EE3A1 Assignment

Student ID: 1095307

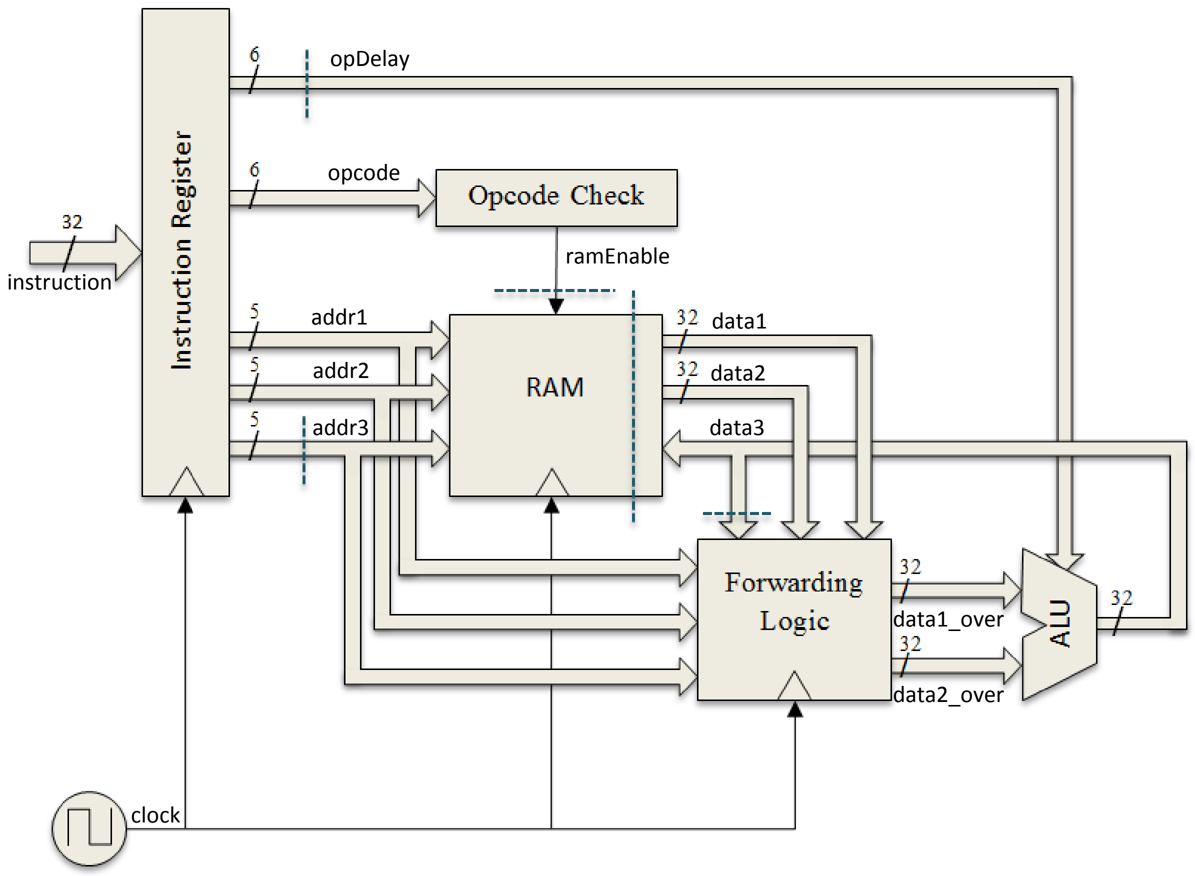
Contents

Introduction

The aim of this assignment is to design a working microprocessor in VHDL. The design should be able to run a specified program that uses opcodes and data that is dependent on one's student ID. The following assignment was worked on together with student 1048258, however the results discussed in this report are based on the student ID 1095307.

System Design

The final system design (as described in "proc.vhd") can be depicted with the aid of the following diagram.



Essentially, the only inputs are a 32-bit instruction and a clock line that are fed in through, in this case, a test bench that acts as a main program. The entire system is combined in the file "proc.vhd" and the test bench contained in "procTest.vhd" (see appendix 1).

The instruction register block ("instructReg.vhd") splits the instruction into its individual parts. Each instruction is structured as 6 bits for the opcode, 5 bits each for addresses 1, 2, and 3, and 11 unused bits, for example "001010_00001_00010_00011_00000000000". The instruction register also contains two D-type flip-flops to delay addr3 by one clock cycle and output another line that is a delayed version of opCode.

