Simple Multi-Cycle MIPS Processor

Erlend Hestnes, MSc. EE student NTNU Victor Iversen, MSc. EE student NTNU
and Christoffer Ramstad-Evensen, MSc. EE student NTNU

***Abstract*—This report presents a functional implementation of a reduced instruction set MIPS processor, capable of doing R-type, J-type, LUI, Beq, Load and Store operations. The report elaborates on the different sub modules needed to achieve a simple, yet functional design with these operations. The presented MIPS processor is synthesizable and has been thoroughly tested on a Xilinx Spartan-6 FPGA.**

I. Introduction

MICROPROCESSORS are found almost everywhere in today's modern society, where applications range from basic home appliances to more advanced smartwatches. The MIPS processor was first introduced in 1981, and is considered to be one of many early RISC architectures that sparked the digital revolution. Even today, one can still find embedded systems that feature different iterations of the MIPS processor core. The MIPS architecture[1] has also influenced many of today's modern microprocessors.

The design goal for this project is to create a functional reduced instruction set MIPS processor. This is realized by using a bottom-up approach. Starting with the construction of each individual module by itself, then ensuring proper functionality by creating a testbench for the module. The modules are then connected together in the top level design. The entire design is then tested, instruction by instruction, until proper functionality is achieved.

II. Solution

THE top-level architecture of a MIPS processor is described in the Figure 1. The main components of the architecture are the Program Counter (PC), Register File, Arithmetic Logic Unit (ALU), Data Memory (DM), Instruction Memory (IM), Control Unit and the ALU Control Unit. This section will describe the design of all the components included in the design with the goal of achieving a functional MIPS processor.

A software utility called Hostcomm was used for testing on the FPGA. It uses serial communication to write instructions and data to the IM and DM. It runs the processor and lets the user read from the IM and DM.

A. MIPS Instructions Set Architecture

The MIPS processor uses the MIPS Instruction Set Architecture (ISA). The ISA defines three different types of instructions depending on what should be performed by the processor. Table I describes the three instruction types and what their bits represents. The *opcode* bits determines the instruction type and r_s, r_t and r_d are register addresses for source one, source two and destination, respectively. The *funct* bits are used to determine the ALU operation. The *shamt* bits will be ignored as they are not used for the design. For further reading on the MIPS ISA read the *MIPS Instruction Set*[1].

TABLE I
Instruction formats.

	Instruction bits					
R	opcode(6)	r_s(5)	r_t(5)	r_d(5)	shamt(5)	funct(6)
I	opcode(6)	r_s(5)	r_t(5)	constant or address(16)		
J	opcode(6)	address(26)				

Table II gives the integer value of the *opcode* and the processor operations given the instruction type for the design. The AND, OR, add and sub operations refers to the AND, OR, addition and the subtraction operations, respectively. Set-on-less-than, branch-equal, load-upper-immediate, load, store and jump operations are denoted slt, beq, lui, lw, sw and j, respectively.

TABLE II
Decimal value of Opcode and MIPS operations.

Type	Opcode	Operations
R	0	add, sub, AND, OR, slt
I	4, 15, 35 or 43	beq, lui, lw, sw
J	2	j

B. Control Unit

The Control Unit serves as the supervisor of the processor. It is responsible for setting the control signals based on the current instruction so that the processor performs the correct operations. To do so, the Control Unit needs the *opcode* bits from the instruction and a processor enable signal as input. Based on this input the Control Unit is able to set the output depicted in Table III.

TABLE III
Description of control signals from the Control Unit.

Control Signal	Description
reg_dest	Select the r_d or r_t address as write register.
reg_write	Enable register write.
branch	Enable branch.
jump	Enable jump.
mem_write	Enable write to data memory.
mem_to_reg	Select data from memory or the register to write to register.
alu_op	Signal to the ALU Control Unit which instruction type is present.
alu_src	Select ALU input source.
pc_mux	Enable write to PC
shift	Enable shift when performing a lui operation.

Three main stages for the processor operation where chosen; fetch, execute and stall. This was possible because of the simplicity of the MIPS processor. From a hardware perspective these stages fits in a Finite State Machine (FSM), which

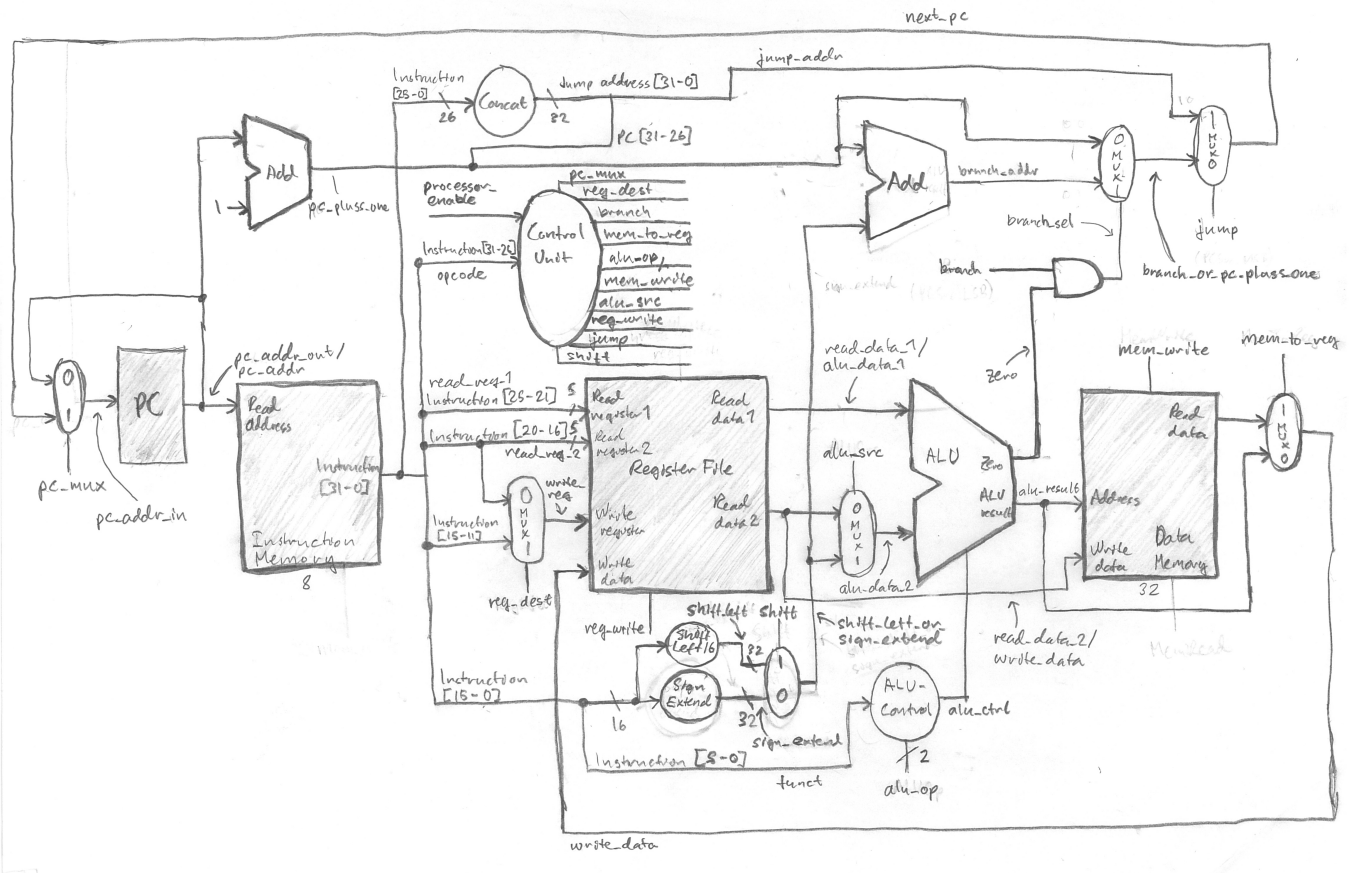

Fig. 1. Top level architecture of a MIPS processor

is implemented in this design. An FSM provides simplicity in keeping signals synchronized and provides an abstraction which is understandable. The complete FSM for this MIPS processor is illustrated in Figure 2. Notice that the FSM in

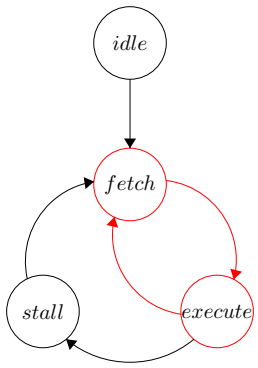

Fig. 2. State Machine of CPU Control Unit. Common Case marked in red

Figure 2 includes the state Idle. The processor only enters this state when it is disabled. As soon as the processor is enabled it proceeds to the Fetch state. The FSM is designed as a Mealy-Machine since it is dependent on both the current state and the inputs.

