Ethan Chung

Student ID: 2160550

Electronic Engineering

Digital Assignment

Summary:

This report goes over the VHDL code designed to run like a microprocessor containing three distinct parts. These are the instruction stream, the memory, and the ALU. The aim was to utilise VHDL code to implement a working 32-bit microprocessor that runs a given set of instructions. The microprocessor breaks down instructions that are given to it into 3 5-bit memory addresses and a 6-bit opcode. The ALU successfully runs the set instructions and outputs the correct answer for the given instruction set of 18EB5. The ALU achieves this by using no-ops in the instruction stream that allow any registers that are being accessed to have the stored value updated if needed.

Table of Contents:

Introduction:

The given task was to code a working microprocessor using VHDL that could run a given set of instructions. The parts of the microprocessor that were required are as follows: the ALU, the memory with write-back ability, the decoder, and the instruction stream. During the coding, the microprocessor was designed to have the structure shown in figure 1.

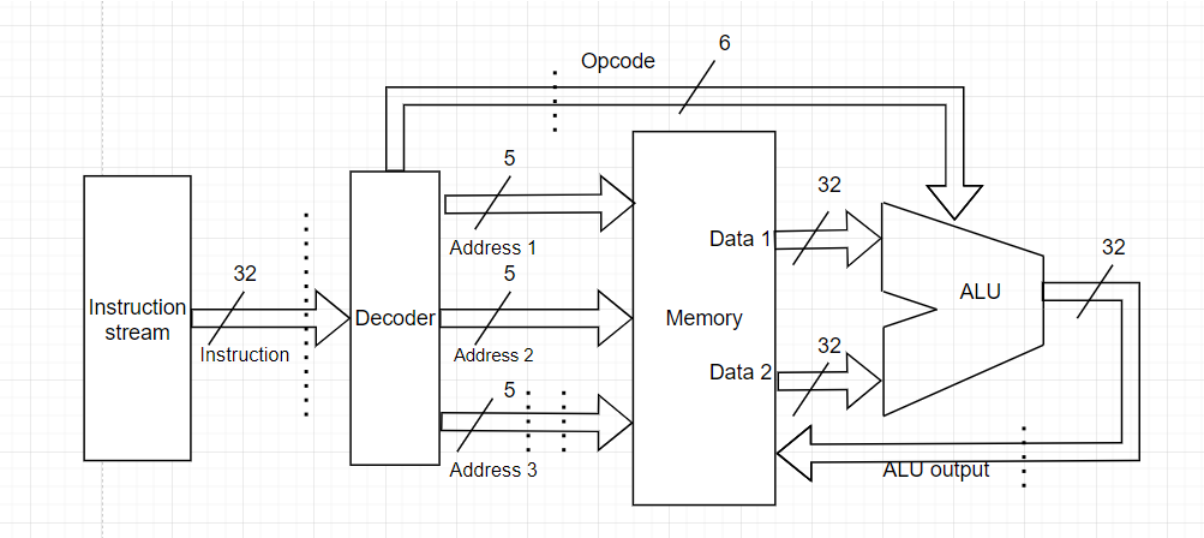

Figure 1: Microprocessor structure implemented through code

The dotted lines represent registers in the pipeline. These are added so that all data that is used at either the ALU or memory arrive at the same time and remove any wrong instruction readings.

Design Functionality:

The instruction set given was to do the following instructions with the final value being stored in memory register 31 (r31):

r2+r3+r4+r5+r6-r7-r8-r9-r10-r11+r12+r13+r14+r15+r16-r17-r18-r19-r20-r21+r22+r23+r24+r25+r26-r27-r28-r29-r30

The microprocessor can complete this instruction set and output the answer of 18EB5 in 885 ns at a clock frequency of 100 GHz.

The coded microprocessor can do other operations other than add and subtract. These are abs(a), - a, abs(b), - b, a OR b, NOT a, NOT b, a AND b, and a XOR b, where a and b are the inputs to the ALU. The microprocessor also contains a zero and negative flag that is raised when a value of zero is outputted or if the output is a negative value. It also can decode a 32-bit instruction into a 6-bit opcode and 3 separate 5-bit addresses. Two of the addresses are used to access memory locations in the memory and the third is used to define which memory location will be written back to. The other 11 bits are discarded and are not read. The microprocessor can recall up to two values from the attached memory and write to all addresses apart from address 0, with memory register 0 acting as a constant 0 value that cannot be overwritten. The microprocessor structure has been pipelined with an internal clock that allows the flow of each register along the pipeline without issues as all parts of the microprocessor are synced using the same clock. The microprocessor also has a valid signal where it can check whether the opcode is valid or if the write-back register is zero. The microprocessor overwrites the output value of the ALU if the write-back register is set to zero, it sets the output as zero and saves it in register zero. In this way, the register is always zero. It also contains an enable signal that would be used to

enable and disable the flow of the instruction stream, however, has not been fully utilised in the microprocessor due to issues with getting the instruction stream to continue.

Main Challenges:

The main challenges that appeared in this assignment were:

Setting up r0 so anything being written to it, would automatically be changed to zero, as well as any invalid opcodes having the output value changed to 0 and is written back to r0

Pipelining the microprocessor and getting them to line up, solving data hazards within the instruction stream

Getting the microprocessor to pause if an instruction contains a register that will have a value written back to it within the next two cycles, i.e., dealing with the data hazards caused by the instruction set.

How Challenges Were Solved:

To solve the problem of anything being written to r0 would be set to 0, a signal was used in the memory that would either accept the data from the output of the ALU or be set to 0 depending on if the signal labelled valid was a 1 or 0. This allowed the calculation to be wiped if it was trying to be written back to r0. The valid signal was set up so that if any opcode that was unused or if the write-back location was set to r0, then the output from the ALU was changed to 0 and written back to register 0.

For the pipelining, I draw out a diagram of the system and looking at each part, drew on where different buffers were. This allowed me to visualise where the buffers were and where each piece of data was at each stage. This was then rearranged so that the pipelining of the system lined up.

To allow time for the requested memory location's value to be updated, the code was reordered to reduce the time where the microprocessor was idling. Also, no-ops were introduced into the instruction stream to allow time before the next memory location is accessed.