Addr1 and *addr2* are then input into RAM ("RAM.vhd"), which reads in 3 addresses, and on the subsequent clock cycle outputs data1 (the data located at *addr1*), *data2* (the data located at *addr2*), and writes *data3* to the address that is at the time being input on *addr3*. As the write operation is one clock cycle later, *addr3* is delayed as a result of this in order for the *data3*, which is a function of *data1* and *data2*, to match up with the same instruction that yielded *data1* and *data2*.

RAM is also fed an enable flag, which it delays by one clock cycle, which controls **only** the write operation (writing *data3* to the memory location pointed to by *addr3*). This is to ensure that, given an invalid opCode, or indeed a "NoOp" instruction, data from the previous clock cycle is not processed giving a recursive-esque accumulation effect if a read address is the same as the write address. For the program in question this is not a problem as instructions come at one clock cycle intervals effectively denying the processor the chance to repeat the same instruction. For other applications however, it may very well be an issue.

OpCheck ("opCheck.vhd") merely reads the *opCode* and outputs a 1 if it is valid and a 0 if it is invalid all asynchronously. The ALU ("ALU.vhd") is similar in this manner in that it immediately and asynchronously outputs an answer given and opcode to define the operation and two 32-bit numbers to perform the operation on.

So far, the design would have worked perfectly fine albeit sub-optimal, yet an additional block was added for optimisation as is explained in the next section of this report. The forwarding logic block ("fwdLogic.vhd"), takes the same address inputs that the RAM block takes, checks for potential data hazards (i.e. when a memory block that is still dirty and has not been updated with its new value yet is to be read), and overrides the *data1* and *data2* lines, that are fed into the ALU, if the hazard exists. The override lines are the same as the data lines that RAM outputs if the hazard does not exist but when it does, then *addr1* and/or *add2* are the same as *addr3* and the forwarding logic block will fix the hazard by instead using the previous *data3* as the new *data1* and/or *data2* values instead of waiting for the RAM values to be updated the next clock cycle. The previous *data3* line is read from a D-type flip-flop to prevent an infinitely recursive feedback loop. In other words, instead of *data3* being a function of *data3* which is constantly changing asynchronously, it becomes a function of *data3Reg*. Clearly in the worst case scenario, or best case depending on your perspective, the forwarding logic block cuts processing time clear in half.

Main Challenges and Solutions

The main challenges in this assignment were mostly related to speed optimisation. The final goal was to create a pipeline where each operation effectively takes a single clock cycle giving the best possible throughput. There were, however, other issues and minor considerations to be taken account of. Some have been mentioned earlier, such as the registers or D-type flip-flops that help make inputs and outputs have the right values at the right time, and do not need to be discussed in detail due to their self-explanatory nature and the fact that they are not problems as such. For some minor reiteration on timing, see appendix 2. The following are other problems that arose and their solutions in a chronological fashion.

Initialisation

As can be seen in the simulation screenshots (appendix 2), when the simulation begins, any uninitialized lines will have garbage values. This becomes a problem because it is possible that, for instance *data3*'s, garbage value is written into RAM memory overwriting important data.

The solution to this is the enable flag from the opCheck entity that only allows data to be written when the inputs are valid. It is conceivable that the RAM entity would have two control lines for both reading and writing or something equivalent yet, in the scope of this assignment, such an addition would be redundant and overcomplicating as it is irrelevant what is being read from the RAM since nothing is being written. The unnecessary complexity stems from the fact that if reading **and** writing is disabled when "enable" is low, a valid opcode followed immediately by and invalid one will cause the answer to only be written the next time a valid opcode appears as writing happens one clock cycle later. Thus, controlling only the write operation allows for more control.

It is also possible to include the *opcode* register that provides *opDelay* to the ALU in the opCheck entity and, from an abstract design perspective, it would in fact make more sense to do so, however that would imply branching the clock line yet again to clock opCheck so for the sake of simplicity, this was done in the instruction register block as it is being clocked anyway.

Hazards – Software

As mentioned earlier, the final design uses a hardware block that rids the processor of a myriad of issues including data hazards. Software based solutions were however explored and implemented successfully. An immediately apparent issue is the fact that it takes two clock cycles from when an *opcode* and three addresses are given to when the memory block at *addr3* is actually updated but the instructions should ideally come at one clock cycle intervals. The amateur solution to this is to simply space every instruction at two clock cycle intervals to be absolutely sure but, when working at such a low level, this would mean effectively slicing the processor speed potentially in half.