When the processor is enabled it immediately enters the Fetch state and initiates a fetch for the first instruction on the IM output. In the next cycle the processor enters the Execute state and will determine which control signals to set based on the current instruction type. The processor does this by checking the integer value of the *opcode* as listed in Table II. The appropriate control signals and state are set based on the instruction type. If the instruction type is an R-Type, the Control Unit must make sure that the processor reads the data at registers r_s and r_t and that they are loaded properly into the ALU. Appropriate control signals to the ALU Control Unit must be set and the Register File must be given the correct destination address r_d. Signals for enabling a register write is also needed. The R-Type instructions always have the same data path, but for the I-Type instruction the data paths differs based on the operation. For the lw operation the data at the r_s register must be read and the next input of the ALU must be the sign extended result of the address bits of the instruction. The correct destination, r_t, must be given to the Register File and because the DM needs one clock cycle to read or write to memory, the Control Unit must make sure that the processor stalls for on clock cycle. The sw and lui operation is simliar to the lw operation except that for the sw operation a memory write is needed as to a register write for the lw operation and for the lui operation the address bits is a constant that is shifted 16 bits left. For the beq operation the ALU must be loaded with data from the r_s and r_t registers to branch if equal. The J-Type instruction only needs to forward the jump address to the PC by enabling the *jump* signal. It is common for all data paths to enable a new input on the PC in the last operation

cycle.

A detailed description of the control signal set by the Control Unit is depicted in Table IV. Notice that signals which

TABLE IV
CONTROL SIGNALS OF THE CONTROL UNIT.

State	Instruction Type	Signals	Next State
Fetch	All	-	Execute
Execute	R-Type	reg_dest reg_write pc_mux alu_op = ”10”	Fetch
	J-Type	jump pc_mux	
	LUI	alu_src reg_write shift pc_mux alu_op = ”00”	
	Beq	branch pc_mux alu_op = ”01”	
	Load	alu_src mem_to_reg reg_write alu_op = ”00”	Stall
	Store	alu_src mem_write alu_op = ”00”	
Stall	Load Store	pc_mux	Fetch

are not set in Table IV have to be set to logical zero.

Gray encoding is used as an optimization for speed in the FSM. The common case is the R-Type instructions which only visits the fetch and the execute state. The optimized encoding for the design is described in Table V.

TABLE V
OPTIMIZED ENCODING FOR THE FSM IN FIGURE 2.

State	Idle	Fetch	Execute	Stall
Encoding	10	00	01	11

To make the RTL of the Control Unit an expression of the next state is made. This can be achieved by making a table as the one depicted in Table VI, illustrating the state transitions depending on the *opcode*, the processor enable bit and the current state. Looking closely at Table VI reveals bits that can

TABLE VI
NEXT STATE TABLE.

State	S_1S_0	$O_5O_4O_3O_2O_1O_0$	P_0	Next State	$S_1'S_0'$
Idle	10	xxxxxx	0	Idle	10
Idle	10	xxxxxx	1	Fetch	00
Fetch	00	xxxxxx	0	Idle	10
Fetch	00	xxxxxx	1	Execute	01
Execute	01	xxxxxx	0	Idle	10
Execute	01	000000	1	Fetch	00
Execute	01	001111	1	Fetch	00
Execute	01	000100	1	Fetch	00
Execute	01	000010	1	Fetch	00
Execute	01	100011	1	Stall	11
Execute	01	101011	1	Stall	11
Stall	11	xxxxxx	0	Idle	10
Stall	11	xxxxxx	0	Fetch	00

be regarded as don’t care. Table VII includes the minimum

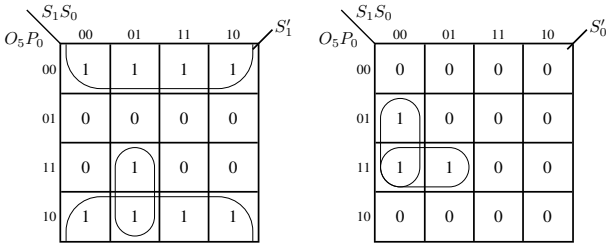

Fig. 3. Karnaugh diagram of the next state.

information necessary to make a boolean expression for the FSM. Setting the information in Table VII in two Karnaugh di-

TABLE VII
NEXT STATE TABLE WITH MINIMUM INFORMATION.

S_1S_0	O_5P_0	$S_1'S_0'$
00	x0	10
00	x1	01
01	x0	10
01	01	00
01	11	11
10	x0	10
10	x1	00
11	x0	10
11	x1	00

agrams will optimize this further. From the Karnaugh diagrams in Figure 3 the next state expression is given by Equation 1 and 2.

$$S_1' = \overline{P_0} + \overline{S_1}S_0O_5 \quad (1)$$

$$S_0' = \overline{S_1}O_5P_0 + \overline{S_1S_0}P_0 \quad (2)$$

Finally the RTL of the Control Unit is generated from the next state expression as depicted in Figure 4.

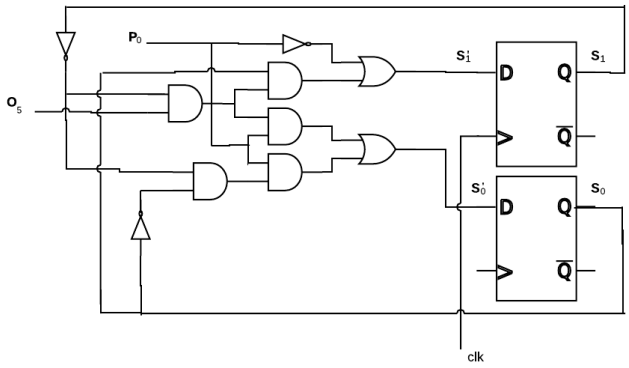

Fig. 4. RTL of the next state generation of the Control Unit.

C. ALU Control

The ALU Control Unit is responsible for the functionality of the ALU. It determines the ALU function based on the *alu_op* signals from the Control Unit and the *funct* bits from the instructions in Table I. Table VIII gives the desired output of the ALU Control for the different operations. With the

TABLE VIII
ALU OUTPUT FOR DIFFERENT OPERATIONS.

Opcode	ALU Op	Operation	Funct	ALU Function	ALU Control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	sub	0110
R-Type	10	add	100000	add	0010
		subtract	100010	sub	0110
		AND	100100	and	0000
		OR	100101	or	0001
		slt	101010	slt	0111

information in Table VIII the combinatorial circuit of the ALU Control Unit can be generated. The input parameters for the circuit will be the *alu_op* and the *funct* bits, but only the four LSB's of the *funct* bits as the two MSB's are common for each instruction or don't care.

Equation 3, 4 and 5 names the respective bits in the different signals and will be used through the generation of the ALU Control Unit circuit.

$$alu\_op = a_1 a_0 \tag{3}$$
$$funct = q_3 q_2 q_1 q_0 \tag{4}$$
$$alu\_ctrl = o_3 o_2 o_1 o_0 \tag{5}$$

By using Karnaugh diagram a minimum circuit is achieved. Figure 5 illustrates the Karnaugh diagram of the R-Type instruction logic. From the Karnaugh diagram in Figure 5

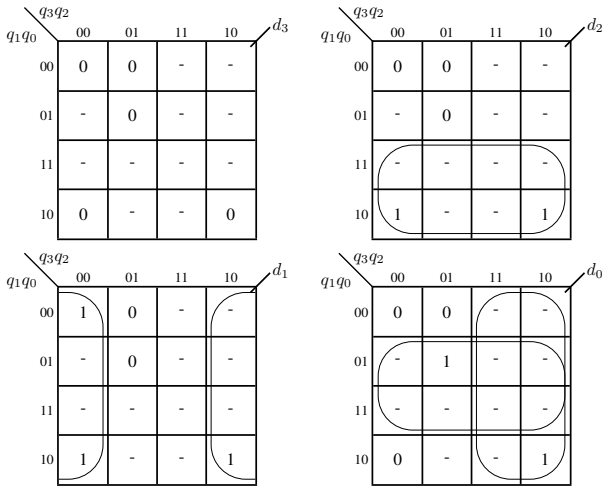

Fig. 5. Karnaugh diagram of the funct input.

Equation 6, 7, 8 and 9 are derived.

$$d_3 = a_0\overline{a_0} \tag{6}$$
$$d_2 = f_1 \tag{7}$$
$$d_1 = \overline{f_2} \tag{8}$$
$$d_0 = f_3 + f_0 \tag{9}$$

To cover the other instructions the *alu_op* signals have to be taken to account. Putting the $alu\_ctrl = d_3 d_2 d_1 d_0$ signals in a separate Karnaugh diagram with the *alu_op* signals the final *alu_ctrl* signals can be generated. The result is illustrated in Figure 6.

