Module 5 - Project Report

Omar Madkour (2887320), Bogdan Dimitrie Pantiru (3016889), Sebastian Dorobantu, Andy Zhou (2978067), Youssef Ali (2872617), David Grimm(2919192)

9 November 2023

Contents

Contents

1 Introduction

The aim of this project is to design a 16 bit processor with a 16 bit word size. The design is inspired by the RISC processor, using a Harvard Architecture, in which the memory is divided into an instruction memory and a main (data) memory. The application consists of a calculator capable of realizing arithmetic (sum, subtract, shift left) and logical (and, or, nand, xor, xnor) operations on signed numbers (two's complement). Moreover, there are unique features that has been incorporated to the design, including a fully centralized asynchronous bus with its ability of performing transactions between two bus slaves rather than the ordinary master-slave transaction; and automatic overflow correction mechanisms in the ALU to enhance the precision of arithmetic and logic operations.

2 Global Schematic of the datapath

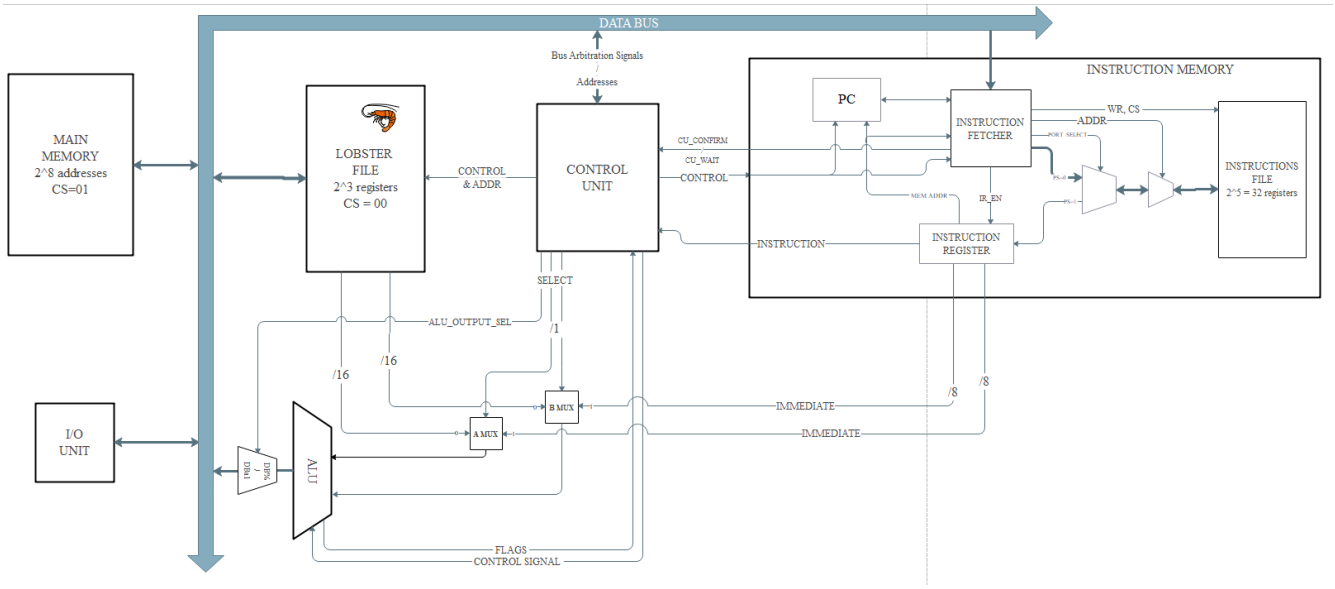

Figure 1: Datapath of the processor

2.1 Design choices

In the figure above the datapath of the processor that we aim to design is shown. In this paragraph, it shall be analyzed and the motivation for certain choices will be provided.

As most other processors of Harvard architecture, the processor is comprised of the main memory, register file, control unit, ALU and instruction memory. The register file is made out of 8 registers that are meant to temporarily store values of variables on which the ALu is performing a certain arithmetic or logical operation. Which operation is being done is controlled by the control unit which is also in charge of the bus if granted by the bus arbiter as well as takes instructions from the instruction memory by the use of synchronization and control bits. The instruction

memory has the instruction file which contains the set of instructions, a program counter which increments when an instruction is executed, the instruction fetcher which can also control the bus to fetch instructions from the main memory and the instruction register which contains the current instruction. The data bus is a fully centralized asynchronous data bus with the ability to make data transactions between two memory elements without them having any control of their own.

The design choices such as implementing a data bus were made to make the final design as close as possible to one of a commercial processors. However, because it was known from the beginning that the application was going to be a calculator, it was decided to implement a more complex ALU. The ALU will be able to handle negative numbers by the use of the two's complement and it will have automatic overflow handling. When an overflow occurs, the overflow bits can be stored in the ALU and outputted within the next instructions by branching. Even though the application was considered during the design process, the processor was not made only to fit a certain application but rather to be a stand-alone processor.

3 Instruction Set Architecture

	Instruction	OP-code	Operation	Assembly code
0	NOP	0000	Skips a clock cycle	NOP
1	Branch on 0	0001	Branch if equal to 0	be
2	Branch always	0001	PC $\leftarrow$ address	ba
3	Branch on Overflow	0010	PC $\leftarrow$ address $\Leftrightarrow$ V=1	bov
4	Shift left	0011	Rs1 $\leftarrow$ Rs1 << 1	shl
5	Shift right	0100	Rs1 $\leftarrow$ Rs1 >> 1	she
6	OR	0101	Rs1 = Rs1 | Rs2	orcc
7	ORN	0110	Rs1 = Rs1 | !Rs2	orncc
8	AND	0111	Rs1 $\leftarrow$ Rs1 & Rs2	andcc
9	ADD	1000	Rs1 $\leftarrow$ Rs1 + Rs2	addcc
10	SUB	1001	Rs1 $\leftarrow$ Rs1 $-$ Rs2	sub
11	Load	1010	Rd $\leftarrow$ Mem[Rs1]	ld
12	Store	1011	Mem[Rs1] $\leftarrow$ Rs2	st
13	Output Overflow	1100	Rs1 $\leftarrow$ Overflow of last operation	oov

Table 1: Instruction set

Next, the format for the 16-bit instructions needs to be set.