Solving this through software means may require methods such as re-ordering instructions so that instructions that involve reading from a register that has been previously written to happen at least two clock cycles apart. This can get increasingly complex and cause the design to suffer in portability.

For this particular assignment register 31, a spare register, can be utilised as a temporary register to be used in the time that it takes register 0, the main sum register, to update. In this case, one would subtract register 1 from register 31 and store the answer in register 31 then, while it is being written to, add register 2 to register 0 and store the answer in register 0 then continue with the next odd register and repeat for instance.

This would however mean that a "NoOp" instruction would be needed before the final operation, adding register 0 and register 31 and storing the answer in register 31, in order to allow the final value to update. Once again this would mean that the program would need 33 clock cycles before it writes the final answer as opposed to the perfect 32. These small inefficiencies cannot be avoided when using purely software-based methods and as such the ultimate way to avoid data hazards is through hardware extensions that trump software on cost-benefit.

To test the reordering of instructions, explicitly setting the instructions in a defined order is a bad design choice. For example:

```
                -- reg[31] = reg[31] - reg[1]
instruction <=  B"001110_11111_00001_11111_00000000000",
                -- reg[0] = reg[0] + reg[2]
                B"001010_00000_00010_00000_00000000000" AFTER 20 NS,
                -- reg[31] = reg[31] - reg[3]
                B"001110_11111_00011_11111_00000000000" AFTER 30 NS,
                ...
```

If the delays do not match the clock speed, certain instructions could be processed multiple times on not at all giving the effect of an accumulator in this case as a read address (addr1) is also being written to and is constantly changing (as discussed earlier when explaining the rationale behind "NoOp"s) or simply give wrong results. A better solution for this particular program is to determine the values algorithmically:

```
-- Add if index is even, subtract if odd, NoOp if greater than 30.
opcode <= B"111111" WHEN index > 30 ELSE B"001010" WHEN (index MOD 2 = 0) ELSE B"001110";
-- Read from register 31 if index is even and 0 if index is odd.
addr1 <= B"11111" WHEN (index MOD 2 = 0) ELSE B"00000";
-- addr2 gets 5-bit index.
addr2 <= STD_LOGIC_VECTOR(TO_UNSIGNED(index, 5));
-- Write to address 31; addr3 could alternatively just get addr1 as they are the same.
addr3 <= B"11111" WHEN (index MOD 2 = 0) ELSE B"00000";
-- Concatenate all components; addr1 can be used instead of addr3 but is not for clarity.
instruction <= opcode & addr1 & addr2 & addr3 & "00000000000";
```

A similar method to this is used in the final design and the source of "index" becomes apparent in the code listing. In essence, index is incremented on every clock cycle as opposed to after specific delays and will allow for any clock speed as the instruction is built concurrently and asynchronously purely dependent on the value of "index".

Hazards – Hardware

The same problems discussed previously can be solved much more elegantly in hardware. This also fulfils the goal of giving the design a throughput of one clock cycle per instruction. As discussed under the “System Design” heading, a forwarding logic block is added that overrides RAM outputs when it detects that they have not been updated yet (i.e. are still dirty). This is done by comparing the values of the *addr1* and *addr2* inputs to RAM with the delayed *addr3* value and thus determine whether or not the memory blocks at the respective addresses are ready.

For this particular program, literally every instruction causes a forwarding logic override but realistically this would not be the case. With the forwarding logic block implemented, it is no longer necessary to alternate between registers 0 and 31 and everything can just be added to and subtracted from register 31. This means that at the end of the program, register 31 will contain the final answer anyway which meets the requirements.

Testing and Results

In order to determine if the design is working properly, every last component was tested individually before being combined to create the processor. The test benches are included in the code listing and are walked through in the comments. The real issue is making sure the timing is right and that the registers/D-type flip-flops are in the correct places. This is very difficult to debug on the fly, so it was worked out exactly on paper and implemented afterwards.

Accordingly, the program was simulated and the first few operations analysed with the aid of a calculator (see appendix 2 diagrams for full details). If the first few instructions yielded correct results, then the last few are subsequently analysed in the same fashion. It is at this point when one discovers if the program ends too late or if the enable flag rises too early. If the final result is correct, then by definition, all intermediate results are correct. The final value obtained is **-26942** in decimal (2’s compliment) which matches the expected value that was calculated on the same Excel spread sheet containing memory data that was provided (see appendix 2 for proof).