The issue of the data hazards in the instruction stream where if a following instruction wanted to access the same registers that it was going to write back to was not solved using the optimum solution. However, part of the solution was implemented, this was the enable signal. This would allow the instruction stream to be enabled and disabled to allow for the required registers to be written back to before allowing the instruction stream to flow again. The main issue faced here was that the enable signal would change to 0 but the instruction stream would carry on for a few cycles before pausing. After pausing, it was not possible to get the instruction stream to start back up, so this idea was left out of the design of the microprocessor. Instead, two no-ops were introduced between the instructions, and the instructions were re-ordered so that the instructions alternated between instructions that dealt with adding register values and instructions that subtracted register values.

Proof of Working:

```
-- Start of main body of code
architecture dataflow of instruction_stream is
begin

      process is
       begin


       --wait until the next rising edge of the clock and for enable to be 1 to proceed
       wait until rising_edge(clk) and enable = '1';
       instruct <= B"001010_00010_00011_00001_00000000000"; -- r2+r3 goes into r1

       wait until rising_edge(clk) and enable = '1';  -- no-op
       wait until rising_edge(clk) and enable = '1';  -- no-op


       wait until rising_edge(clk) and enable = '1';
       instruct <= B"001010_00111_01000_11111_00000000000";-- r7+r8 goes into   r31
```

These wait statements act as the no-ops in the instruction stream

Figure 2: Code showing how no-ops are represented

Figure 1 shows how an instruction was written for the microprocessor to deal with. It also shows how the no-ops were shown and used within the instruction set. As can be seen in the figure, there first 6 bits are read as the opcode, the next set of 5 bits are read as memory location 1, the 5 bits after are read as memory location 2, the final 5-bit segment is read as to which memory location the output should be stored for the given instructions. The remaining 11-bits are unused so for ease of reading and to reduce confusion, have all been set as 0. The main instruction set was separated into add instructions and subtract instructions. All the addition instructions are added together into register 1 and all subtract instructions are added together into register 31. However, any register that has been written needs three cycles before it can be accessed with the correct value stored. If the register is accessed within the next cycle, a data hazard occurs as the wrong data will be used in the next calculation. The above figure shows how the no-ops have been added to the instruction stream. This means that each instruction takes 3 clock cycles to be completed which is shown in the figure below. The instruction set has also been rewritten to reduce the number of no-ops needed. The new instruction stream now is r2+r3-r7-r8+r4-r9+r5-r10+r6-r11+r12-r17+r13-r18+r14-r19+r15-r20+r16-r21+r22-r27+r23-r28+r24-r29+r26-r27+r30. Then the final instruction is r1-r31 to give the same result as the instruction set given. This removes the data hazards as the no-ops and instruction rearrangement allow for enough time for the register to update so it holds the correct value.

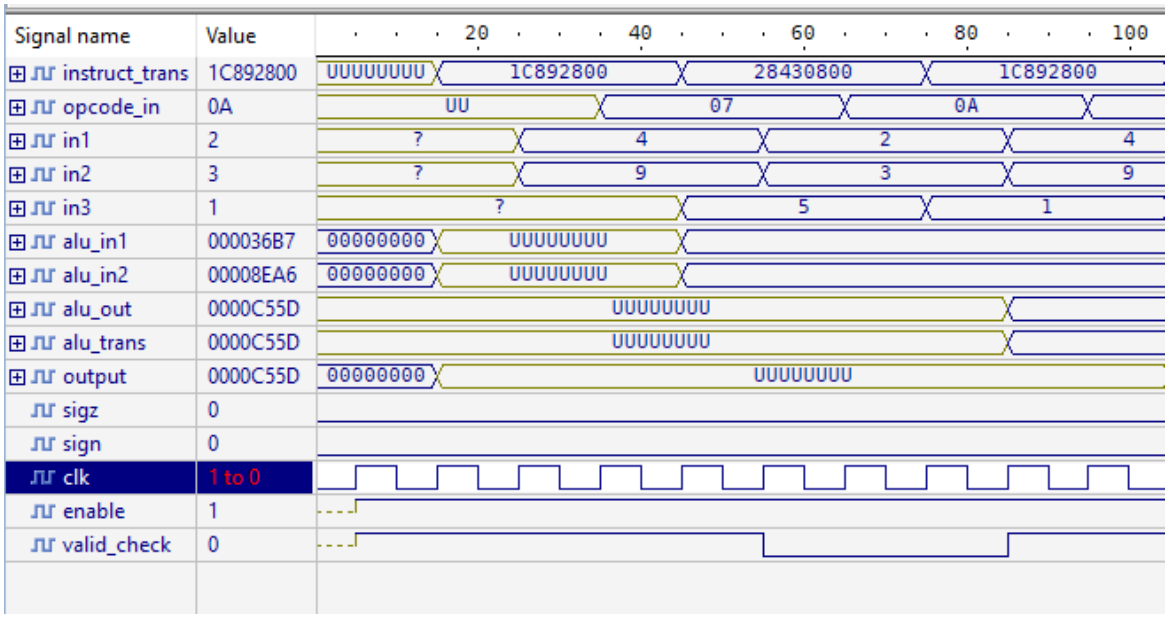

Figure 3: Simulation screenshot showing valid signal changing

The above figure shows that when an invalid opcode is in the instruction stream, the valid signal changes to 0 after 1 clock cycle due to the pipeline having one cycle. This is used to stop any invalid operations or if the write-back address has been set to 0 from corrupting the workings of the microprocessor. When valid is set to 0, the writeback address is changed to 0 and the ALU output is changed to 0 as well. This acts as a reset flag.

