Digital Hardware project

2 operand machine with hardcoded ALU output

Luc Ndeze	Matthijs Reus	Bas Zutt	Marianna Protopsalti
s2509865	s2519402	s2603888	s2510855

Miguel Beleza Leong Seixas e Sousa
s2551586

November 10, 2021

Contents

1 Introduction

The processor is designed to be able to calculate prime numbers using the sieve of Eratosthenes. The processor is a 2 address machine with a 16 bits central processing unit (CPU) with 12 instructions. It contains 64 16-bit registers of which one hard coded is to store the result of every instruction. The jump address and main memory address are fetched from a selected register. An assembler is used to convert the assembly code into machine code, which the processor executes.

2 Instruction Set

The instruction set architecture of the processor consists of 12 instructions. The selection of the 12 instructions is made based on the requirements of the application (see section 2.5) and looking at the fundamentals of other processors and functions. For the processor created in this project a hard coded store register (%r63) is used to store the result of the arithmetic operations as well as the generate, inverse, load and store operations (Formats 2,5 figure 1).

Figure 1: Machine code formats.

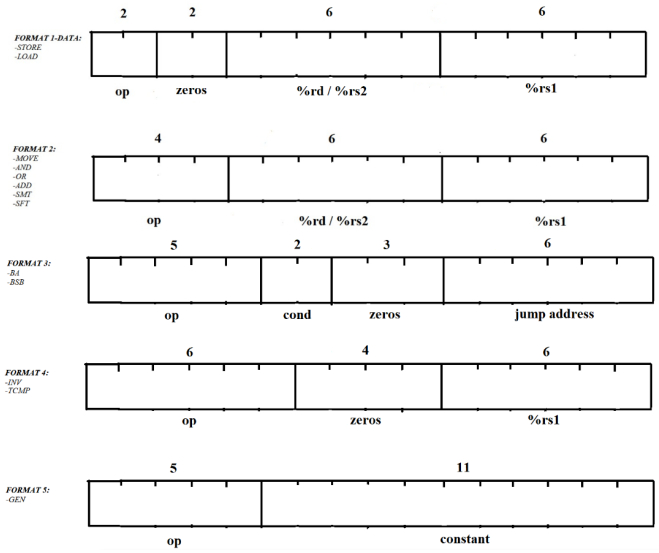


Table 1: ISA formats.

Format 1

Operation	*Instruction*
01	STORE
00	LOAD

Format 2

Operation	*Instruction*
1000	MOVE
1001	AND
1010	OR
1011	ADD
1100	SFTL
1101	SFTR

Format 3

Operation	*Instruction*
11110	BA
11111	BSB

Format 4

Operation	*Instruction*
111000	INV
111001	TCMP

Format 5

Operation	*Instruction*
11101	GEN

Table 2: Description of instructions.

Operations:	**Description:**
load	Load information from main memory into a register
store	Store information from a register into main memory
move	Move information from one register to another register
ba	Branch always
bsb	Branch based on a status bit
or	Perform a bitwise logic OR-operation
and	Perform a bitwise logic AND-operation
st	Smaller than
add	Perform addition
inv	Invert all the bits
tcmp	Invert all the bits and perform addition with 1
sft	Shift to the right or left (if bit 5=0 ->shift right, if bit 5=1, shift left)

2.1 Memory Instructions

The memory instructions work by retrieving a main memory address from a register. This register is selected by %rs1. The data is stored into or loaded from register %rs2/%rd. Having the main memory address in a register is advantageous to loop through all the main memory addresses by incrementing the register the address is in.

2.2 Initializing a constant

A separate generate function (format 5, figure 1) is used to load a constant into a register, removing the need for register-constant or register-register operation designation. This results in space for a total of 6 bit register addresses in the machine code. The constant will be loaded into %r63.

2.3 Moving register data

All ALU results are stored in %r63 and adding a constant is not possible. So in order to move data from one register into another the move function is used.

2.4 Inverting numbers

The invert(INV) function can be useful for logical operations like creating a NAND, NOR, XOR and XNOR. The two's complement function allows for easy implementation of negative numbers, since it allows a number to become negative in 1 line of code instead of 4(inverting, move the result, generating a 1 and adding it). This function is useful for subtracting one from an initially generated i and then branch out when i becomes 0 and is also used in the DIV/MOD function.

2.5 Application

The chosen application is calculating prime numbers. This calculation is computationally intensive and a well covered subject. The method used for calculating prime numbers is the sieve of Eratosthenes. This method makes a list of a range of numbers with the lower boundary being two. It then removes all values where $Xmod2 = 0$. It picks the next remaining value in the list and goes over all the values again checking for $XmodNextValue = 0$, until there is no next value in range. To be able to find more prime numbers the list is initialized with numbers non-divisible by 2 and 3, increasing our potential upper bound 6 times. The ultimate goal of this project is to calculate up to the 500th prime number, which is 3571. We split the code into 4 parts:

- 1. DIV/MOD function
- 2. Initializing list with numbers non-divisble by 2 and 3
- 3. Removing all empty entries in main memory and consolidating the data
- 4. The sieve of Eratosthenes being the file combining all functions.

The control unit also controls the operation mode of the main memory by changing the mem write/mem read bits.

3 Processor Design

3.1 Schematic of Data path

The microarchitecture of a processor is responsible for fetching the instructions to be followed, decode the operation code, read the operands and sources from either the register or main memory, execute the instruction and finally store the result. Below in Figure 2, is a figure describing the datapath of the processor. The actual code is divided into a similar way as the dataPath. The control unit and instruction set are combined into one vhdl file. The register file, the ALU, the main memory and the microstore have their own vhdl files. The data section consists of the register file and ALU, which can be referred to as the datapath, therefore portmapped in datapath. The control section of the microarchitecture is the microstorage, control unit and instruction set, which is portmapped in the processor file. The uniqueness of the design is visible in the ALU output, Bus C, being mapped directly to one register. Furthermore, the Bus A containing the address from the data of a register file is another visible unicum.