Performance-wise the throughput is one clock cycle per instruction as expected (see appendix 2 for proof). In terms of design, no discernable limitations exist, however the actual clock speed is solely dependent on hardware. In simulation combinatorial logic is expected to happen instantly yet this is naturally not the case in practice. The maximum clock speed would be limited by how long it takes for all signals to stabilise, such as the ALU output which might fluctuate during the operation and, if clocked to quickly, a fluctuation might be read as the answer rather than the actual final answer. This would depend on low level design (e.g. individual adders) and hardware factors such as logic gate delays and even temperature.

Conclusion

Ultimately, a working, optimised processor was designed to meet the specifications of this assignment as much as possible. In the process of the design, not only were the ins and outs of VHDL mastered, but concepts such as pipelining and hazards, among many others, learned. It is upon these fundamentals that modern processors are built.

End.

Appendices

Appendix 1 – Code Listing

Common header

```
--------------------------------------------------------------------------
-- Title:               EE3A1 Assignment                                --
-- Authors:             XXXXX (1095307) & XXXXX (1048258)               --
-- Instructor:          Dr. Steven Quigley                              --
-- Date Demonstrated:   15/12/12                                        --
--------------------------------------------------------------------------
```

ALU.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY alu IS
    PORT (  a, b : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            opcode : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
            c : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) );
END ENTITY alu;

ARCHITECTURE std OF alu IS
BEGIN
    -- Opcodes and their function corresponding to student ID 7 (last).
    c <= a + b    WHEN opcode="001010"
    ELSE a - b    WHEN opcode="001110"
    ELSE abs(a)   WHEN opcode="000110"
    ELSE -a       WHEN opcode="000111"
    ELSE abs(b)   WHEN opcode="001001"
    ELSE -b       WHEN opcode="001111"
    ELSE a OR b   WHEN opcode="000010"
    ELSE NOT a    WHEN opcode="000011"
    ELSE NOT b    WHEN opcode="000100"
    ELSE a AND b WHEN opcode="001000"
    ELSE a XOR b WHEN opcode="001011"
    ELSE; -- No op.
END ARCHITECTURE std;
```

ALUTest.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY aluTest IS
END ENTITY aluTest;

ARCHITECTURE std OF aluTest IS
    SIGNAL in1, in2, out1 : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL in3 : STD_LOGIC_VECTOR (5 DOWNTO 0);
BEGIN
    g1: ENTITY work.alu(std)
        PORT MAP (a=>in1, b=>in2, opcode=>in3, c=>out1); --Named association.
    in3 <= "000000"; --add
    in1 <= X"55555555";
    in2 <= X"AAAAAAAA";
END ARCHITECTURE std;
```

RAM.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY ram IS
    PORT (  addr1, addr2, addr3 : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
            output1, output2 : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
            input : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            enable, clock : IN STD_LOGIC );
END ENTITY ram;

ARCHITECTURE std OF ram IS
    TYPE ram_array IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(31 DOWNTO 0);
    -- Data in RAM corresponding to student ID 0 (second last).
    SIGNAL ram_data: ram_array := ( X"00000000",
                                    X"0000110C",
                                    X"00002F4E",
                                    X"00014028",
                                    X"0001774D",
                                    X"00010601",
                                    X"00001E24",
                                    X"00009BE2",
                                    X"00016969",
                                    X"000155EA",
                                    X"00006FFD",
                                    X"000021E5",
                                    X"00004F19",
                                    X"00018529",
                                    X"00012CD0",
                                    X"00017DBF",
                                    X"00014190",
                                    X"00010852",
                                    X"0001161F",
                                    X"0000CEB3",
                                    X"00016F83",
                                    X"0000713C",
                                    X"0001100C",
                                    X"0000B4B8",
                                    X"00011F48",
                                    X"00016F25",
                                    X"0000EFCB",
                                    X"000050DC",
                                    X"00003430",
                                    X"000099C2",
                                    X"000026BD",
                                    X"00000000");
    SIGNAL enableReg : STD_LOGIC := '0'; -- To delay the enable signal by one clock cycle.
BEGIN
    PROCESS (clock) -- NB: Can be made faster with variables.
    BEGIN
        IF (RISING_EDGE(clock)) THEN
            enableReg <= enable;
            -- NB: CONV_INTEGER(...) is not very portable.
            -- Read.
            output1 <= ram_data(CONV_INTEGER(addr1));
            output2 <= ram_data(CONV_INTEGER(addr2));
            -- Write.
            IF (enableReg = '1') THEN
                ram_data(CONV_INTEGER(addr3)) <= input;
            END IF;
        END IF;
    END PROCESS;
END ARCHITECTURE std;
```

RAMTest.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY ramTest IS
END ENTITY ramTest;

ARCHITECTURE std OF ramTest IS
    SIGNAL addr1, addr2, addr3 : STD_LOGIC_VECTOR(4 DOWNTO 0);
    SIGNAL output1, output2, input : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL enable, clock : STD_LOGIC;
BEGIN
    g1: ENTITY work.ram(std)
        PORT MAP (addr1, addr2, addr3, output1, output2, input, enable, clock);
    enable <= '0', '1' AFTER 25 NS;
    addr1 <= B"00000", B"00010" AFTER 50 NS;
    addr2 <= B"00001";
    addr3 <= B"00010";
    input <= X"FFFFFFFF";

    PROCESS
    BEGIN
        clock <= '1';
        WAIT FOR 5 NS;
        clock <= '0';
        WAIT FOR 5 NS;
    END PROCESS;
END ARCHITECTURE std;
```

