetch of the next instruction to prevent unnecessary memory access due to the indirect operation.

31 28	27 26	25	24	23 0
Cond	OP(10)	Ind	L	Offset

Figure 4 Branch Instruction Format

Figure 4 highlights the branch instruction bit-by-bit. Branch target address (BTA) computation is as different than the single cycle and pipelined, computer which can be accepted as the general convention.

$$BTA = PC + 1 + offset$$

1.3 – Memory Instructions

For the memory instructions, bits in ARM ISA such as I, P, U, etc. are kept for the possibility of extending the capabilities of the computer beyond what is described in the lab manual. Similar to data processing instructions, the register specifying bits are 3-bits instead of 4 as it in ARM ISA. Hence, 19th bit can be, and is used for the 'load immediate to register' instruction. This instruction takes the last 8 bits as the immediate value and loads the immediate value to the register specified in the Rd section. In the real implementation, this operation is carried out in 'shift operations' section by taking the shift as 0.

31 28	27 26	25	24	23	22	21	20	19	18 16	15	14 12	11 0
Cond	OP(01)	I'	P	U	B	W	L	Im	Rn	0	Rd	Offset / Immediate

Figure 5 Memory Reference Instruction Format

2 Datapath Design

2.1 Update the Datapath

In *Digital Design and Computer Architecture Harris&Harris*, the datapath in figure 4 is given.

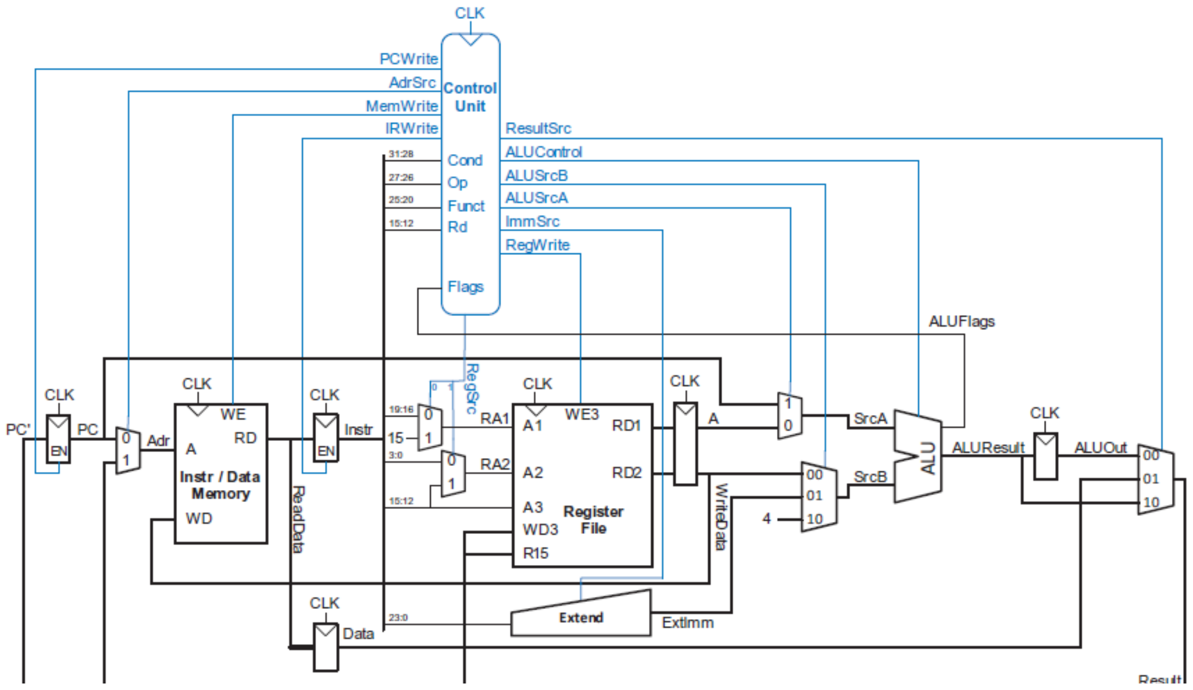


Figure 6 Harris&Harris Multicycle Computer Datapath

It is important to note that this datapath can execute most of the instructions that we expect our datapath to be capable of. Yet, certain instructions in the manual cannot be performed. In particular, indirect memory and arithmetic operations, branch with link, and shift operations cannot be executed no matter how the controller is altered.

To be able to perform all the instructions provided in the manual, this datapath must be extended. We finalize our datapath design by taking the following steps:

- Connect 'Data' register output, the data read from the memory, to the 11 port of the MUX before the SrcB input of the ALU. This is done to execute 'branch indirect', 'addition indirect', and 'subtraction indirect' instructions.
- Add an enhanced version of 'Barrel Shifter' component which is employed in single cycle computer, lab 3. It should be placed right before the 00 input of the ALU SrcB port.
- Replace the 2x1 MUX before the SrcA with a 4x1 MUX. Connect PC to 00, RD1 register output to 01, and a constant 0 to 10. In this way, 'load immediate to register', and the indirect operations can be performed. Furthermore, the shift result of RD2 is taken from the output of the ALU by adding it with 0.
- Replace 15 with 7 since our register file has 8 registers, and the last register is used to calculate branch target addresses.
- Add a 2x1 MUX right before WD3 input port of the register file to be able to write the current value of the PC to R6 (LR).
- Add a 2x1 MUX before the A3 input port of the register file to be able to select the destination register as R6 (LR). This is essential since in the BL instruction, there is no field to specify the destination register to be R6 (LR)

Finalize the datapath by taking these steps.

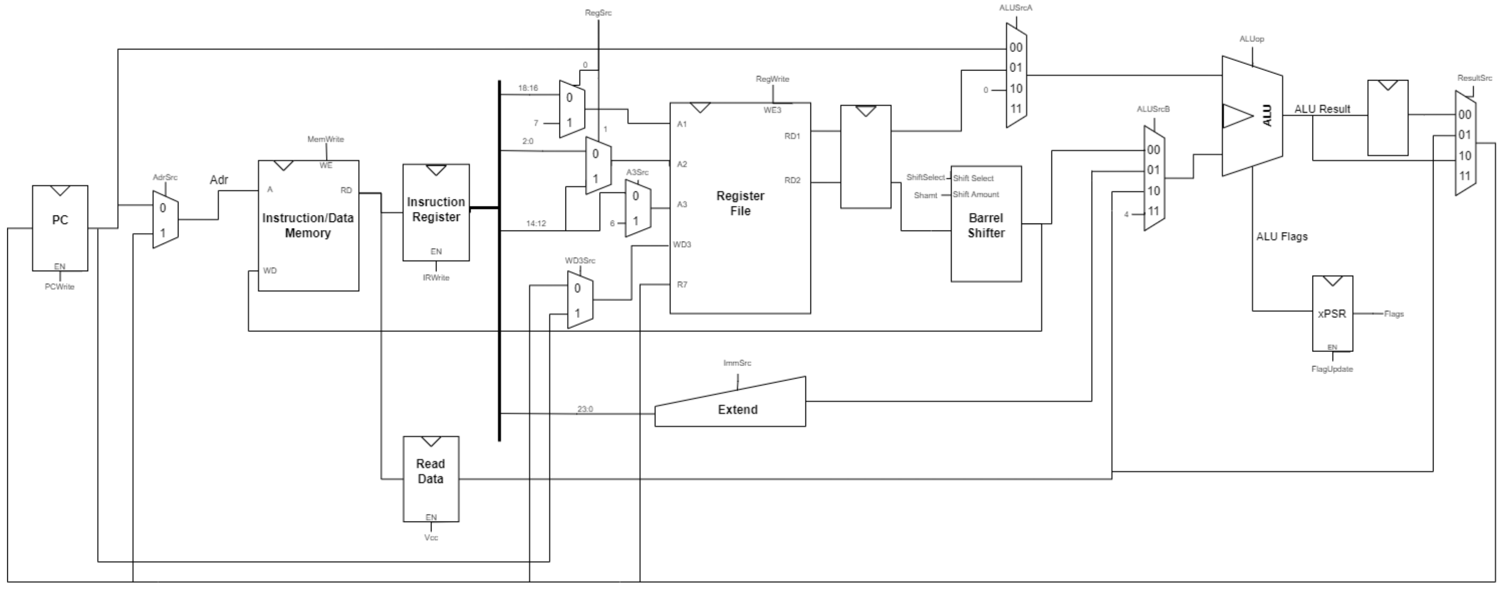


Figure 7 Final Datapath of the Multicycle Computer

With this final version of the datapath, we are capable of achieving the required indirect branch, indirect arithmetic operations, and branch with link. For the indirect branch operation, after the decode stage, the extended immediate value is selected as the source B port of the ALU (ALUSrcB = 01), and source A port of the ALU is selected as 0 (ALUSrcA = 10). The memory address is calculated by adding these two, and in the next cycle, the offset value present in the memory address is brought into 'Read Data' register. In the cycle after, BTA is calculated in directly written back to the PC by providing the control signals ALUSrcA = 01, ALUSrcB = 10 (the read data), and ResultSrc = 10. Other indirect operations are carried out similarly. Rx = Ry -(op)- M(Rz) operations, we calculate the memory address in the ALU and read the data into the 'Read Data' register, and then carry out the necessary arithmetic operation by selecting the data register as the second operand of the ALU.

We also made sure our datapath is capable of handling MOV PC LR (BX LR) operation, which is carried out quite similar to shift instructions. We believe this instruction is essential since without being capable of returning to the address of the link register, branch with link operation has no advantage over direct branch. In general, our design approach of multicycle computer is on the side of completeness, robustness, and functionality. With small adjustments in datapath, and having 32-bit instructions, we can have a complete computer, capable of handling loops, etc. Detailed state machine for instructions and control signals for each state will be provided in this report.

2.2 Component Updates

We use the same components which are used in laboratory work 3. Yet, since we changed the register lengths and the instruction set architecture, it is necessary to alter these components to be able to properly use in the multicycle computer.

2.2.1 Register File

Our register file will have 8 registers in total, and each has 8-bit length. In the previous implementation, we had 16 registers each having 32-bit length. Number of multiplexers for output selection and write enables must change. Furthermore, in the previous lab, we took R15, the last register, as the program counter for branch instructions. In this lab, this register becomes R7. Required changes has been made, and the testbench result is given figure 8.

2.2.2 Instruction/Data Memory

In the single cycle computer, instruction and data memories are implemented separately, both having 32-bit lengths of data in each row. Instruction memory had no write capability for it will be externally programmed like a ROM, where the data memory needed to have write capability for it is a must with store instructions.

In multicycle computer, we keep the memory word length 32-bits since we intend to use it for both instructions and data, and instruction length is 32-bits. Number of words in the memory is set to 128 for the length of the addresses we can employ is at max 8-bits, which makes the addressing of elements greater than 128 impossible. The testbench results for the updated instruction/data memory can be found in figure 9.

2.2.3 Barrel Shifter

In laboratory work 3, the single cycle computer assignment, we only had left and right logical shift operations. Yet, the requirements of multicycle computer shift operation implementations include arithmetic shift right, and rotation operations. Capabilities of barrel shifter is extended to meet these requirements. It is a combinational circuit, requiring no clock edges for output update, as it was in our previous work. The testbench results can be found in figure 10.

2.2.4 Immediate Extender

In single cycle computer assignment, we implemented immediate extender to be capable of applying rotation immediate operation for it is the way specified on ARM instruction set. Yet, for the multicycle computer case with the ISA created, the immediate extender does not need to have this additional functionality. For all the cases that the immediate value is required such as indirect branch, memory reference instructions, and data processing instructions with immediate only the last 8 bit of the instruction will be employed. This component is simplified only to handle its required function and its testbench results can be found in figure 11.

2.3 Testbench Results of Updated Components

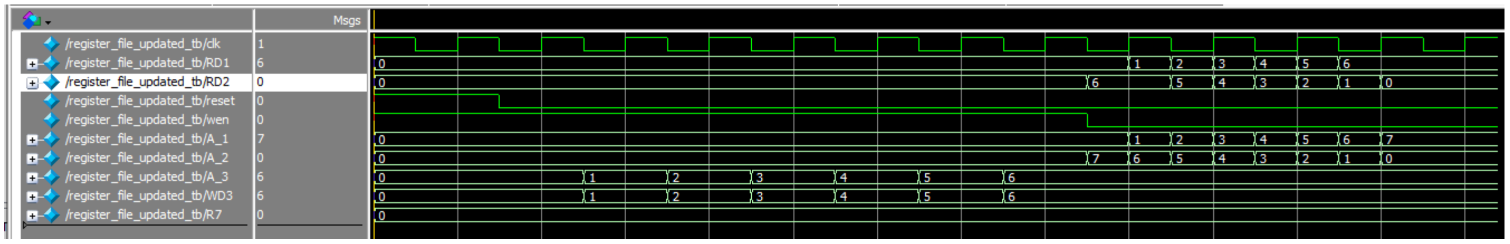

Msgs
/register_file_updated_tb/clk 1
/register_file_updated_tb/RD1 6
/register_file_updated_tb/RD2 0
/register_file_updated_tb/reset 0
/register_file_updated_tb/wen 0
/register_file_updated_tb/A_1 7
/register_file_updated_tb/A_2 0
/register_file_updated_tb/A_3 6
/register_file_updated_tb/WD3 6
/register_file_updated_tb/R7 0

Figure 8 Register File Testbench

We loaded the register Ri with value i in the first cycle and printed their values in increasing indices in one hand with A1 and decreasing indices on the other hand with A2.

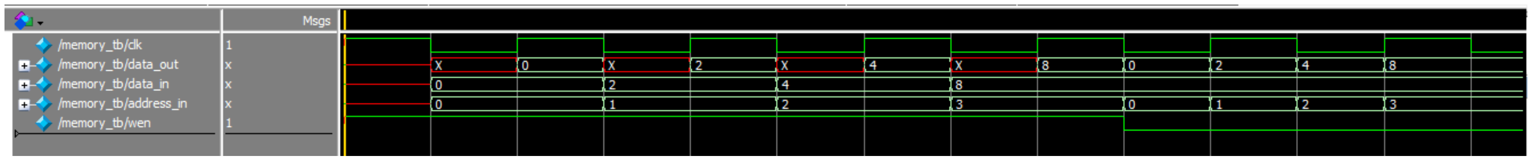
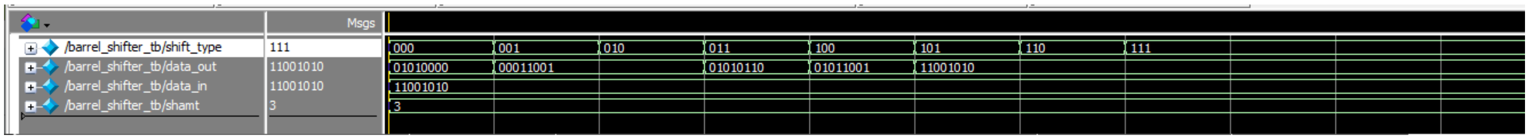


Figure 9 Memory Testbench

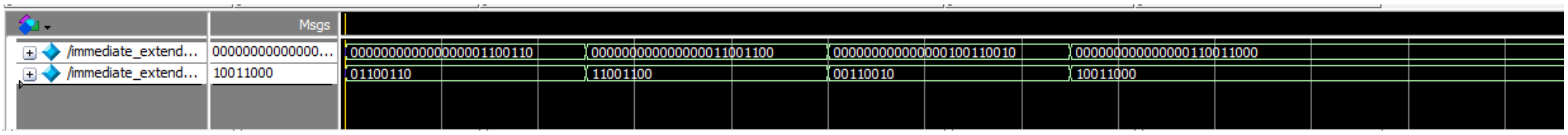
For the memory testbench, in the first half, we carry out store operation to address i with the value 2*i, in the first half. In the second half, these values are printed, and they can be observed.



```
                                Msgs
/barrel_shifter_tb/shift_type   111        000       001       010   011       100       101       110   111
/barrel_shifter_tb/data_out     11001010   01010000  00011001        01010110  01011001  11001010
/barrel_shifter_tb/data_in      11001010   11001010
/barrel_shifter_tb/shamt        3          3
```

Figure 11 Barrel Shifter Testbench

Each shift operation is tested and verified with shift amount of 3.



```
                      Msgs
/immediate_extend...  00000000000000...  000000000000000001100110  000000000000000011001100  000000000000000000110010  000000000000000010011000
/immediate_extend...  10011000           01100110                  11001100                  00110010                  10011000
```

Figure 10 Immediate Extender

Only the last 8-bits of the input is reflected to the output.

