FACULTY OF ENGINEERING AND NATURAL SCIENCES

A RISC ARCHITECTURE BASED 8 BITS COMPUTER DESIGN AND IMPLEMENTATION ON FPGA USING VHDL

Reported By

Ömer KARSLIOĞLU

May, 2022

ANKARA

ACKNOWLEDGEMENTS

I would like to thank Asst. Prof. Gökhan Koray GÜLTEKİN for advising me throughout this project and for sharing his valuable knowledge and experience with me throughout my undergraduate education and contributing to my development.

2022 , 28 May **Ömer KARSLIOĞLU**

A RISC ARCHITECTURE BASED 8 BITS COMPUTER DESIGN AND IMPLEMENTATION ON FPGA USING VHDL

ABSTRACT

The general aim of this project is to design a computer system that requires low cost, has high efficiency 8-bit register cells and has RISC instruction set architecture. This computer system is suitable for algorithms that can perform various arithmetic and logical operations and make decisions as a result of operations. The main components of this designed architecture are CPU, memory system, timer units and ICU. This processor supports 37 instructions and has various peripheral features such as interrupt and PWM generating. This designed computer was run on FPGA.

Keywords: RISC, FPGA, HDL, Computer Architecture, Memory, Timer, Interrupt, Memory

Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACRONYYMS	FULL FORM
ISA	Instruction Set Architecture
RISC	Reduced Instruction Set Computer
HDL	Hardware Description Language
IR	Instruction Architecture
PC	Program Counter
MAR	Memory Address Register
CCR	Condition Code Register
ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
RAM	Read Access Memory
ROM	Read Only Memory
IMM	Immediate Addressing
DIR	Direct Addressing
ICU	Interrupt Control Unit
TIM	Timer Peripheral Unit

1- Introduction

Today, the usage area of processors is very wide. Processors are used in all products from high-tech products to low-tech products. The demand for processors is increasing day by day. As a result of these demands, there are deficiencies in terms of both design and production. This 8-bit RISC-based computer system that I have designed can be integrated into many technological products that people use in daily life.

The most important goal of this computer I have designed and the processor in it is that it is very suitable for use in low-cost applications and is effective in terms of use. In order to understand how these features are provided, we need to examine this computer that I have designed in more detail.

RISC instruction set architecture is simpler than CISC and all instruction sizes are fixed and the same in RISC. Thus, RISC-based designs are not complex. Although uncomplicated RISC-based processors are less successful in powerful operations, they consume less power than CISC and are therefore generally effective.

The designed computer architecture generally consists of CPU, memory system, ICU and timer units. Let's examine these units in more detail.