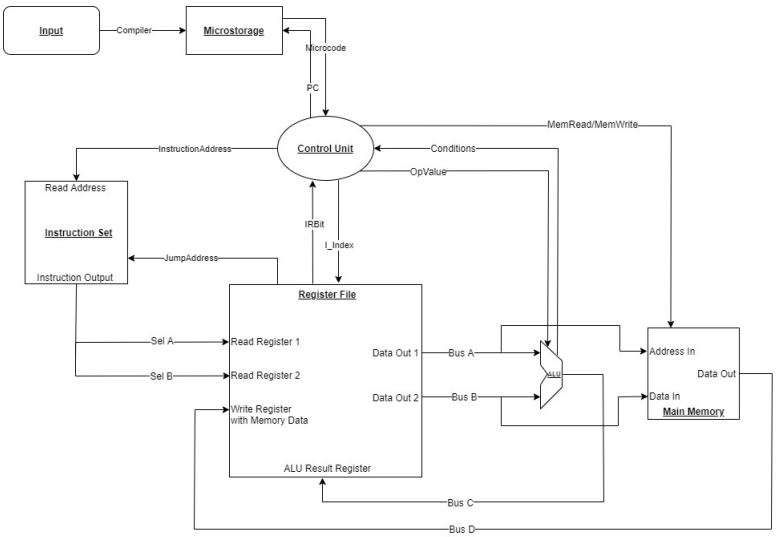


Figure 2: General Processor Data Path

3.2 Controller

The controller is the core of the processor, controlling all data flowing through the processor as seen in the datapath. The controller first decodes the microcode received from the microstore, the decoded input format is found in the instruction set, which has been implemented inside the controller using cases. From there it finds the corresposding format and sends the necessary decoded microcode data through Sel_A and Sel_B, and the current instruction index (I_Index), such that the register can then process the data as necessary and allocate the appropriate data to the Buses. The buses will then either go to the ALU or main memory, if to the ALU, an operation code is provided to the ALU by the controller, allowing the correct operation to be undergone. If the buses go to main memory, the controller will either send a memory write or memory read bit to the main memory and the main memory will read accordingly. Some components of the processor respond back to the controller, the register file does it through the IR_Bit, which tells the controller an instruction has been done and the ALU communicates the condition bits used for branching. Lastly the program counter is built into the controller and is communicated to the microstore, such that a microcode can be retrieved and decoded in the controller.

3.3 Datapath

The Datapath file portmaps the ALU and registerFile together and has an entity that can be portmapped to the processor file.

3.3.1 Register File

The register file gets the input i_index from the control unit and this selects the operating mode of the register file. It makes sure that for each instruction the right action is taken. The register file also creates the actual place where the registers are stored, hence its name. Bus_C (output ALU) is also connected to the registerFile and will always store the ALU results into %r63.

3.3.2 ALU

The ALU receives an OPvalue and uses cases to do the correct operation and puts the result on Bus_C.

3.4 Main Memory

For the Main Memory an array is created of 4096 rows where each row is 8 bits. That is because in the ISA-formats (figure 1), there are 11 bits to assign to a main memory address and using a hidden bit allows to have $2^{12} = 4096$ addresses. If a 'word' is stored into main memory, the main memory always uses two consecutive addresses to store it in, where the 8 lowest significant bits of the word are always stored in an address that is dividable by two since this is concatenated with '0'. This makes it possible for the test environment to make the main memory byte addressable and still to display an entire word on the FPGA.

3.5 Another application

A user of the processor can use the assembly code syntax in figure 5, with explanations, to make another program. Within the C++ code for the assembler the local datapath needs to be changed, this position is marked with an documentation comment. Running the assembler will convert the assembly program into machine code. The results will be stored in the target.txt and this should be pasted over in the microstorebhv.vhdl file. Lastly, compiling and simulating the testbench file will be required to run the code. After these steps another application will run with this processor.

4 Test Environment

In order to test the created processor physically on the FPGA, a test environment is created (see figure 3). From left to right, the first switch is entering the debug mode. The following 6 switches can be used to set a certain step size, which is determined by multiplying the scalars of the switches that are set to '1'. Switch 8 and 9 are used to specify whether we want to read the content or address of either the main memory, the registers, the microstore or the control branch logic for which the formats are specified in the figure. The last switch determines whether we want to display the address or the content of that address.
Next, we use, again from left to right, the first button to implement the step mode where one instruction is fed to the ALU per button click. The second button is used to decrement and the third button is used to increment the address that we want to display or which we want to display the content from. The fourth button is used to reset all the addresses such that in every mode (main memory, register, etc.) address zero is displayed again instead of having to use the decrement button multiple times to go back to address zero.
The test environment is created such that complete 'words' will be displayed. This means that if a certain 'word' address is created, that address will be concatenated with '0' and '1' respectively to create two byte addressable consecutive addresses (the same way in which a 'word' is stored into main memory) and both addresses will be displayed on the six 7 segment displays. This allow the test environment to display complete 'words' (as long as it is aligned addressed) and still be byte addressable instead of only displaying the 8-bit content of 1 address.
Both the registerfile and the microstore don't require such a construction since only main memory was chosen to be byte addressable.

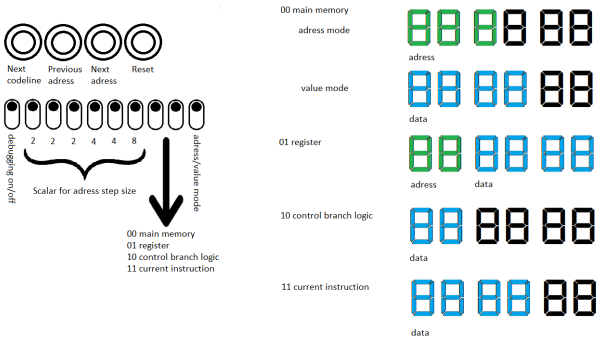


Figure 3: Test environment.

4.1 ModelSim

The test environment succesfully compiled in ModelSim, showing the expected behaviour. However, due to time lack only the step mode couldn't be implemented succesfully.