2.4 Datapath Inputs and Outputs

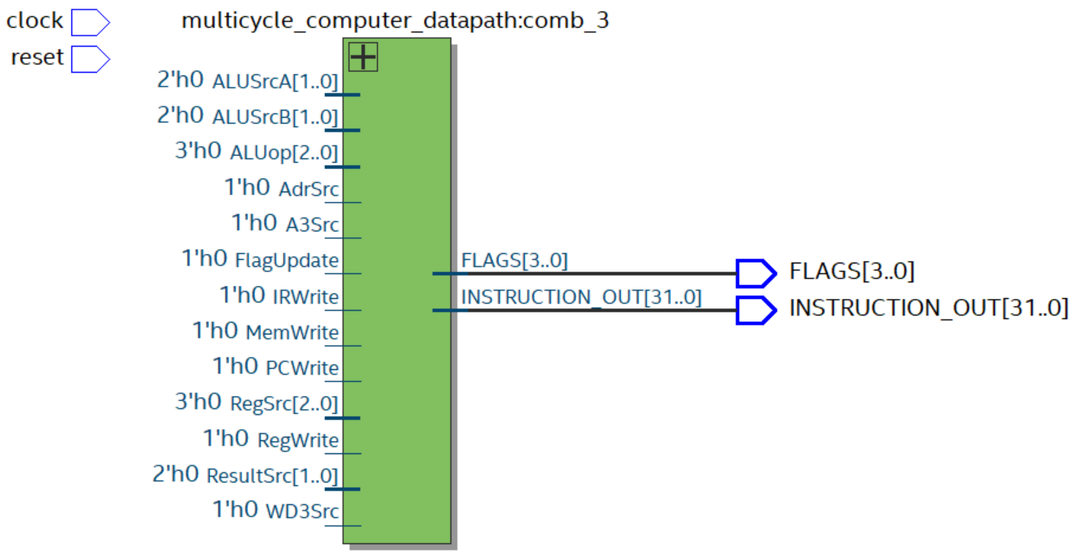


Figure 12 Datapath Inputs and Outputs

In figure 12, inputs and outputs of the datapath are laid out to clearly show the control signals and controller inputs. Program status register (flags) and the instruction itself will be used by the controller to generate the required signals for each cycle in the state machine of operation.

3. State Machine

In single cycle computer, all instructions took only one clock cycle, where for the multicycle computer, all instructions take at least 3 clock cycles. Each cycle for the instructions is unique and they provide different control signals.

Control signals will be generated depending on the state and the instruction itself. That is to say, controller will produce control signals combinationally depending on the state, the instruction decode fields, and flag update, etc.

The control signals AdrSrc, IRWrite, MemWrite, PCWrite, ALUSrcA, ALUSrcB, and ResultSrc will be directly produced with the state knowledge, whereas the control signals RegSrc, A3Src, FlagUpdate, WD3Src, RegWrite, ShiftSelect and ALUop will be produced by using the state information and the instruction itself. Meaning, the controller will consist of two main units, namely, the state machine and the ALU/Instruction decoder. The state machine created for our multicycle can be observed in figure 13.

It is possible to see that our controller is not a Moore machine, for which the output signals are only function of the current state, but rather it is a Mealy machine, for which the output signals are function of the current state and the instruction itself.

The controller details, and the way we construct it from it sub-components will be highlighted later in this study.

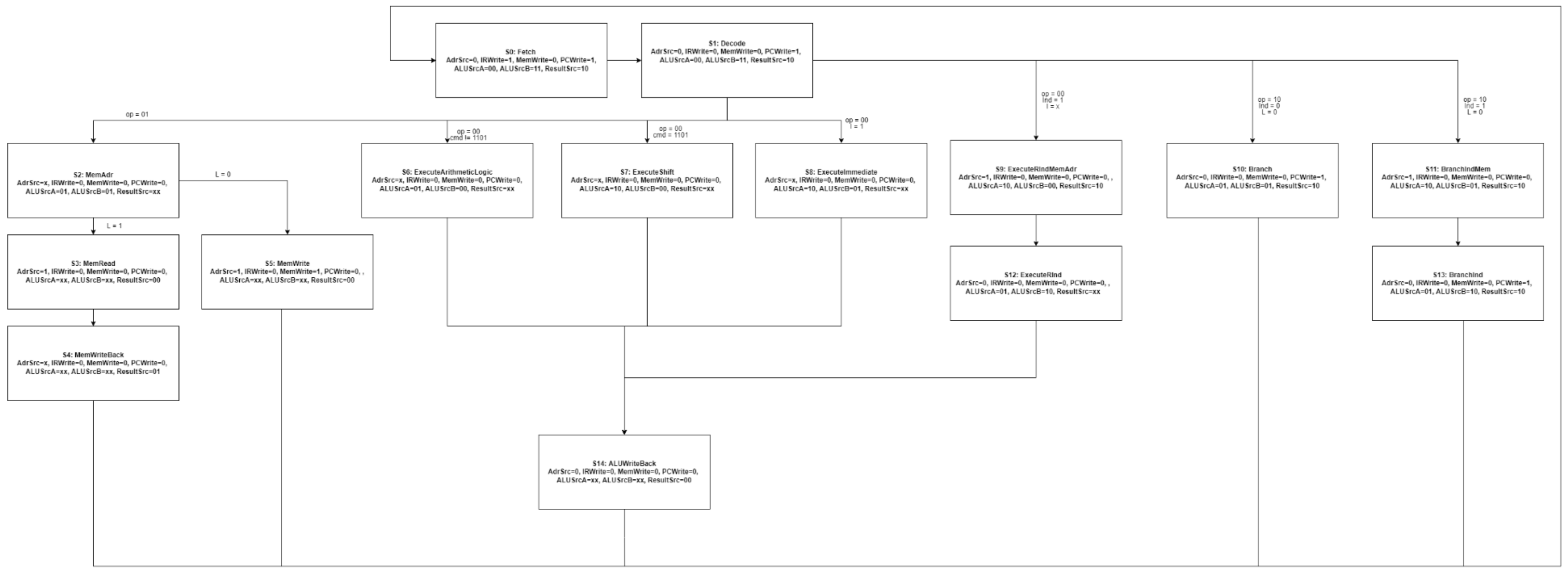
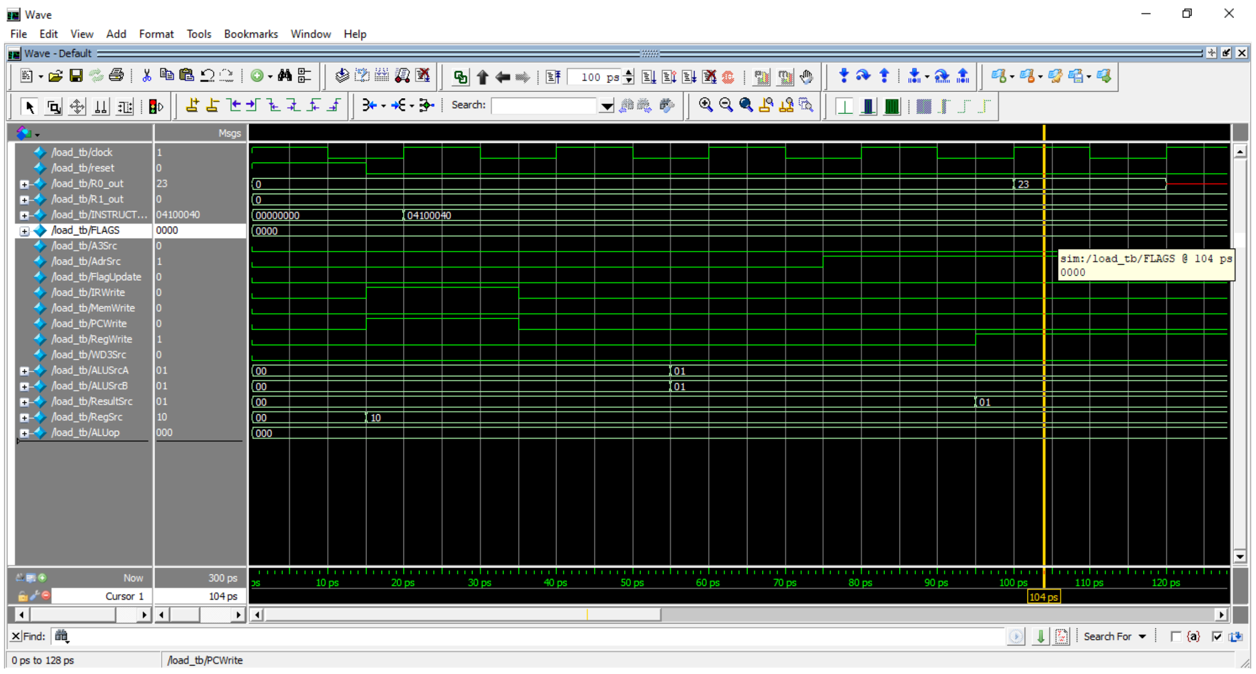

S0: Fetch
AdrSrc=0, IRWrite=1, MemWrite=0, PCWrite=1,
ALUSrcA=00, ALUSrcB=11, ResultSrc=10
S1: Decode
AdrSrc=0, IRWrite=0, MemWrite=0, PCWrite=1,
ALUSrcA=00, ALUSrcB=11, ResultSrc=10
op = 01
S2: MemAdr
AdrSrc=x, IRWrite=0, MemWrite=0, PCWrite=0,
ALUSrcA=01, ALUSrcB=01, ResultSrc=xx
L = 0
L = 1
S3: MemRead
AdrSrc=1, IRWrite=0, MemWrite=0, PCWrite=0,
ALUSrcA=xx, ALUSrcB=xx, ResultSrc=00
S4: MemWriteBack
AdrSrc=x, IRWrite=0, MemWrite=0, PCWrite=0,
ALUSrcA=xx, ALUSrcB=xx, ResultSrc=01
S5: MemWrite
AdrSrc=1, IRWrite=0, MemWrite=1, PCWrite=0, ,
ALUSrcA=xx, ALUSrcB=xx, ResultSrc=00
op = 00
cmd != 1101
S6: ExecuteArithmeticLogic
AdrSrc=x, IRWrite=0, MemWrite=0, PCWrite=0,
ALUSrcA=01, ALUSrcB=00, ResultSrc=xx
op = 00
cmd = 1101
S7: ExecuteShift
AdrSrc=x, IRWrite=0, MemWrite=0, PCWrite=0,
ALUSrcA=10, ALUSrcB=00, ResultSrc=xx
op = 00
I = 1
S8: ExecuteImmediate
AdrSrc=x, IRWrite=0, MemWrite=0, PCWrite=0,
ALUSrcA=10, ALUSrcB=01, ResultSrc=xx
op = 00
Ind = 1
I = x
S9: ExecuteRIndMemAdr
AdrSrc=1, IRWrite=0, MemWrite=0, PCWrite=0, ,
ALUSrcA=10, ALUSrcB=00, ResultSrc=10
S12: ExecuteRInd
AdrSrc=0, IRWrite=0, MemWrite=0, PCWrite=0, ,
ALUSrcA=01, ALUSrcB=10, ResultSrc=xx
op = 10
Ind = 0
L = 0
S10: Branch
AdrSrc=0, IRWrite=0, MemWrite=0, PCWrite=1,
ALUSrcA=01, ALUSrcB=01, ResultSrc=10
op = 10
Ind = 1
L = 0
S11: BranchIndMem
AdrSrc=1, IRWrite=0, MemWrite=0, PCWrite=0,
ALUSrcA=10, ALUSrcB=01, ResultSrc=10
S13: BranchInd
AdrSrc=0, IRWrite=0, MemWrite=0, PCWrite=1,
ALUSrcA=01, ALUSrcB=10, ResultSrc=10
S14: ALUWriteBack
AdrSrc=0, IRWrite=0, MemWrite=0, PCWrite=0,
ALUSrcA=xx, ALUSrcB=xx, ResultSrc=00

Figure 13 The State Machine

4. Instruction Testbenches

With the state machine and control signals for each state is laid down, we can create testbenches for each instruction.

4.1 Load Instruction



```
/load_tb/clock          1
/load_tb/reset          0
/load_tb/R0_out         23
/load_tb/R1_out         0
/load_tb/INSTRUCT...    04100040
/load_tb/FLAGS          0000
/load_tb/A3Src          0
/load_tb/AdrSrc         1
/load_tb/FlagUpdate     0
/load_tb/IRWrite        0
/load_tb/MemWrite       0
/load_tb/PCWrite        0
/load_tb/RegWrite       1
/load_tb/WD3Src         0
/load_tb/ALUSrcA        01
/load_tb/ALUSrcB        01
/load_tb/ResultSrc      01
/load_tb/RegSrc         10
/load_tb/ALUop          000

sim:/load_tb/FLAGS @ 104 ps
0000

Now       300 ps
Cursor 1  104 ps

0 ps to 128 ps    /load_tb/PCWrite
```

Figure 14 Load Instruction Testbench

For the testbench, for which the result of is observable in figure 14, the instruction and data memory is initialized with the following code segment:

```
initial begin
        mem[0] = 32'h04100040; // LDR R0, [R0, #64]
        mem[64] = 32'd23;
end
```

In the first clock cycle, we pull the reset high. As can be observed in figure 14, after 5 positive edges of the clock, we write the value in memory location 64, which is decimal value of 23, to R0.

4.2 Store Instruction

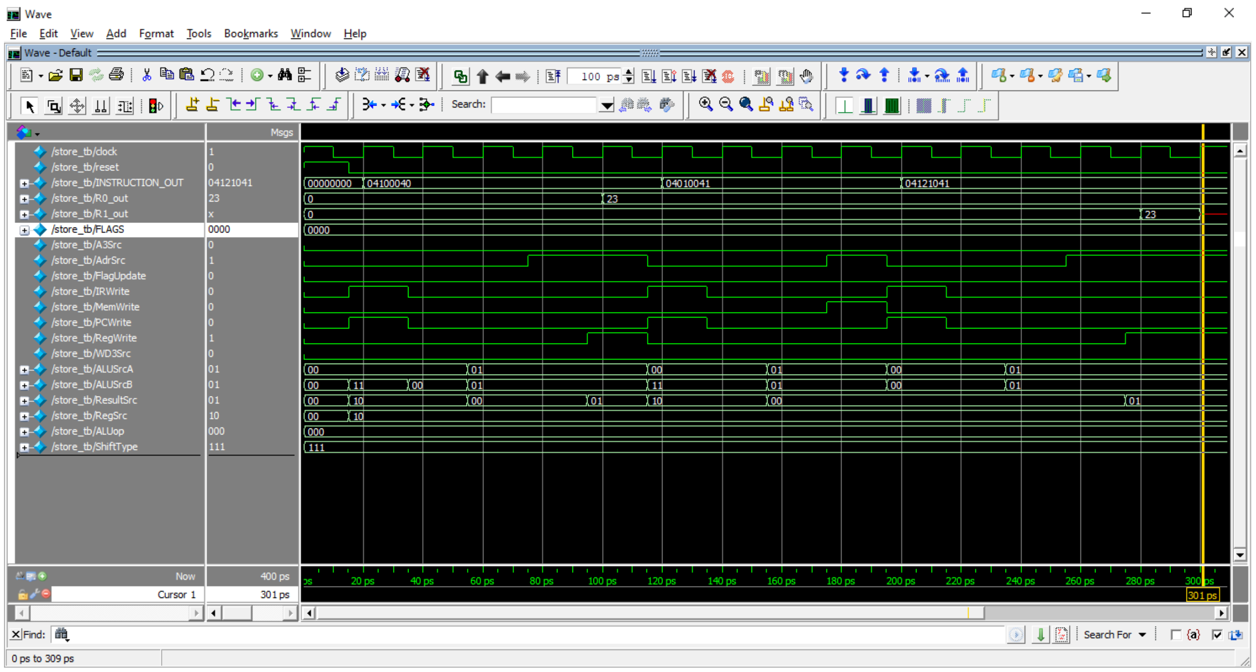


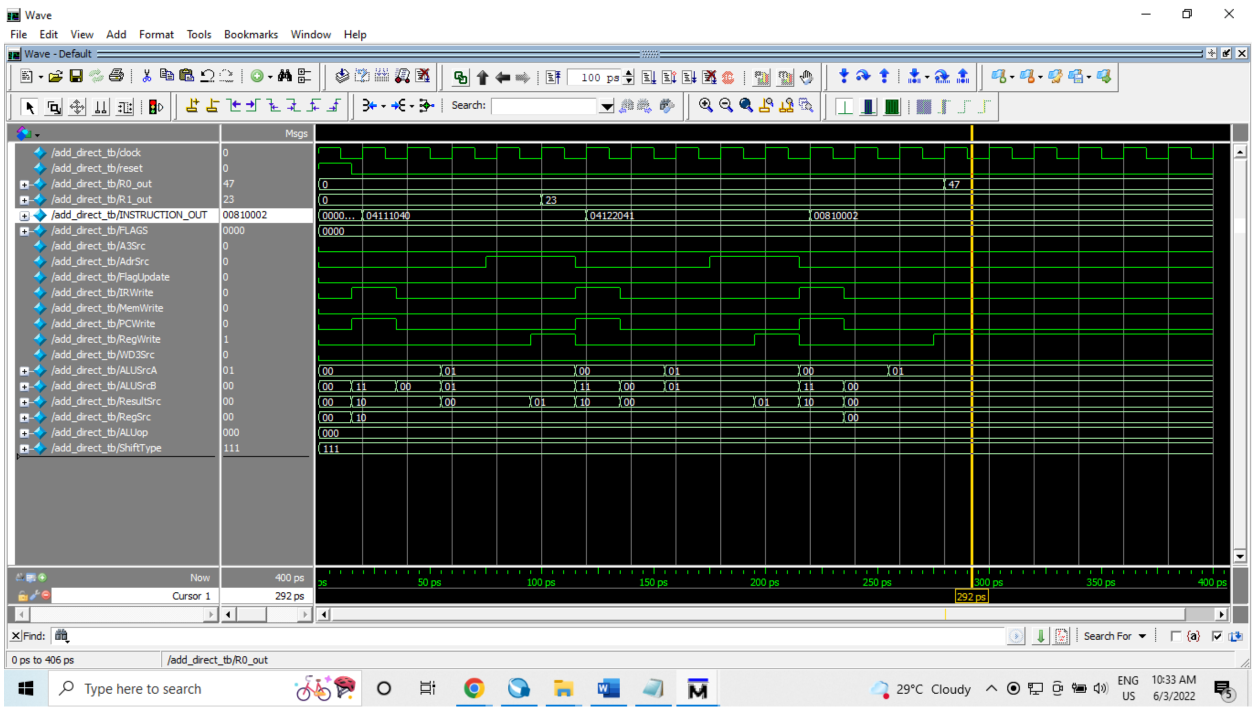
Figure 15 Store Instruction Testbench

For the testbench, for which the result of is observable in figure 15, the instruction and data memory is initialized with the following code segment:

```
initial begin
        mem[0] = 32'h04100040; // LDR R0, [R0, #64]
        mem[1] = 32'h04010041; // STR R0, [R1, #65]
        mem[2] = 32'h04121041; // LDR R1, [R2, #65]
        mem[64] = 32'd23;
end
```

We validate the store instruction with 3 instructions, which is expressed in ARM Assembly code in the comment sections of the memory initialization above. In the second instruction, we perform the load operation, and in the end the 3rd instruction, we expect to determine whether the store instruction works or not. Initially, we have the decimal value of 23 in memory location 64. We first load this value to R0 as it was done in load testbench. Then, we store this value to memory location 65. Finally, we load the value in memory location 65 to R1, to validate that we were capable of storing the value 23 in memory location 65. In the end of the testbench, observe the value of R1_out, which is 23 as intended. Figure 15 assures that store operation works as intended.