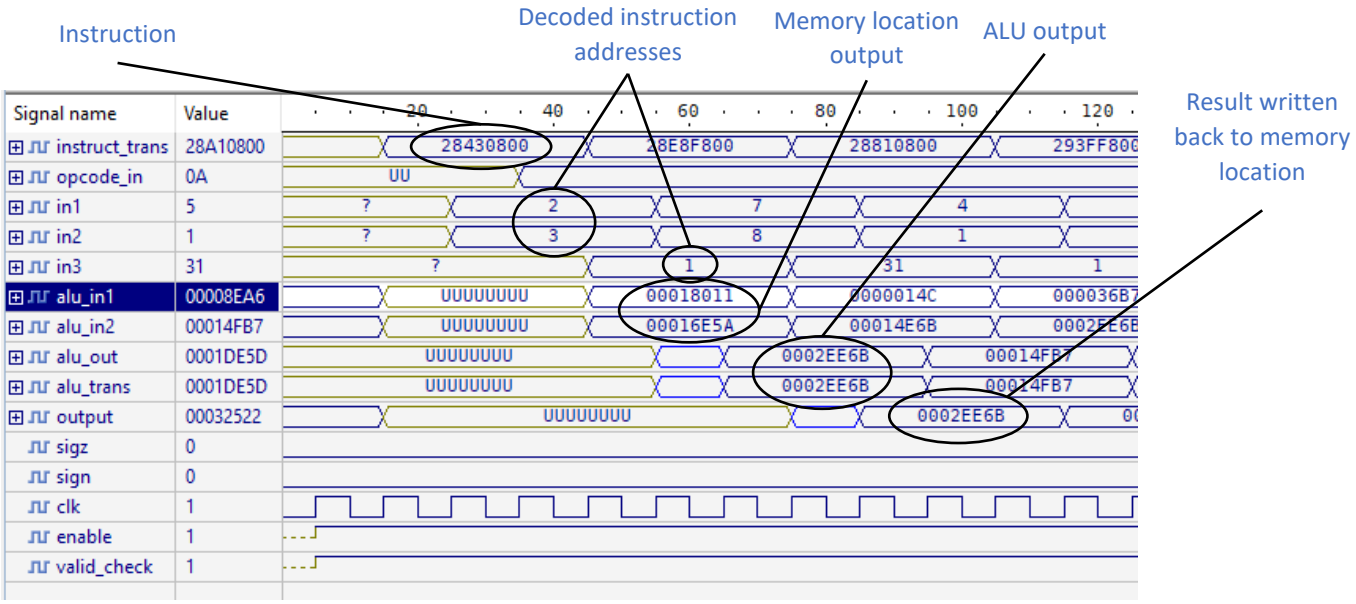

Figure 4: Simulation showing signal values and the flow through the pipeline

Figure 3 shows how an instruction flows through the pipeline. The instruction comes through the instruction stream and one cycle later is broken down into the necessary data, which is the opcode, which memory locations to get the data from, and which memory location to write back to which is delayed by one cycle. In the next cycle, the data from the memory locations are fetched and the write-back memory location is released from the previous register so is not in sync with the data. On the next rising edge of the clock, the ALU performs the given operation on the fetched data, then on the next cycle is written back into the given write-back memory location. The output reads that write-back memory location on the next cycle so that the write-back operation can be checked to ensure it is working correctly.

Figure 3 also shows that at the start of the simulation, all the variables are initially U. Once both the enable signal and valid signal are seen as 1, the instruction stream is allowed to proceed. Once the clock after both signals are initialised as 1, the first instruction comes in. This then prompts the other signals to change from U to the required values from the instruction. Some signals take longer to change from U as it takes 1 to 3 cycles for the signal to travel through the pipeline depending on which signal it is.

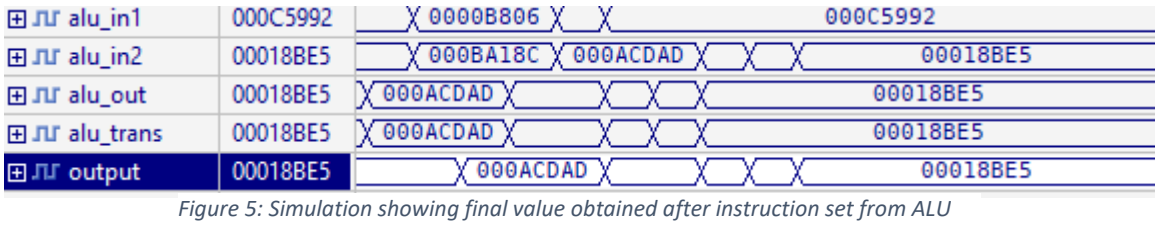

Figure 5: Simulation showing final value obtained after instruction set from ALU

Comparing the ALU output values from figure 3 and those from figure 5, we can see that r2+r3=18011+16E5A=2EE6B and r7+r8=14C+14E6B=14FB7 for both showing that the ALU is outputting the correct answers.

Memory location	Value	running total (+) (Hex)	running total (-) (Hex)	Running Total (Hex)	
0	0				
1	0				
2	98321				
3	93786	2EE6B		2EE6B	
4	14007	32522		32522	
5	75501	44C0F		44C0F	
6	93561	5B988		5B988	
7	332	0		5B83C	
8	85611	0	14FB7	469D1	
9	36518	0	1DE5D	3DB2B	
10	78586	0	31157	2A831	
11	47484	0	3CAD3	1EEB5	
		0	0	0	
12	64810	6B6B2	0	2EBDF	
13	15808	6F472	0	3299F	
14	26091	75A5D	0	38F8A	
15	17754	79FB7	0	3D4E4	
16	81289	8DD40	0	5126D	
		0	0	0	
17	75245	0	4F0C0	3EC80	
18	56096	0	5CBE0	31160	
19	96786	0	745F2	1974E	
20	16458	0	7863C	15704	
21	45256	0	83704	A63C	
		0	0	0	
22	347	8DE9B	0	A797	
23	62628	9D33F	0	19C3B	
24	64460	ACF0B	0	29807	
25	53889	BA18C	0	36A88	
26	47110	C5992	0	4228E	
		0	0	0	
27	11867	0	8655F	3F433	
28	48355	0	92242	33750	
29	31874	0	99EC4	2BACE	
30	77545	0	ACDAD	18BE5	
31	0	0	0	18BE5	
totals			18BE5	18BE5	101349

Figure 6: Calculation of correct final value

Figure 3 shows how the final value was calculated by converting all the values in the memory to hexadecimal and then following the instruction stream stated before. Following the instruction stream, the final answer is 00018BE5 in hex form which is 101349 in decimal. As stated before, the final ALU output matches that of the calculated final value proving that the ALU runs and the instruction for the pipeline is correct. Comparing figure 3 with figures 6, 7, 8 & 9 in the appendix, it can be seen that the microprocessor correctly calculates the values detailed in the "running total (+) (Hex)" and "running total (-) (Hex)" in the table shows that with the added no-ops, the microprocessor can successfully the instructions.