First the main memory is tested by giving the switch string the right bit pattern (the one with the right format for main memory) using a button signal that oscillates every $20ns$ to represent the press and release of a button (see figure 15). It is visible that the address increments correctly with the given step size of 8 addresses. The 7 segment displays then show the 2 consecutive addresses that together represent the stored 'word'. The same happens for when the decrement button is pressed. The register mode and microstore mode are simulated in exactly the same way for which results are shown in figure 16 and figure 14 respectively. Unfortunately the content of the addresses couldn't yet be displayed since the test environment has not been compiled with a filled memory.
In order for the test environment to be succesfully compiled, a QSF file also had to be made. Therefore the datasheet of the FPGA was used [6].

5 Simulation and Synthesis

5.1 ModelSim

For the simulations, we began by testing the code without a microstore, in order to have correctly functioning operations before implementing an assembler. This meant that we began by testing our individual formats with the following testbench (see figures 12,13). After resolving some datapath issues and without taking timings into account, we had a working simulation (see figure 11), the results in main memory and the register file can be seen here (see figures 9,10). With all operation formats working, aside from branching, a microstore was intruduced and an assembler was created.

5.1.1 Assembler

The assembler was written in C++ using CodeBlocks. The assembler makes the assembly code case insensitive and removes all empty lines, tabs and spaces. Then it splits the code up at the places of the comma's and sends the first three elements into another function. This function string compares the op code to convert the op code to binary. Each op code has its own case in which it calls functions to convert the register address or generate constant to a binary number of corresponding

size and fills up the zeroes. The code then adds the necessary code to be able to directly implement the target.txt into the microstorebhv.vhdl.

5.1.2 Simulations with Microstore

Now that a microstore was added we began by testing the division code for the final application, however many timing issues were found, these were solved by adding a few clock iterations for each instruction, such that all components had enough time to perform their operations correctly. Once the timings were fixed, the branching was simulated both for branching on bit and branching always, which functioned with a few simple changes in relation to timings. The following was the final simulation for the division using the assembled microstore (see figure 8) and its register results (see figure6) for the following assembly code(sim of div assembly code). Finally we were ready to test our final application, sadly we did not have enough to time to analyse this simulatie17).

5.2 Quartus

The entire processor synthesized successfully. There was a warning about the clock not being specified. This is because we did not make a .qsf and .sdc specifying a clock and the pin connections. The synthesized design looked as shown below and it was as expected, since all signals were correctly connected.

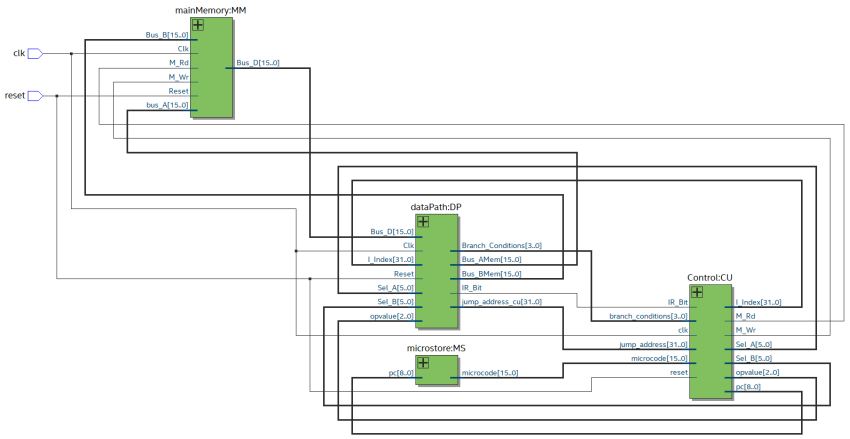


Figure 4: Quartus Design Schematic

6 Discussion

Branching was changed with regards to the project plan from jumping to a fixed address to jumping directly to an address in a register. This allows for a changing branching location, used primarily in a function being called twice. The instruction also changed for the load/store functions. The main memory address is also found in a register specified in the machine code. This allows all the main memory addresses to be accessed in a single loop by simply incrementing the address in the register. The simulation of the DIV/MOD function operated as expected, as $42DIV8 = 5, 42MOD8 = 2$. The assembly code for the entire sieve was created. However, when simulated it did not work as intended and there was no time to debug the code. Calculating 500 prime numbers was therefore a

bit of an overestimation of what our team could design, program and debug in a bit more than a week. Moreover, the debug mode was tested successfully, yet it had not been implemented into the most recent version of our processor. There was no time to test the design on the FPGA.

7 Conclusions

In this project a processor is designed and simulated. The processor is a 2 address machine with a hardcoded output register for the ALU. The word size for instructions and data is 16 bits. The register file has 64 registers and main memory 2047 addresses with one hidden bit for debug mode byte addressability. The load/store command is done uniquely by not accessing the main memory address in the machine code, but instead by referring to a register address containing the main memory address. The branch function also looks into a register for the jump address. We were not able to calculate 500 prime numbers. More time would is needed for debugging. The DIV was successfully simulated, however the MOD function does could not be simulated in time, which is a fundamental step towards the sieve. The assembler successfully converted assembly code into machine code. Overall the project is not a full success, since the complete application is not functional, however a working processor that can perform all formats from the instruction set, in simulation, was created.

References

[1] Wikipedia, *Prime numbers* https://en.wikipedia.org/wiki/Prime_number

[2] cplusplus.com, *vector::size* https://www.cplusplus.com/reference/vector/vector/size/

[3] stackoverflow, *How to get an element at specified index from c++ List* https://stackoverflow.com/questions/16747591/how-to-get-an-element-at-specified-index-from-c-list/27977053 may 2013.

[4] V. Krishna, P. Jain, A. Prakesh, *How to find prime numbers* https://www.vedantu.com/maths/how-to-find-prime-numbers

[5] M.J. Murdocca and V.P. Heuring, *Computer Architecture and Organization*, John Wiley & Sons, Inc. , 2007.