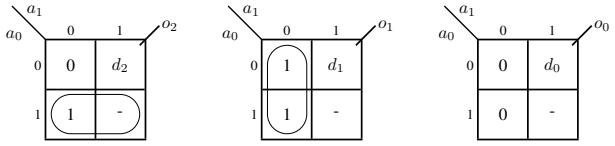

Fig. 6. Karnaugh diagram of the combined alu_op and funct bits.

Derived from Figure 6 Equation 10, 11, 12 and 13 describes the final boolean expressions for the ALU Control Unit.

$$o_3 = a_0\overline{a_0} \tag{10}$$
$$o_2 = a_0 + a_1 f_1 \tag{11}$$
$$o_1 = \overline{a_1} + \overline{f_2} \tag{12}$$
$$o_0 = a_1(f_3 + f_0) \tag{13}$$

The final circuit is given in Figure 7

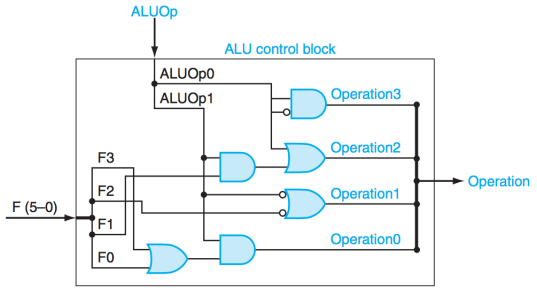

Fig. 7. ALU Control Unit RTL [4].

D. Program Counter

The PC is a register that stores the address of the next instruction to be fetched from the IM. Before fetching the next instruction to be executed, the PC is incremented by 1 to apply the address of the next instruction to the address input of the IM. The control unit decides when the PC should be incremented, based on the current state of the FSM within the Control Unit shown in Figure 2. The input line of the PC is fed from a multiplexer that is controlled by the Control Unit, which controls if the PC is to be updated with a new value or if it should hold on to its current value. This multiplexer is shown in Figure 1.

In case of performing a branch as a result of a branch instruction (beq), PC is updated with the branch address, and in case of a jump instruction, the PC is updated with the jump address. The Control Unit sets the control signals that enables updating the PC with the branch address or the jump address dependent on the instruction being executed. The control flow in case of a branch or jump is further explained in Section II-B.

The PC was designed as a single 32-bit register enabled by a clock-input. The PC would be updated on each rising edge of the clock, where the multiplexer controlled by the Control Unit, decided if the PC would be updated with its previous value, or obtain a new value that would result in a new instruction being fetched from the IM. The output of the PC was connected directly to the address-port of the IM.

Figure 1 shows how the output of the PC was connected to the address-port of the IM.

E. Register File

The register file stores the operands that are used in calculations within the ALU and stores the result of the calculations. The register file has got two read ports and one write port. To control which of the 32 registers that is connected to the different ports, 5-bit address lines are used for the two read ports and the write port. The register file has got a control line named *reg_write* which is controlled by the Control Unit and is asserted whenever the register file is written to. The value of register zero is always zero. Thus, whenever a zero is needed in a calculation within the ALU, this register is used.

In case of an R-Type instruction, the operands that are going to be used in the calculation inside the ALU, is output from the register file by applying the address of the operands on the *read_register* inputs. The register where the result of the calculation will be stored is fed to the *write_register* input of the register file.

If the fetched instruction is an I-Type instruction, the register port *read_register_2* is used in case of a write-instruction to address the register value that is going to be written to memory. In case of a load/store operation, the value applied to *read_register_1* input must be zero, since the output from the register file to the ALU on the port *read_data_1* needs to be zero for the ALU to compute the correct memory address to load from in case of a load-instruction, and where to write to in case of a store-instruction. When performing a load or a store, the ALU will perform an addition of the two outputs from the register file, where the output from *read_data_2* contains the address for the load or store operation. Thus, when the value read from the port *read_register_1* is zero, the correct address is computed by the ALU. In case of a branch instruction (beq), the instruction will address two registers and read their values thorugh the outputs *read_register_1* and *read_register_2* that are compared by the ALU, where the ALU decides if a branch is going to be performed.

For J-format instructions, the register file is not used since the jump address is calculated directly from the instruction without the need to go through the ALU. In this case, the control line *reg_write* must be deasserted to make sure that nothing is written back to the register file so that the data within the register file is protected from being overwritten.

When designing the register file, it was decided to follow the MIPS standard by assigning 32 registers of length 32 bits to the register file. This way, 5-bit address lines for reading and writing would be able to address all the individual registers. Since the ALU would need to read from two registers simultaneously, the register file needed to have two read-ports. One write-port was sufficient since this MIPS-implementation would either require to write the result from an ALU operation to the register, or a value loaded from the DM. Hence, one 5-bit address line would be sufficient for selecting which register to write to in the register file. The Register File was given a 1-bit control line *reg_write* that was needed to control when the register file would be written to. To be able to select which registers to read from through the two 5-bit address lines, a multiplexer connected to each of the 32-bit registers for each of the read-ports was implemented. For selecting which register to write to by using the 5-bit *write_reg*-line, a decoder was selected. The decoder would convert the 5-bit binary address and select the register corresponding to the given address. To enable writing to the selected register, the *reg_write*-line needed to be asserted and the clock-input needed to have a rising-edge. This was solved by letting the clock and the *reg_write*-signal go through an and-gate and then letting the result go through another and-gate together with the line from the decoder that belonged to the specific register. A conceptual drawing of the register file reflecting the intentional design is shown in Figure 8.

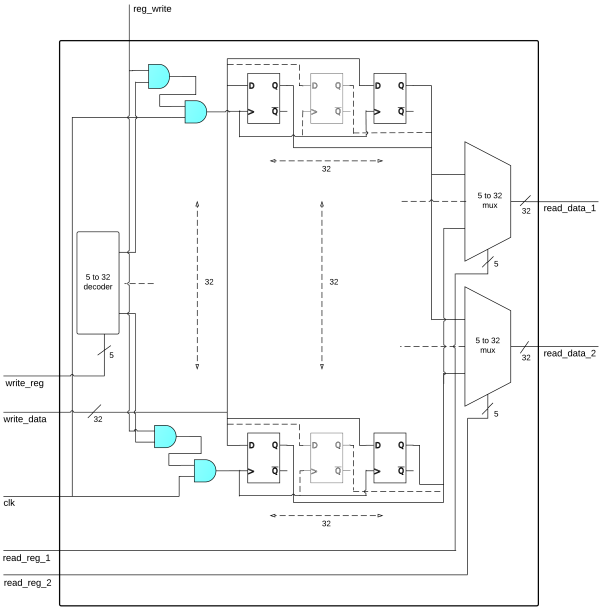

Fig. 8. Conceptual drawing for the register file

F. ALU

The MIPS processor in this design consists of three ALUs. However, only one of them is fitted with add, subtract, AND, OR and slt (set on less than) operators. This module is located between the DM and the Register File in Figure 1. The other two ALUs are respectively used for incrementing the program counter and for calculating the branch address. Henceforth, when the term ALU is used, it is referring to the module with five operators.

A 4-bit control signal from the ALU Control Unit determines which operator that the ALU should use. A detailed view of the operators and the corresponding ALU control signals can be seen in Table VIII. The ALU is capable of using one of five operators on two signed 32-bit integer operands ($data\_1$ and $data\_2$) to either produce a single 32-bit signed integer result, or to check for a branch condition. The branch condition is determined by using the subtract operator on the two input operands. If the result from the subtract operation is equal to zero, meaning that the two operands are equal,

the zero output signal is set high. The zero output signal in combination with the branch control signal determines whether the processor should update the PC with the branch address or do the usual increment.

The ALU itself is a combinational circuit. A series of multiplexers determines which route the two input signals ($data\_1$ and $data\_2$) should take, based on the signal from the ALU control module. The mode of operation is then based upon which route is taken, since the operators are attached at the end of each route as logic blocks. This is depicted in Figure 9.