4.3 Direct Addition Instruction



	Msgs
/add_direct_tb/clock	0
/add_direct_tb/reset	0
/add_direct_tb/R0_out	47
/add_direct_tb/R1_out	23
/add_direct_tb/INSTRUCTION_OUT	00810002
/add_direct_tb/FLAGS	0000
/add_direct_tb/A3Src	0
/add_direct_tb/AdrSrc	0
/add_direct_tb/FlagUpdate	0
/add_direct_tb/IRWrite	0
/add_direct_tb/MemWrite	0
/add_direct_tb/PCWrite	0
/add_direct_tb/RegWrite	1
/add_direct_tb/WD3Src	0
/add_direct_tb/ALUSrcA	01
/add_direct_tb/ALUSrcB	00
/add_direct_tb/ResultSrc	00
/add_direct_tb/RegSrc	00
/add_direct_tb/ALUop	000
/add_direct_tb/ShiftType	111

Now 400 ps
Cursor 1 292 ps

0 ps to 406 ps /add_direct_tb/R0_out

Figure 16 Direct Addition Instruction Testbench

For the testbench, for which the result of is observable in figure 16, the instruction and data memory is initialized with the following code segment:

```
initial begin
        // INSTRUCTION SECTION
        mem[0] = 32'h04111040; // LDR R1, [R1, #64]
        mem[1] = 32'h04122041; // LDR R2, [R2, #65]
        mem[2] = 32'h00810002; // ADD R0, R1, R2


        // DATA SECTION
        mem[64] = 32'd23;
        mem[65] = 32'd24;
        mem[66] = 32'd25;
        mem[67] = 32'd26;
        mem[68] = 32'd27;
        mem[69] = 32'd28;
        mem[70] = 32'd29;
end
```

Firstly, we load the value in memory location 64 to R1, and then we load the value in memory location 65 to R2. Finally, we add the values in R1 and R2, and write the result to R0. As can be seen in figure 16, correct result of addition, 47, is obtained in the output of R0.

4.4 Direct Subtraction Instruction

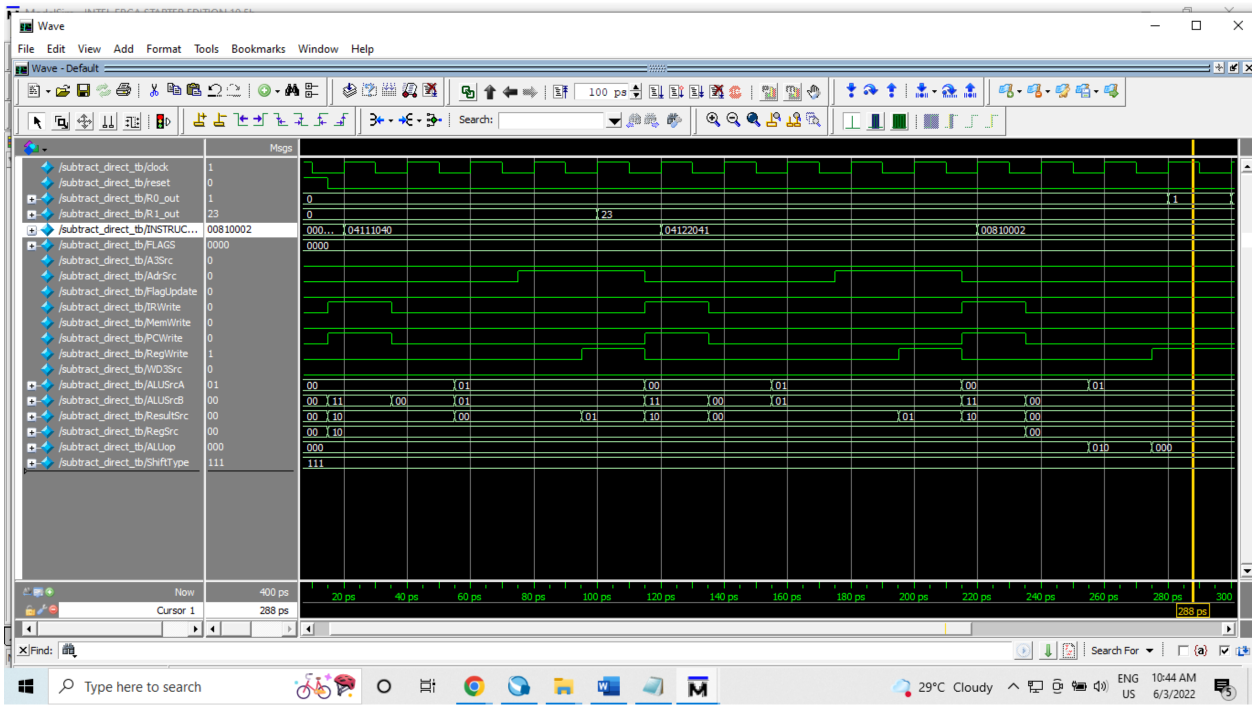

Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Msgs
/subtract_direct_tb/clock 1
/subtract_direct_tb/reset 0
/subtract_direct_tb/R0_out 1
/subtract_direct_tb/R1_out 23
/subtract_direct_tb/INSTRUC... 00810002
/subtract_direct_tb/FLAGS 0000
/subtract_direct_tb/A3Src 0
/subtract_direct_tb/AdrSrc 0
/subtract_direct_tb/FlagUpdate 0
/subtract_direct_tb/IRWrite 0
/subtract_direct_tb/MemWrite 0
/subtract_direct_tb/PCWrite 0
/subtract_direct_tb/RegWrite 1
/subtract_direct_tb/WD3Src 0
/subtract_direct_tb/ALUSrcA 01
/subtract_direct_tb/ALUSrcB 00
/subtract_direct_tb/ResultSrc 00
/subtract_direct_tb/RegSrc 00
/subtract_direct_tb/ALUop 000
/subtract_direct_tb/ShiftType 111
04111040
04122041
00810002
Now 400 ps
Cursor 1 288 ps
288 ps

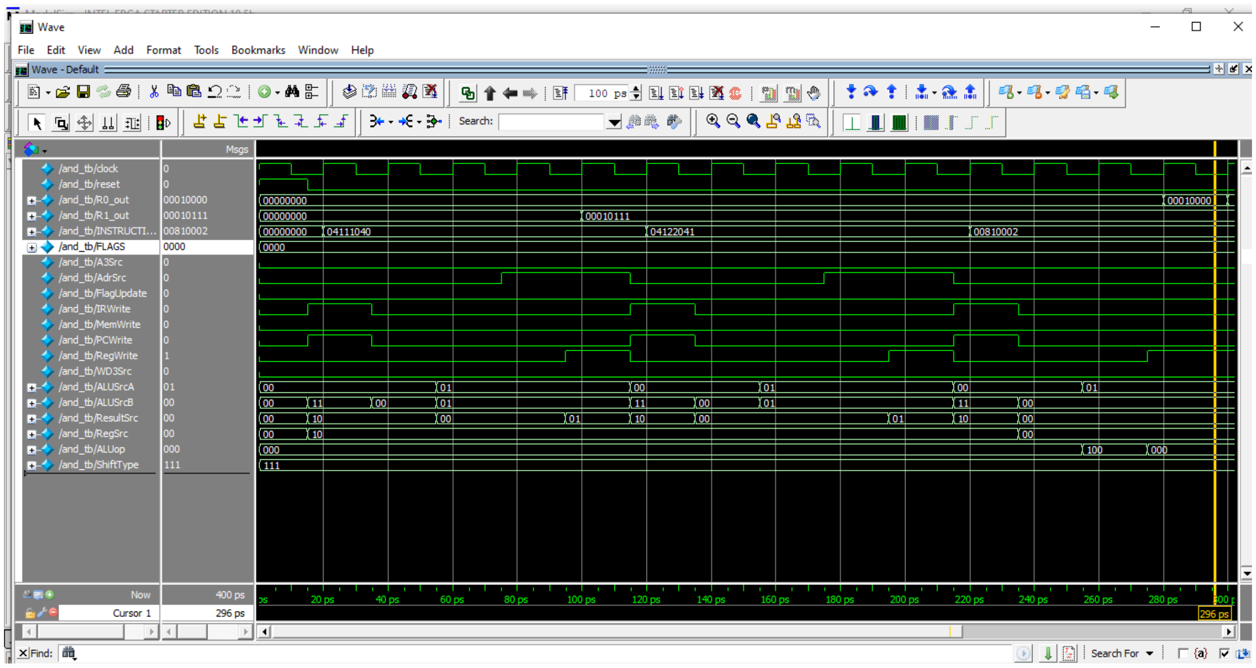
Figure 17 Direct Subtraction Instruction Testbench

For the testbench, for which the result of is observable in figure 17, the instruction and data memory is initialized with the following code segment:

```
initial begin
        // INSTRUCTION SECTION
        mem[0] = 32'h04111040; // LDR R1, [R1, #64]
        mem[1] = 32'h04122041; // LDR R2, [R2, #65]
        mem[2] = 32'h00410002; // SUB R0, R1, R2


        // DATA SECTION
        mem[64] = 32'd23;
        mem[65] = 32'd24;
        mem[66] = 32'd25;
        mem[67] = 32'd26;
        mem[68] = 32'd27;
        mem[69] = 32'd28;
        mem[70] = 32'd29;
end
```

As we did in the direct subtraction testbench, we first load the values in memory addresses 64, 65 to R1 and R0, respectively. Then, we perform the subtraction in the second instruction. As can be observed from figure 17, the correct result 1 in the output of R0 is obtained.

4.5 AND Instruction



	Msgs
/and_tb/clock	0
/and_tb/reset	0
/and_tb/R0_out	00010000
/and_tb/R1_out	00010111
/and_tb/INSTRUCTI...	00810002
/and_tb/FLAGS	0000
/and_tb/A3Src	0
/and_tb/AdrSrc	0
/and_tb/FlagUpdate	0
/and_tb/IRWrite	0
/and_tb/MemWrite	0
/and_tb/PCWrite	0
/and_tb/RegWrite	1
/and_tb/WD3Src	0
/and_tb/ALUSrcA	01
/and_tb/ALUSrcB	00
/and_tb/ResultSrc	00
/and_tb/RegSrc	00
/and_tb/ALUop	000
/and_tb/ShiftType	111

Figure 18 AND Instruction Testbench

For the testbench, for which the result of is observable in figure 18, the instruction and data memory is initialized with the following code segment:

```
initial begin
        // INSTRUCTION SECTION
        mem[0] = 32'h04111040; // LDR R1, [R1, #64]
        mem[1] = 32'h04122041; // LDR R2, [R2, #65]
        mem[2] = 32'h00010002; // AND R0, R1, R2


        // DATA SECTION
        mem[64] = 32'd23;
        mem[65] = 32'd24;
        mem[66] = 32'd25;
        mem[67] = 32'd26;
        mem[68] = 32'd27;
        mem[69] = 32'd28;
        mem[70] = 32'd29;
end
```

As in the previous data processing instructions’ testbenches, we load R1 with the value 8’b00010111 (decimal 23), and R2 with 8’b00011000 (decimal 24). Result of the AND operation must be 8’b00010000, and it can be observed in the output of R0 when all of the instructions are completed.

4.6 OR Instruction

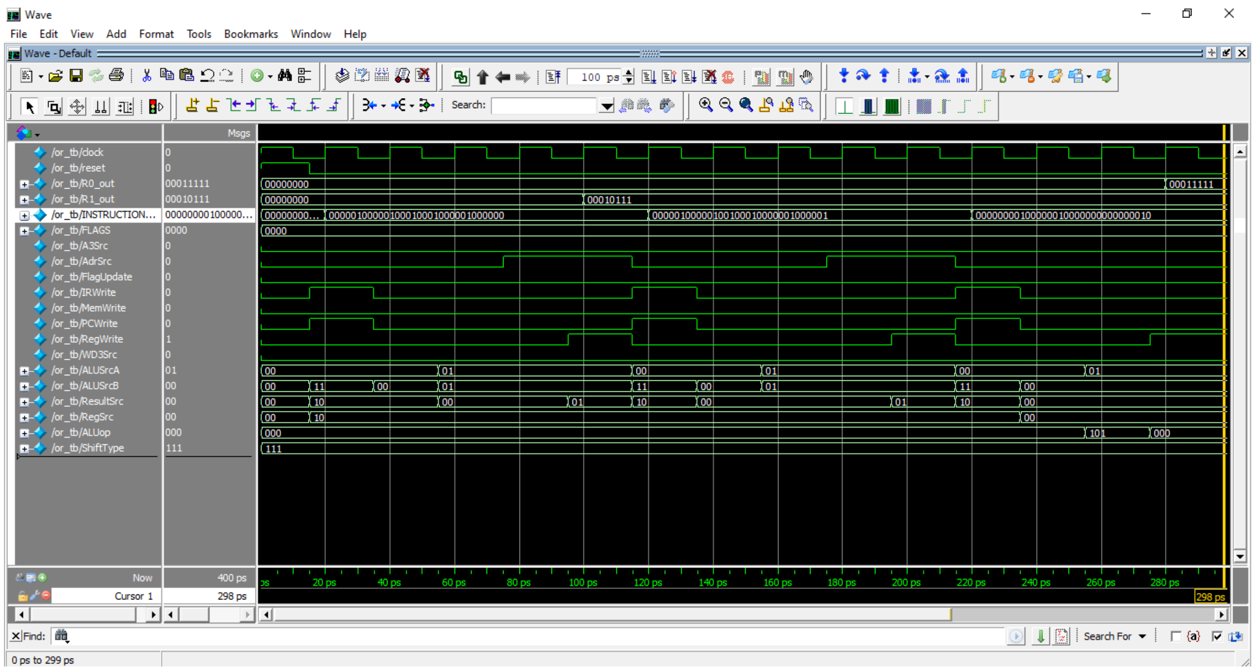


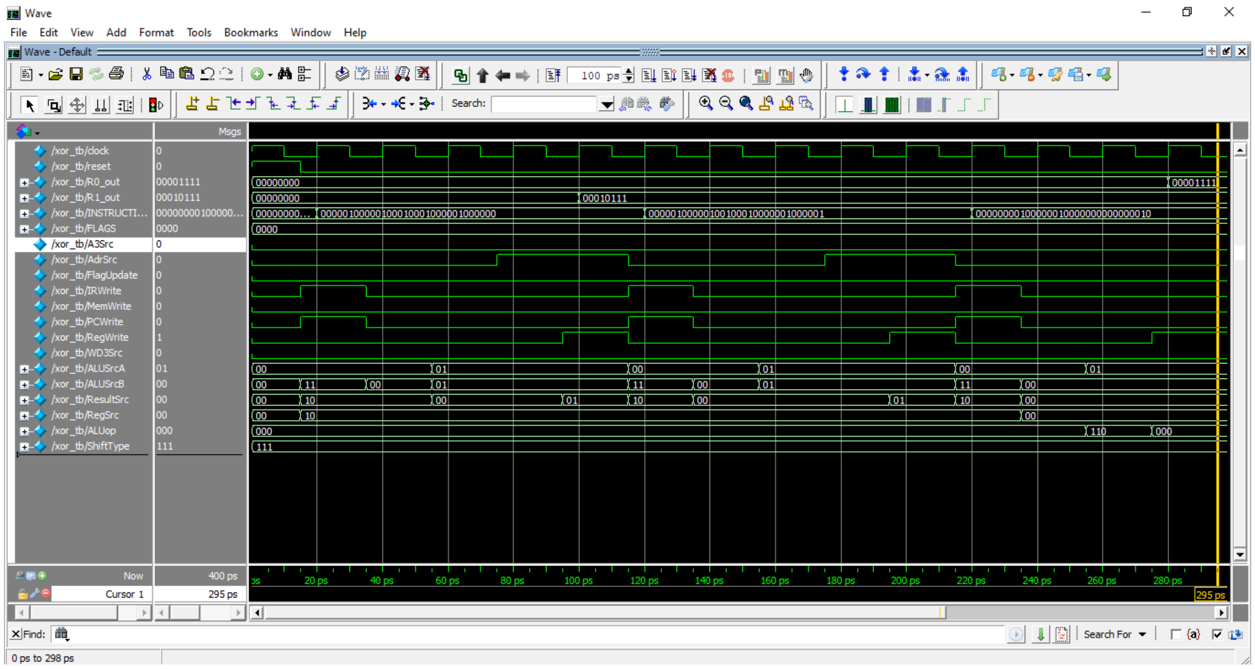
Figure 19 OR Instruction Testbench

For the testbench, for which the result of is observable in figure 19, the instruction and data memory is initialized with the following code segment:

```
initial begin
        // INSTRUCTION SECTION
        mem[0] = 32'h04111040; // LDR R1, [R1, #64]
        mem[1] = 32'h04122041; // LDR R2, [R2, #65]
        mem[2] = 32'h01810002; // ORR R0, R1, R2


        // DATA SECTION
        mem[64] = 32'd23;
        mem[65] = 32'd24;
        mem[66] = 32'd25;
        mem[67] = 32'd26;
        mem[68] = 32'd27;
        mem[69] = 32'd28;
        mem[70] = 32'd29;
end
```

As in the previous data processing instructions' testbenches, we load R1 with the value 8'b00010111 (decimal 23), and R2 with 8'b00011000 (decimal 24). Result of the OR operation must be 8'b00011111, and it can be observed in the output of R0 when all of the instructions are completed.