[6] Terasic Technologies *DE1-Soc user manual*https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-1004282204-de1-soc-user-manual.pdf , 2003-2014

8 Appendix

Figure 5: Assembly syntax

Assembly code syntax	
LOAD, 5, 8	Load the address in register 5 into register 8
STORE, 2, 7	Store to address in register 2 from register 7
MOVE 12, 6	Move contents of register 12 into register 6
OR, 3, 15	Perform bitwise logical OR-operation with the contents of register 3 and register 15
AND, 5, 11	Perform bitwise logical AND-operation with the contents of register 5 and register 11
ADD, 2, 3	Add the contents of register 2 with the contents of register 3
INV, 4	Invert the contents of register 4
TCMP, 9	Take the two's compliment of the contents of register 9 by inverting the bits and adding 1
SHIFTR, 14, 8	Shift the contents of register 14 an amount of the contents of register 8 to the right
SHIFTL, 23, 5	Shift the contents of register 23 an amount of the contents of register 5 to the left
BA, 6	Branch always to the address that register 6 contains
BSB, 7	Branch on status bit to the address that register 7 contains

Figure 6: Simulation of Division result

```
sim:/testbench/processor/DP/RF/register_file @ 9869407 ps
63 :  0  0  0  0  0  0  0  0
55 :  0  0  0  0  0  0  0  0
47 :  0  0  0  0  0  0  0  0
39 :  0  0  0  0  0 36 19 29
31 :  0  0  0  0  0  0  0  0
23 :  0  0  0  0  0  0  0  0
15 :  0  0 -8  2  0  1  0  0
 7 :  0  0  0  2  5  8 42  0
```

Figure 7: Simulation of division

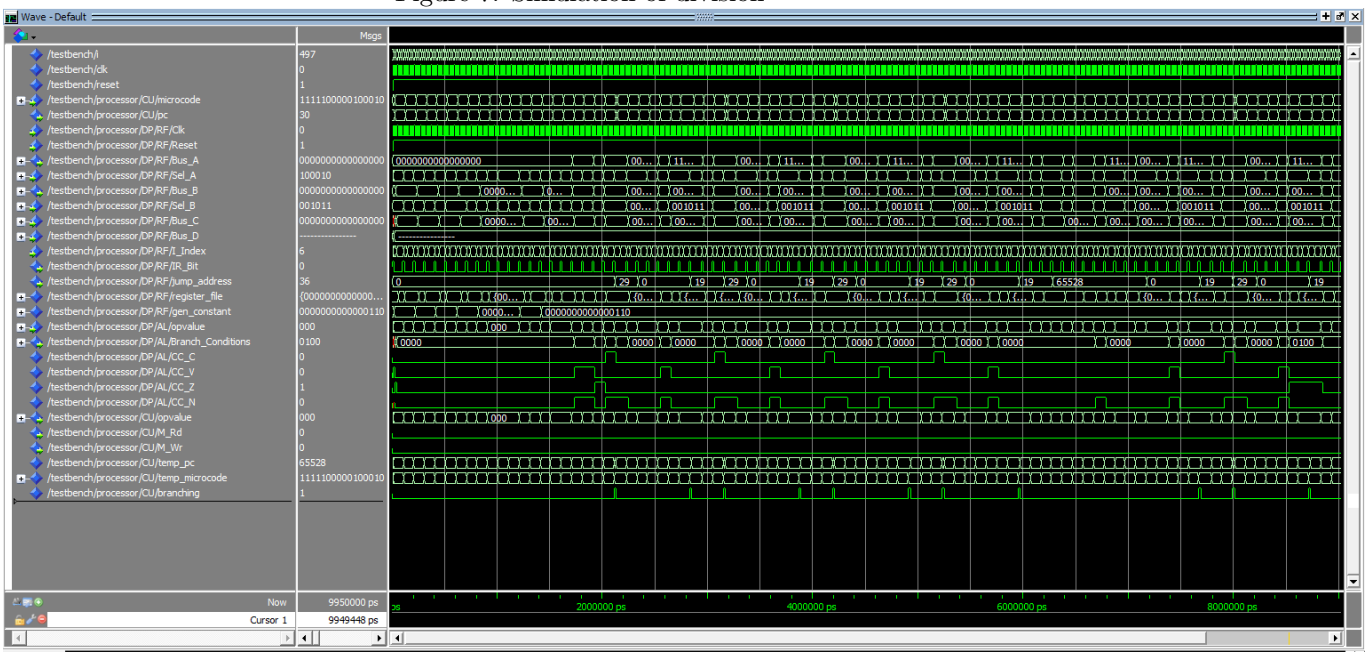


Figure 8: Assembly code of division

```
GEN, 42
MOVE, 63, 1 --X
GEN, 8
MOVE, 63, 2 --Y
GEN, 29
MOVE, 63, 32 --jump1B
Gen, 19
MOVE, 63, 33 --jump2B
Gen, 36
MOVE, 63, 34 --Bout
MOVE, 0, 3
MOVE, 0, 4
GEN, 1
MOVE, 63, 10
GEN, 15
MOVE, 63, 11
MOVE, 1, 12
TCMP, 2
MOVE, 63, 13
SFTR, 12, 11 --jump2B
ADD, 63, 13
BSB, N, 32
SFTL, 10, 11
ADD, 3, 63
MOVE, 63, 3
SFTL, 13, 11
ADD, 12, 63
MOVE, 63, 12
MOVE, 12, 4
ADD, 11, 0 --jump1B
BSB, Z, 34
MOVE, 63, 11
TCMP, 10
ADD, 11, 63
MOVE, 63, 11
BA, 33
```