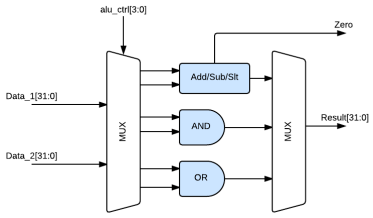

Fig. 9. RTL of ALU module

A more detailed RTL schematic of the add, subtract and slt operator-block is presented in Figure 10. The module

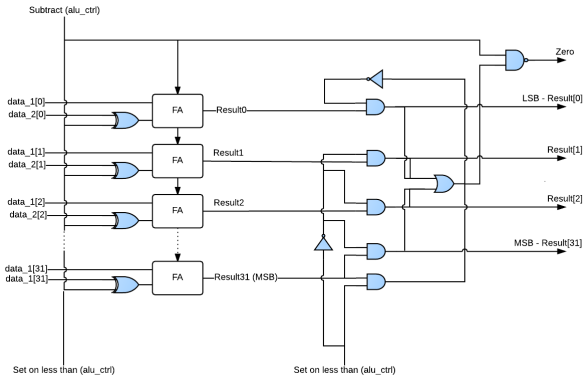

Fig. 10. RTL of add, subtract and slt block

consists of 32 full-adder blocks chained up in a ripple carry configuration. The XOR gates in front of the adders are used for subtract operations. The subtraction is done by two's complement. The AND gates on the right side of Figure 10 are used for slt operations, and the OR and NAND gate is used to determine whether the result is zero or not. An slt operation works by checking the MSB of the result from a subtract operation. If the MSB is one, the result is negative by two's complement. A negative result implies that $data\_2$ needs to be a bigger number than $data\_1$. To indicate this, a zero is put on the LSB. The LSB of the result is one when $data\_1$ is bigger than $data\_2$.

As seen from Figure 1, a multiplexer is connected to one of the inputs of the ALU. For store-operations this multiplexer takes input from the sign-extend unit and the shift unit. In this case, the output from the ALU is used as a DM address.

G. Memory Modules

A generated Xilinx IP core is used for both the IM and the DM module. This features a dual-port memory block, which is configured to store 256 32-bit signed integer data values. The module is of type dual-port, due to the fact that it should be accessible by both the implemented MIPS processor and the Hostcomm utility for testing purposes. The memory module is depicted in Figure 11.

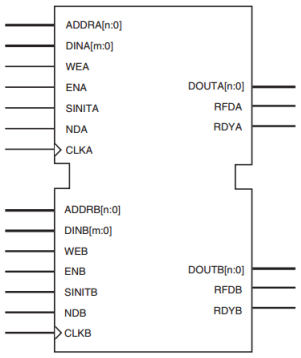

Fig. 11. Xilinx dualport memory module

Port A is used for the MIPS and Port B for the Hostcomm utility. The module has one control signal (WEA) to determine read or write operations. Write operations are active high, and read operations are active low. The WEA wire is connected to the mem_write signal from the Control Unit.

III. Results

THIS section presents the generated RTL's and waveforms for all the modules design in the Solution Section II. The results will be discussed in the Discussion Section IV.

A. Control Unit

The generated RTL from the synthesis in Xilinx is illustrated in Figure 12 and 13. Notice the wire marked in red in the figures which connects the two images of the RTL through the or gate in Figure 12. The complete RTL of the Control

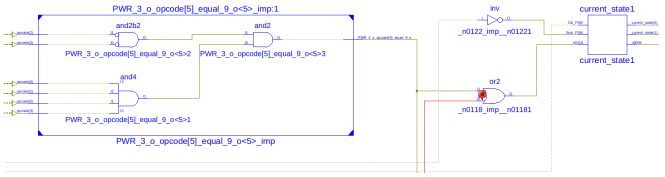

Fig. 12. Control Unit RTL

Unit is found in Appendix -A, 20.

Figure 14 depicts the correct behaviour of the Control Unit through simulation. The wavefrom of the testbench covers all the valid *opcode* combinations and is checked for the correct output. All the checks are met and no errors are asserted.

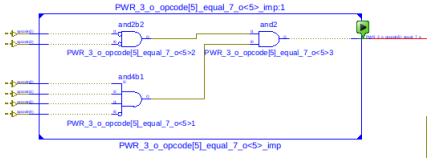

Fig. 13. Control Unit RTL

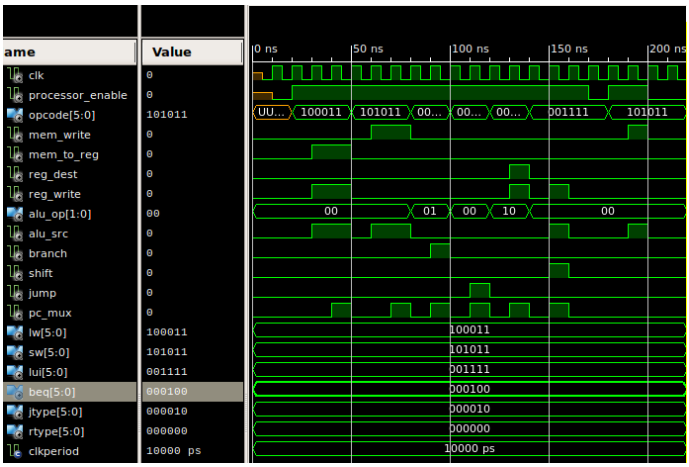

Fig. 14. Waveform of the Control Unit Testbench.

B. ALU Control Unit

The simple RTL of the ALU Control Unit is depicted in Figure 15.

Figure 16 depicts the correct behaviour of the ALU Control Unit through simulation. The ALU Control Unit was tested for all combinations of inputs and all checks gave no errors.

C. Program Counter

The RTL generated by the synthesis tool for the PC is shown in Appendix -A, Figure 17. A single 32-bit register was generated with the input connected to *addr_in* and the output to *addr_out*, with the opportunity to clear the contents of the register through the *CLR*-line.

A testbench was used to test the functionality of the PC and to make sure that the module performed as expected. In the testbench it was tested to write to the PC register, and that it updated its value on the rising edge of the clock. Clearing the saved value was also tested by asserting the reset-line. The resulting waveform from the testing of the PC is shown in Appendix -B in Figure 21.

D. Register file

The generated RTL for the Register File is shown in Appendix -A, Figure 18. The 32-bit registers are shown as individual blocks, with separate clock-enable (CE) inputs. The decoder was realized by the synthesis tool in Xilinx by using 5-input AND-gates with the output from the decoder connected to an AND-gate together with the *reg_write*-signal, where the output from decoder was inverted. The output from the AND-gate was further connected to an OR-gate together

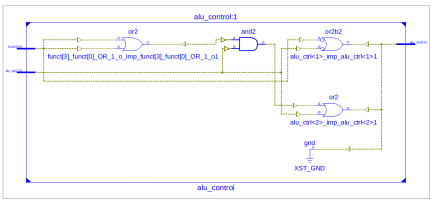

Fig. 15. RTL of the ALU Control Unit.

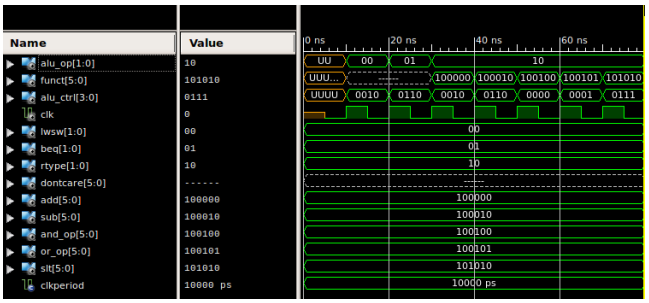

Fig. 16. Waveform of the ALU Control Unit Testbench.

with the inverted *reg_write*-signal. Before being connected to the CE-input of the registers, the output from the OR-gate was inverted. The design requirement of register zero to always have value zero, was realized by the synthesis tool by omitting register zero, and let the input to the multiplexers for read-address 0 be connected to ground. The reset-input for the Register File was connected to the *CLR*-input on all the individual registers as shown in the figure.

The Register File was simulated using a testbench with the resulting waveform shown in Appendix -A, Figure 22. By using the testbench, it was tested to write to the Register File, and read the written information through both read-ports. As shown in the figure, a simultaneous read from the same register was also conducted. The reset function of the Register File was also tested by checking that all registers read as zero after the *reset* had been asserted.

E. ALU

An ALU module with with five different arithmetic operators has been successfully designed in this project. The generated RTL schematic of the ALU is depicted in Appendix -A, Figure 19. The schematic consists of a series of multiplexers and dedicated logic blocks for each of the specified operators. The functionality of the ALU module has been thoroughly tested. The waveforms generated from the testbench can be found in Appendix -B, Figure 23 and **??**. The testbench applies different stimuli operands for each test condition, this is done three times for each of the five operators.

F. Simulation of the design

The complete MIPS processor passed the provided top-level testbench. The waveforms are depicted in Appendix -B, Figure 25, 26, 27 and 28.