Efficiency:

The Microprocessor has a clock period of 10 ns and therefore a clock speed of 100 GHz. It takes 925 ns to get the output of 00018BE5, which is made of 28 instructions. However, in the instruction stream, the instructions were reordered, and no-ops were introduced into the stream to allow for time for the microprocessor to write back to a register. Two no-ops were added between each instruction meaning that there were a total of 84 instructions and that the efficiency of the current microprocessor is 33.33%.

Limitations:

The instruction stream uses no-ops to allow for registers to be updated before the instructions can continue along the pipeline, this means that the instructions are only applicable to pipelines of the same structure. With the added no-ops, the instruction set contains only 33.33% of useful instructions. This is considerably low for a microprocessor. Another limitation of this microprocessor setup is that both registers 1 and 31 are used, if another type of instruction was added into the instruction stream and was needed for another calculation, then one of the registers would have to be used for that purpose instead.

Improvements:

To improve the microprocessor, the enable signal could be properly added so that it pauses the instruction stream if an instruction tries to access a register that will be updated. This would allow this processor to process any set of instructions and allow the given instruction set to be given to any other microprocessor. This would also speed up the computation time as the microprocessor may not need to wait so long for the register to be updated. It would increase the efficiency of the microprocessor as 100% of the instructions would be useful. Also, the instruction stream could be set up in such a way that only one out of registers r1 and r31 is used for the given instruction set, which would allow for one extra register to be used if a separate calculation was needed to be done as well as the given instruction set.

Appendix:

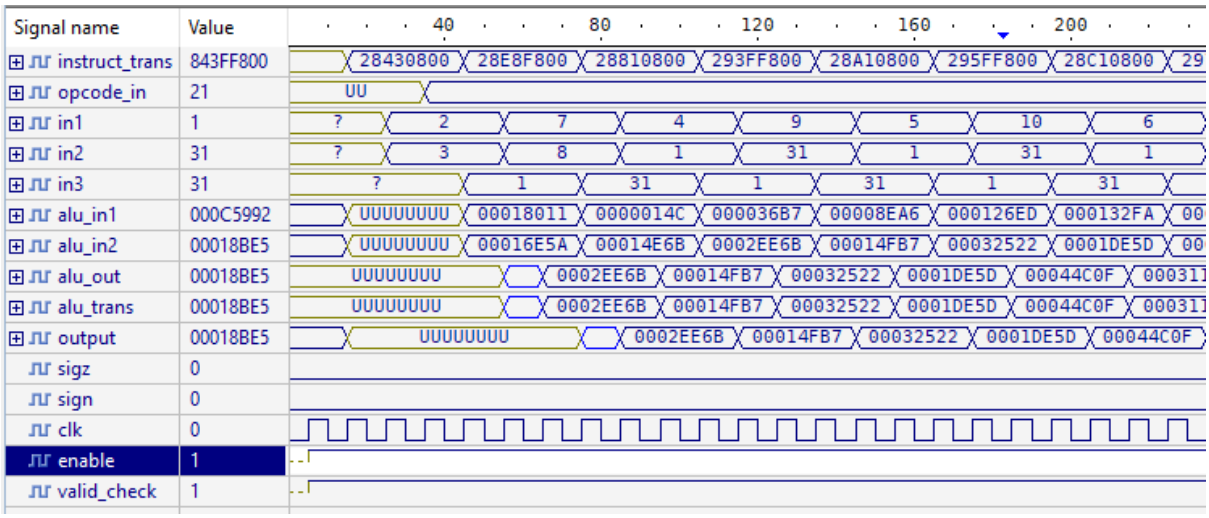

Figure 7: Full simulation screenshot (a)

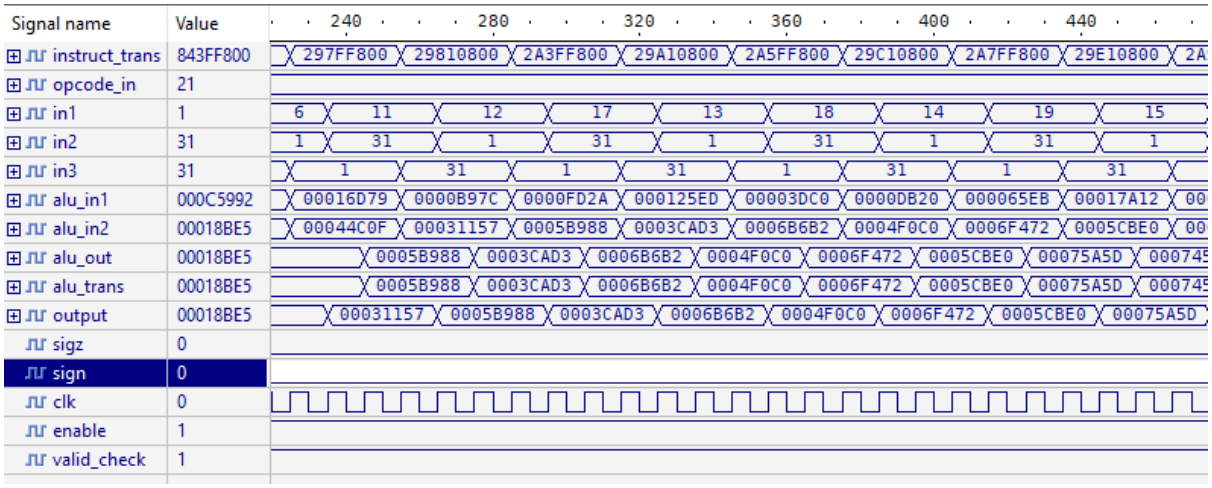

Figure 8: Full simulation screenshot (b)