Figure 9: Simulation without microstore of main memory

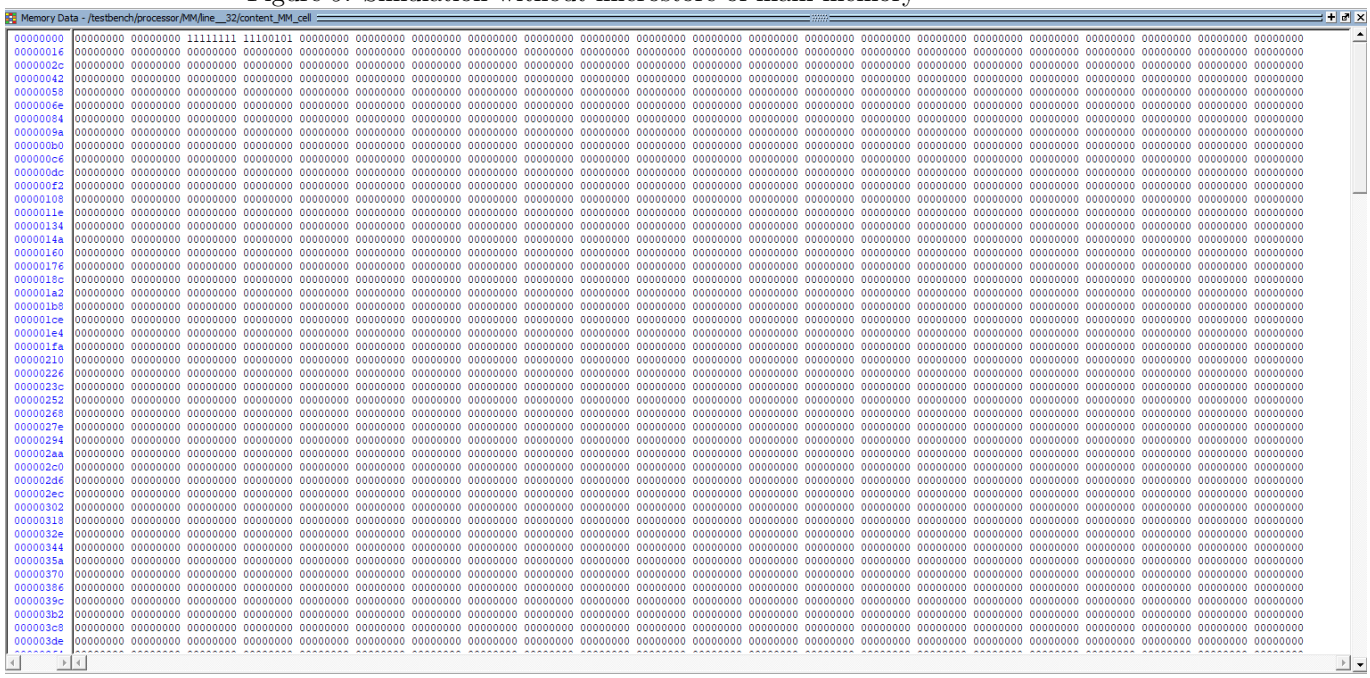


Figure 10: Simulation without microstore of register file

```
sim:/testbench/processor/DP/RF/register_file @ 75999999943 ps
63 : 1111111111100101 1111111111100101 1111111111100101 0000000000000000 0000000000000000 0000000000000000 0000000000000000 ...
55 : 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 ...
47 : 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 ...
39 : 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 ...
31 : 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 ...
23 : 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 ...
15 : 0000000000000000 0000000000000000 0000000000000000 0000000000000000 1111111111100101 0000000000011011 0000000000110110 ...
 7 : 0000000000001010 0000000000010001 0000000000010001 0000000110010001 1111111111111111 0000000000000001 1111111111111110 ...
```