G. Testing on FPGA

The design was implemented on the FPGA and controlled through the Hostcomm Utility. During testing on the FPGA, the same instructions used for testing the design in simulation, were loaded into the IM and the required data to compute the same results as when running the testbench, were loaded into the DM. After running the implementation on the FPGA, the contents of the DM was read using the Hostcomm Utility and is shown in Appendix -C, Figure 29. The implementation was also tested with fewer instructions as shown in Figure 30. In this test, it was tested to load values from the DM, perform simple calculations on the values, and write them back to memory. Figure 29 shows the test performed with the DM containing a zero in position zero, while Figure 30 displays the same test, but with the DM containing a value of one in position zero.

IV. Discussion

THE MIPS architecture is fairly simple and with simplified functionality it becomes a realtivly small CPU. Even so, it is important when designing such a CPU, to be able to get control of the whole design. This i probably the biggest challenge.

A. Control Unit

It is important that the Control Unit works properly as it is critical for the correct behaviour of the processor. Looking at the generated RTL in Figure 12 and 13 it deviates from the proposed RTL in Figure 4 These RTL's illustrates only the next state conditions for the Control Unit, but the control signals are dependent on the state of the Control Unit. The deviations indicates a poor design and lack of considerations. Xilinx gives warnings on the design inferring latches on all outputs. This should be accounted for. One way of inproving the design should be to make sure all cases in a case-statement are covered and that all signals are set.

B. ALU Control Unit

The generated RTL for the ALU Control Unit illustrated in Figure 15 is similar to the one proposed in Figure 7. Compared to the proposed solution the AND gate for the MSB of the output in the generated RTL is removed. This does not change the behaviour of the circuit as the output should logic zero. In the generated RTL optimizations for output bit two and zero is combined in on expression. Operations is still the same. This is confirmed through the waveform in Figure 16.

C. Program Counter

By comparing the conceptual design for the PC with the RTL generated by the synthesis tool in Xilinx, it is clear that the generated RTL and it's performance studied by using the testbench corresponded to the initial design criteria. When studying the memory module used for the IM in this implementation, it is seen that the address-line for this memory module is 8-bit. This means that the register within the PC could be reduced to 8-bit, since the number of lines required to address all the positions in the IM is 8.

D. Register file

The generated RTL for the Register File is shown as Figure 18 in Appendix-A. Observe that the generated RTL in Appendix-A is similar to the conceptual drawing in Figure 8. One difference is the omitted zero register in the RTL generated by the sythesis tool, which was replaced by a direct connection to ground. As register zero is supposed to always have a value of zero, and since a logical low signal is considered as zero, this leads to a more efficient implementation since it contains one register less than the conceptual design. The *reset*-line that was added to the Register File in the RTL, was a result of it being added to the vhdl code for the module after the conceptual design was set. When trying to synthesize the module without adding a *reset*, it was discovered that the synthesis tool implemented the Register File in RAM present within the FPGA, rather than using D-Flip-Flops for creating the module as shown in 18. The convenience of having a way to clear the Register File together with the intention of using D-Flip-Flops for the registers led to the chosen implementation of the register file.

In the generated RTL for the Register File, the number of AND-gates that together form the decoder, does not correspond to the number of registers. The generated RTL indicates that the output from some of the AND-gates, are used to enable writing for several registers. If this is the case, it could cause problems since intending to write to one register, may overwrite values in other registers. To further clarify this, a testbench writing to one register, then checking the value of all other registers could be used.

E. ALU

The proposed RTL schematic of the ALU in Figure 9 is somewhat different to the generated RTL from Xilinx ISE in Appendix -A. The multiplexed path is similar, but it seems that the Xilinx ISE synthesizer applies some optimizations to the HDL and uses dedicated blocks for the add/subtract, slt and equal operators. How these blocks were implemented was not possible to see from the Xilinx RTL viewer, so no comparison with Figure 10 could be made. However, it is reasonable to assume that a great deal of optimizations was applied to these blocks as well.

Only standard VHDL operators was used to implement the logic blocks of the ALU, and there was made no attempt to implement more advanced features such as a CLA (carry-lookahead) as described in lecture 2 [3]. However, according to Xilinx [5] , all modern FPGAs have dedicated logic for all standard operators. So in almost every case, a standard A + B operation in VHDL will be much faster than any self implemented CLA.

F. Simulation of the design

As the Figure 25, 26, 27 and 28 illustrates, the simulation was sucessful.

G. Testing on FPGA

When the design was implemented on the FPGA and controlled through the Hostcomm Utility, with the same instructions as used in the testbench for the design, the values read

back from the DM did not correspond to the values that the simulation produced for the same instructions. When testing the design with fewer instructions as shown in Appendix -C, Figure 29 and 30, the design produced the expected values in the DM when it was run with a value of one in DM position 0 as shown in Figure 30. Without this value, the DM did not receive the expected values as depicted in Figure 29.

When synthesizing the design for testing on the FPGA, warnings were produced by the synthesis tool in Xilinx. The warnings concerned latches used for the control signals in the Control Unit. These warnings could be studied in order to isolate the problem in the design that causes it to produce unexpected values in the DM when tested on the FPGA. Testing using simulation with even simpler and fewer instructions and testing the same instructions on the FPGA could also assist in isolating the problem within the design.

V. Conclusion

A simple MIPS processor with a reduced instruction set has been presented in this report. The design passed the behavioural simulation with zero errors and synthesized without any errors. However, testing on the FPGA revealed that the design did not work as intended. The cause of this may be the inferred latches within the Control Unit, since warnings were generated during synthesis.

References

[1] http://en.wikipedia.org/wiki/MIPS_instruction_set
MIPS instruction set
[2] *Load Upper Immediate*,
http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html
[3] Donn Morrison, *VHDL and the basic of logic design*
TDT4255 Computer Design Lecture 2: VHDL and the basic of logic design
[4] Donn Morrison, *The processor*
TDT4255 Computer Design Lecture 4: The processor
[5] http://forums.xilinx.com/t5/Spartan-Family-FPGAs/Carry-look-ahead-logic-in-FPGAs/td-p/120744
Is a self implemented CLA adder faster than a standard A + B operation