4.7 XOR Instruction



```
/xor_tb/clock          0
/xor_tb/reset          0
/xor_tb/R0_out         00001111
/xor_tb/R1_out         00010111
/xor_tb/INSTRUCTI...   00000000100000...
/xor_tb/FLAGS          0000
/xor_tb/A3Src          0
/xor_tb/AdrSrc         0
/xor_tb/FlagUpdate     0
/xor_tb/IRWrite        0
/xor_tb/MemWrite       0
/xor_tb/PCWrite        0
/xor_tb/RegWrite       1
/xor_tb/WD3Src         0
/xor_tb/ALUSrcA        01
/xor_tb/ALUSrcB        00
/xor_tb/ResultSrc      00
/xor_tb/RegSrc         00
/xor_tb/ALUop          000
/xor_tb/ShiftType      111
```

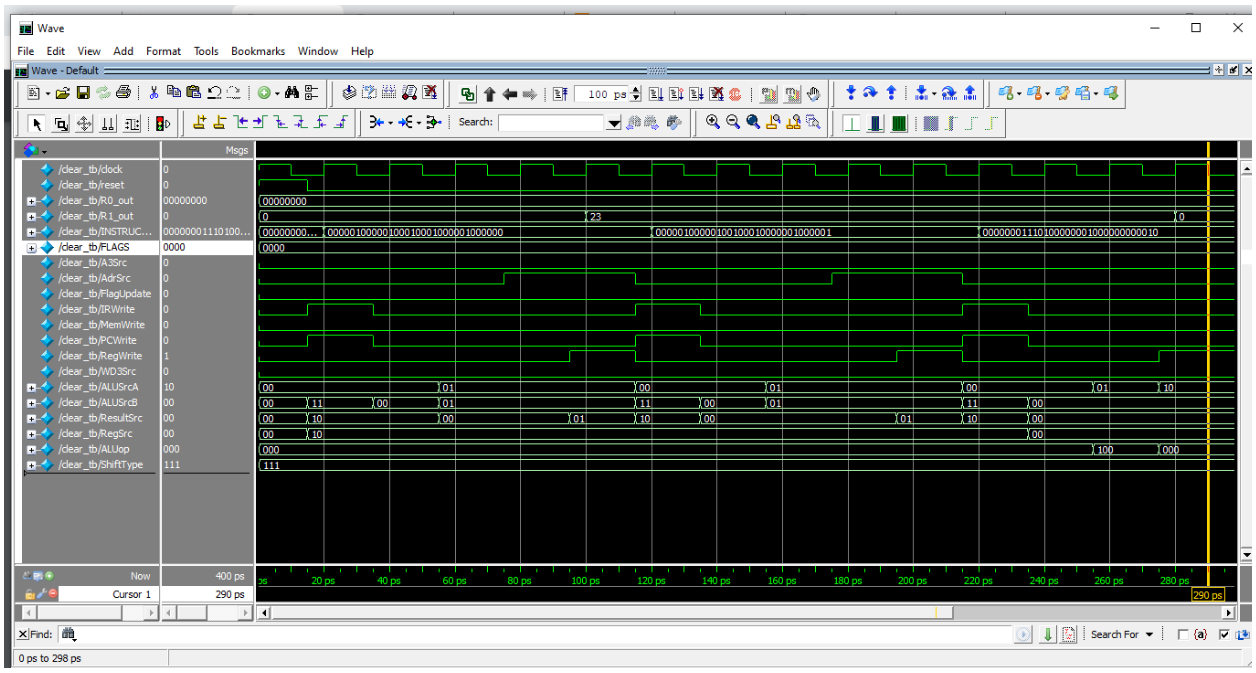
Figure 20 XOR Instruction Testbench

For the testbench, for which the result of is observable in figure 20, the instruction and data memory is initialized with the following code segment:

```
initial begin
        // INSTRUCTION SECTION
        mem[0] = 32'h04111040; // LDR R1, [R1, #64]
        mem[1] = 32'h04122041; // LDR R2, [R2, #65]
        mem[2] = 32'h00110002; // XOR R0, R1, R2


        // DATA SECTION
        mem[64] = 32'd23;
        mem[65] = 32'd24;
        mem[66] = 32'd25;
        mem[67] = 32'd26;
        mem[68] = 32'd27;
        mem[69] = 32'd28;
        mem[70] = 32'd29;
end
```

As in the previous data processing instructions' testbenches, we load R1 with the value 8'b00010111 (decimal 23), and R2 with 8'b00011000 (decimal 24). Result of the XOR operation must be 8'b00001111, and it can be observed in the output of R0 when all of the instructions are completed.

4.8 Clear Instruction



	Msgs
/clear_tb/clock	0
/clear_tb/reset	0
/clear_tb/R0_out	00000000
/clear_tb/R1_out	0
/clear_tb/INSTRUC...	00000001110100...
/clear_tb/FLAGS	0000
/clear_tb/A3Src	0
/clear_tb/AdrSrc	0
/clear_tb/FlagUpdate	0
/clear_tb/IRWrite	0
/clear_tb/MemWrite	0
/clear_tb/PCWrite	0
/clear_tb/RegWrite	1
/clear_tb/WD3Src	0
/clear_tb/ALUSrcA	10
/clear_tb/ALUSrcB	00
/clear_tb/ResultSrc	00
/clear_tb/RegSrc	00
/clear_tb/ALUop	000
/clear_tb/ShiftType	111

Now 400 ps
Cursor 1 290 ps
0 ps to 298 ps

Figure 21 Clear Instruction Testbench

For the testbench, for which the result of is observable in figure 20, the instruction and data memory is initialized with the following code segment:

```
initial begin
        // INSTRUCTION SECTION
        mem[0] = 32'h04111040; // LDR R1, [R1, #64]
        mem[1] = 32'h04122041; // LDR R2, [R2, #65]
        mem[2] = 32'h01D01002; // CLR, R1


        // DATA SECTION
        mem[64] = 32'd23;
        mem[65] = 32'd24;
        mem[66] = 32'd25;
        mem[67] = 32'd26;
        mem[68] = 32'd27;
        mem[69] = 32'd28;
        mem[70] = 32'd29;
end
```

We kept the first 2 load instructions which was necessary in previous data processing instructions. In this schema, we first load R0 and R1 with values 23 and 24 respectively. Then, we carry out the clear operation for R1. As can be seen in figure 21, R1 is cleared after the CLR instruction.

4.9 Shift Instructions

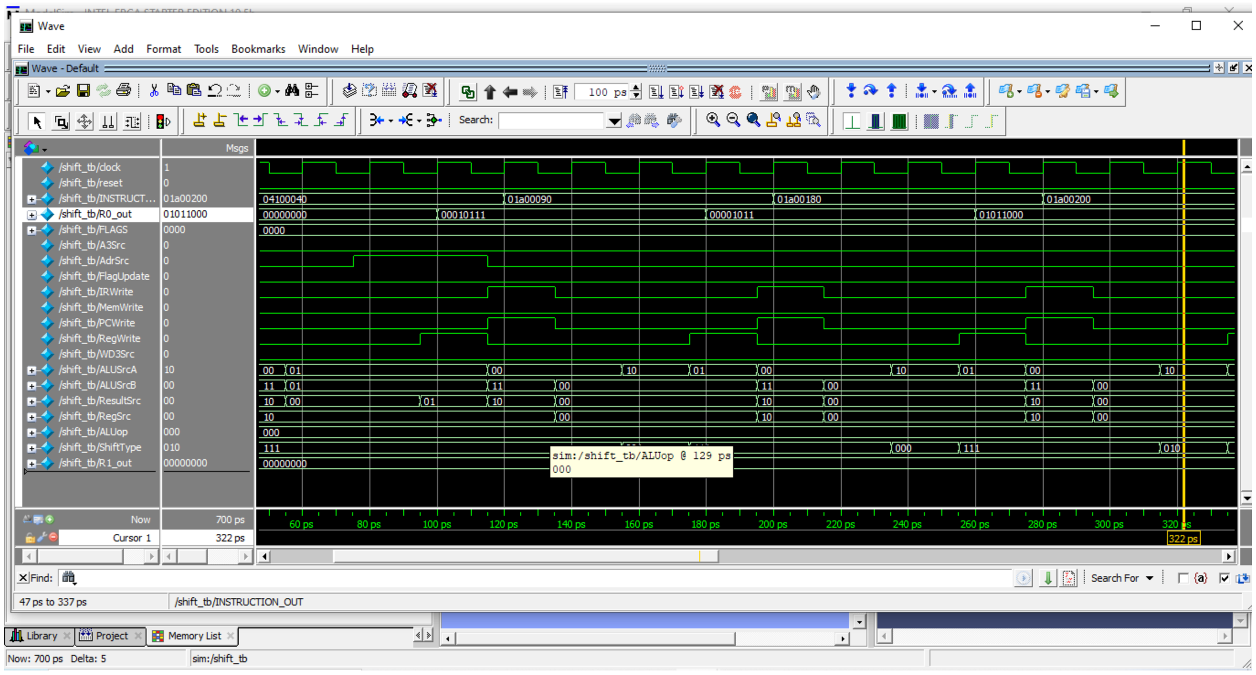


Figure 22 Logical Shift Right, Logical Shift Left, Arithmetic Shift Right Testbench

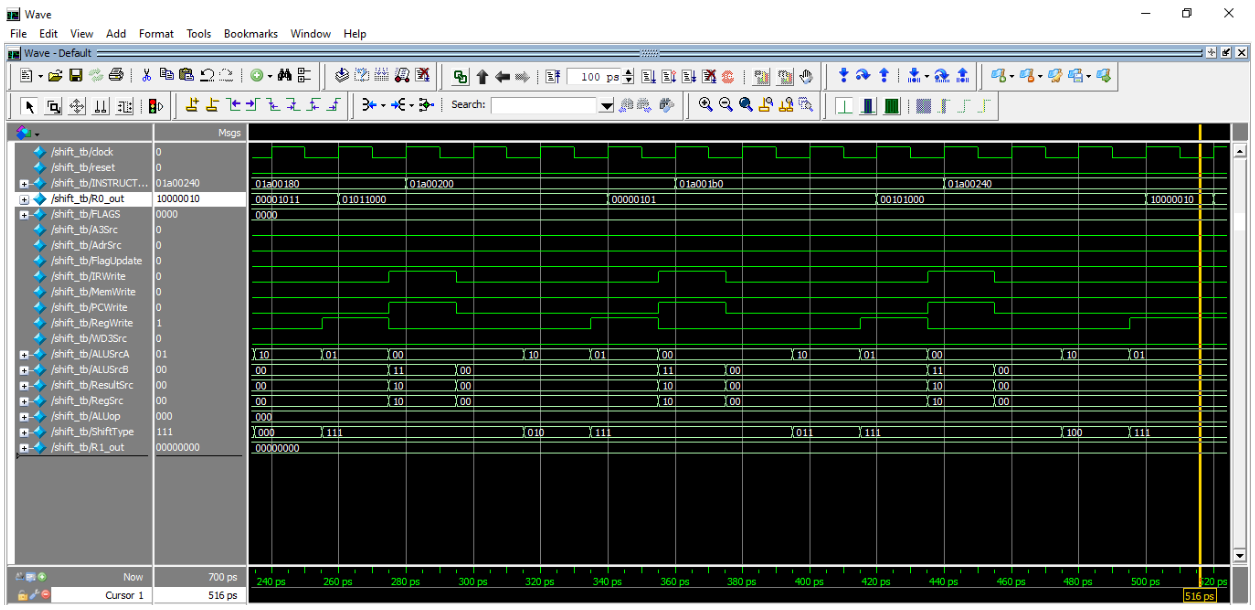


Figure 23 Rotate Left and Rotate Right

For the testbench, for which the result of is observable in figure 20, the instruction and data memory is initialized with the following code segment:

```
initial begin
        // INSTRUCTION SECTION
        mem[0] = 32'h04100040; // LDR R0, [R0, #64]
        mem[1] = 32'h01A00090; // LSR R0, R0, #1
        mem[2] = 32'h01A00180; // LSL R0, R0, #3
        mem[3] = 32'h01A00240; // ASR R0, R0, #4
        mem[4] = 32'h01A001B0; // ROL R0, R0, #3
        mem[5] = 32'h01A00240; // ROR R0, R0, #4

        // DATA SECTION
        mem[64] = 32'd23;
        mem[65] = 32'd24;
        mem[66] = 32'd25;
        mem[67] = 32'd26;
        mem[68] = 32'd27;
        mem[69] = 32'd28;
        mem[70] = 32'd29;
end
```

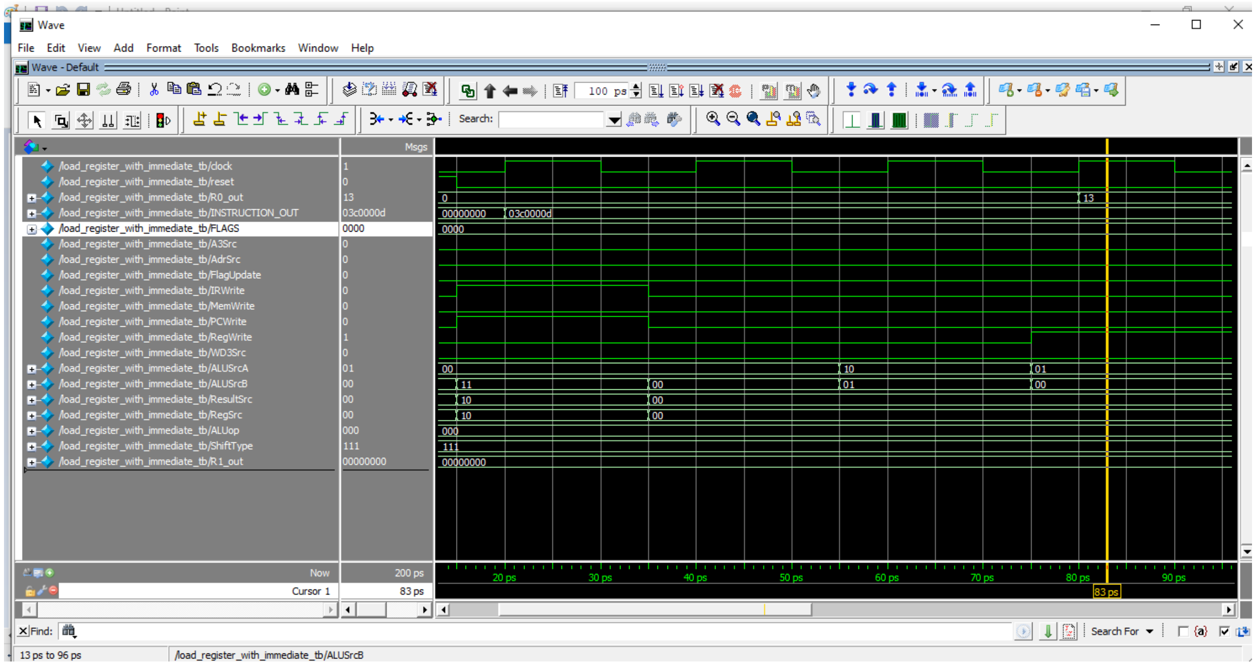
The given instructions in the comment section are executed in order. By tracking the value of the R0 output in figures 22 and 23, we validate that each shift operation works as intended.

4.10 Load Register with Immediate Instruction

The instruction "load immediate to register", which is a MOV instruction in ARM ISA, is given under the title of "Memory Operations" in the manual. For the datapath we designed, this instruction is easier to be executed as a data processing instruction with an immediate. The MOV instruction opcode in ARM ISA is used for shift instructions only in the ISA we designed, and this instruction must be separated from other immediate reference instructions since it uses different selects for ALU input MUXs.

```
initial begin
        // INSTRUCTION SECTION
        mem[0] = 32'h03C0000D; //MOV R0, #13

        // DATA SECTION
        mem[64] = 32'd23;
        mem[65] = 32'd24;
        mem[66] = 32'd25;
        mem[67] = 32'd26;
        mem[68] = 32'd27;
        mem[69] = 32'd28;
        mem[70] = 32'd29;
end
```

The instruction/data memory is initialized as above. The MOV instruction we written has opcode 1110, and in the immediate field it has decimal value of 13. Our instruction attempts to load R0 with immediate value of 13.