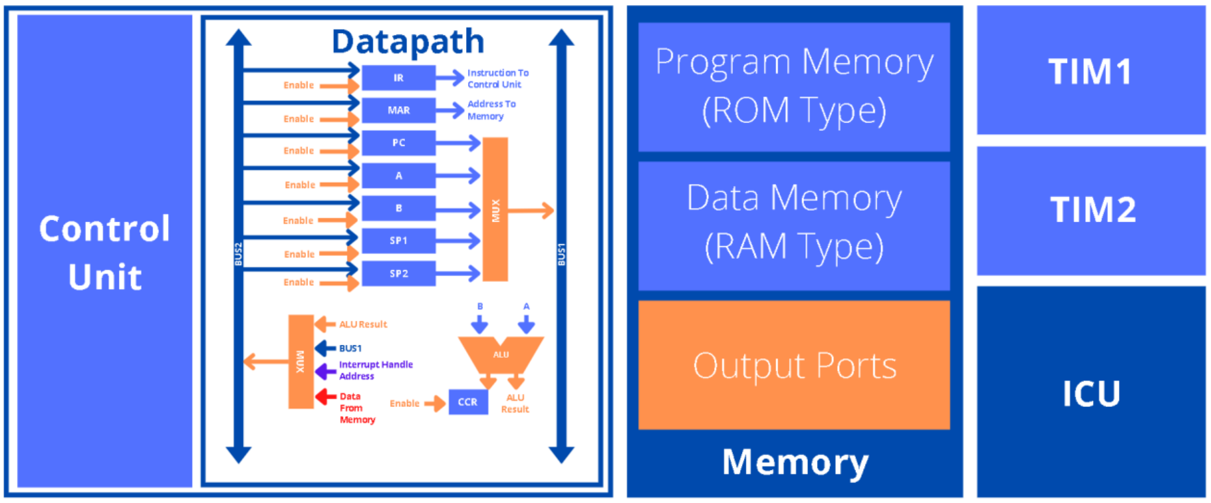


Figure 1 The RISC Architecture Based 8-Bits Computer Schematic: System diagram shows the main units of the computer.

In the schematic you have seen above, Control Unit and Datapath units are hierarchically combined in the same module and the CPU is created.

I can say that the memory system consists of three separate modules: ROM, RAM and output ports. program memory and data memory are in L1 cache structures and are separate from each other as you can see. The reason it is separate is to prevent instructions and data from coming from the same source. This will make the processor run faster. Separating these memories helped me to make my design look like Harvard Architecture.

In the same schematic also shows TIM1, TIM2 and ICU peripherals.

1.1- CPU
It is the hardware unit that manages the entire computer system and where instructions are processed.

1.1.1- Control Unit
It is the hardware unit that controls the Datapath according to the detected instruction. How to compute all the instructions is microarchitecturally designed in the Control Unit. It also directly controls the interrupt status.

1.1.2- Datapath
It is the hardware unit where instructions are processed with fetch-decode-execute steps. It also establishes the relationship between the CPU and the memory system.

There are two BUS structures in Datapath. While one of these bus structures assigns values to the registers, the other one sends the data from the registers to the memory or returns that value in the Datapath.

1.1.2.1- Registers
There are seven (7) registers in total in Datapath.

- IR: It is the register where the read instruction is saved.
- PC: It is the register that points to the opcode or operand to be read from the program memory. In normal program flow, the PC value usually increases by one.
- MAR: It is the register that directly transmits the address information to the memory.
- A-B: They are general purpose registers.
- SP1-SP2: These are the registers used for interrupt status and to hold addresses in branch operations. While SP2 is only for interrupt, SP1 is user access register.

1.1.2.2-ALU
It is the execution unit in which arithmetic and logical operations are performed. In order to better understand the ALU capacity of the processor, the instruction table should be examined.

1.2- Memory System
There are three memories in total in the memory module: RAM, ROM, Output Ports. The general operating logic of the memory system is to whichever memory unit the incoming address data is matched with, the data pointed to by the address is drawn from that memory unit and transferred to the data output port of the memory unit.

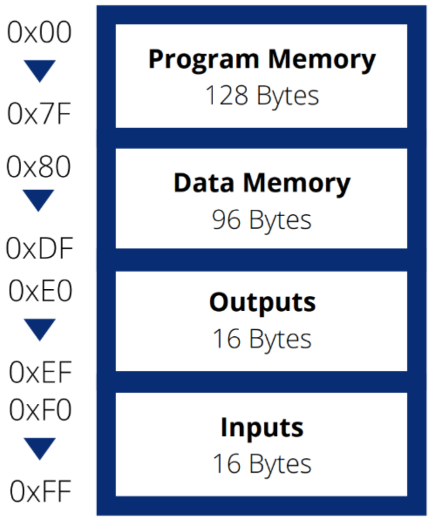


Figure 2 Memory Map Design: The schematic of the memory structure designed to show the map address ranges is as follows.

The addresses of all memory units are designed to follow each other. Thus, a more effective solution was created. All memory can be accessed with a single address signal.