The following tables show the microword format of this processor. A table entry 'x' means that this field doesn't matter for this particular instruction, while a '-' denotes a space for one bit. N/U denotes an unused field for this specific instruction. The constants are 8 bits for arithmetic and logic instructions and 9 bits for the sethi instruction.

Special instruction INIT:

OP code	REG A	COND	Address
0000	0 0 0	0	0000 0000
1100	- - -	x	xxxx xxxx

Table 2: Special instructions

OP code	REG A	COND	Disp8
0001	x x x	0	− − − − − − − −−
0001	− − −	1	− − − − − − − −−

Table 3: Branch Instructions

OP code	REG A	COND	N/U	REG B
0010	− − −	0	xxxxx	− − −
0011	− − −	0	xxxxx	− − −

Table 4: Shift value in regA by 2^{REGB}

OP code	REG A	COND	Imm8
0010	− − −	1	− − − − − − − −−
0011	− − −	1	− − − − − − − −−

Table 5: Shift value in register by 2^{Imm8}

OP code	REG A	COND	N/U	REG B
0100	− − −	0	xxxxx	− − −
0101	− − −	0	xxxxx	− − −
0110	− − −	0	xxxxx	− − −
0111	− − −	0	xxxxx	− − −
1000	− − −	0	xxxxx	− − −

Table 6: Arithmetic/Logic instructions with two registers

OP code	REG A	COND	Constant
0100	− − −	1	− − − − − − − −−
0101	− − −	1	− − − − − − − −−
0110	− − −	1	− − − − − − − −−
0111	− − −	1	− − − − − − − −−
1000	− − −	1	− − − − − − − −−

Table 7: Arithmetic/Logic instructions with a register and constant

OP code	REG A	COND	N/U	REG B
1001	− − −	0	xxxxx	− − −

Table 8: Load value from Memory regB into regA

OP code	REG A	COND	Address
1001	− − −	1	− − − − − − − −−

Table 9: Load value from main memory address into regA

OP code	REG A	COND	N/U	REG B
1010	− − −	0	xxxxx	− − −

Table 10: Store value from regA into main memory address in regB

OP code	REG A	COND	Address
1010	− − −	1	− − − − − − − −−

Table 11: Store value from regA into main memory address

?? outlines the arithmetic and logic functions performed by the calculator. Users input function codes, which prompt the calculator to send a sequence of instructions to the main memory. To illustrate, when the NAND function is chosen, the calculator initially sends an AND operation instruction, followed by an ORN instruction to the main memory. In the case of multiplication and division functions, it employs shift left (equivalent to multiplying by two) and shift right (equivalent to dividing by two) instructions, respectively. The calculator can perform multiplication with numbers other than two. It's worth noting that the original intention was to support decimals and negative numbers, but these features were ultimately omitted in favor of a more straightforward approach.

	Instruction	Function-code	Operation
1	Add	0000	Out = Num1 + Num2
2	Subtract	0001	Out = Num1 − Num2
3	Multiply by power of 2	0010	Out = Num1 $*$ 2^n
4	OR	0101	Out = Num1 | Num2
5	AND	0110	Out = Num1 & Num2
6	ORN	0111	Out = Num1 | !Num2
7	NAND	1000	Out = !(Num1 & Num2)
8	XOR	1001	Out = Num1 $\oplus$ Num2
9	XNOR	1010	Out = !(Num1 $\oplus$ Num2)
10	OOV	1011	Out = overflow of last op

Table 12: Calculator Functions

4 Register file

The register file or Local Output Binary Storage Temporary Execution Register reads or writes memory addresses from main memory. The selected architecture consists of a 8x16 register file, 8 registers of 16 bits.

It is made up of 5 inputs, 3 inputs provided by the bus and 2 inputs provided by the control unit. The 3 inputs provided by the bus are: the 16 bit data (*data*); the address (*address*), from which the 3 right most bits select the register that the data will be written or read to/from; and a write enable (*wr_enable*), which whenever is active, it writes into the register, otherwise it reads. Furthermore, the 2 inputs provided by the control unit are: regA and regB. These are used to select the data read from the register file that will be the inputs to the ALU.

There are 3 outputs from the register file. Two of them (outA and outB) will be the inputs to the ALU and the last one (outC) goes to the main bus.

Additionally, there is a final output (outD) which indicates if the outputs are done. It operates as a flag to inform the control unit that the outC is on the main bus as well as knowing when the inputs to the ALU are ready.

Figure 2 shows the time diagram of the register file simulated in ModelSim.

There are different addresses stored in registers 2, 4, 5 and 7. In this case, the registers which are accessed are 2, 4 and 7 (inputs from regA, regB and address respectively). These addresses are shown in the outputs (outA, outB and outC). Out D, is set to 1, indicating that the output is available.