	Msgs
/load_register_with_immediate_tb/clock	1
/load_register_with_immediate_tb/reset	0
/load_register_with_immediate_tb/R0_out	13
/load_register_with_immediate_tb/INSTRUCTION_OUT	03c0000d
/load_register_with_immediate_tb/FLAGS	0000
/load_register_with_immediate_tb/A3Src	0
/load_register_with_immediate_tb/AdrSrc	0
/load_register_with_immediate_tb/FlagUpdate	0
/load_register_with_immediate_tb/IRWrite	0
/load_register_with_immediate_tb/MemWrite	0
/load_register_with_immediate_tb/PCWrite	0
/load_register_with_immediate_tb/RegWrite	1
/load_register_with_immediate_tb/WD3Src	0
/load_register_with_immediate_tb/ALUSrcA	01
/load_register_with_immediate_tb/ALUSrcB	00
/load_register_with_immediate_tb/ResultSrc	00
/load_register_with_immediate_tb/RegSrc	00
/load_register_with_immediate_tb/ALUop	000
/load_register_with_immediate_tb/ShiftType	111
/load_register_with_immediate_tb/R1_out	00000000

Now 200 ps
Cursor 1 83 ps
13 ps to 96 ps
/load_register_with_immediate_tb/ALUSrcB

Figure 24 Load Register with Immediate Value Testbench

Observing the output of R0 in figure 24, we can conclude that the instruction works as intended since it achieved to load R0 with 13.

4.11 Branch Instruction

In our ISA, branch target address calculation is different than the way that it is done in ARM ISA since we do not employ byte-addressable memory. The calculation is as follows:

$$BTA = 1 + offset$$

To show the operation of branch instruction, we execute a branch operation, and then set IRWrite control signal to 1 to deduce what the next PC became. In our schema, PC is 0 when branch instruction is to be executed. The offset given in the instruction is 5, meaning the target PC should be 6, according to the BTA calculation formula. After the B instruction is executed, IRWrite is set to 1, which enable us to deduce the value of the PC after a clock edge. The instruction/data memory initialization is as following:

```
initial begin
                    // INSTRUCTION SECTION
                    mem[0] = 32'hE8000005; // B
                    mem[1] = 32'h03C00001; // MOV R0, #1
                    mem[2] = 32'h03C00002; // MOV R0, #2
                    mem[3] = 32'h03C00003; // MOV R0, #3
                    mem[4] = 32'h03C00004; // MOV R0, #4
                    mem[5] = 32'h03C00005; // MOV R0, #5
                    mem[6] = 32'h03C00006; // MOV R0, #6
                    mem[7] = 32'h03C00007; // MOV R0, #7
                    mem[8] = 32'h03C00008; // MOV R0, #8

                    // DATA SECTION
                    mem[64] = 32'd23;
                    mem[65] = 32'd24;
                    mem[66] = 32'd25;
                    mem[67] = 32'd26;
                    mem[68] = 32'd27;
                    mem[69] = 32'd28;
                    mem[70] = 32'd29;
          end
```

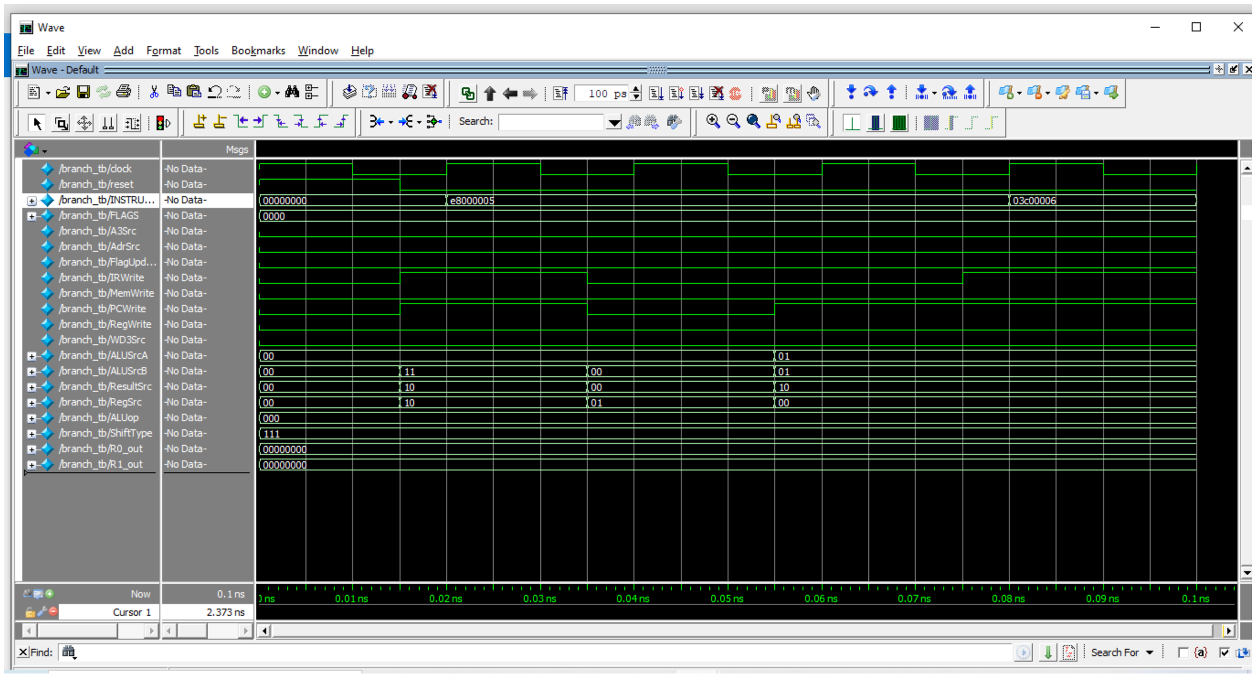
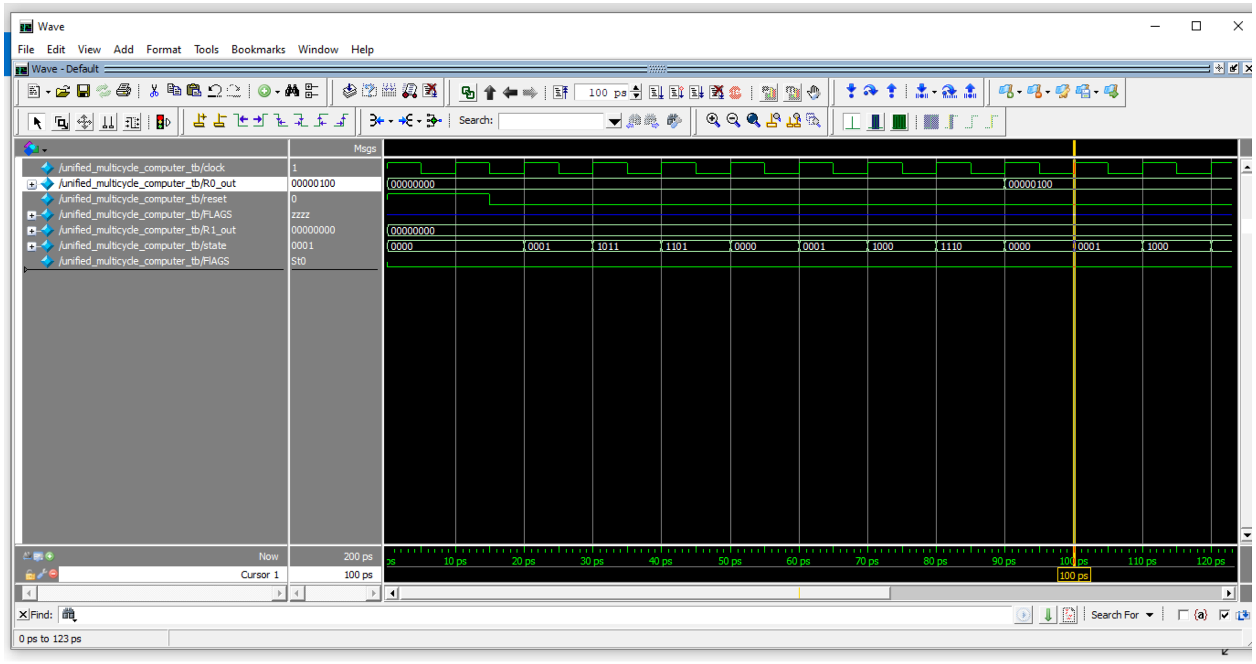


Figure 25 Branch Instruction Testbench

By observing the value of INSTRUCTION_OUT in the testbench provided in figure 25, we can conclude that PC is set to the correct value since INSTRUCTION_OUT became 32'h03C00006, meaning PC became 6, as intended.

4.12 Branch Indirect



Signal	Msgs
/unified_multicycle_computer_tb/clock	1
/unified_multicycle_computer_tb/R0_out	00000100
/unified_multicycle_computer_tb/reset	0
/unified_multicycle_computer_tb/FLAGS	zzzz
/unified_multicycle_computer_tb/R1_out	00000000
/unified_multicycle_computer_tb/state	0001
/unified_multicycle_computer_tb/FlAGS	St0

R0_out: 00000000 → 00000100
R1_out: 00000000
state: 0000, 0001, 1011, 1101, 0000, 0001, 1000, 1110, 0000, 0001, 1000

Now 200 ps; Cursor 1 100 ps; 0 ps to 123 ps

Figure 26 Branch Indirect Testbench

We wrote certain combination of instructions to determine whether the branch indirect operation works as intended. In our implementation, branch indirect instruction retrieves the data which is located in the address given in offset field of the branch instruction. Then, this data is directly for BTA. In the testbench, branch instruction offset field is set to 3. Meaning, the next instruction address after the branch will be PC+4. We wrote “MOV R0, #4” instruction to the address of the BTA. In summary, for the successful branch indirect operation, we should observe R0 output value to be 4, after the execution of another instruction after branch. Observing figure 26, we can conclude that branch indirect operation is successfully executed since the value 4 is observed after the execution of instruction after branch indirect. The instruction/data memory initialization is as following:

```
initial begin
        // INSTRUCTION SECTION
        mem[0] = 32'hEA000047; // BIND [71]
        mem[1] = 32'h03C00001; // MOV R0, #1
        mem[2] = 32'h03C00002; // MOV R0, #2
        mem[3] = 32'h03C00003; // MOV R0, #3
        mem[4] = 32'h03C00004; // MOV R0, #4
        mem[5] = 32'h03C00005; // MOV R0, #5
        mem[6] = 32'h03C00006; // MOV R0, #6
        mem[7] = 32'h03C00007; // MOV R0, #7
        mem[8] = 32'h03C00008; // MOV R0, #8

        // DATA SECTION
        mem[64] = 32'd23;
        mem[65] = 32'd24;
        mem[66] = 32'd25;
        mem[67] = 32'd26;
        mem[68] = 32'd27;
        mem[69] = 32'd28;
        mem[70] = 32'd29;
        mem[71] = 32'd4;
end
```

4.13 Branch with Link

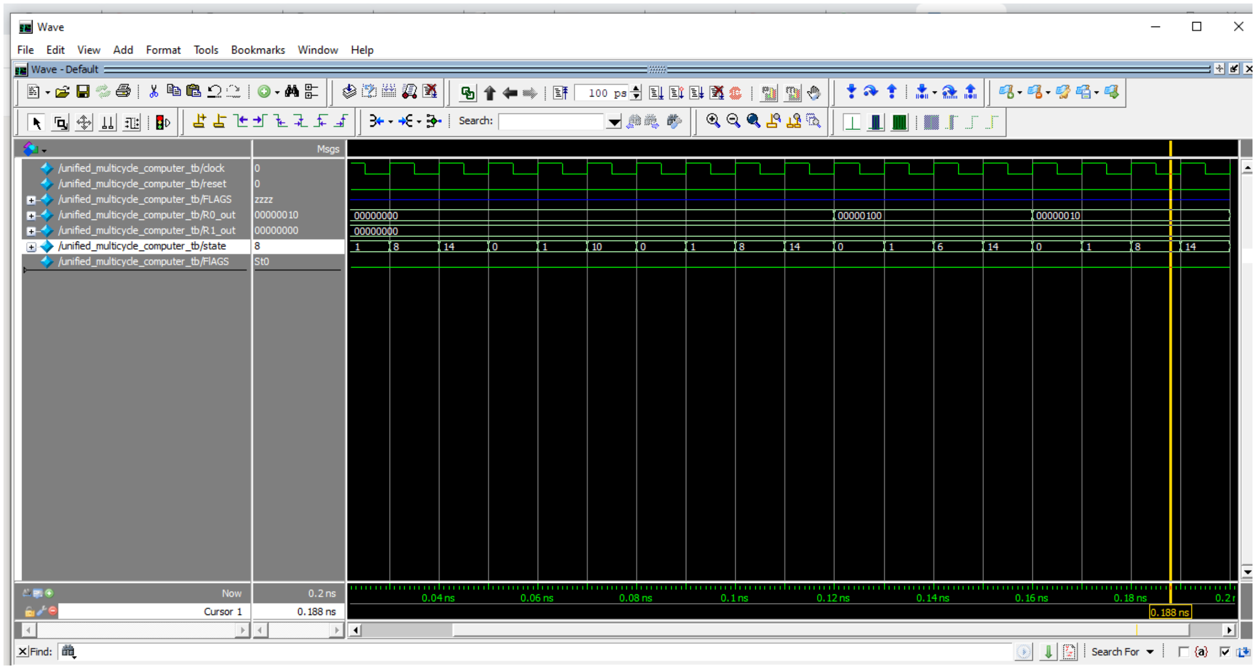

Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
100 ps
Msgs
/unified_multicycle_computer_tb/clock 0
/unified_multicycle_computer_tb/reset 0
/unified_multicycle_computer_tb/FLAGS zzzz
/unified_multicycle_computer_tb/R0_out 00000010
/unified_multicycle_computer_tb/R1_out 00000000
/unified_multicycle_computer_tb/state 8
/unified_multicycle_computer_tb/FlAGS St0
00000000
00000100
00000010
00000000
1 8 14 0 1 10 0 1 8 14 0 1 6 14 0 1 8 14
Now 0.2 ns
Cursor 1 0.188 ns
0.04 ns
0.06 ns
0.08 ns
0.1 ns
0.12 ns
0.14 ns
0.16 ns
0.18 ns
0.188 ns
Find:
Search For

Figure 27 Branch with Link Testbench

To be able to show both we are able to branch to the target address and save the current PC value in the link register, we created a plot. The data/instruction memory is initialized as following:

```
initial begin
        // INSTRUCTION SECTION
        mem[0] = 32'h03C00000; // MOV R0, #0
        mem[1] = 32'hE9000002; // BL #2
        mem[2] = 32'h03C00002; // MOV R0, #2
        mem[3] = 32'h03C00003; // MOV R0, #3
        mem[4] = 32'h03C00004; // MOV R0, #4
        mem[5] = 32'h00060006; // AND R0, R6, R6 (R0 <- R6)
        mem[6] = 32'h03C00006; // MOV R0, #6
        mem[7] = 32'h03C00007; // MOV R0, #7
        mem[8] = 32'h03C00008; // MOV R0, #8

        // DATA SECTION
        mem[64] = 32'd23;
        mem[65] = 32'd24;
        mem[66] = 32'd25;
        mem[67] = 32'd26;
        mem[68] = 32'd27;
        mem[69] = 32'd28;
        mem[70] = 32'd29;
        mem[71] = 32'd3;
end
```

We first load R0 with 0, which is only a fill instruction. Then, we perform branch with link where the offset is 2. Using the BTA calculation, we branch to the address 4, in which MOV R0, #4 instruction is stored. The next instruction performs this operation, and we can observe the R0 output to be 4. Meaning, we can branch successfully. To be able to determine whether we can store the value into link register, we perform the operation AND R0, R6, R6, which transfers the value of R6 to R0. So, in figure 27, in the last part of output of R0, we observe the value R6, which is 2. In LR, we store the PC+1 while the branch is taken. With this testbench, we validate that BL operation can be carried out with issue.

4.14 Conditional Branch Instructions

When the controller for multicycle computer is designed and unified with the datapath, there will be a control signal BranchTaken, which is the output of condition checker. In these testbenches, we provide the control signals externally, which means for branch instructions, we externally provide BranchTaken signal, so to speak. Due to this, conditional branches will be tested when the controller is implemented since it does not make any sense now to test them.

5. Controller Design

The datapath is validated to be operating the way we designed with ISA we created. The controller we are to design will consist of 6 sub-components. Namely, the state machine, ALU decoder, instruction decoder, state and instruction dependent control signals module, xPSR write enable module, and a condition checker. It is important to note that some of our control signals is dependent only on the instruction, where most of them also depend on the state. The main motivation behind this division is to separate those control signals to simplify the overall design, for which otherwise we would have a very complicated single controller module.

5.1 State Machine

The state machine ASM chart is provided in figure 13. After the realization of design in Verilog, the state machine RTL view results as it can be observed in figure 28.