opCheck.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY opCheck IS
    PORT (  opcode : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
            ramEnable : OUT STD_LOGIC );
END ENTITY opCheck;

ARCHITECTURE std OF opCheck IS
BEGIN
    -- Enable RAM write if opcode is valid (delayed by one clock-cycle in RAM).
    ramEnable <= '1' WHEN opcode="001010"
                        OR opcode="001110"
                        OR opcode="000110"
                        OR opcode="000111"
                        OR opcode="001001"
                        OR opcode="001111"
                        OR opcode="000010"
                        OR opcode="000011"
                        OR opcode="000100"
                        OR opcode="001000"
                        OR opcode="001011"
                        ELSE '0';
END ARCHITECTURE std;
```

instructReg.vhd

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY instructReg IS
    PORT (  instruction : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            opcode, opDelay : OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
            addr1, addr2, addr3 : OUT STD_LOGIC_VECTOR(4 DOWNTO 0);
            clock : IN STD_LOGIC );
END ENTITY instructReg;

ARCHITECTURE std OF instructReg IS
    SIGNAL opcodeReg : STD_LOGIC_VECTOR(5 DOWNTO 0);
    SIGNAL addr3Reg : STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN
    opcode <= opcodeReg;
    PROCESS (clock)
    BEGIN
        IF (RISING_EDGE(clock)) THEN
            opcodeReg <= instruction(31 downto 26);
            -- One clock-cycle delay to synchronise with RAM read.
            -- NB: Done here as opposed to opCheck to avoid clocking it.
            opDelay <= opcodeReg;
            -- Split the instruction into its constituent parts.
            addr1 <= instruction(25 downto 21);
            addr2 <= instruction(20 downto 16);
            addr3Reg <= instruction(15 downto 11);
            -- One clock-cycle delay to synchronise with RAM write.
            addr3 <= addr3Reg;
        END IF;
    END PROCESS;
END ARCHITECTURE std;
```

fwdLogic.vhd

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY fwdLogic IS
    PORT (  addr1, addr2, addr3 : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
            dataIN1, dataIN2, data3 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            dataOUT1, dataOUT2 : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
            clock : IN STD_LOGIC );
END ENTITY fwdLogic;

ARCHITECTURE std OF fwdLogic IS
    -- Register for data3 to not go into an infinite feedback loop.
    SIGNAL data3Reg : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL override1, override2 : BOOLEAN := FALSE;
BEGIN
    -- When an override flag is active, dataOutX gets the previous data3 value.
    dataOUT1 <= data3Reg WHEN override1 ELSE dataIN1;
    dataOUT2 <= data3Reg WHEN override2 ELSE dataIN2;

    PROCESS (clock)
    BEGIN
        IF (RISING_EDGE(clock)) THEN
            data3Reg <= data3;

            override1 <= FALSE;
            override2 <= FALSE;

            -- If addrX is still being updated, override output with the previous data3.
            IF (addr1 = addr3) THEN
                override1 <= TRUE;
            END IF;