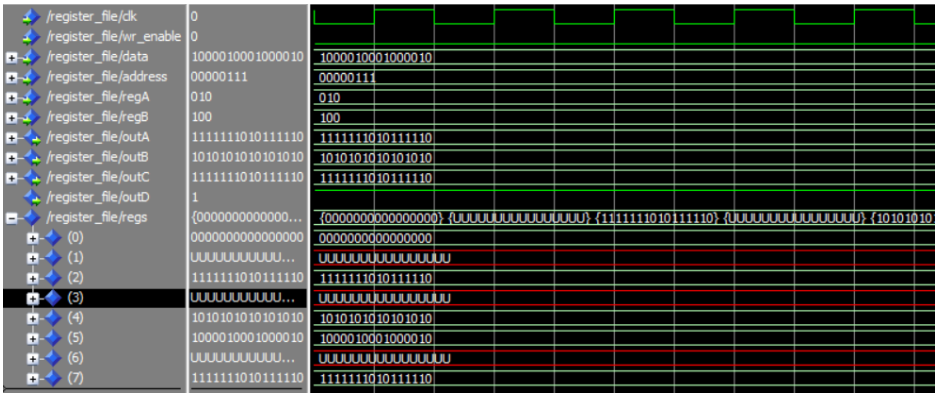

Figure 2: Register File time diagram simulation in ModelSim

5 Memory

The purpose of the memory is to temporarily hold data that will be used by the processor. The inputs of the desired application will be stored in memory before being used in logical operations.

The structure of the memory is 256 addresses of 16 bits. Data can be written just when the write bit (left most bit of the address input) is active along with the correct chip select bits (bit 9 and 8 are expected to be 10) and the index which specifies the actual address in the memory where data will be stored . The output is dependent on the write bit. If the write bit is 0, output data will be read from the address.The *Bus_sync* outputs flag which one of the 2 Adress inputs from the bus have been used. Below is provided the Dataflow of the memory 3.

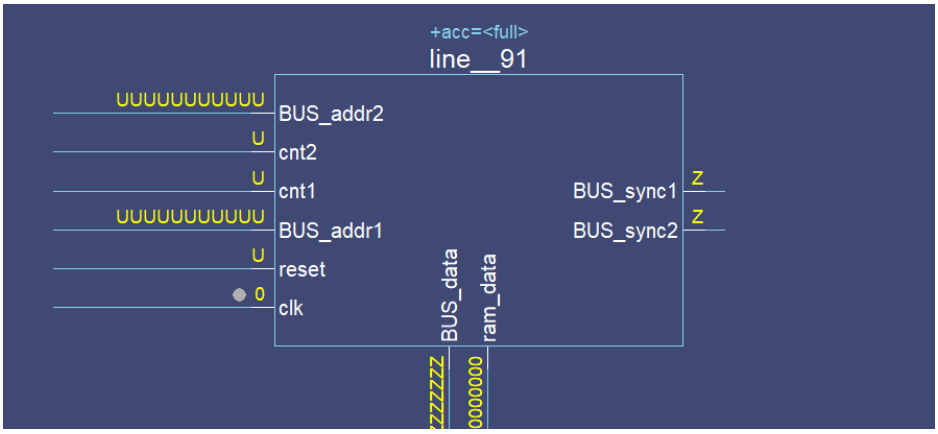

Figure 3: Memory writing mode

The test of the memory can be conducted by forcing a 16-bit array in BUS_data. By forcing the address array in the format "11000000001" write is enabled, correct chip is selected and address with index 1 is selected in the memory, in which the data is stored. If a different address in the memory is chosen and cnt1 is incremented to 1, the first byte from the memory takes the first byte from the data. If index 1 is selected again and write is disabled, by incrementing cnt2 to 1 and decrementing cnt1 back to 0, the last byte of memory address 1 becomes the output. Thus, the full functionality of the memory is proven, being byte addressable for both reading and writing.(4).

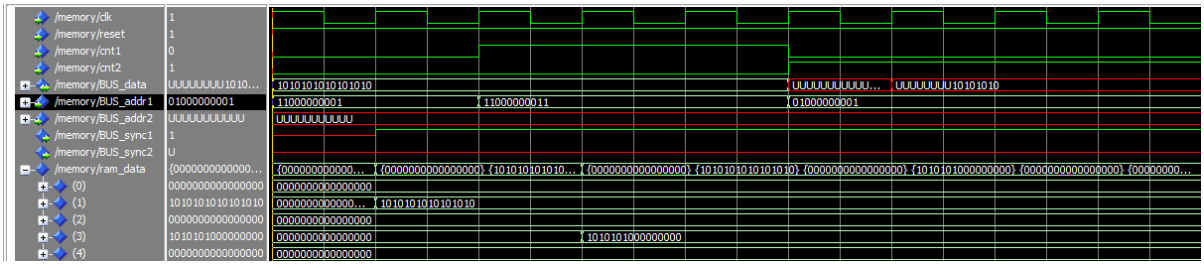

Figure 4: Ram Simulation

6 Bus

For this processor, a simple, fully centralized, asynchronous bus is implemented with its only special feature being its ability to have transactions between two bus slaves rather than just master-slave transactions. This is achieved by adding a second address bus and by using a part of the address bits as a chip select for slaves. Three devices need controlled access to the bus, the instruction fetcher, the IO unit and the control unit. Therefore, an arbiter with 3 priority levels is implemented. The highest priority goes to the control unit at level 0. Priority level 1 goes to the instruction fetcher, followed by the interface, which has the lowest priority.

The bus consists of a 16-bit wide data bus, two 11-bit address busses, request lines, grant lines and synchronization and busy lines.

The arbiter behaves in the following manner: when a request is sent, if the busy signal is active no grant is given. If the busy signal is 0, based on the priority level a grant is given, and the state is changed to granted. When the bus starts being used for data transfer the arbiter moves from state granted to state In_Prog. After the process ends and the busy signal is 0 again, the arbiter

returns to state idle and is ready for the next request. It is possible as well if multiple requests are waiting for the arbiter to go directly to the next granted state and skip the idle state.