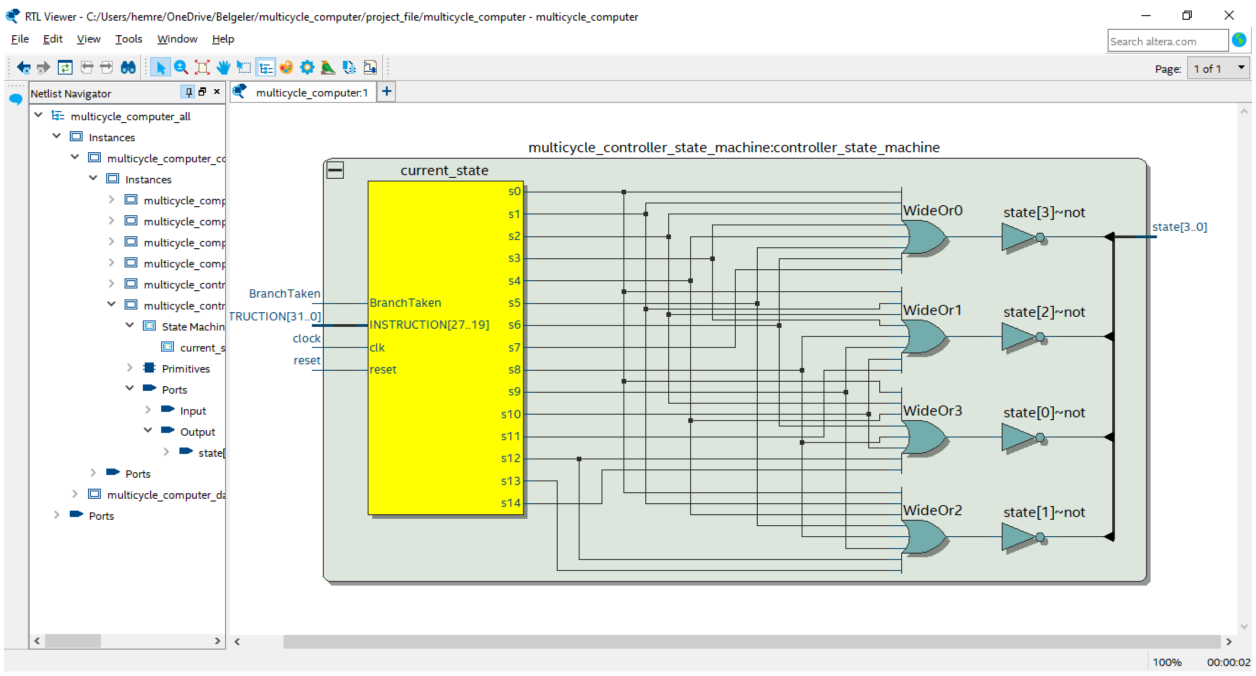


Figure 28 State Machine Module RTL View

The state transition diagram can be observed in figure 29. Observe figure 29 and figure 13 together for they are together comprehensive description of the state machine we designed for the multicycle computer.

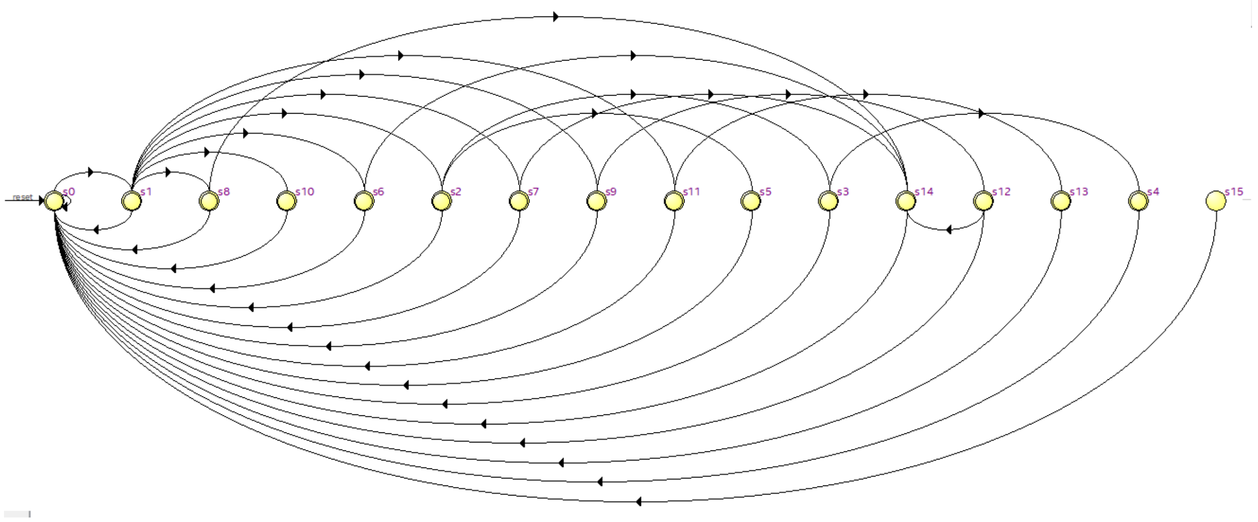


Figure 29 State Machine Transition Diagram

5.2 ALU Decoder

The ALU operation select is dependent both on the state and the instruction itself. It is implemented in an independent module to simplify the design. Inputs of this module are the current state and the instruction itself. It generates the control signal ALUop with these two inputs.

5.3 Instruction Decoder

The instruction decoder module generates the control signal RegSrc and assigns the ShiftType input to the barrel shifter module in the datapath. It only takes the instruction as input for these signals are only dependent on the instruction.

5.4 xPSR Write Enable Module

This module determines whether the instruction updates the program status flags (NZCV), which is controlled by the signal FlagUpdate. It generates this signal with current state and instruction information.

5.5 Controller Condition Checker

In the ISA, there are conditional branch instructions. In the condition checker module, we generate the BranchTaken signal, which is fed back to the state machine. If the branch is decided not to be taken, state machine goes to fetch stage from the decode stage of a branch instruction.

5.6 State and Instruction Dependent Control Signals Module

Rest of the signals we use in the datapath are generated by this module, which takes inputs of current state and the instruction itself. While most of the control signals are generated in this module, the other controller modules simplified the conditional statements in this module.

5.7 Controller RTL View

The controller RTL view can be found in figure 30.

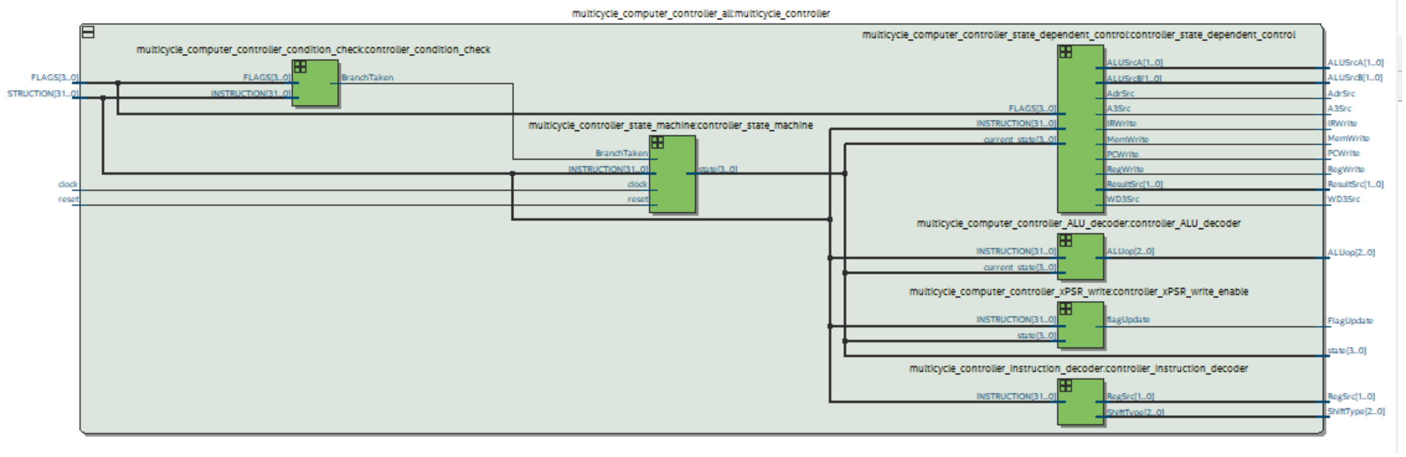


Figure 30 The Controller RTL View

After the design of controller module, the multicycle computer datapath and controller RTL is as it is highlighted in figure 31.

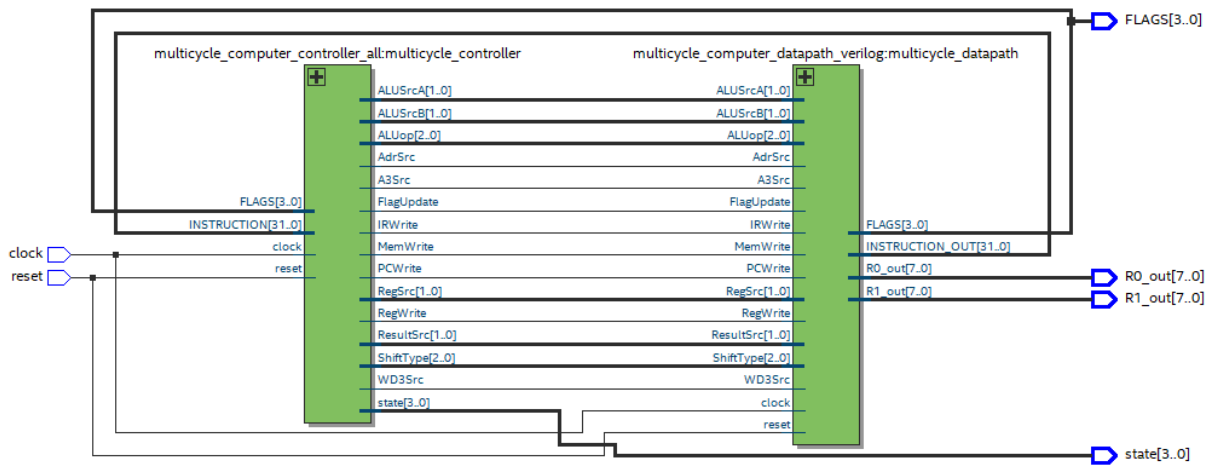
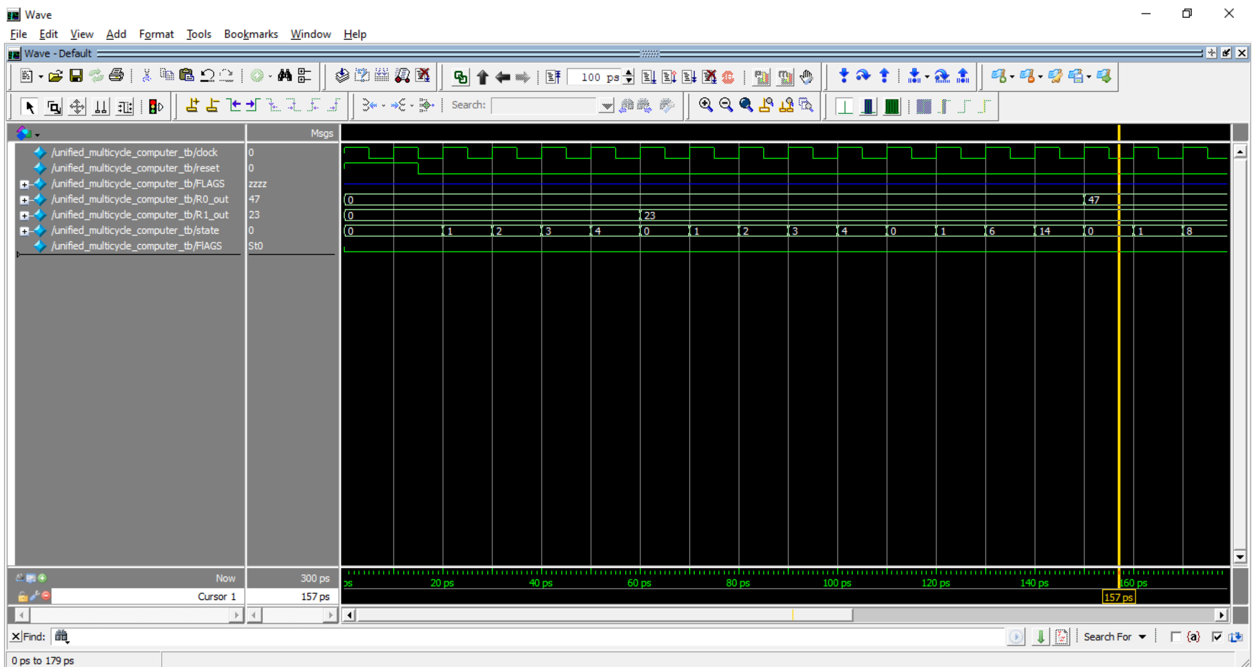


Figure 31 Multicycle Computer RTL View

6. Operation Verification of Unified Computer

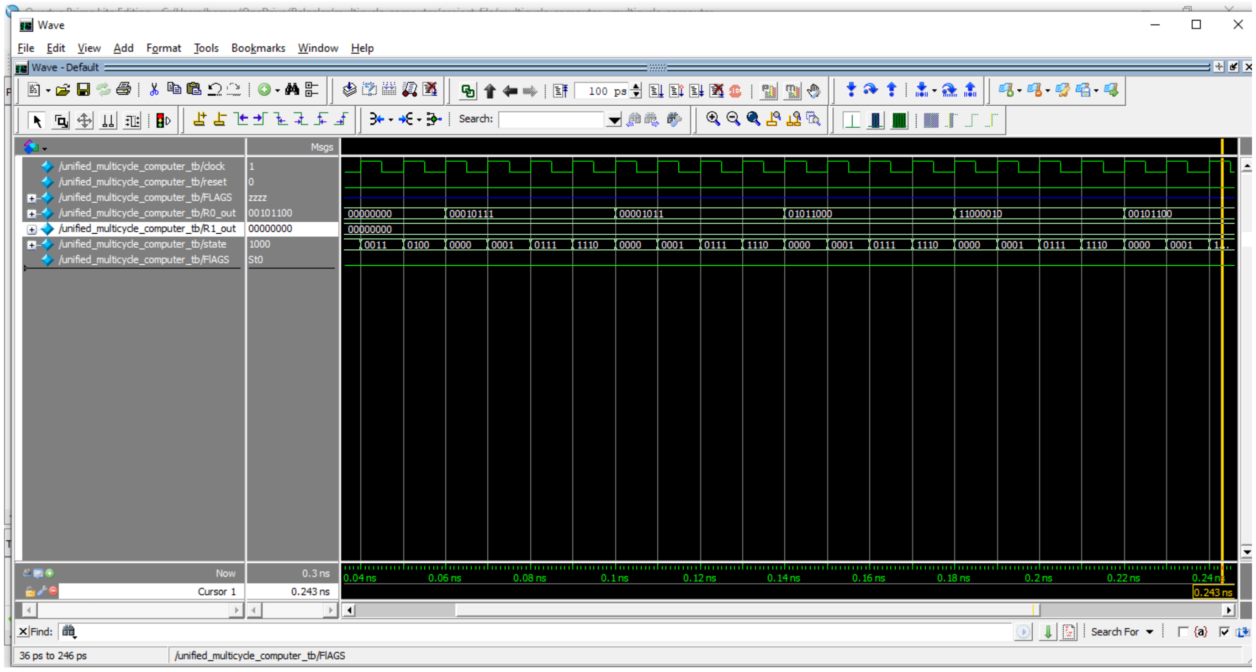
After unifying the datapath and the controller as highlighted in figure 31, some (not all) of the instructions are tested in testbench to verify the operation.