            IF (addr2 = addr3) THEN
                override2 <= TRUE;
            END IF;
        END IF;
    END PROCESS;
END ARCHITECTURE std;
```

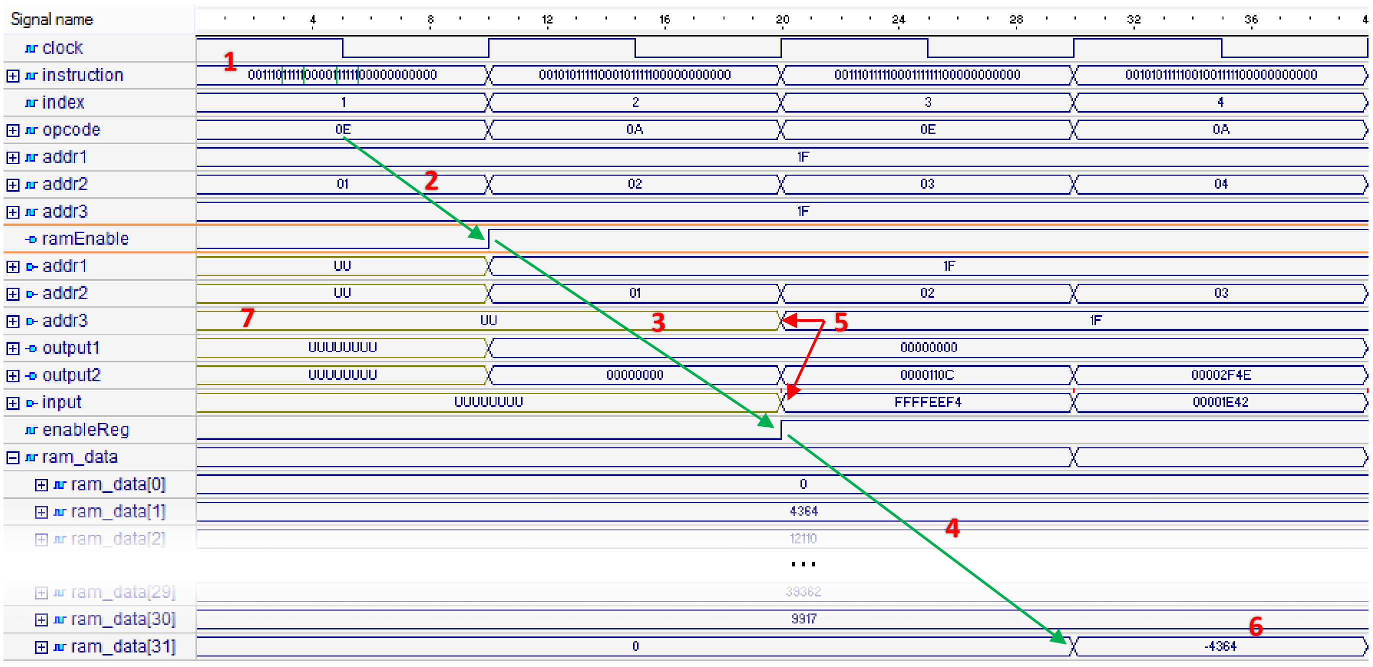
proc.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY proc IS
    PORT (  instruction : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            clock : IN STD_LOGIC );
END ENTITY proc;

ARCHITECTURE std OF proc IS
    SIGNAL opcode, opDelay : STD_LOGIC_VECTOR(5 DOWNTO 0);
    SIGNAL ramEnable : STD_LOGIC;
    SIGNAL addr1, addr2, addr3 : STD_LOGIC_VECTOR(4 DOWNTO 0);
    SIGNAL data1, data2, data3 : STD_LOGIC_VECTOR(31 DOWNTO 0);
    -- Overidden outputs.
    SIGNAL data1_over, data2_over : STD_LOGIC_VECTOR(31 DOWNTO 0);
BEGIN
    -- Wire all blocks together.
    g1: ENTITY work.instructReg(std)
        PORT MAP (instruction, opcode, opDelay, addr1, addr2, addr3, clock);
    g2: ENTITY work.opCheck(std)
        PORT MAP (opcode, ramEnable);
    g3: ENTITY work.ram(std)
        PORT MAP (addr1, addr2, addr3, data1, data2, data3, ramEnable, clock);
    g4: ENTITY work.fwdLogic(std)
        PORT MAP (addr1, addr2, addr3, data1, data2, data3,
                  data1_over, data2_over, clock);
    g5: ENTITY work.alu(std)
        PORT MAP (data1_over, data2_over, opDelay, data3);
END ARCHITECTURE std;
```