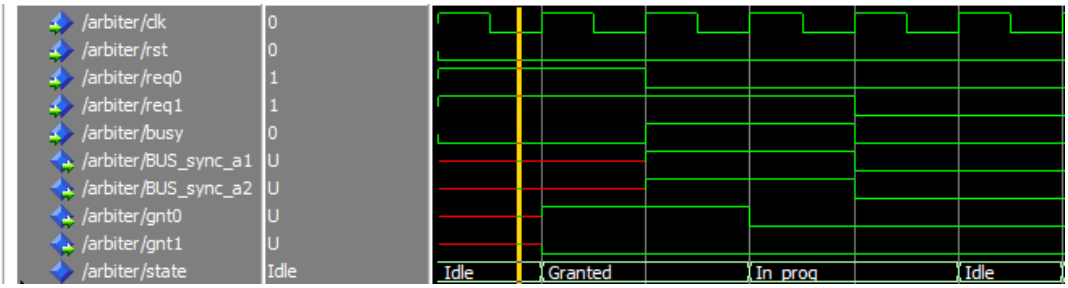

Figure 5: Arbiter Simulation

In the provided simulation (5) it can be observed that the grant behaves by the priority levels assigned. Taking into account the busy signal and reset, it is clear that all states behave as described.

7 ALU

The Arithmetic and Logic unit performs the functions described in calculator function table **??**, once a valid function code, as well as two valid 16-bit operators, are received by the ALU. For all operations but shift left (Multiply by 2^n), operators A and B can be negative or positive. If a number A is to be multiplied by 2^B, B can range between integer values 1 and 16. If any arithmetic operation results in an overflow, the overflow is stored in a 16-bit register. If requested with function code "1011", the ALU output is overwritten with the 16-bit overflow register.
If the first bit of the 3-bit output $flag_{vector}$ is 1, the result is zero. If the second bit is 1, the result is negative and if the third bit is 1, overflow is detected.
In figure 6, two numbers ($ALU_A = -32$ and $ALU_B = 9$) are added together, giving result $ALU_{out} = -23$. The function code is changed to subtraction and after that, to left shift. For all operations, the result and the flag are correctly output.
Once ALU_B is increased to 12, the output is zero, overflow is detected and stored in a register ($overflow_{outbuf}$). This register is overwritten, once a new operation is performed. After the overflow is requested as output, the 16 LSBs (the normal output) are overwritten by the 16 MSBs (the contents of the overflow register). After the overflow is read out, ALU_A and ALU_B are compared with the "XNOR" function and the output is as expected.

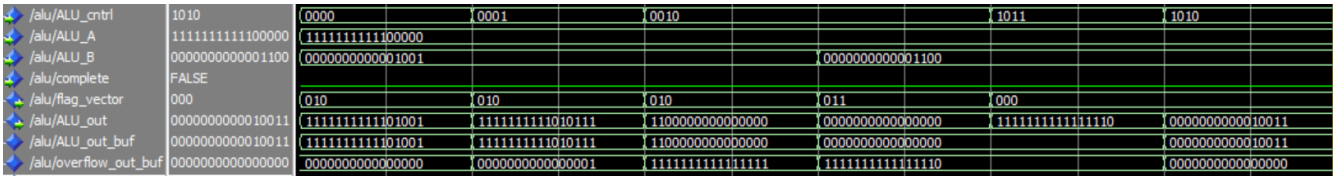

Figure 6: ALU simulation

Due to the ALU being completely programmed with combinational logic, there is no clock signal. This allows the ALU to, at least theoretically, perform any operation within a few delta delays and significantly increase efficiency.

8 Control Unit

The control unit stands as a pivotal component within the processor, orchestrating the seamless collaboration among its various parts. Its functionality is characterized by its inputs and outputs, with the majority of the time invested into making it concentrated on the correlations between them.

In terms of its interaction with instruction memory, the control unit establishes connections that facilitate the exchange of status signals. These signals play a role in allowing the instruction memory to determine the next instruction, as well as signals serving as input to the CU for the instruction requested.

Furthermore, the control unit interfaces with the Arithmetic Logic Unit (ALU) through a more complex network of signals. These signals enable the control unit to selectively determine the ALU inputs, choosing between registers and immediate values. Additionally, a signal set is dedicated to carrying the transaction flags to the CU, as elaborated in the ALU section (see Section 7), along with a 4-bit bus that dictates the ALU's designated operation.

Notably, the control unit possesses specialized connections to the Register File, enabling it to directly designate registers for output to the ALU, without going through the bus.

The orchestration of these inputs and outputs is decided by a Finite State Machine(FSM), handling the fetch-execute cycle. This cycle consists of two primary states within the FSM: the fetch state and the execute state, each further decomposed into four distinct sub-states.

The fetch state encompasses both a send and receive sub-state. In the send state, the control unit issues a request for a new instruction, while the receive state awaits the availability of the instruction before transitioning back to the overarching execute state.

Within the execute state, two sub-states are distinguished: the exec state, which configures the required signals to initiate the transaction dictated by the opcode, and the fin state, which awaits confirmation of the transaction's successful completion.

For a visual representation, refer to Figure 7, showcasing a simulated timing diagram in ModelSim. The exemplified timing diagram involves the execution of a branch instruction followed by an arithmetic instruction.