Figure 11: Simulation without microstore

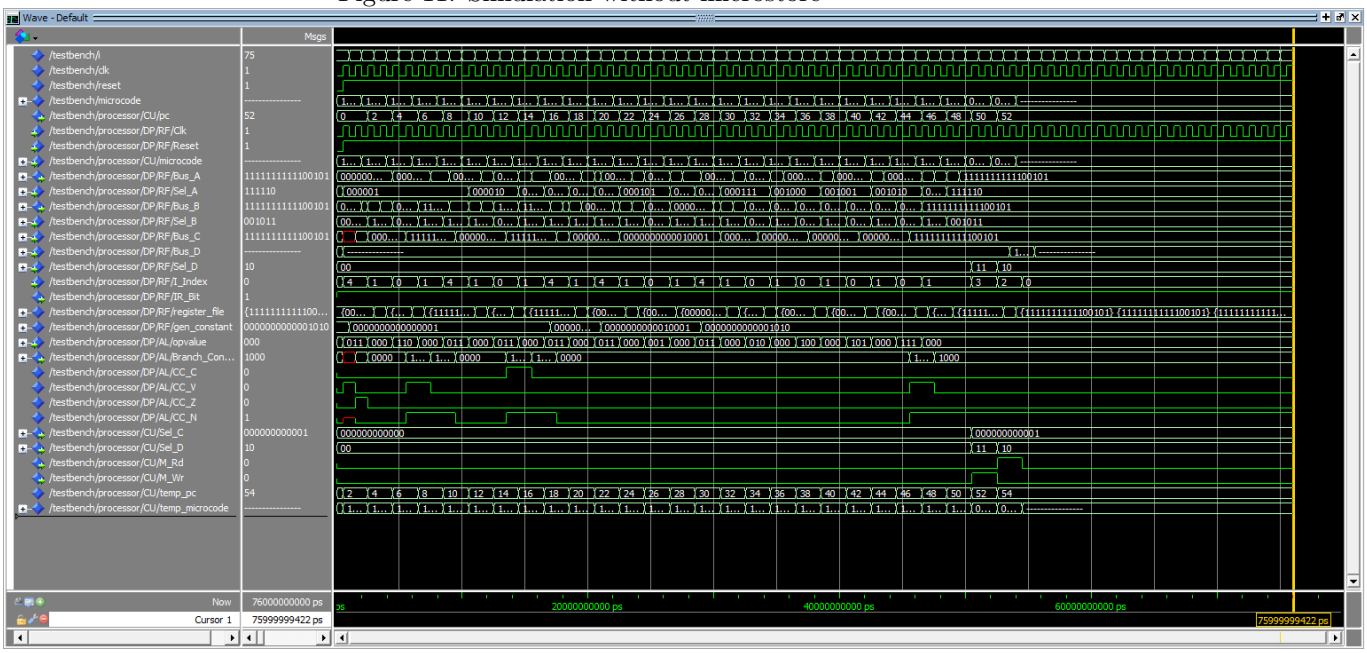
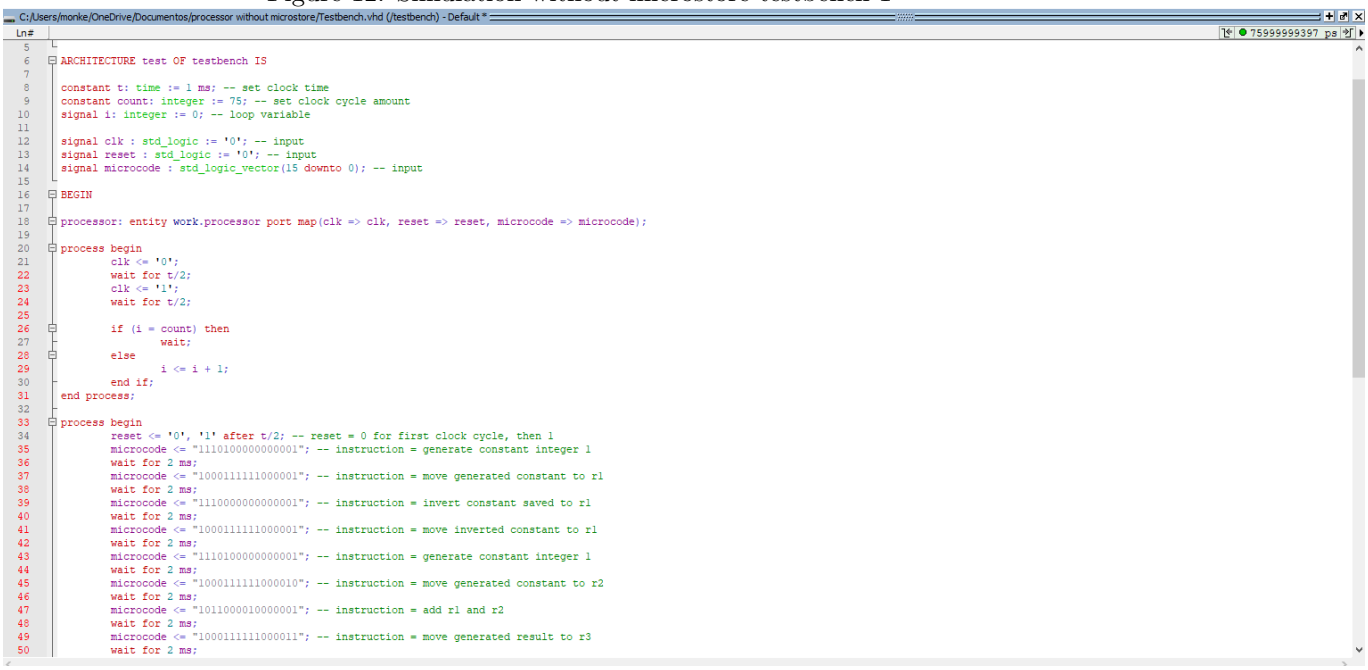
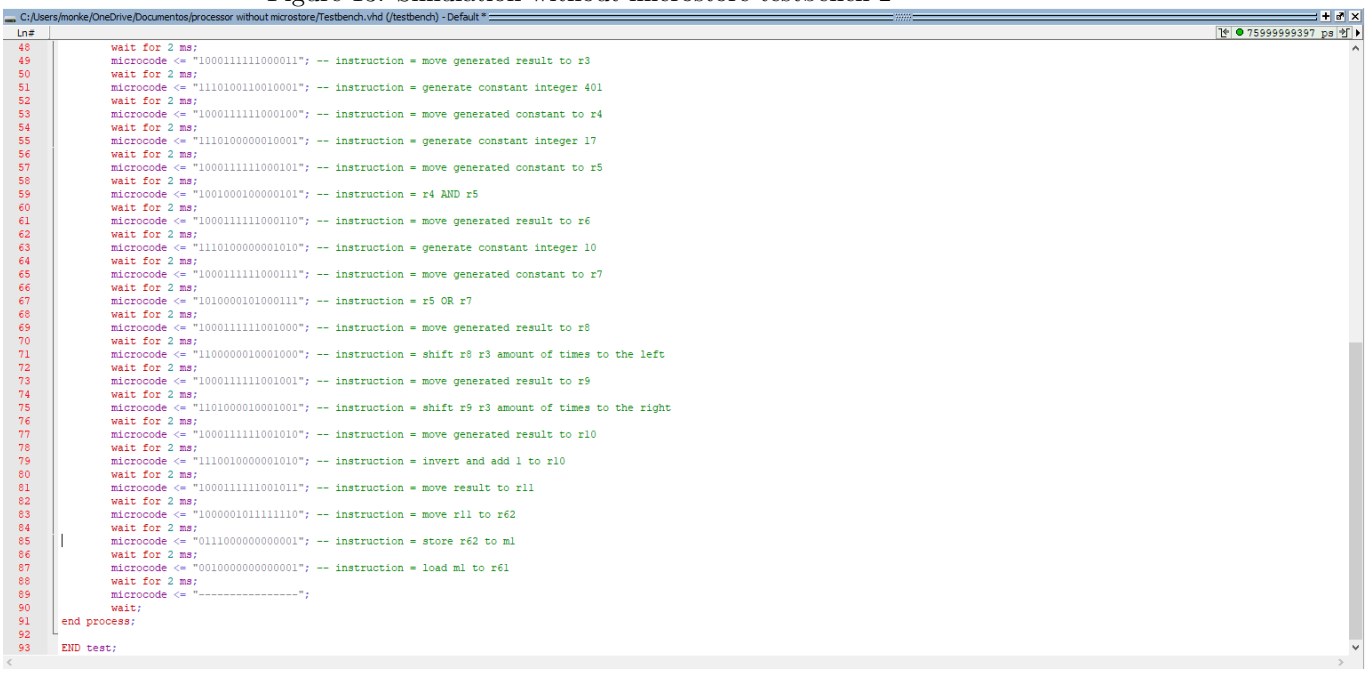