APPENDICES

A. Generated RTL

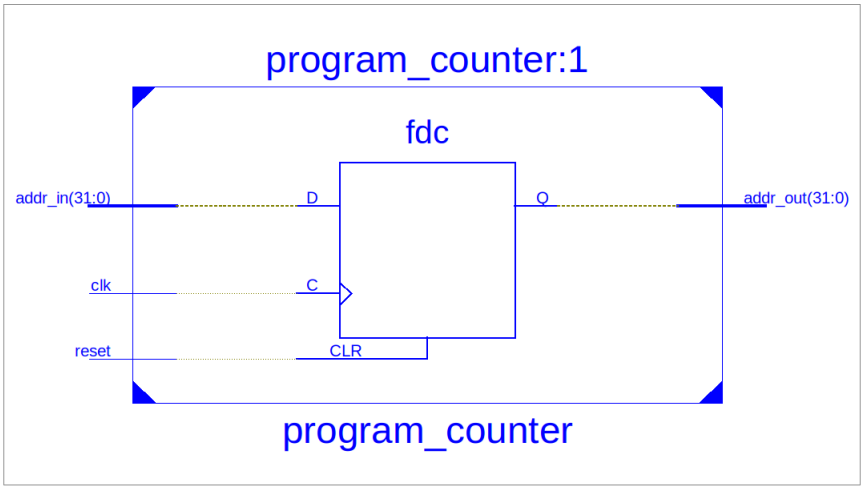

Fig. 17. Generated RTL for the Program Counter

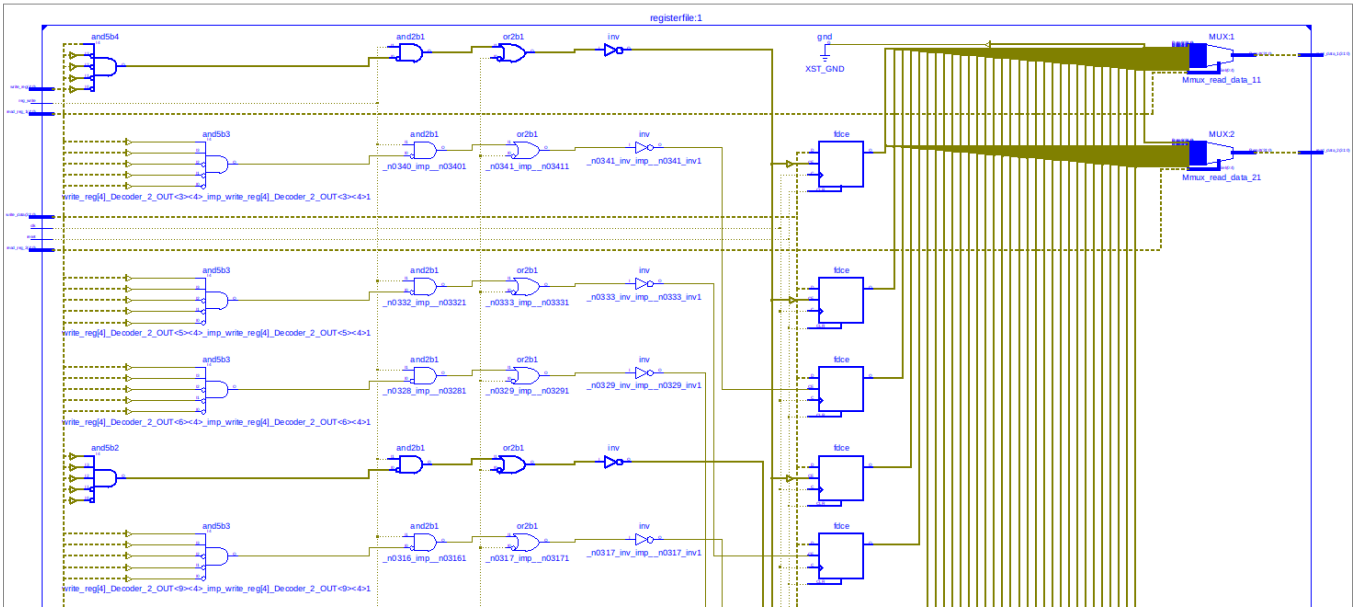

Fig. 18. Generated RTL for the Register File. Upper portion is shown.

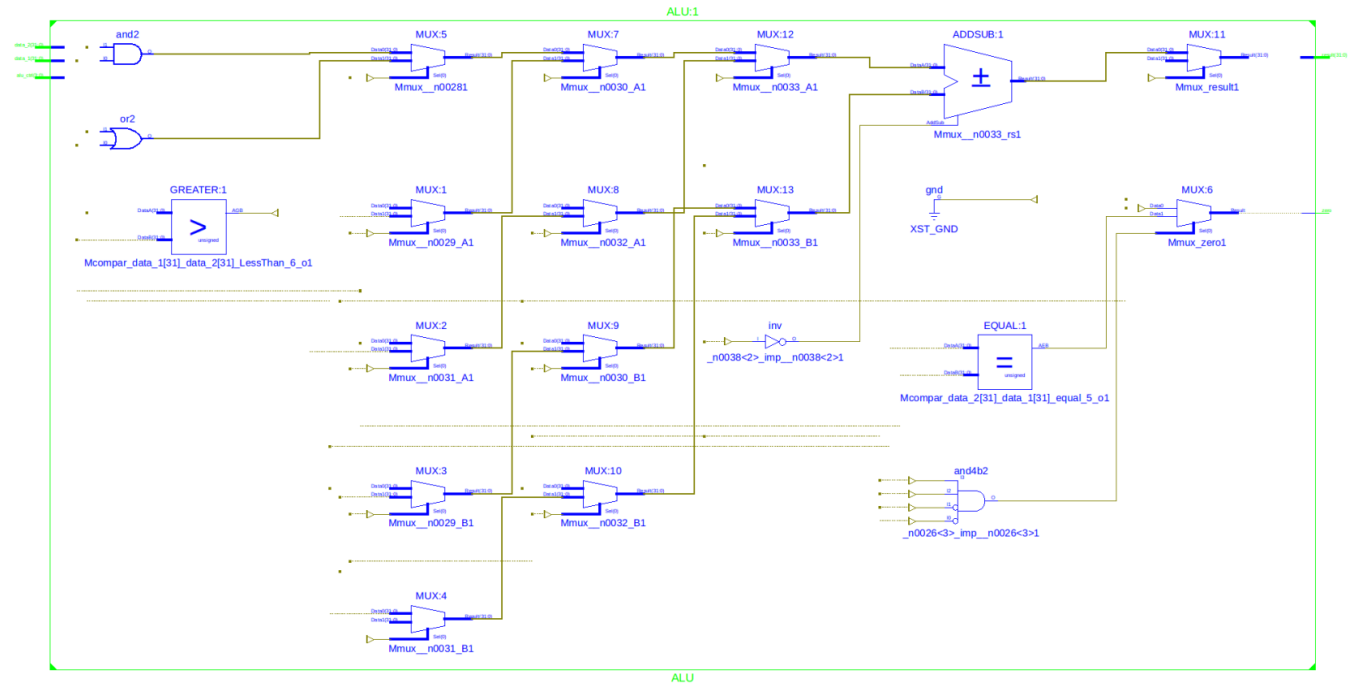

Fig. 19. Generated RTL for the ALU

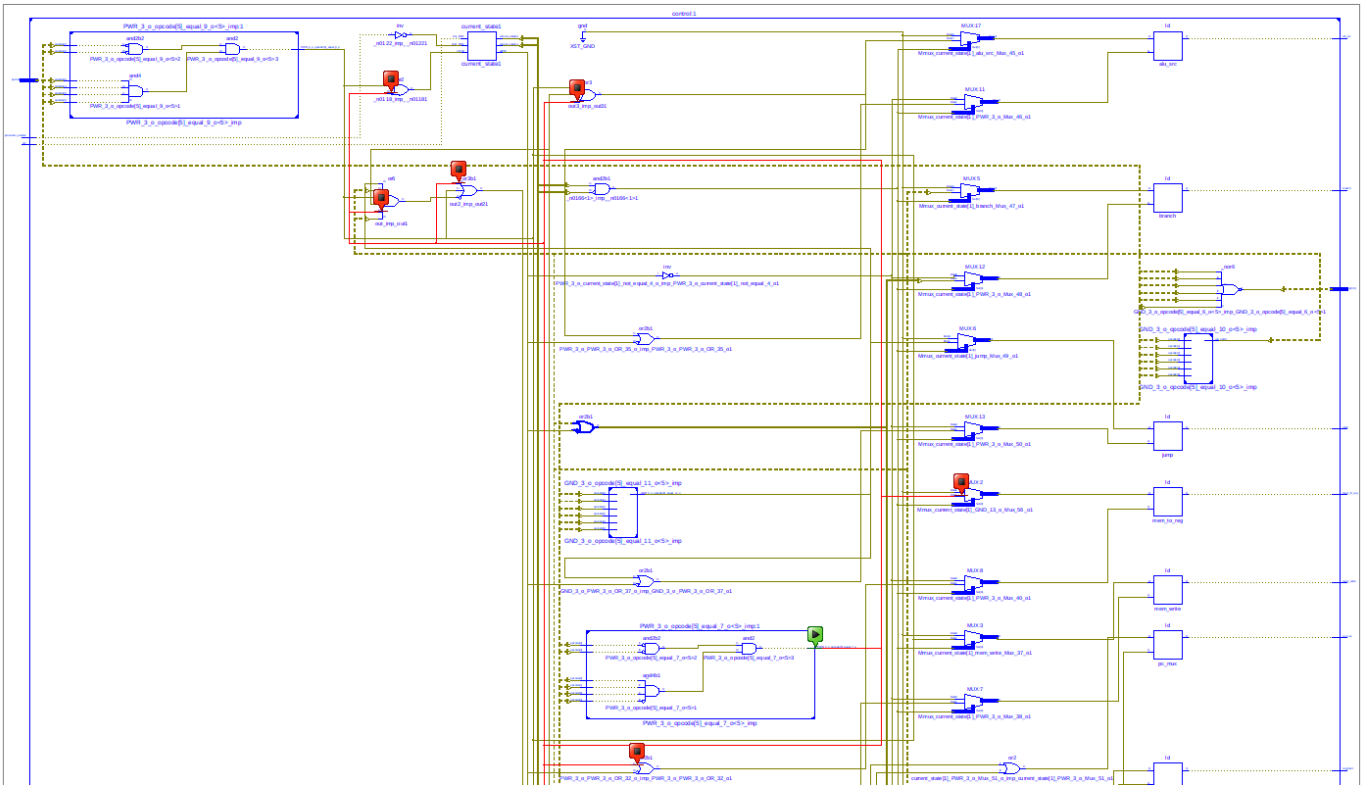

Fig. 20. Generated RTL for the Control Unit.