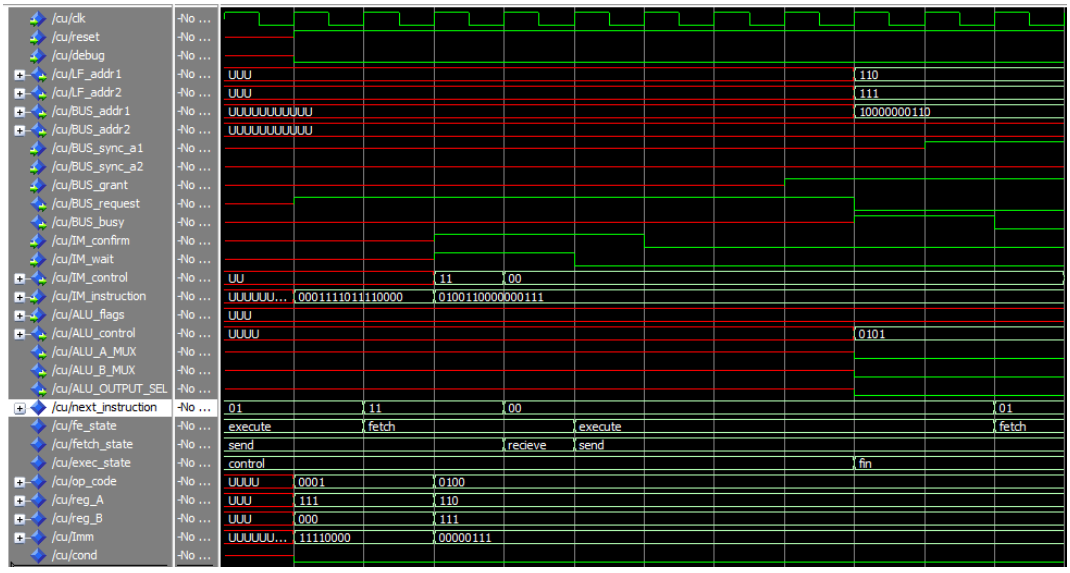

Figure 7: CU time analysis in ModelSim

9 Application

The chosen application for the processor is a calculator. It must be able to realize basic operations such as addition, subtraction, multiplication, and division. These operations will be carried out using the arithmetic (add, sub), logical (and, or) as well as the shift left/right instructions. Due to the chosen 16-bit processor architecture, the operations must be carried out using words (16 bits).

The FPGA has a limited amount of I/O, consisting of ten switches, four buttons, and six 7-segment displays. For the four buttons, each button is assigned a function such that there will be an increment, change display, enter, and reset button. To start the calculation process all ten switches must be set to zero (switched off).

After putting in the number and the enter button is pressed to confirm it, four of the ten switches will be used to indicate the function code and the enter button will be pressed again to confirm the operation. Finally, the same procedure can be applied to the next number that the operation is going to be applied to. Figure 9 shows how the FPGA will be used for the calculator application.

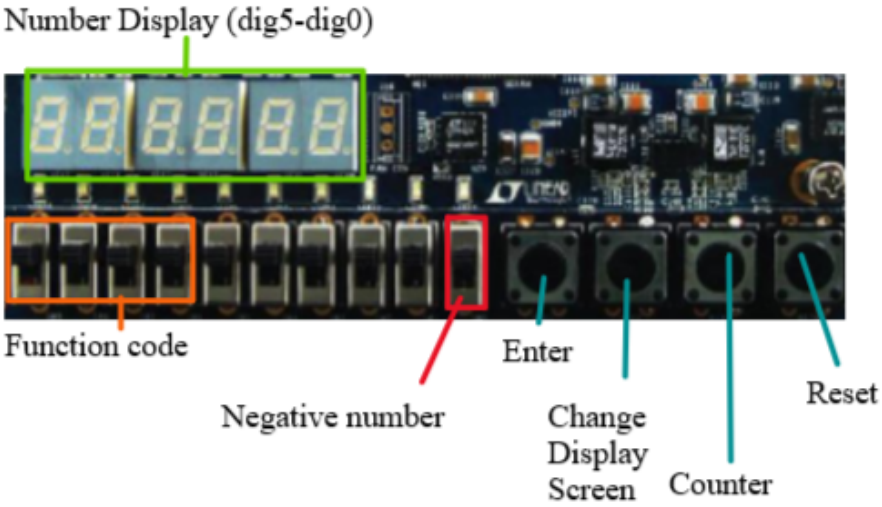

Figure 8: Calculator operation in FPGA

For debugging, nine switches will be used to enter the address that is to be read. The displays will show the address and contents of this address in HEX.

10 Interface

The interface for the calculator allows the user to use the buttons (button0, button1, button2, and button3) as well as the last four switches to operate the calculator as illustrated in figure 9. For example, to input a number the user can change the displays by pressing button2 and then increment the value of the display or choose a number from 0 to 9 by pressing the counter button or button1. Then after the user inputs the final number a decision can be made whether to make the number signed (negative) by switching switch0 on or leave it off to make the number positive. Finally, the user can press button3 to change the number displayed on the hexdisplays

to its binary value in 16 bits and enter the number. The same procedure is done for the next number however the user can now operate the last four switches to input a function code for the calculator to carry out. Additionally, the corresponding LEDs for each of the four switches turn on or off depending on the state of the switch so the user can know which binary value of the function code is represented. Finally, button3 is pressed again to enter the function code and the 16-bit binary value for the next number. The figures below give a visual explanation of this process with simulations to show the binary values sent when button3 is pressed.