Figure 12: Simulation without microstore testbench 1




```
ARCHITECTURE test OF testbench IS

constant t: time := 1 ms; -- set clock time
constant count: integer := 75; -- set clock cycle amount
signal i: integer := 0; -- loop variable

signal clk : std_logic := '0'; -- input
signal reset : std_logic := '0'; -- input
signal microcode : std_logic_vector(15 downto 0); -- input

BEGIN

processor: entity work.processor port map(clk => clk, reset => reset, microcode => microcode);

process begin
        clk <= '0';
        wait for t/2;
        clk <= '1';
        wait for t/2;

        if (i = count) then
                wait;
        else
                i <= i + 1;
        end if;
end process;

process begin
        reset <= '0', '1' after t/2; -- reset = 0 for first clock cycle, then 1
        microcode <= "1110100000000001"; -- instruction = generate constant integer 1
        wait for 2 ms;
        microcode <= "1000111111000001"; -- instruction = move generated constant to r1
        wait for 2 ms;
        microcode <= "1110000000000001"; -- instruction = invert constant saved to r1
        wait for 2 ms;
        microcode <= "1000111111000001"; -- instruction = move inverted constant to r1
        wait for 2 ms;
        microcode <= "1110100000000001"; -- instruction = generate constant integer 1
        wait for 2 ms;
        microcode <= "1000111111000010"; -- instruction = move generated constant to r2
        wait for 2 ms;
        microcode <= "1011000010000001"; -- instruction = add r1 and r2
        wait for 2 ms;
        microcode <= "1000111111000011"; -- instruction = move generated result to r3
        wait for 2 ms;
```

Figure 13: Simulation without microstore testbench 2



```
48          wait for 2 ms;
49          microcode <= "1000111111000011"; -- instruction = move generated result to r3
50          wait for 2 ms;
51          microcode <= "1110100110010001"; -- instruction = generate constant integer 401
52          wait for 2 ms;
53          microcode <= "1000111111000100"; -- instruction = move generated constant to r4
54          wait for 2 ms;
55          microcode <= "1110100000010001"; -- instruction = generate constant integer 17
56          wait for 2 ms;
57          microcode <= "1000111111000101"; -- instruction = move generated constant to r5
58          wait for 2 ms;
59          microcode <= "1001000100000101"; -- instruction = r4 AND r5
60          wait for 2 ms;
61          microcode <= "1000111111000110"; -- instruction = move generated result to r6
62          wait for 2 ms;
63          microcode <= "1110100000001010"; -- instruction = generate constant integer 10
64          wait for 2 ms;
65          microcode <= "1000111111000111"; -- instruction = move generated constant to r7
66          wait for 2 ms;
67          microcode <= "1010000101000111"; -- instruction = r5 OR r7
68          wait for 2 ms;
69          microcode <= "1000111111001000"; -- instruction = move generated result to r8
70          wait for 2 ms;
71          microcode <= "1100000010001000"; -- instruction = shift r8 r3 amount of times to the left
72          wait for 2 ms;
73          microcode <= "1000111111001001"; -- instruction = move generated result to r9
74          wait for 2 ms;
75          microcode <= "1101000010001001"; -- instruction = shift r9 r3 amount of times to the right
76          wait for 2 ms;
77          microcode <= "1000111111001010"; -- instruction = move generated result to r10
78          wait for 2 ms;
79          microcode <= "1110010000001010"; -- instruction = invert and add 1 to r10
80          wait for 2 ms;
81          microcode <= "1000111111001011"; -- instruction = move result to r11
82          wait for 2 ms;
83          microcode <= "1000001011111110"; -- instruction = move r11 to r62
84          wait for 2 ms;
85          microcode <= "0111000000000001"; -- instruction = store r62 to m1
86          wait for 2 ms;
87          microcode <= "0010000000000001"; -- instruction = load m1 to r61
88          wait for 2 ms;
89          microcode <= "----------------";
90          wait;
91      end process;
92
93      END test;
```