procTest.vhd

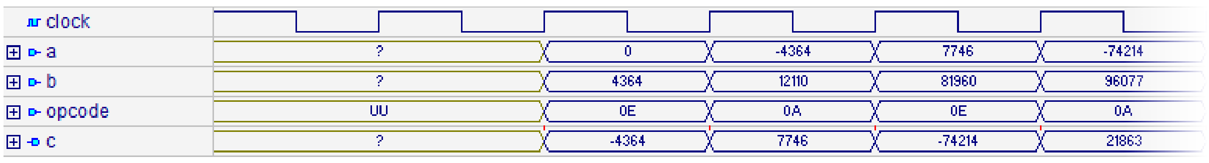
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY procTest IS
END ENTITY procTest;

ARCHITECTURE std OF procTest IS
    SIGNAL instruction : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL clock : STD_LOGIC;
    -- The following signals are used for generating instructions.
    SIGNAL opcode : STD_LOGIC_VECTOR(5 DOWNTO 0);
    SIGNAL addr1, addr2, addr3 : STD_LOGIC_VECTOR(4 DOWNTO 0);
    SIGNAL index : INTEGER := 1; -- Initialise to 1 to read from register 1, not 0.
BEGIN
    g1: ENTITY work.proc(std)
        PORT MAP (instruction, clock);

    -- Algorithm corresponding to student ID 3 (third last; even-odd).
    -- Add for every even index, subtract for every odd, no-op when index>30.
    opcode <= B"111111" WHEN index > 30 ELSE B"001010" WHEN (index MOD 2 = 0) ELSE B"001110";
    -- Read value of sum thus far from register 31.
    addr1 <= B"11111";
    -- addr2 gets 5-bit index.
    --addr2 <= CONV_STD_LOGIC_VECTOR(index, 5);
    addr2 <= STD_LOGIC_VECTOR(TO_UNSIGNED(index, 5));
    -- Write to address 31; addr3 could alternatively just get addr1 as they are the same.
    addr3 <= B"11111";
    -- Concatenate all components.
    instruction <= opcode & addr1 & addr2 & addr3 & "00000000000";

    PROCESS (clock)
    BEGIN
        -- Increment index on every rising edge of the clock.
        IF (index < 31 AND RISING_EDGE(clock)) THEN
            index <= index + 1;
        END IF;
    END PROCESS;