Signal	Msgs
/unified_multicycle_computer_tb/clock	0
/unified_multicycle_computer_tb/reset	0
/unified_multicycle_computer_tb/FLAGS	zzzz
/unified_multicycle_computer_tb/R0_out	47
/unified_multicycle_computer_tb/R1_out	23
/unified_multicycle_computer_tb/state	0
/unified_multicycle_computer_tb/FlAGS	St0

Now 300 ps; Cursor 1 157 ps; 0 ps to 179 ps

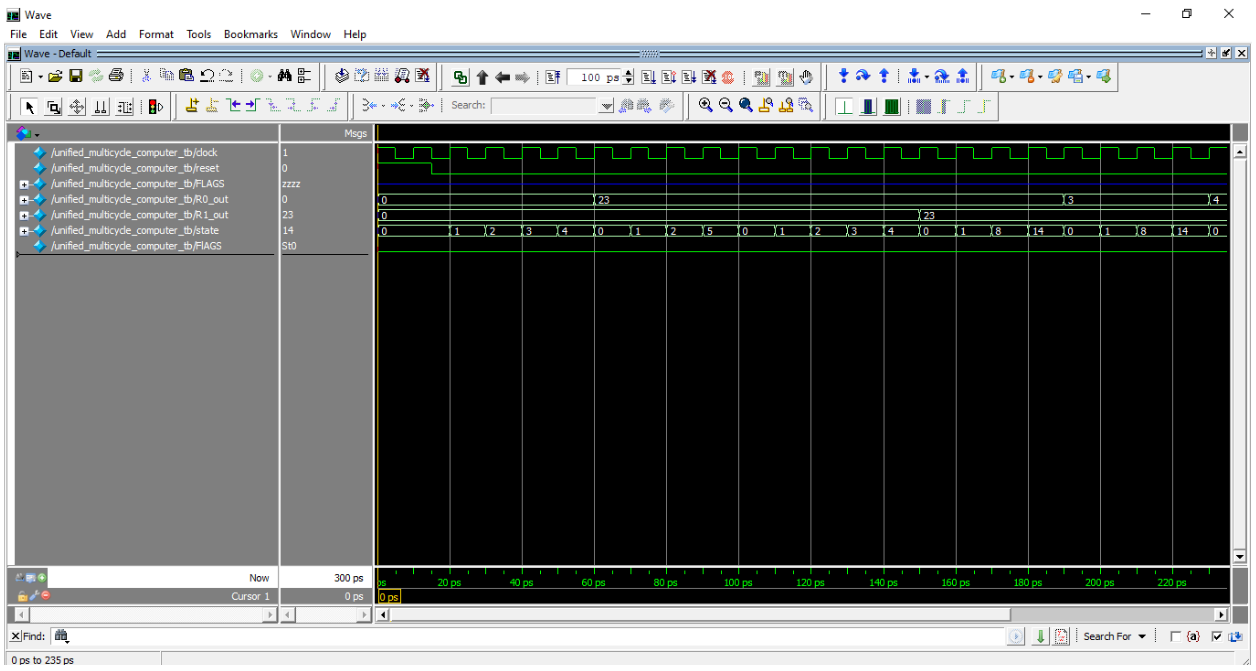
Figure 33 Unified Computer ADD Operation (R0 <- 23+24)



Signal	Msgs
/unified_multicycle_computer_tb/clock	1
/unified_multicycle_computer_tb/reset	0
/unified_multicycle_computer_tb/FLAGS	zzzz
/unified_multicycle_computer_tb/R0_out	00101100
/unified_multicycle_computer_tb/R1_out	00000000
/unified_multicycle_computer_tb/state	1000
/unified_multicycle_computer_tb/FlAGS	St0

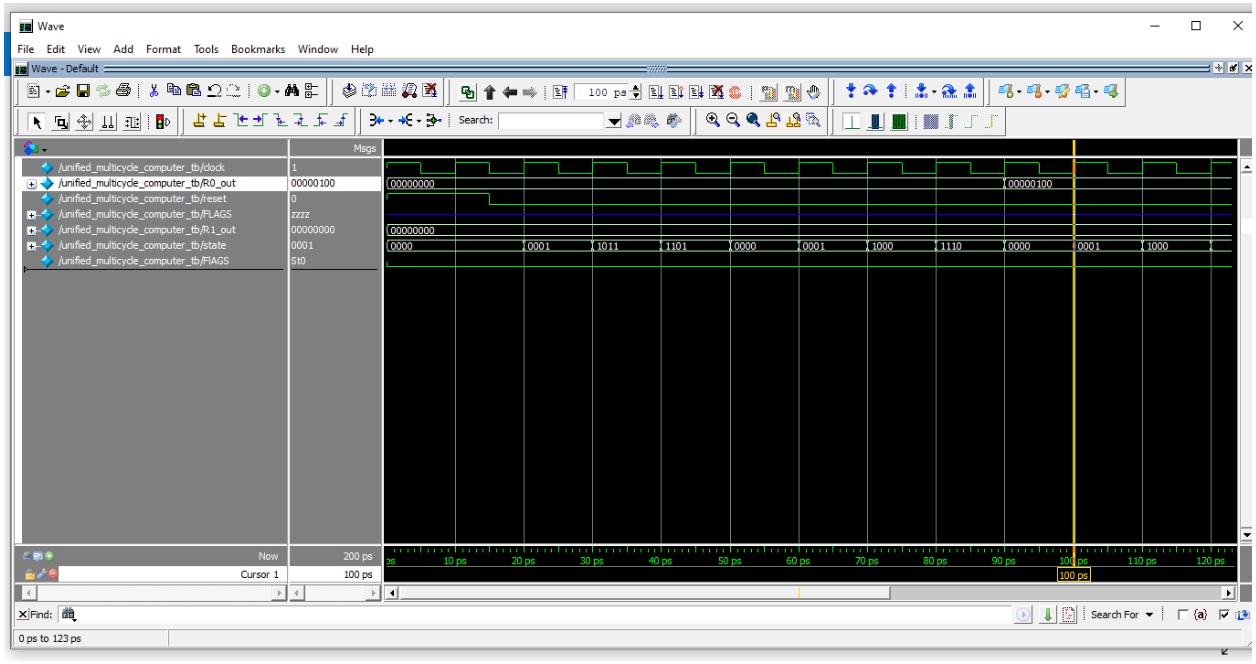
Now 0.3 ns; Cursor 1 0.243 ns; 36 ps to 246 ps

Figure 32 Unified Shift Instructions (R0<-23, LSR R0 #1, LSL R0, #3, ROL R0, #3, ROR R0, #4)



```
/unified_multicycle_computer_tb/clock      1
/unified_multicycle_computer_tb/reset      0
/unified_multicycle_computer_tb/FLAGS      zzzz
/unified_multicycle_computer_tb/R0_out     0
/unified_multicycle_computer_tb/R1_out     23
/unified_multicycle_computer_tb/state      14
/unified_multicycle_computer_tb/FlAGS      St0

Now       300 ps
Cursor 1  0 ps
0 ps to 235 ps
```

Figure 35 Unified Computer Store Operation (LDR R0, #23, STR R0, [65], LDR R1, [65])



```
/unified_multicycle_computer_tb/clock      1
/unified_multicycle_computer_tb/R0_out     00000100
/unified_multicycle_computer_tb/reset      0
/unified_multicycle_computer_tb/FLAGS      zzzz
/unified_multicycle_computer_tb/R1_out     00000000
/unified_multicycle_computer_tb/state      0001
/unified_multicycle_computer_tb/FlAGS      St0

Now       200 ps
Cursor 1  100 ps
0 ps to 123 ps
```

Figure 34 Unified Computer Branch Indirect (same in figure 26)

With figure 32-34, we verified the operation of the unified computer.

7. Instruction Mnemonics and Machine Codes

In the following table, we present all possible instructions and their machine code equivalent with the ISA we designed. Refer to figures 1,2,3,4, and 5 for the binary translation tables for each instructions.

Table 1 Instruction Mnemonics and their Machine Codes

Instruction Mnemonic	**RTL**	**Machine Code (HEX)**
ADD R0, R1, R2	R0 <- R1+R2	0x00810002
ADDIN R0, R1, [R2]	R0 <- R1+MEM[R2]	0x00890002
SUB R0, R1, R2	R0 <- R1-R2	0x00410002
SUBIN R0, R1, [R2]	R0 <- R1-MEM[R2]	0x00490002
AND R0, R1, R2	R0 <- R1&R2	0x00010002
ORR R0, R1, R2	R0 <- R1|R2	0x01810002
XOR R0, R1, R2	R0 <- (R1&R2')|(R1'^R2)	0x00110002
CLR R0	R0 <- 0	0x01D00002
ROL R0, R0, #3	R0 <- (R0<<3)|(R0>>(8-3))	0x01A001B0
ROR R0, R0, #4	R0 <- (R0<<(8-4))|(R0>>4)	0x01A00240
LSL R0, R0, #4	R0 <- R0<<4	0x01A00200
ASR R0, R0, #4	R0 <- R0>>>4	0x01A00240
LSR R0, R0, #1	R0 <- R0>>1	0x01A00090
B #5	PC <- PC+1+offset	0xE8000005
BL #2	PC <- PC+1+offset, LR<-PC+1	0xE9000002
BIND [71]	PC <- MEM[offset]	0xEA000047
BEQ #5	PC <- PC+1+offset if Z==1	0x08000005
BNE #5	PC <- PC+1+offset if Z==0	0x18000005
BCS #5	PC <- PC+1+offset if C==1	0x28000005
BCC	PC <- PC+1+offset if C==0	0x38000005
LDR R0, [R0, #64]	R0 <- MEM[R0+64]	0x04100040
STR R0, [R1, #65]	MEM[R1+65] <- R0	0x04010041
MOV R0, #8	R0 <- 8	0x03C00008

All these machine codes can be verified to be operational by initializing the instruction/data memory with them, which is done in testbench section of the datapath.

8. Validation Through Microprogrammed Control

8.1 2’s Complement

The following memory initialization is made to achieve the given task.

```
initial begin
                // INSTRUCTION SECTION
                mem[0] = 32'h03C00005; // MOV R0, #5
                mem[1] = 32'hE900001E; // BL #31

        //**********************************************************************
                        // 2's complement subroutine
                                // take R0 as the parameter and return its leave its complement to R0
                mem[32] = 32'h03C01000; // MOV R1, #0
                mem[33] = 32'h00410000; // SUB R0, R1, R0
                mem[34] = 32'h0401607E; // STR R6, [R1, #127] (MEM[127] <- LR)
                mem[35] = 32'hEA00007E; // BIND #127 (PC <- LR)
                                        // Last two operations(mem 34, 35) are for BX LR

        //**********************************************************************



                // DATA SECTION
                mem[64] = 32'd23;
                mem[65] = 32'd24;
                mem[66] = 32'd25;
                mem[67] = 32'd26;
                mem[68] = 32'd27;
                mem[69] = 32'd28;
                mem[70] = 32'd29;
                mem[71] = 32'd3;
        end
```

In the comment sections of the memory initialization, the code in our mnemonics is given. We took the complement of decimal 5, and all operation can be observed in the testbench result given in figure 36.

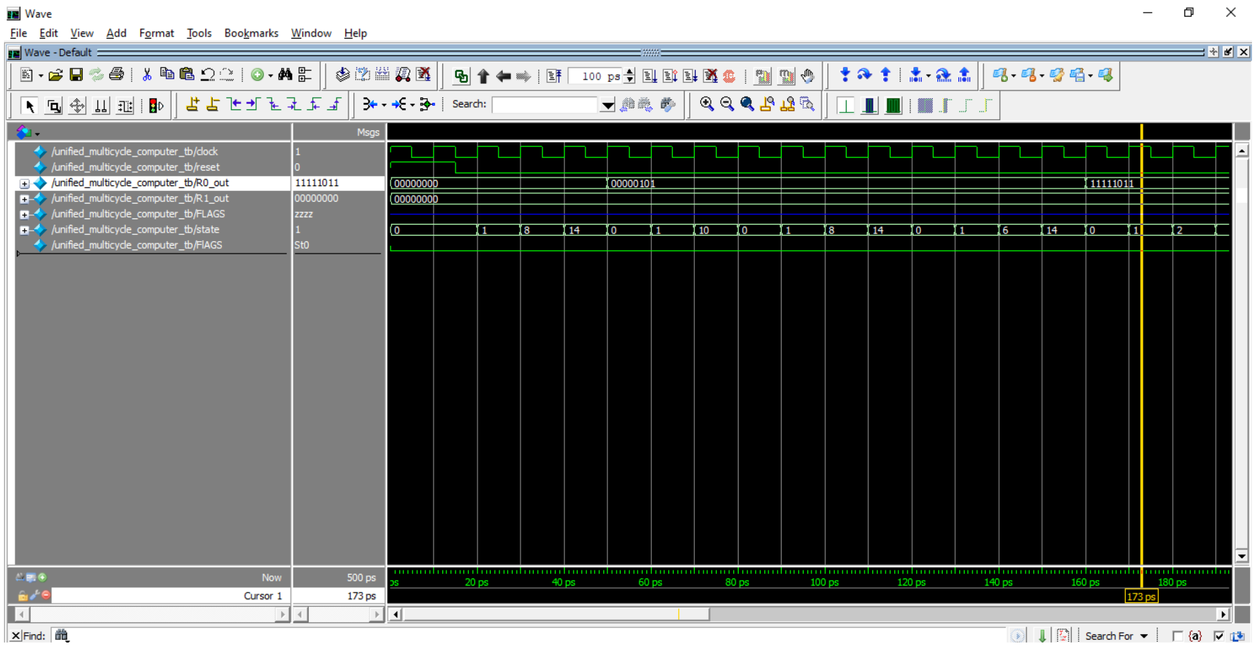

/unified_multicycle_computer_tb/clock 1
/unified_multicycle_computer_tb/reset 0
/unified_multicycle_computer_tb/R0_out 11111011
/unified_multicycle_computer_tb/R1_out 00000000
/unified_multicycle_computer_tb/FLAGS zzzz
/unified_multicycle_computer_tb/state 1
/unified_multicycle_computer_tb/FlAGS St0
Now 500 ps
Cursor 1 173 ps

Figure 36 Testbench Result of Task 1

8.2 Array Sum

The following memory initialization is made to achieve the second task specified in the manual. The assembly code in our mnemonics is provided in the comment section.

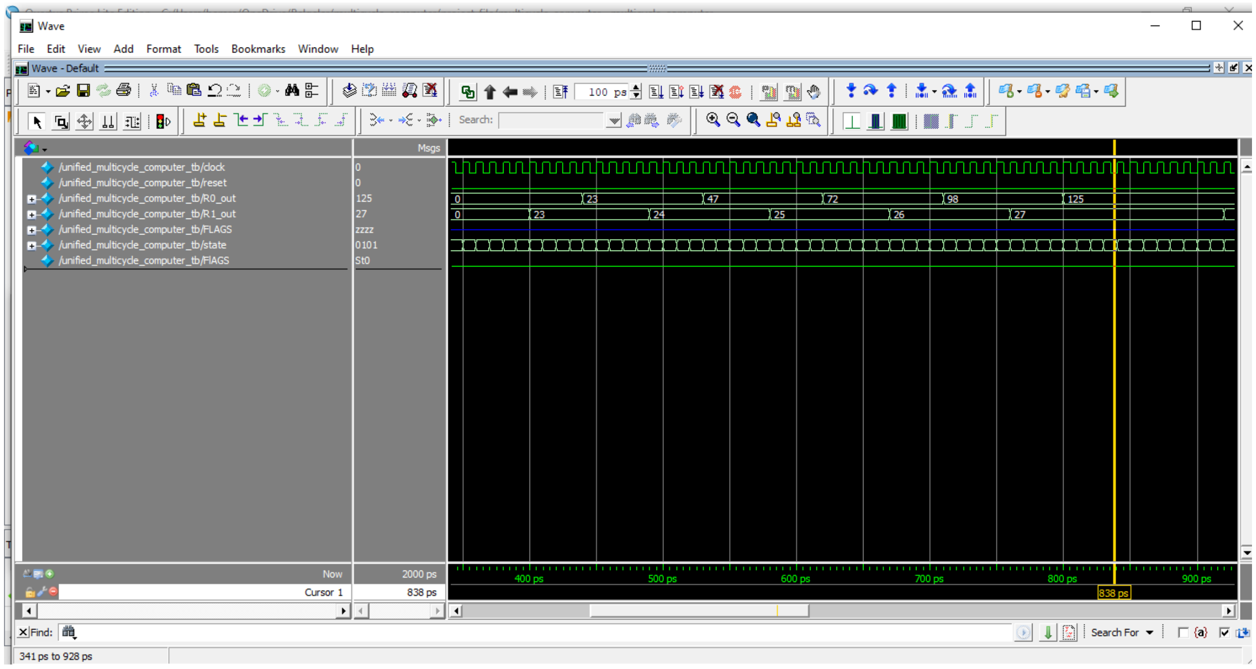

/unified_multicycle_computer_tb/clock 0
/unified_multicycle_computer_tb/reset 0
/unified_multicycle_computer_tb/R0_out 125
/unified_multicycle_computer_tb/R1_out 27
/unified_multicycle_computer_tb/FLAGS zzzz
/unified_multicycle_computer_tb/state 0101
/unified_multicycle_computer_tb/FlAGS St0
Now 2000 ps
Cursor 1 838 ps
341 ps to 928 ps