Figure 9: Interface operation

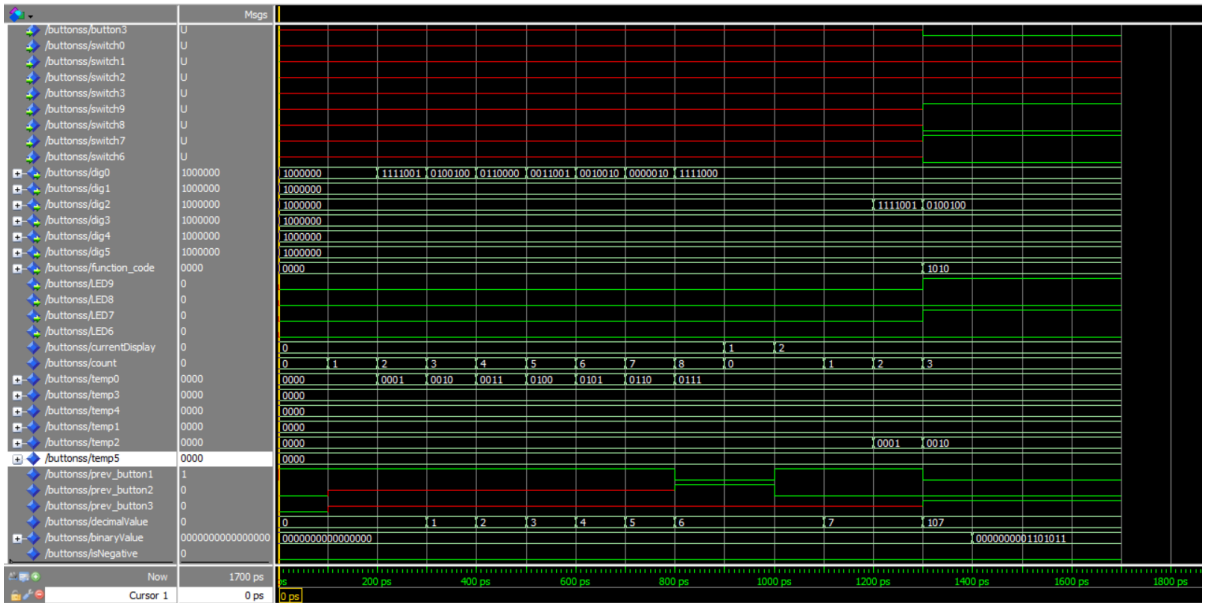

Figure 10: Interface simulation

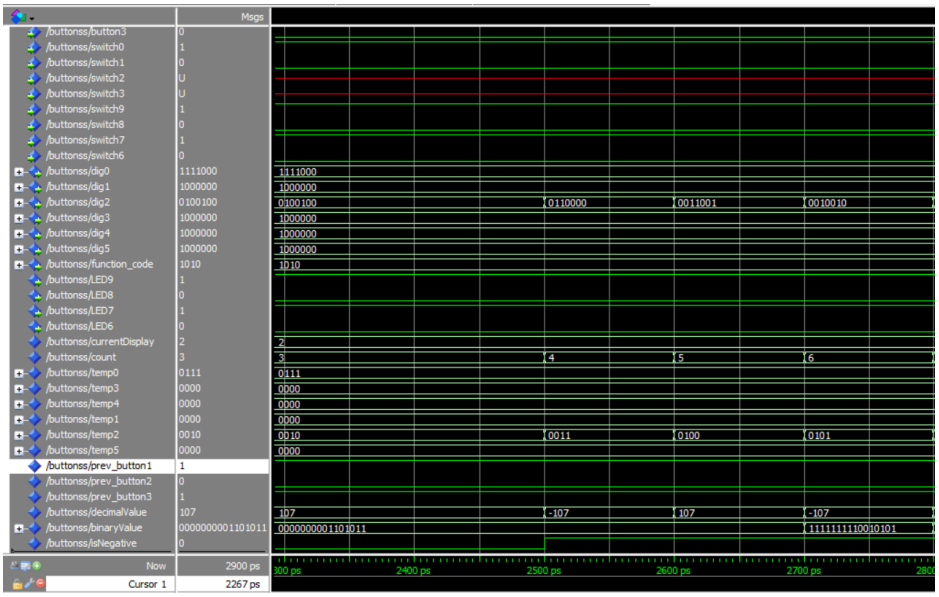

Figure 11: Signed number simulation (switch0 on)

As can be seen from the figures above, when buttons 1 and 2 are pressed to show the decimal value "107" on the displays and the switches 9 to 6 are used to input a function code (1010). It is expected that when button3 is pressed this decimal value is switched to its binary value and stored. Moreover, this binary value is expected to be "0000000001101011" and its signed value for "-107" should be "1111111110010101". This operation can be seen in figures 10 and 11. Nevertheless, the following figure 12 shows the same code but modified to send the number entered to be stored in a specific memory address as well as making a bus request to be able to send the following data and memory address on the data bus and memory bus.

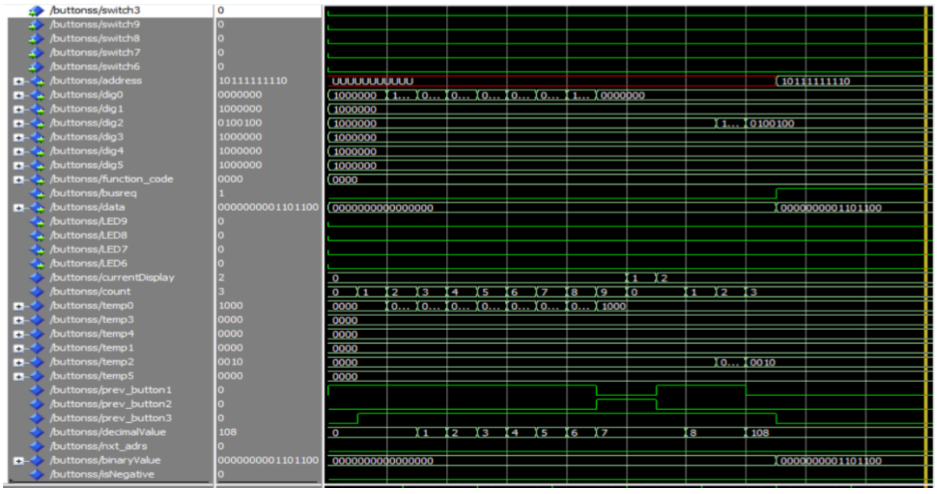

Figure 12: Memory address and bus request.