B. Testbench results

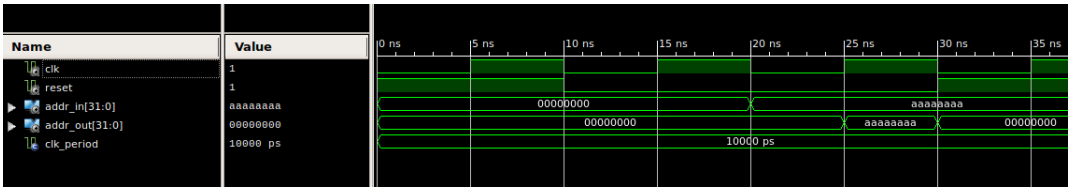

Fig. 21. Testbench results for the program counter

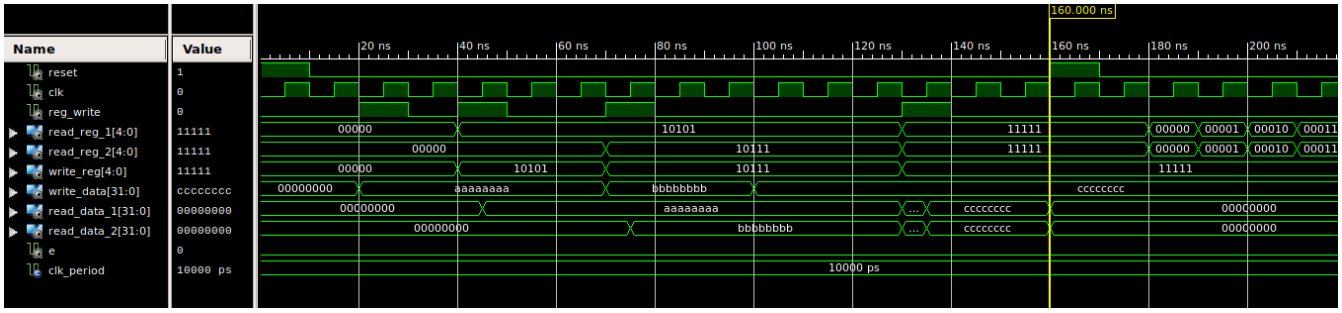

Fig. 22. Testbench results for the register file

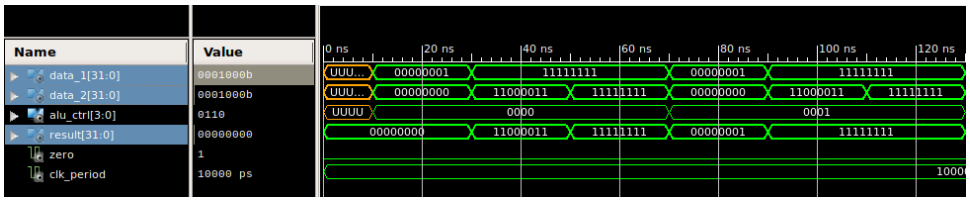

Fig. 23. Waveform for ALU module (AND and OR operations)

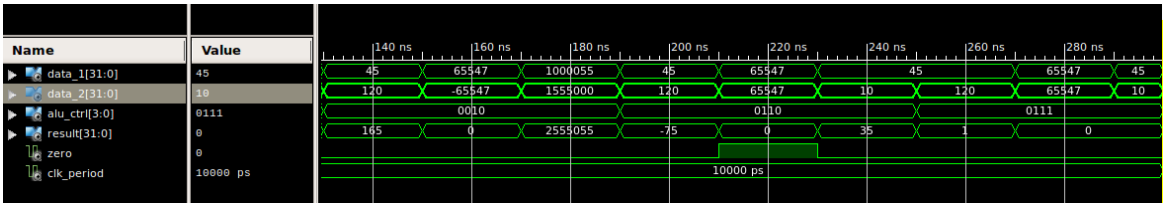

Fig. 24. Waveform for ALU module (add, subtract and slt operations)