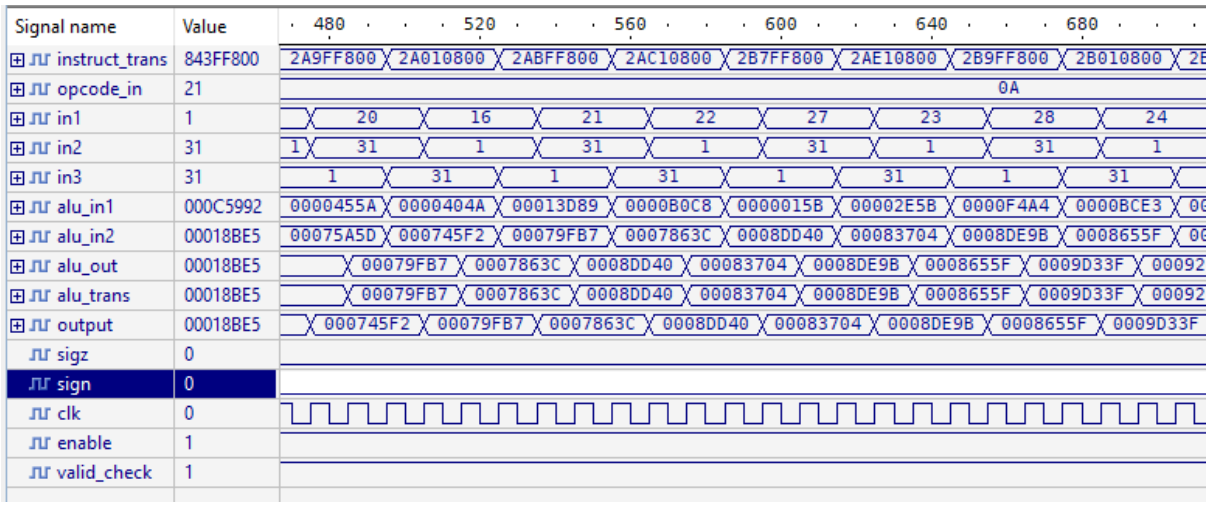

Figure 9: Full simulation screenshot (c)

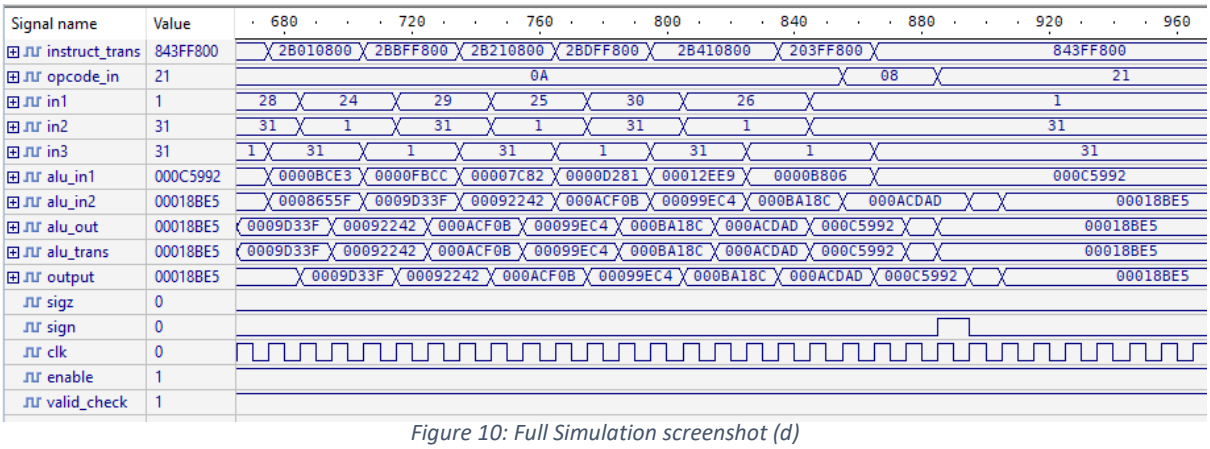

Figure 10: Full Simulation screenshot (d)

```
14  ----------------------------------------------------------------------------
15  --************** Microprocessor ********************
16  -- setting up the libraries used
17  library ieee;
18  use ieee.std_logic_1164.all;
19  use ieee.std_logic_signed.all;
20  use ieee.numeric_std.all;
21
22  entity compalu is
23  end entity compalu;
24
25  architecture test of compalu is
26  --setting up signals used within the Microprocessor program
27  signal instruct_trans: std_logic_vector(31 downto 0);
28  signal opcode_in: std_logic_vector(5 downto 0) :="000000";
29  signal in1, in2, in3: std_logic_vector(4 downto 0);
30  signal alu_in1, alu_in2, alu_out, alu_trans, output: std_logic_vector(31 downto 0) :=X"00000000";
31  signal sigz, sign: std_logic := '0';
32  signal clk: std_logic;
33  signal enable: std_logic;
34  signal valid_check: std_logic;
35  begin
36
37     process is       --constantly changing clk value when code is running
38     begin            -- Period per full clock cycle = 10 ns
39      clk <= '0';
40      wait for 5 ns;
41      clk <= '1';
42      wait for 5 ns;
43     end process;
44
45  g1: entity work.instruction_stream(dataflow) -- run architecture named dataflow in program instruction_stream
46      port map( instruct => instruct_trans, clk => clk,  -- transfer named registers into named signals
47      enable => enable, valid => valid_check);
48
49     g2: entity work.decoder_test(divide) -- run architecture named divide in program named decoder_test
50         port map ( instruct_input => instruct_trans, instruct1 => in1, instruct2 => in2,
51         instruct3buff2 => in3, opcode => opcode_in, clock => clk, valid_trans => valid_check,
52         enable_trans => enable, output_check => output);
53
54        g3: entity work.memory_test(dataflow) -- run architecture named dataflow in program named memory_test
55            port map ( addr1 => in1, addr2 => in2, d_out1 => alu_in1, d_out2 => alu_in2,
56            addr3 => in3, d_in => alu_trans, check => output, valid => valid_check,
57            clock => clk, enable_input => enable);
58
59
60        g4: entity work.alu_test(dataflow) -- run architecture named dataflow in program named alu_test
61            port map ( a => alu_in1, b => alu_in2, opcode => opcode_in, c => alu_out,
62            z => sigz, n => sign, clock => clk, enable_input => enable);
63
64  alu_trans <= alu_out;    -- transfer value of alu_out to alu_trans to be written back to memory
65
66  end architecture test;
```