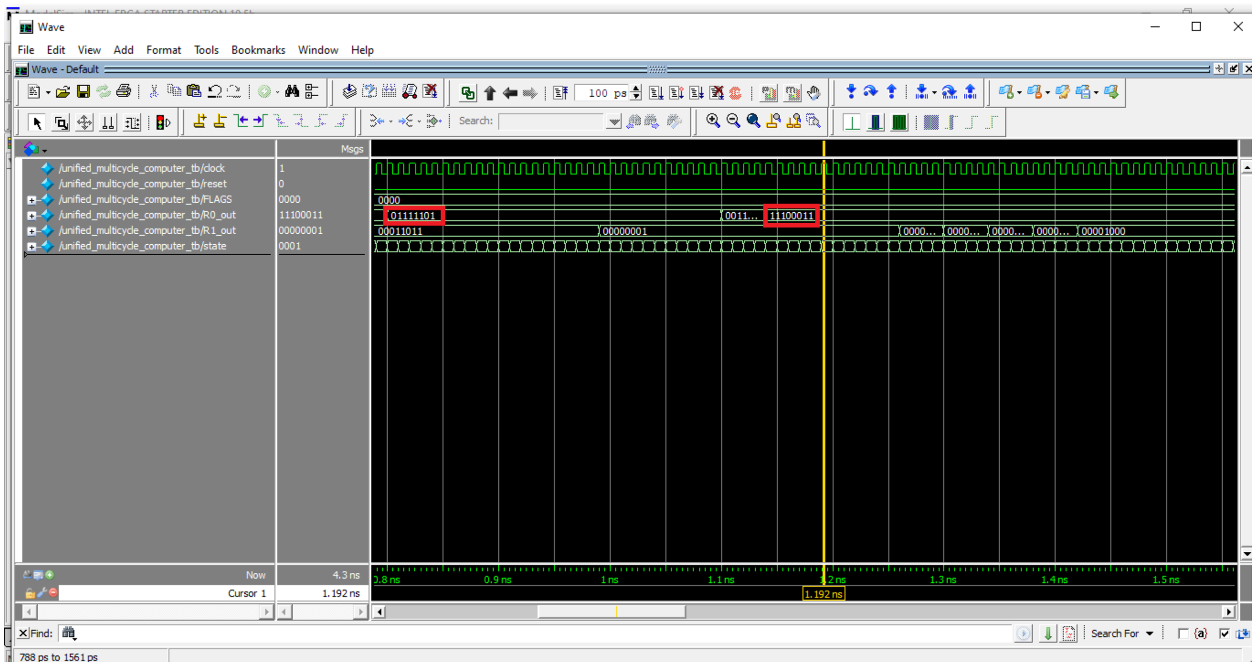
Figure 37 Testbench Result of Task 2

```
initial begin
            // INSTRUCTION SECTION
            mem[0] = 32'h03C00005; // MOV R0, #5
            mem[1] = 32'hE900001E; // BL #31 (2's complement)
            mem[2] = 32'hE9000021; // BL #35, #0


      //*******************************************************************
                  // 2's complement subroutine
                  // Refer to previous sub-section
      //*******************************************************************


      //*******************************************************************
                  // Array sum subroutine
                        // return the sum to R0
            mem[36] = 32'h03C00000; // MOV R0, #0
            mem[37] = 32'h03C02000; // MOV R2, #0
            mem[38] = 32'h04121040; // LDR R1, [R2, #64]
            mem[39] = 32'h00810000;  // ADD R0, R0, R1 (NUM 1)
            mem[40] = 32'h04121041; // LDR R1, [R2, #65]
            mem[41] = 32'h00810000; // ADD R0, R0, R1 (NUM2)
            mem[42] = 32'h04121042; // LDR R1, [R2, #66]
            mem[43] = 32'h00810000; // ADD R0, R0, R1 (NUM3)
            mem[44] = 32'h04121043; // LDR R1, [R2, #67]
            mem[45] = 32'h00810000; // ADD R0, R0, R1 (NUM4)
            mem[46] = 32'h04121044; // LDR R1, [R2, #68]
            mem[47] = 32'h00810000; // ADD R0, R0, R1 (NUM5)
            mem[48] = 32'h0402607E; // STR R6, [R2, #127] (MEM[127] <- LR)
            mem[49] = 32'hEA00007E; // BIND #127 (PC <- LR)

                              // Last two operations(mem 48, 49) are for BX LR

      //*******************************************************************

            // DATA SECTION
            mem[64] = 32'd23;
            mem[65] = 32'd24;
            mem[66] = 32'd25;
            mem[67] = 32'd26;
            mem[68] = 32'd27;
            mem[69] = 32'd28;
            mem[70] = 32'd29;
            mem[71] = 32'd3;
      end
```

The resulting sum (125 = 23+24+25+26+27) is obtained in the end of the subroutine as can be observed in figure 37.

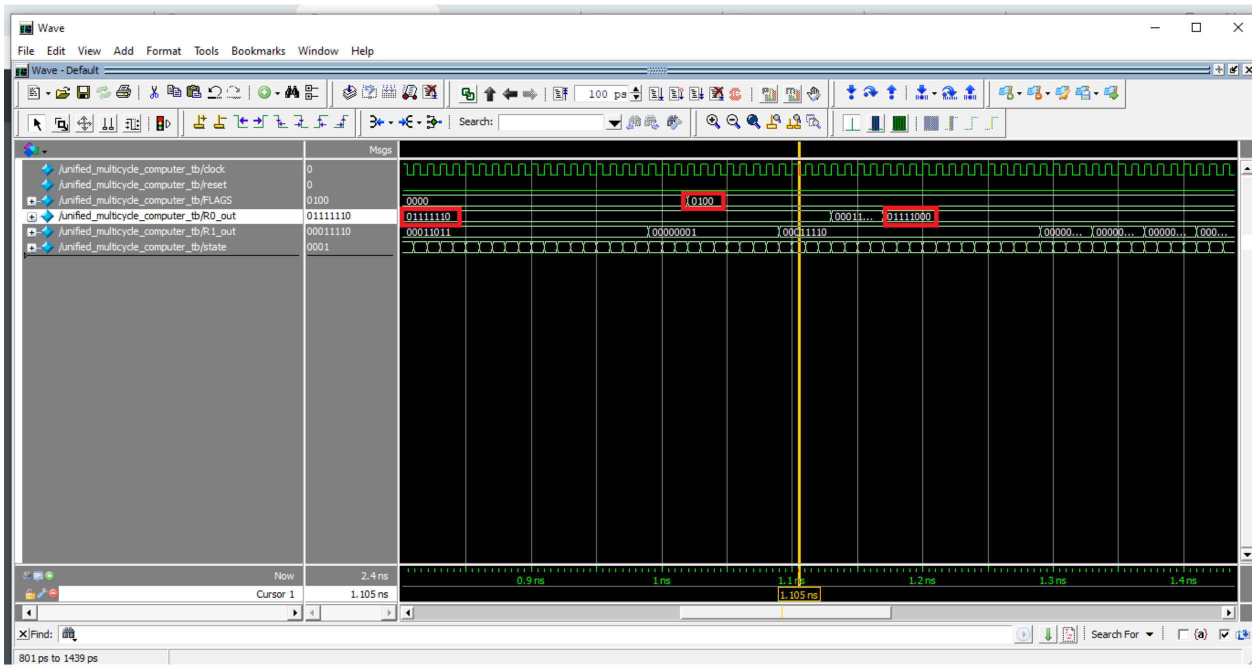
8.3 Evennes-Oddidty Check

The following memory initialization is made to achieve the required task. The assembly code with mnemonics of our ISA is presented in the comments section.



```
/unified_multicycle_computer_tb/clock    1
/unified_multicycle_computer_tb/reset    0
/unified_multicycle_computer_tb/FLAGS    0000
/unified_multicycle_computer_tb/R0_out   11100011
/unified_multicycle_computer_tb/R1_out   00000001
/unified_multicycle_computer_tb/state    0001
Now      4.3 ns
Cursor 1 1.192 ns
```

Figure 39 Testbench Result of Task 3 (Odd Case)



```
/unified_multicycle_computer_tb/clock    0
/unified_multicycle_computer_tb/reset    0
/unified_multicycle_computer_tb/FLAGS    0100
/unified_multicycle_computer_tb/R0_out   01111110
/unified_multicycle_computer_tb/R1_out   00011110
/unified_multicycle_computer_tb/state    0001
Now      2.4 ns
Cursor 1 1.105 ns
```

Figure 38 Testbench Result of Task 3 (Even Case)

```
initial begin
        // INSTRUCTION SECTION
        mem[0] = 32'h03C00005; // MOV R0, #5
        mem[1] = 32'hE900001E; // BL #30 (2's complement)
        mem[2] = 32'hE9000021; // BL #33 (Array sum)
        mem[3] = 32'hE900002E; // BL #46 (Evennes-oddity check)
        mem[4] = 32'h03C01004; // MOV R1, #0
        mem[5] = 32'h03C01005; // MOV R1, #0
        mem[6] = 32'h03C01006; // MOV R1, #0
        mem[7] = 32'h03C01007; // MOV R1, #0
        mem[8] = 32'h03C01008; // MOV R1, #0



//***********************************************************************
                // 2's complement subroutine
                // refer to sub-section 1
//***********************************************************************
//***********************************************************************
                // Array sum subroutine
                // refer to sub-section 2
//***********************************************************************
                // Evennes-oddity check subroutine
                        //      R0 has the value to be checked whether it is even or not
        mem[50] = 32'h03C02000; // MOV R2, #0
        mem[51] = 32'h03C01001; // MOV R1, #1
        mem[52] = 32'h00113000; // ANDS R3, R1, R0
        mem[53] = 32'h18000005; // BNE #5
        mem[54] = 32'h03C0101E; // MOV R1, #(0001.1110)
        mem[55] = 32'h00010000; // AND R0, R0, R1
        mem[56] = 32'h01A00100; // LSL R0, R0, #2
        mem[57] = 32'h0402607E; // STR R6, [R2, #127] (MEM[127] <- LR)
        mem[58] = 32'hEA00007E; // BIND #127 (MEM[127] <- LR)
        mem[59] = 32'h01A00090; // LSR R0, R0, #1
        mem[60] = 32'h01A00240; // ROR R0, R0, #4
        mem[61] = 32'h0402607E; // STR R6, [R2, #127] (MEM[127] <- LR)
        mem[62] = 32'hEA00007E; // BIND #127 (PC <- LR)
                                // Last two operations(mem 48, 49) are for BX LR
//***********************************************************************
        // DATA SECTION
        mem[64] = 32'd23;
        mem[65] = 32'd24;
        mem[66] = 32'd25;
        mem[67] = 32'd26;
        mem[68] = 32'd27;
        mem[69] = 32'd28;
        mem[70] = 32'd29;
        mem[71] = 32'd3;
end
```

With this testbench, we also validated the operation of conditional branches for we employed a conditional branch BNE in the program we have written.

9. Final Remarks

9.1 Bonus Operations

The bonus operations given in manual, namely, the indirect addition and subtraction are implanted in the ISA designed. Testbench results can be found in figures 40 and 41.

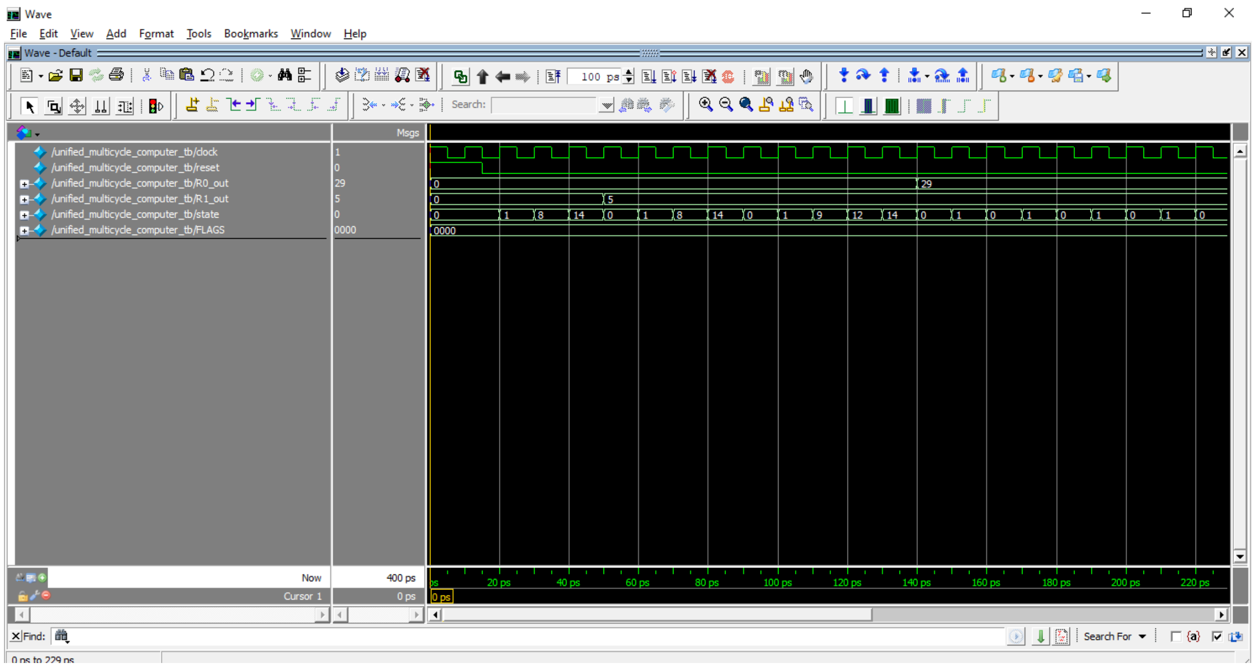

Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
/unified_multicycle_computer_tb/clock 1
/unified_multicycle_computer_tb/reset 0
/unified_multicycle_computer_tb/R0_out 29
/unified_multicycle_computer_tb/R1_out 5
/unified_multicycle_computer_tb/state 0
/unified_multicycle_computer_tb/FLAGS 0000
0 29
0 5
0 1 8 14 0 1 8 14 0 1 9 12 14 0 1 0 1 0 1 0 1 0
0000
Now 400 ps
Cursor 1 0 ps
0 ps
20 ps 40 ps 60 ps 80 ps 100 ps 120 ps 140 ps 160 ps 180 ps 200 ps 220 ps
Find:
Search For
0 ps to 229 ps

Figure 40 ADDIN Instruction Testbench

For ADDIN instruction, the following memory initialization is made.

```
initial begin
        // INSTRUCTION SECTION
        mem[0] = 32'h03C01005; // MOV R1, #5
        mem[1] = 32'h03C02040; // MOV R2, #64
        mem[2] = 32'h00890002; // ADDIN R0, R1, [R2]

        // DATA SECTION
        mem[64] = 32'd24;
end
```

We add the values present in R1 (5) and in the memory address 64. We write the result in R0, which is the correct value of decimal 29.

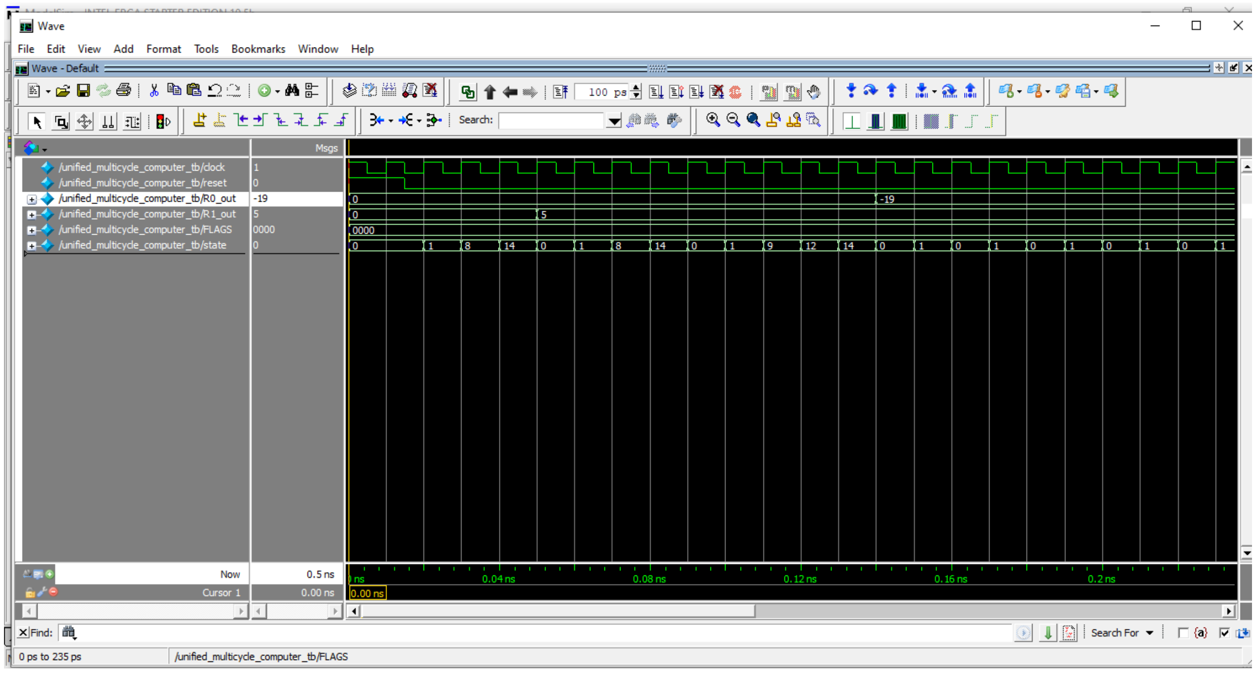

Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Msgs
/unified_multicycle_computer_tb/clock 1
/unified_multicycle_computer_tb/reset 0
/unified_multicycle_computer_tb/R0_out -19
/unified_multicycle_computer_tb/R1_out 5
/unified_multicycle_computer_tb/FLAGS 0000
/unified_multicycle_computer_tb/state 0
Now 0.5 ns
Cursor 1 0.00 ns
0.04 ns
0.08 ns
0.12 ns
0.16 ns
0.2 ns
Find:
Search For
0 ps to 235 ps
/unified_multicycle_computer_tb/FLAGS

Figure 41 SUBIN Instruction Testbench

For SUBIN instruction, the following memory initialization is made.

```
initial begin
        // INSTRUCTION SECTION
        mem[0] = 32'h03C01005; // MOV R1, #5
        mem[1] = 32'h03C02040; // MOV R2, #64
        mem[2] = 32'h00490002; // SUBIN R0, R1, [R2]

        // DATA SECTION
        mem[64] = 32'd24;
end
```

We subtract the value present in memory location 64 (24) from the value present in R1 (5). The result (-19) is written to R0, which can be seen in figure 41.

9.2 ISA Justification

We used 32-bit instructions to be able to extend our capabilities with ISA beyond what we are given in the manual. Additional to the manual, we are able to perform data processing instruction with immediate value, and all conditional branch instruction specified in ARM ISA. With this decision of keeping the instruction length 32-bit, instead of shorter possible one, we also simplified the controller unit since fields in the instructions can be used minimally to determine them with minimum number of if else statements. We can also add other data processing instruction since the opcode field is not exhausted with the instructions at hand, and also add another class of instructions aside from memory, data-processing, and branch instructions since the op field in also not exhausted. In summary, keeping the instruction length longer the optimum case (16-bit instructions might suffice), we had elasticity in adding new operations and simplified the controller unit.