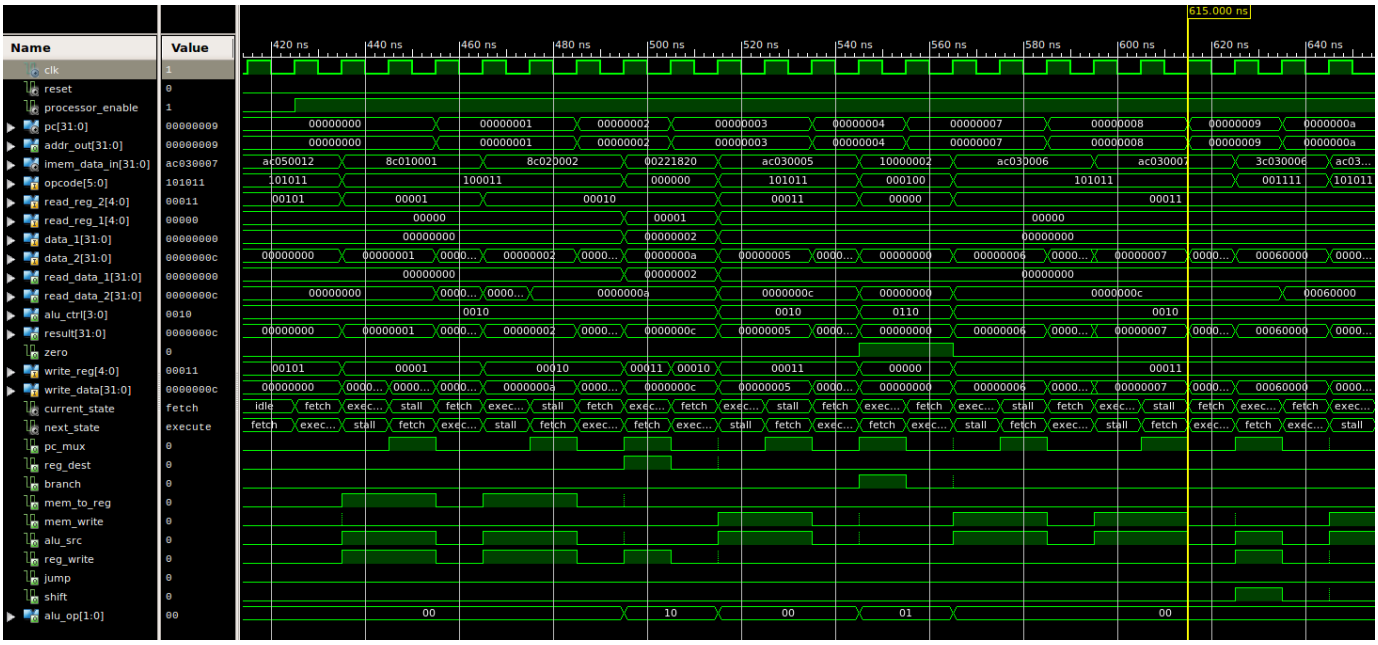

Fig. 25. Testbench results for the top level part 1

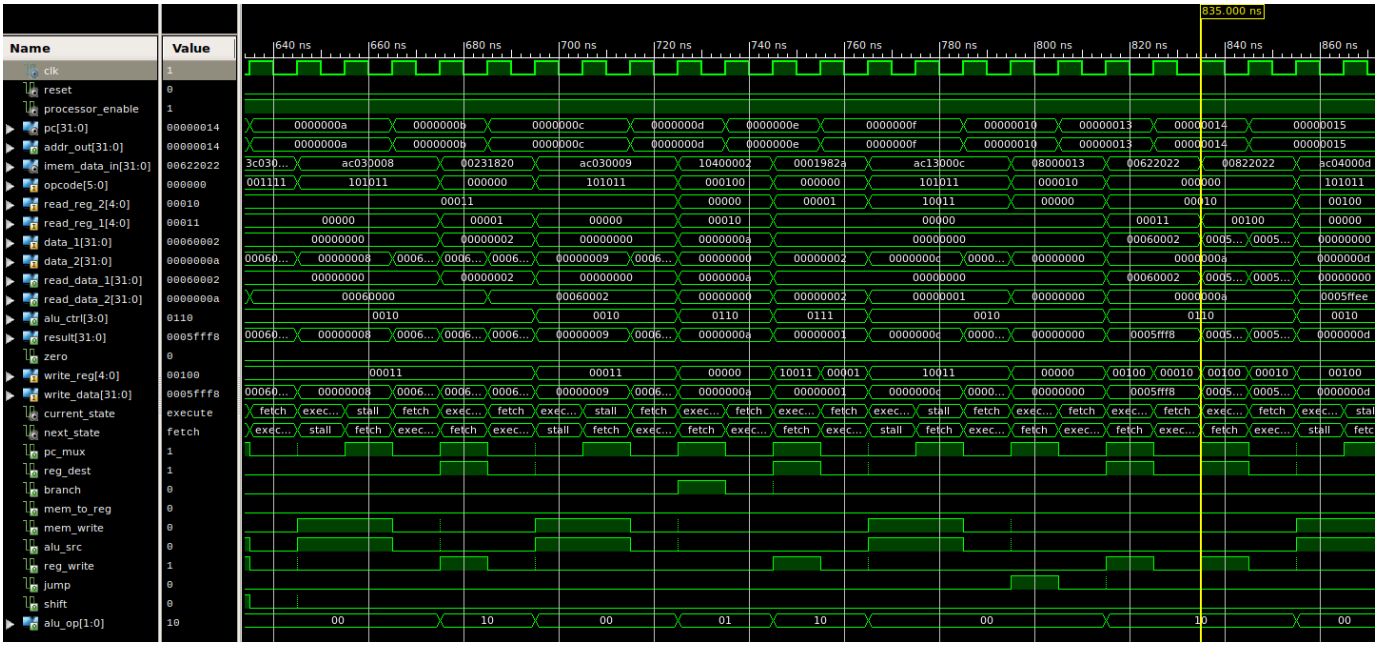

Fig. 26. Testbench results for the top level part 2

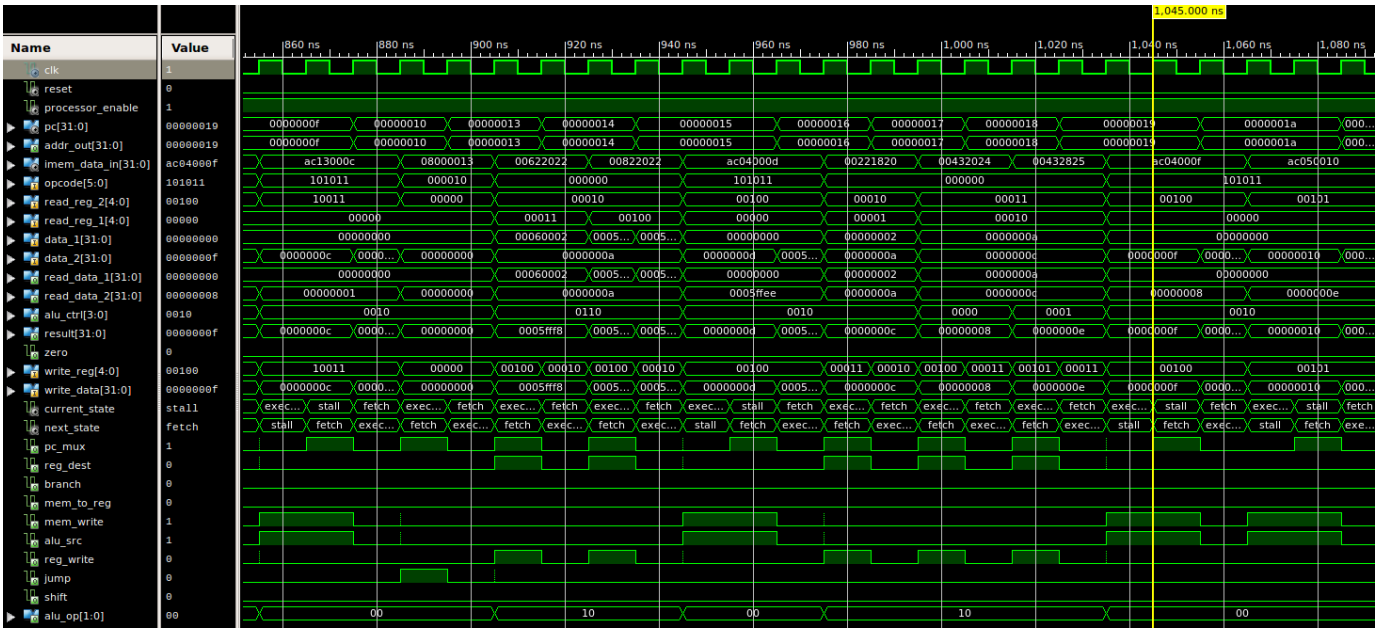

Fig. 27. Testbench results for the top level part 3

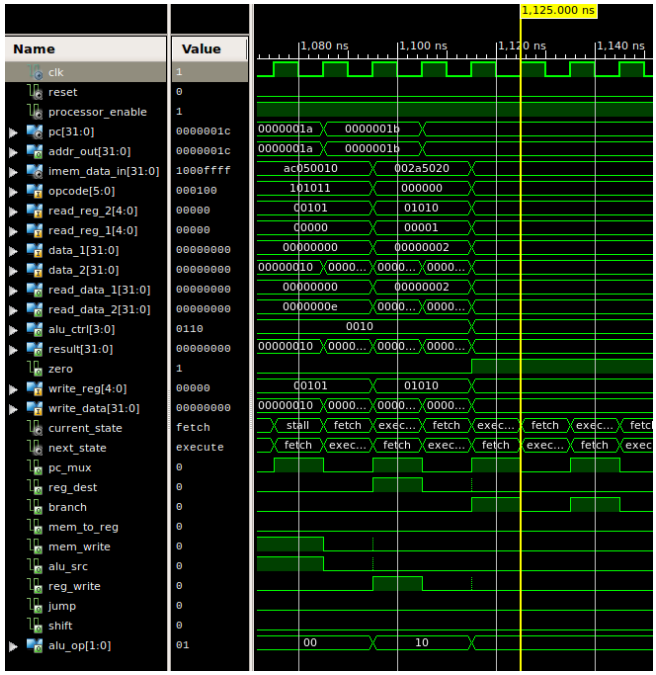

Fig. 28. Testbench results for the top level part 4

C. Testing on the FPGA

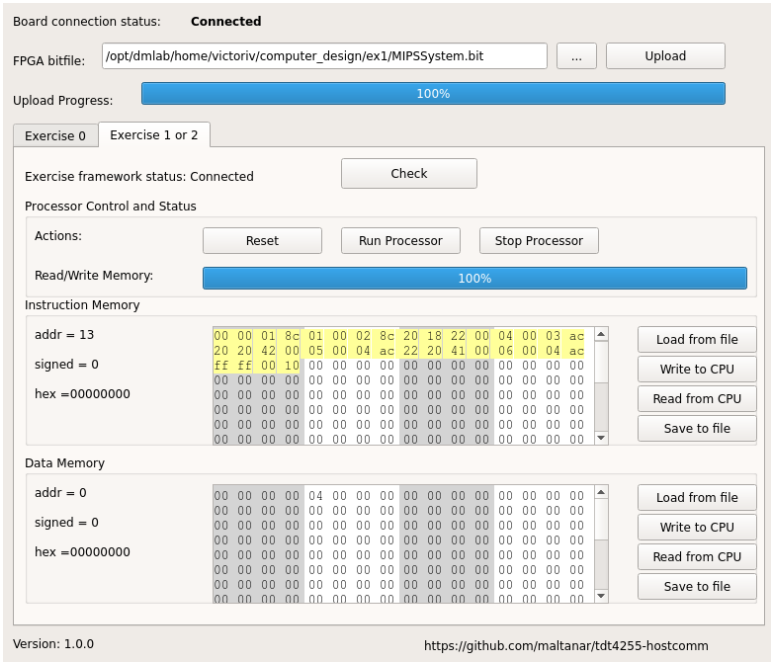

Fig. 29. Testing the implementation on the FPGA by reading values from memory, performing ADD and SUB operations, and writing the results back to memory.

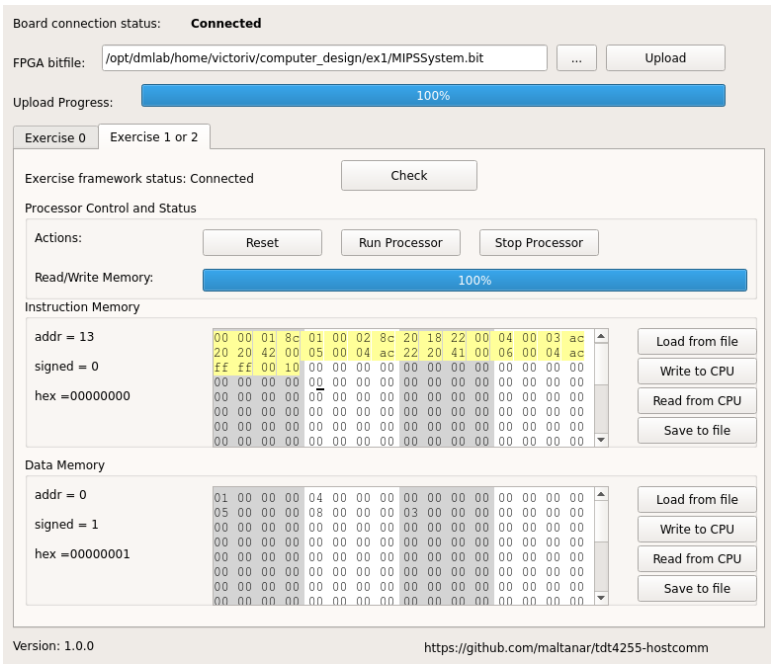

Fig. 30. Testing the implementation on the FPGA by reading values from memory, performing ADD and SUB operations, and writing the results back to memory. DM position zero contains a value of one.