Figure 14: Microcode test mode

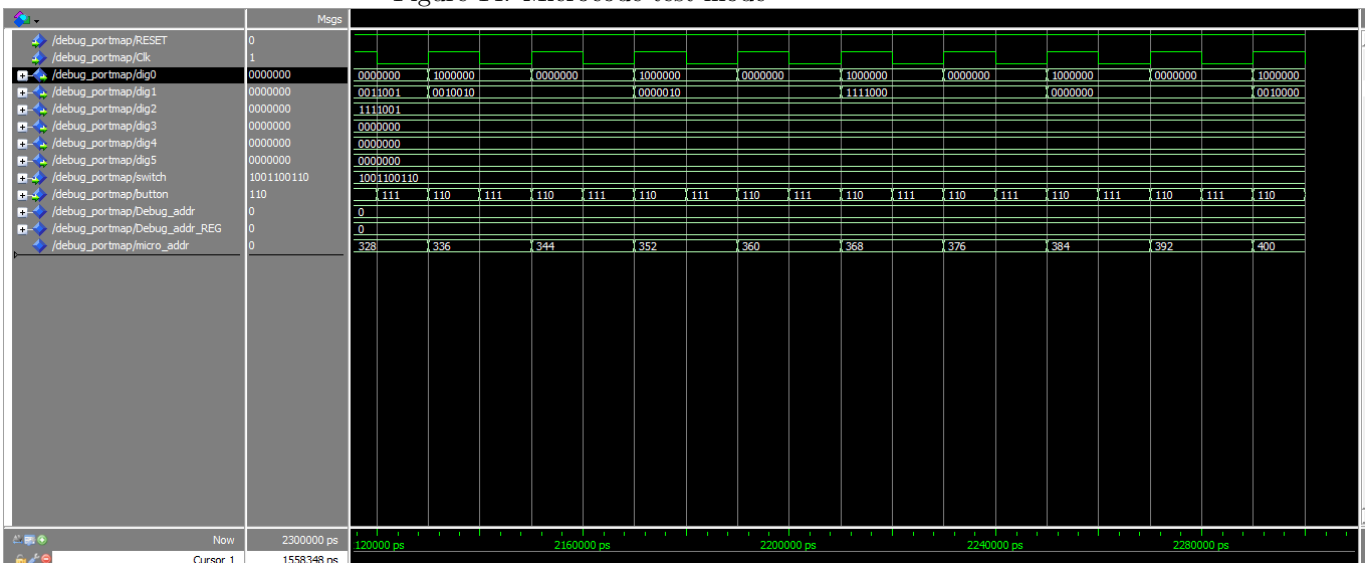


Figure 15: Main Memory test mode

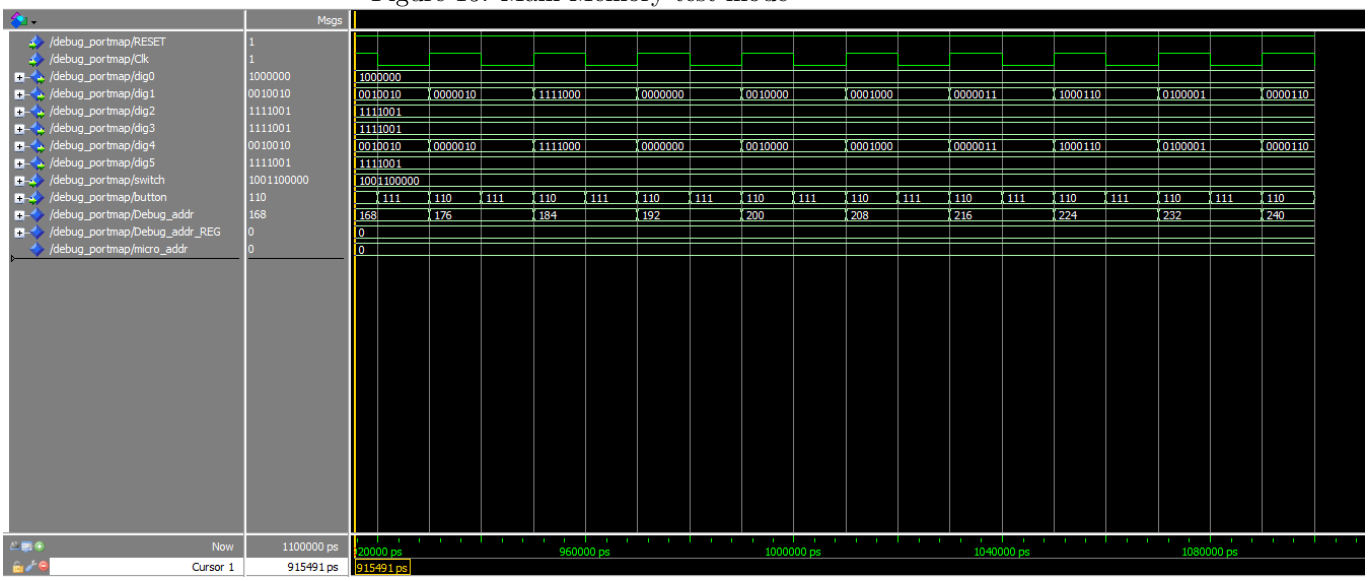


Figure 16: Register test mode

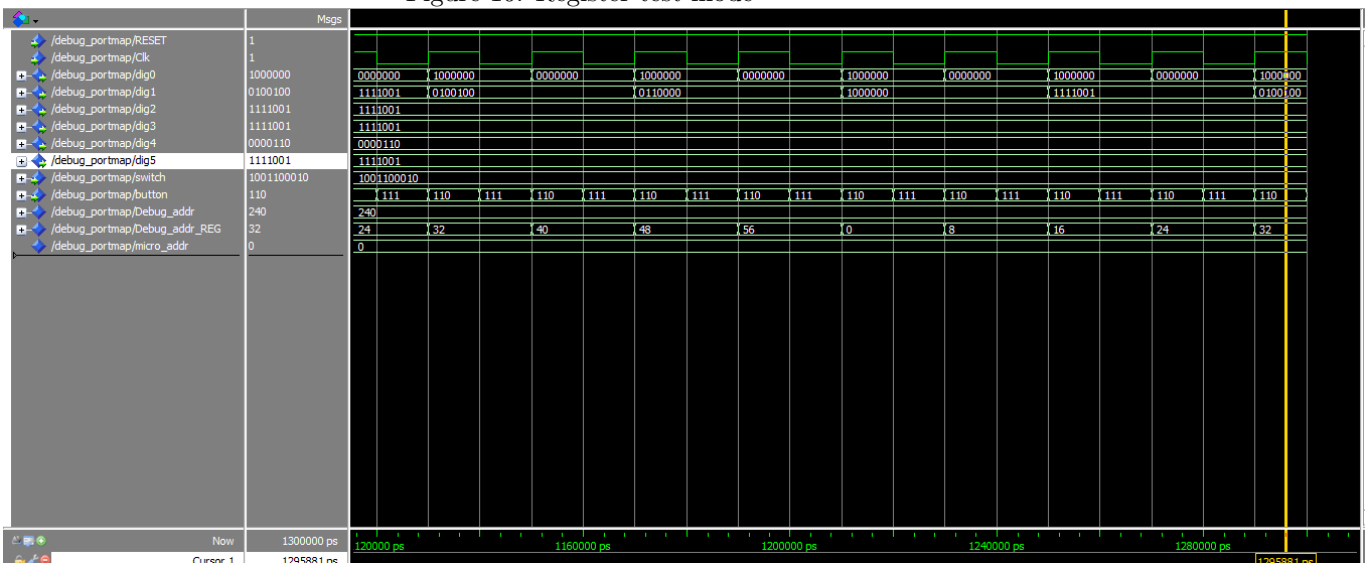


Figure 17: Final simulation