    PROCESS
    BEGIN
        -- Clock at an arbitrary period of 10ns.
        clock <= '1';
        WAIT FOR 5 NS;
        clock <= '0';
        WAIT FOR 5 NS;
    END PROCESS;
END ARCHITECTURE std;
```

Appendix 2 – Simulation



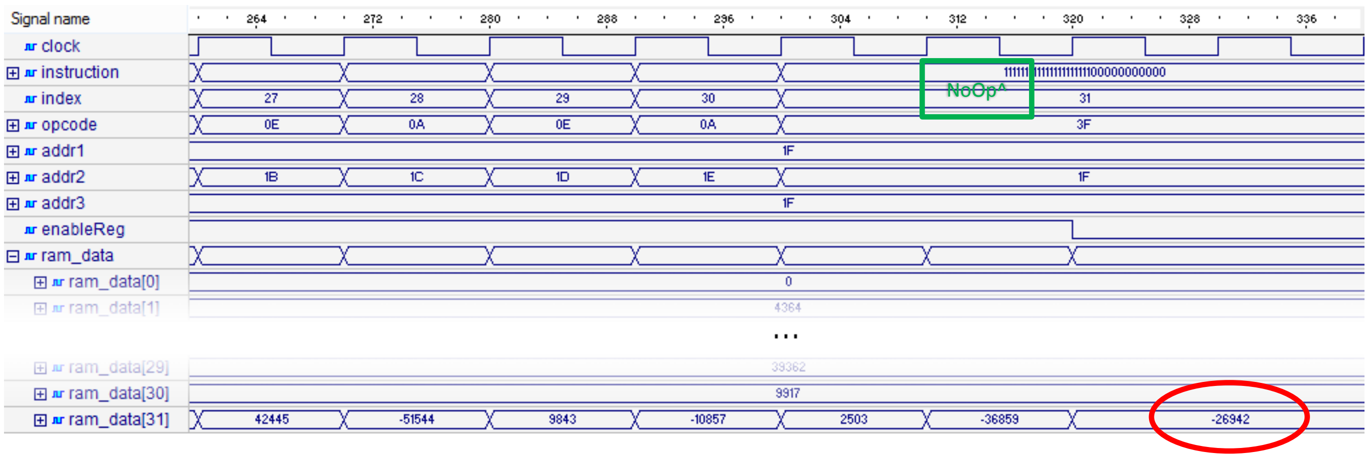
InstructReg, opCheck, and RAM at Start

1. An instruction is fed in and decomposed into an opcode and three addresses.
2. The opcode is checked and deemed valid so ramEnable (in opCheck) is set high.
3. The RAM entity (starting from the second orange line) has already received addr1 and addr2 but not addr3 as it is delayed. On the next rising edge of the clock signal, data1 and data2 are retrieved and sent to the ALU on the same rising edge as can be seen below.
4. It is at this point that the RAM unit enables writing (from the delayed enable flag) and writes the answer returned from the ALU within the third tick (realistically there would be a little delay).
5. The correct answer is returned from the ALU and written to RAM on the same tick.
6. One clock cycle after the enableReg has been set high, the answer appears in register 31 as it should. It takes 3 clock cycles for the answer to be written, but this is negligible when considering overall throughput.
7. All undefined values in the start are either initialised (e.g. ramEnable), or have no effect until they are overwritten (e.g. input which in turn means output1 and output2 also have no effect).

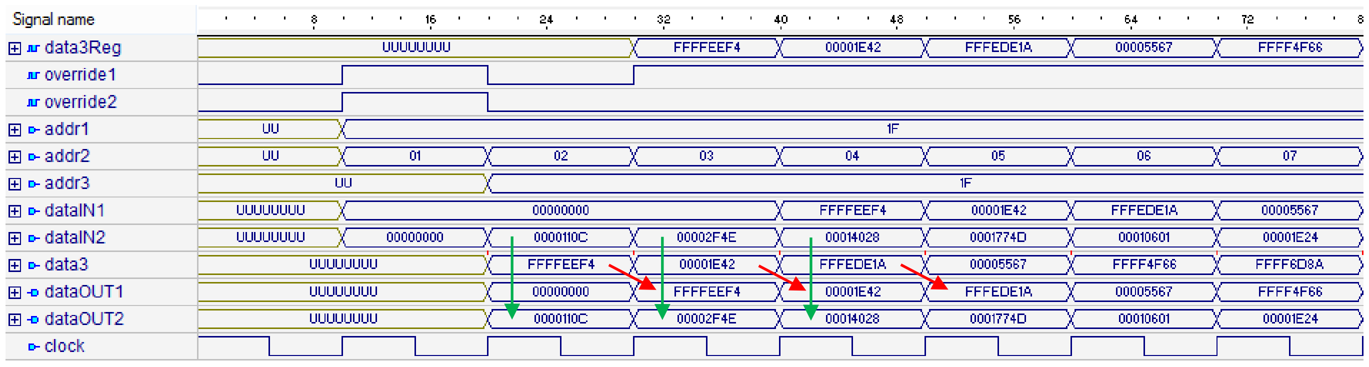


ALU

The ALU (and RAM one clock cycle later) shows that the answers yielded thus far are correct.



InstructReg, opCheck, and RAM at End


Signal name
8 16 24 32 40 48 56 64 72
data3Reg
UUUUUUUU FFFFEEF4 00001E42 FFFEDE1A 00005567 FFFF4F66
override1
override2
addr1
UU 1F
addr2
UU 01 02 03 04 05 06 07
addr3
UU 1F
dataIN1
UUUUUUUU 00000000 FFFFEEF4 00001E42 FFFEDE1A 00005567
dataIN2
UUUUUUUU 00000000 0000110C 00002F4E 00014028 0001774D 00010601 00001E24
data3
UUUUUUUU FFFFEEF4 00001E42 FFFEDE1A 00005567 FFFF4F66 FFFF6D8A
dataOUT1
UUUUUUUU 00000000 FFFFEEF4 00001E42 FFFEDE1A 00005567 FFFF4F66
dataOUT2
UUUUUUUU 0000110C 00002F4E 00014028 0001774D 00010601 00001E24
clock

Forwarding logic in Action

(Not that override1 is pretty much forever true and data1 is being constantly overridden; see report)