Figure 11: Microprocessor code screenshot

```
--********************** Instruction Stream *********************************--
-- Stating and defining libraries used within the instruction stream program
 library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;



entity instruction_stream is
-- setting up reigsters used in the Instruction stream program
    port (instruct: out std_logic_vector(31 downto 0);
    clk: in std_logic;
    enable: in std_logic;
    valid: in std_logic);

end instruction_stream;


-- Start of main body of code
architecture dataflow of instruction_stream is
begin

      process is
       begin


       --wait until the next rising edge of the clock and for enable to be 1 to proceed
       wait until rising_edge(clk) and enable = '1';
       instruct <= B"001010_00010_00011_00001_00000000000"; -- r2+r3 goes into r1


       wait until rising_edge(clk) and enable = '1';  -- no-op
       wait until rising_edge(clk) and enable = '1';  -- no-op


       wait until rising_edge(clk) and enable = '1';
       instruct <= B"001010_00111_01000_11111_00000000000";-- r7+r8 goes into   r31


       wait until rising_edge(clk) and enable = '1';
       wait until rising_edge(clk) and enable = '1';


           wait until rising_edge(clk) and enable = '1';
       instruct <= B"001010_00100_00001_00001_00000000000"; --r4+r1 goes into r1


       wait until rising_edge(clk) and enable = '1';
       wait until rising_edge(clk) and enable = '1';


           wait until rising_edge(clk) and enable = '1';
       instruct <= B"001010_01001_11111_11111_00000000000"; --r9+r31 goes into r31
```

Figure 12: Instruction stream code screenshot (a)

```
67          wait until rising_edge(clk) and enable = '1';
68        instruct <= B"001010_01001_11111_11111_00000000000"; --r9+r31 goes into r31
69
70
71        wait until rising_edge(clk) and enable = '1';
72        wait until rising_edge(clk) and enable = '1';
73
74
75        wait until rising_edge(clk) and enable = '1';
76        instruct <= B"001010_00101_00001_00001_00000000000"; --r5+r1 goes into r1
77
78
79        wait until rising_edge(clk) and enable = '1';
80        wait until rising_edge(clk) and enable = '1';
81
82
83           wait until rising_edge(clk) and enable = '1';
84       instruct <= B"001010_01010_11111_11111_00000000000";  --r10+r31 goes into r31
85
86
87       wait until rising_edge(clk) and enable = '1';
88       wait until rising_edge(clk) and enable = '1';
89
90
91         wait until rising_edge(clk) and enable = '1';
92        instruct <= B"001010_00110_00001_00001_00000000000";  --r6+r1 goes into r1
93
94
95        wait until rising_edge(clk) and enable = '1';
96        wait until rising_edge(clk) and enable = '1';
97
98
99       wait until rising_edge(clk) and enable = '1';
100      instruct <= B"001010_01011_11111_11111_00000000000"; --r11+r31 goes into r31
101
102
103      wait until rising_edge(clk) and enable = '1';
104      wait until rising_edge(clk) and enable = '1';
105
106
107      wait until rising_edge(clk) and enable = '1';
108      instruct <= B"001010_01100_00001_00001_00000000000";  --r12+r1 goes into r1
109
110
111      wait until rising_edge(clk) and enable = '1';
112      wait until rising_edge(clk) and enable = '1';
113
114
115         wait until rising_edge(clk) and enable = '1';
116     instruct <= B"001010_10001_11111_11111_00000000000"; --r17+r31 goes into r31
117
118
119     wait until rising_edge(clk) and enable = '1';
```

Figure 13: Instruction stream code screenshot (b)

```
119         wait until rising_edge(clk) and enable = '1';
120         wait until rising_edge(clk) and enable = '1';
121
122
123
124         wait until rising_edge(clk) and enable = '1';
125         instruct <= B"001010_01101_00001_00001_00000000000";    --r13+r1 goes into r1
126
127
128         wait until rising_edge(clk) and enable = '1';
129         wait until rising_edge(clk) and enable = '1';
130
131
132            wait until rising_edge(clk) and enable = '1';
133         instruct <= B"001010_10010_11111_11111_00000000000"; --r18+r31 goes into r31
134
135
136         wait until rising_edge(clk) and enable = '1';
137         wait until rising_edge(clk) and enable = '1';
138
139
140         wait until rising_edge(clk) and enable = '1';
141         instruct <= B"001010_01110_00001_00001_00000000000";    --r14+r1 goes into r1
142
143
144           wait until rising_edge(clk) and enable = '1';
145           wait until rising_edge(clk) and enable = '1';
146
147
148             wait until rising_edge(clk) and enable = '1';
149         instruct <= B"001010_10011_11111_11111_00000000000"; --r19+r31 goes into r31
150
151
152         wait until rising_edge(clk) and enable = '1';
153         wait until rising_edge(clk) and enable = '1';
154
155
156         wait until rising_edge(clk) and enable = '1';
157         instruct <= B"001010_01111_00001_00001_00000000000";    --r15+r1 goes into r1
158
159
160         wait until rising_edge(clk) and enable = '1';
161         wait until rising_edge(clk) and enable = '1';
162
163
164            wait until rising_edge(clk) and enable = '1';
165        instruct <= B"001010_10100_11111_11111_00000000000"; --r20+r31 goes into r31
166
167
168        wait until rising_edge(clk) and enable = '1';
169        wait until rising_edge(clk) and enable = '1';
170
171
```