11 Test approach

For the debugging process all the input parameters on the FPGA board will be utilized. To start debugging, switch 1 (S_1) must be switched on. Once it is on, the user can enter the address in

binary by turning the switches 2 to 9 ($S_2 - S_9$) on/off corresponding to the indexed address. Switch 2 (S_2) represents 2^0 while the left most switch represents 2^8. Each switch and its corresponding index is shown in Table 13.

Switch	Binary Index
S_1	enter debug
S_2	2^0
S_3	2^1
S_4	2^2
S_5	2^3
S_6	2^4
S_7	2^5
S_8	2^6
S_9	2^7

Table 13: Binary index for each switch in the FPGA

To illustrate the test approach, an example will be given. To read the memory address 57 (111001 in binary) the user has to turn on the following switches: (S_1, S_2, S_5, S_6, S_7), Switch 1 to enter debug mode and the rest of the switches to enter the address. Moreover, the user can also move through consecutive addresses through the assigned increment and decrement buttons on the FPGA board. This provides easy access for debugging and viewing memory content. A visualization of the debugging and test parameters can be seen in Figure 13.

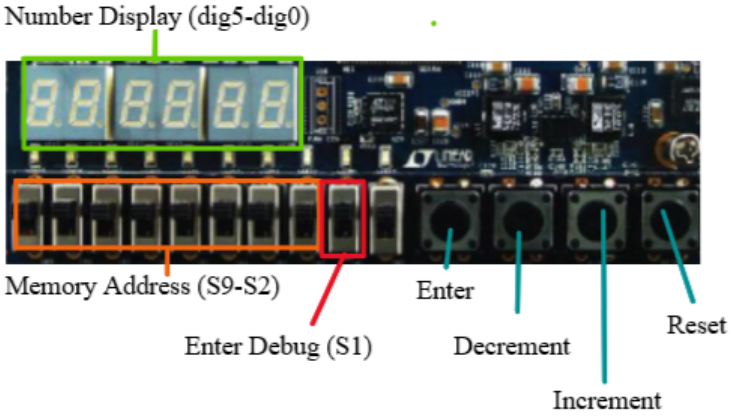

Figure 13: Debugging parameters

The time diagram simulation in ModelSim of the debugging mode is shown in Figure 14. From the I/O unit, the address that wants to be read is '00000010', this address is an input to the address bus (Bus_addr2). Moreover, the increment button is pressed (button 2), hence the new address is '00000011'. The address bus sends the input address to the main memory, fetches the corresponding data (1111111111000100 in binary) and it is an input to the data bus (Bus_data). The data bus sends this data back to the I/O unit, it is then converted into an integer (-60). This case corresponds to a signed number, thus to indicate that the number is negative, the right most LED in the FPGA is turned on (LED0='1'). The number is then converted into hexadecimal (3C) and displayed in the six 7-segment displays ($dig_0, dig_1, dig_2, dig_3, dig_4, dig_5$), dig_0 being the right

most display and dig_5 the left most display. dig_0 and dig_1 show the expected result while the rest of the outputs to the display have a value of '1000000' which indicate that the display is off.

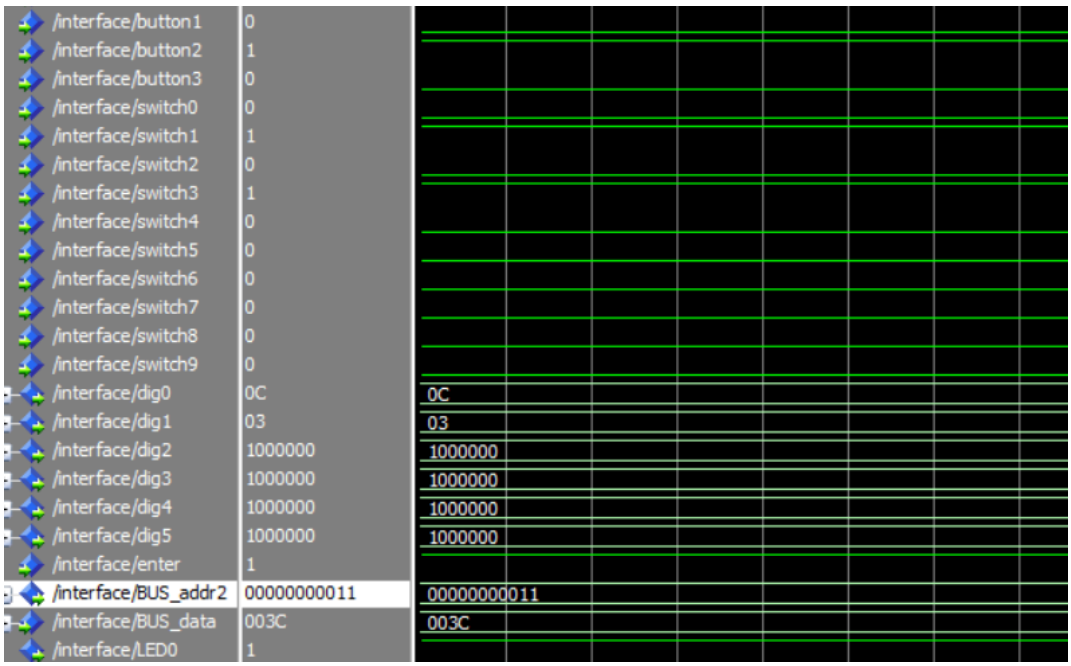

Figure 14: Debugging Simulation

12 Results and Conclusion

While it was not possible to simulate the complete processor due to an overcomplicated bus design and time constraints, each component was successfully simulated on its own. Synthesizing the whole processor was successful, even though the I/O unit was not successfully implemented as a whole. The buttons and switches of the I/O unit kept creating errors that couldn't be resolved in time. The calculator application, written for the processor could therefore not be successfully tested as it relies heavily on the input and output capabilities of the FPGA. When the Interface component was removed, however, it was possible to synthesize the processor and the resulting register transfer level (RTL) graphical representation can be seen below in figure 15.