Figure 14: Instruction stream code screenshot (c)

```
171
172         wait until rising_edge(clk) and enable = '1';
173         instruct <= B"001010_10000_00001_00001_00000000000";   --r16+r1 goes into r1
174
175
176         wait until rising_edge(clk) and enable = '1';
177         wait until rising_edge(clk) and enable = '1';
178
179
180       wait until rising_edge(clk) and enable = '1';
181       instruct <= B"001010_10101_11111_11111_00000000000"; --r21+r31 goes into r31
182
183
184       wait until rising_edge(clk) and enable = '1';
185       wait until rising_edge(clk) and enable = '1';
186
187
188         wait until rising_edge(clk) and enable = '1';
189         instruct <= B"001010_10110_00001_00001_00000000000"; --r22+r1 goes into r1
190
191
192         wait until rising_edge(clk) and enable = '1';
193         wait until rising_edge(clk) and enable = '1';
194
195
196         wait until rising_edge(clk) and enable = '1';
197       instruct <= B"001010_11011_11111_11111_00000000000"; --r27+r31 goes into r31
198
199
200       wait until rising_edge(clk) and enable = '1';
201       wait until rising_edge(clk) and enable = '1';
202
203
204       wait until rising_edge(clk) and enable = '1';
205       instruct <= B"001010_10111_00001_00001_00000000000"; -- r23+r1 goes into r1
206
207
208       wait until rising_edge(clk) and enable = '1';
209       wait until rising_edge(clk) and enable = '1';
210
211
212           wait until rising_edge(clk) and enable = '1';
213       instruct <= B"001010_11100_11111_11111_00000000000";  --  r28+r31 goes into r31
214
215
216       wait until rising_edge(clk) and enable = '1';
217       wait until rising_edge(clk) and enable = '1';
218
219
220       wait until rising_edge(clk) and enable = '1';
221       instruct <= B"001010_11000_00001_00001_00000000000"; --r24+r1 goes into r1
222
223
```

Figure 15: Instruction stream code screenshot (d)

```
223
224         wait until rising_edge(clk) and enable = '1';
225         wait until rising_edge(clk) and enable = '1';
226
227
228             wait until rising_edge(clk) and enable = '1';
229         instruct <= B"001010_11101_11111_11111_00000000000"; --r29+r31 goes into r31
230
231
232         wait until rising_edge(clk) and enable = '1';
233         wait until rising_edge(clk) and enable = '1';
234
235
236         wait until rising_edge(clk) and enable = '1';
237         instruct <= B"001010_11001_00001_00001_00000000000";  --r25+r1 goes into r1
238
239
240         wait until rising_edge(clk) and enable = '1';
241         wait until rising_edge(clk) and enable = '1';
242
243
244             wait until rising_edge(clk) and enable = '1';
245         instruct <= B"001010_11110_11111_11111_00000000000"; --r30+r31 goes into r31
246
247
248         wait until rising_edge(clk) and enable = '1';
249         wait until rising_edge(clk) and enable = '1';
250
251
252         wait until rising_edge(clk) and enable = '1';
253         instruct <= B"001010_11010_00001_00001_00000000000"; --r26+r1 goes into r1
254
255
256         wait until rising_edge(clk) and enable = '1';
257         wait until rising_edge(clk) and enable = '1';
258         wait until rising_edge(clk) and enable = '1';
259
260
261
262         wait until rising_edge(clk) and enable = '1';
263         instruct <= B"001000_00001_11111_11111_00000000000";    --r1-r31 goes into r31
264
265         wait until rising_edge(clk) and enable = '1';
266         wait until rising_edge(clk) and enable = '1';
267
268
269            wait until rising_edge(clk) and enable = '1';
270         instruct <= B"100000_00001_11111_11111_00000000000"; --invalid opcode to hold final answer
271            wait;
272            end process;
273     end architecture;
```

Figure 16: Instruction stream code screenshot (e)

```
13  --------------------------------------------------------------------------------
14  --************************* Decoder ****************************************--
15  -- Stating and defining libraries used within the Decoder program
16  library ieee;
17  use ieee.std_logic_1164.all;
18  use ieee.std_logic_signed.all;
19
20
21  -- setting up registers used with the decoder program
22  entity decoder_test is
23      port ( opcode, opcode_buff1, opcode_buff2: out std_logic_vector(5 downto 0); -- 6 bit register
24      instruct1, instruct2, instruct3: out std_logic_vector(4 downto 0); -- 5 bit register
25      instruct3buff, instruct3buff2 : out std_logic_vector(4 downto 0);
26      instruct_input: in std_logic_vector(31 downto 0); -- 32 bit register
27      instruct_valid, instruct_buff, instruct_buff2: out std_logic_vector(31 downto 0);
28      valid_trans, enable_trans: out std_logic; -- 1 bit register
29      clock: in std_logic;
30      output_check: in std_logic_vector(31 downto 0));
31  end decoder_test;
32
33
34  --beginning of main body of code
35  architecture divide of decoder_test is
36  -- initialising signals used in code
37  signal enable: std_logic;
38  signal valid: std_logic;
39
40  begin
41
42         process is
43         begin
44
45          -- move the instruction stream along pipeline
46             instruct_buff<=instruct_input;
47
48          --input insctruction is broken down into opcode
49          --and instructions 1, 2 and 3
50          opcode_buff1 <= instruct_input(31 downto 26);
51
52          opcode <= opcode_buff1;
53
54          instruct1 <= instruct_input(25 downto 21);
55
56          instruct2 <= instruct_input(20 downto 16);
57
58          instruct3 <= instruct_input(15 downto 11);
59
60
61   -- buffer instruction 3 as would arrive at ALU to early
62  instruct3buff <= instruct3;
```

Figure 17: Decoder code screenshot (a)

```
60
61   -- buffer instruction 3 as would arrive at ALU to early
62  instruct3buff <= instruct3;
63  instruct3buff2 <= instruct3buff;
64
65  -- checks for valid write back location and opcode
66  -- if write back location = 0 or opcode is invalid, change signal valid to 0
67          if instruct3 = "00000" then
68              valid <= '0';
69
70          elsif opcode >= "010000" then
71              valid <= '0';
72
73          elsif opcode = "000100" then
74              valid <= '0';
75
76          elsif opcode = "000111" then
77              valid <= '0';
78
79          elsif opcode = "001001" then
80              valid <= '0';
81
82          elsif opcode = "001100" then
83              valid <= '0';
84  -- if everything ok, enable and valid signals go to 1
85     else
86         enable <= '1';
87      valid <= '1';
88
89      end if;
90  -- transfer signals to regsiters
91              enable_trans <= enable;
92              valid_trans <= valid;
93
94              wait until rising_edge(clock); -- tells program to only proceed when clock is on rising edge
95
96
97      end process;
98
99  end architecture;
100
```

Figure 18: Decoder code screenshot (b)

```
17  ----------------------------------------------------------------------------
18  --*************************** Memory (RAM) ********************************--
19  -- Stating and defining libraries used within the Memory program
20  library ieee;
21  use ieee.std_logic_1164.all;
22  use ieee.std_logic_unsigned.all;
23
24  -- Setting up registers used within the Memory program
25  entity memory_test is
26      port ( addr1, addr2, addr3: in std_logic_vector(4 downto 0);
27      addr3_buff1, addr3_buff2, addr3_buff3: out std_logic_vector(4 downto 0);
28      d_in: in std_logic_vector(31 downto 0);
29      check: out std_logic_vector(31 downto 0);
30      d_out1, d_out2: out std_logic_vector(31 downto 0);
31      valid: in std_logic;
32      clock, enable_input: in std_logic);
33  end entity memory_test;
34
35  --start of main body of code
36  architecture dataflow of memory_test is
37  type rom_array is array (0 to 31) of std_logic_vector(31 downto 0); -- Defining ROM memory as 32 locations each of 32-bit length
38  signal rom_mem: rom_array := (
39          X"00000000", X"00000000", X"00018011", X"00016E5A",         -- Initiallisng data inside each memory location
40          X"000036B7", X"000126ED", X"00016D79", X"0000014C",
41          X"00014E6B", X"00008EA6", X"000132FA", X"0000B97C",
42          X"0000FD2A", X"00003DC0", X"000065EB", X"0000455A",
43          X"00013D89", X"000125ED", X"0000DB20", X"00017A12",
44          X"0000404A", X"0000B0C8", X"0000015B", X"0000F4A4",
45          X"0000FBCC", X"0000D281", X"0000B806", X"00002E5B",
46          X"0000BCE3", X"00007C82", X"00012EE9", X"00000000");
47  -- Setting up signals used in th memory program
48  signal addr1_valid, addr2_valid: std_logic_vector(4 downto 0);
49  signal  d_inbuff: std_logic_vector(31 downto 0);
50  begin
51     process
52     begin
53  -- if instructions are valid, then move the values from certain registers to signals
54     if valid = '1' then
55
56
57          addr3_buff1 <= addr3;
58
59          d_inbuff <= d_in;
60          addr1_valid <= addr1;
61          addr2_valid <= addr2;
62
63  -- if instructions are not valid, then change singal values to 0
64      else
65      addr3_buff1 <= "00000";
66      d_inbuff <=X"00000000";
```

Figure 19: Memory (ROM) code screenshot (a)

```
67
68          end if;
69  -- put data from given memory loaction into registers d_out1 and d_out2
70              d_out1 <= rom_mem(conv_integer(addr1_valid));
71          d_out2 <= rom_mem(conv_integer(addr2_valid));
72
73  -- Move write back address along pipeline
74          addr3_buff2 <= addr3_buff1;
75
76  -- write back the value of d_in into given memory location
77          rom_mem(conv_integer(addr3_buff1)) <= d_in;
78
79  -- put value from given memory address into register check
80  check <= rom_mem(conv_integer(addr3_buff2));
81
82  -- tells program to only proceed when clock is on rising edge
83          wait until rising_edge(clock);
84
85      end process;
86
87  end architecture dataflow;
```

Figure 20: Memory (RAM) code screenshot (b)

```
15  ----------------------------------------------------------------------------
16  --********************************* ALU ************************************--
17  -- Stating and defining libraries used within the ALU program
18  library ieee;
19  use ieee.std_logic_1164.all;
20  use ieee.std_logic_signed.all;
21
22
23  entity alu_test is
24  -- Setting up registers used within the ALU program
25    port ( a, b: in std_logic_vector(31 downto 0);
26                opcode: in std_logic_vector(5 downto 0);
27                c, c_buffer, c_buffer2: out std_logic_vector(31 downto 0);
28                z, n: out std_logic;
29                valid: out std_logic;
30               clock, enable_input: in std_logic);
31  end entity alu_test;
32
33  --main body of the code
34  architecture dataflow of alu_test is
35  --set up signal used within the program
36      signal c_input: std_logic_vector(31 downto 0);
37  begin
38
39      process
40      begin
41
42  --defining what operation to perform on the data inputted into a and b when certain
43  --opcodes have been put in
44    c_input <= a + b when opcode="001010"
45    else a - b when opcode="001000"
46    else abs a when opcode="000110"
47    else -a when opcode="001110"
48    else abs b when opcode="000010"
49    else -b when opcode="001101"
50    else a or b when opcode="001111"
51    else not a when opcode="001011"
52    else not b when opcode="000011"
53    else a and b when opcode="000101"
54    else a xor b when opcode="000001";
55
56  --If instructions are invalid then change output, c, to 0
57     if valid = '0' then
58         c <= X"00000000";
59     else
60  --If instructions are valid, then allow data to transfer from signal to output register
61         c <= c_input;
62         end if;
63
64        if c = X"00000000" then     -- if c is zero, raise z flag
```

Figure 21: ALU code screenshot (a)

```
65            z<= '1';
66        else
67            z <= '0';
68        end if;
69        if c <X"00000000" then   -- if c is negative, raise n flag
70            n <= '1';
71        else
72            n <= '0';
73        end if;
74
75          wait until rising_edge(clock); -- tells compiler to only proceed when clock is on rising edge
76
77        end process;
78
79  end architecture dataflow;
```

Figure 22: ALU code screenshot (b)

References:

[1] Electronic Engineering – Digital – Part 1 assignment sheet – Dr. Steven Quigley - https://canvas.bham.ac.uk/courses/65144/files/13704053/download?wrap=1

[2] Electronic Engineering – Digital – Part 2 assignment sheet – Dr. Steven Quigley - https://canvas.bham.ac.uk/courses/65144/files/13704051?wrap=1

[3] Electronic Engineering – Digital – Part 3 assignment sheet – Dr. Steven Quigley - https://canvas.bham.ac.uk/courses/65144/files/13909223?wrap=1

[4] Electronic Engineering – Digital – Digital Part 2 – Dr. Steven Quigley - https://canvas.bham.ac.uk/courses/65144/files/13538000?wrap=1