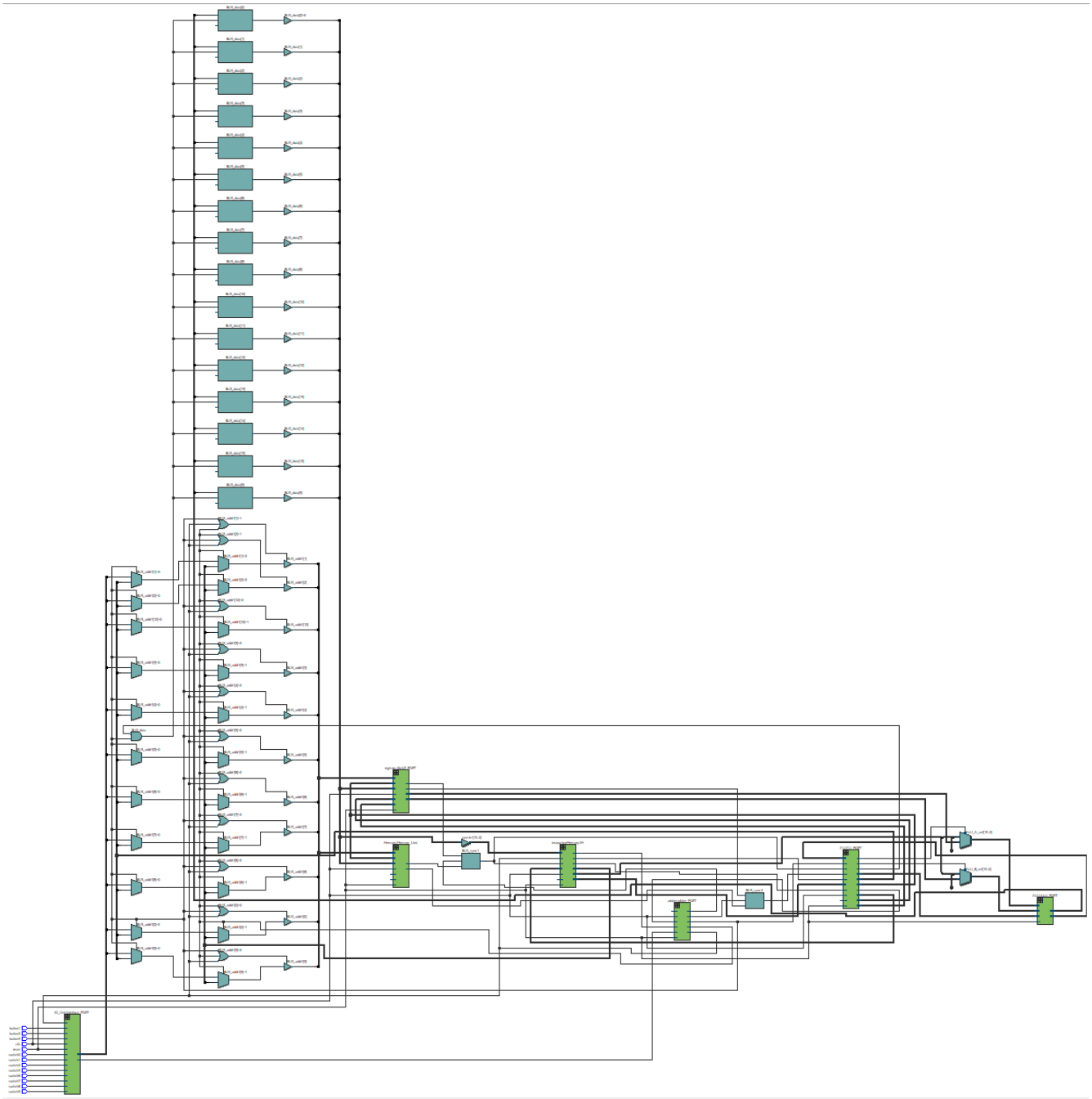

Figure 15: RTL view without I/O unit

It is clear, when looking at the RTL view of the CPU that the BUS is too convoluted and that its implementation is presumably not fully correct. In conclusion, simulating each component individually was successful and gave the expected results. All components were successfully synthesized individually, and synthesizing the full processor was possible. On the other hand, when simulating the processor, it was insufficient due to issues with the BUS and other unexpected complications. Moreover, these complications consumed a lot of time to fix which caused a major schedule delay. Since it was not possible to simulate the complete processor successfully, a test environment was not devised. The timeline did not favour completing the processor simulations and uploading them on the FPGA. Therefore, unfortunately, due to time constraints and integration

problems, as well as an overly ambitious design, the processor was not successfully simulated.

13 Appendix

Sebastian Dorobantu	Control, Datapath
Omar Madkour	Interface and application, BUS
Andy Zhou	Debugging
Youssef Ali	Interface and application
David Grimm	ALU, BUS
Bogdan Pantiru	Memory, Registers

Table 14: Task divison