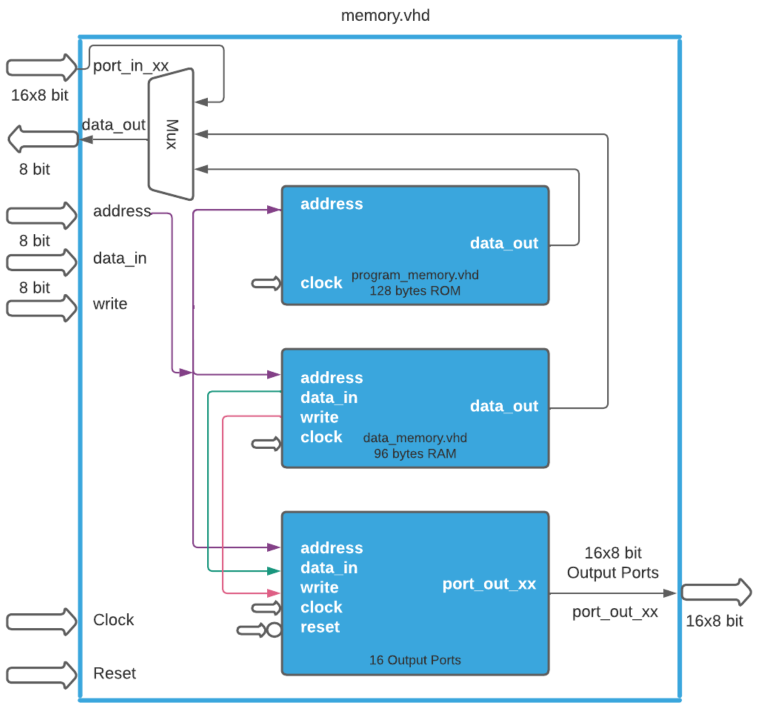


Figure 3 Memory System Diagram: The diagram shows inside of the memory system design.

The entire memory system works in a synchronous structure and the incoming address enters all memory units. The process related to the memory unit with which the address is matched is performed. Peripherals are connected to the 2-15 port range of the output port memory unit. The general IO unit is connected to ports 0-1.

1.3- Peripheral

There are two timer units and ICU units in this computer architecture. While the ICU can directly affect the CPU, the timer can only interact with the CPU together with the memory. For more detailed information, you can find the title of the peripheral you are interested in from this report.

1.4- FPGA Design Flow

The computer architecture project has progressed on FPGA flow.

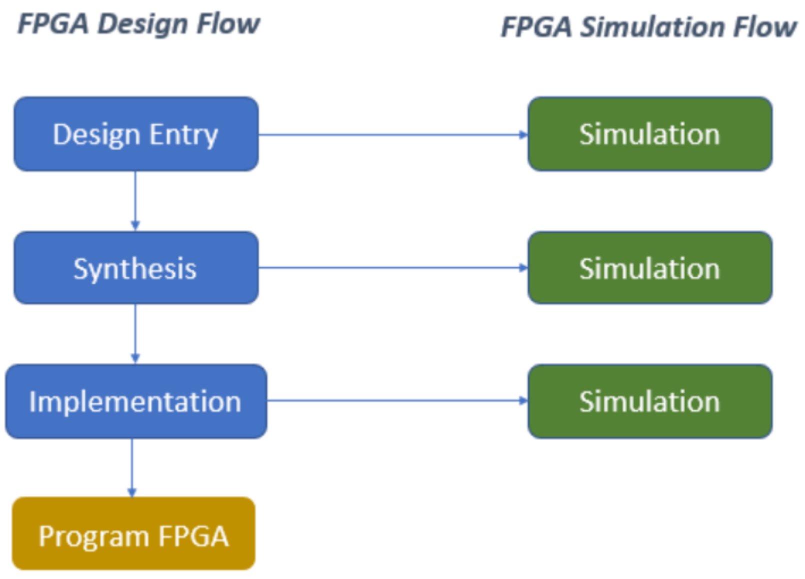


Figure 4 FPGA Design Flow: The diagram is a design flow diagram for FPGA.

According to the FPGA flow, firstly, how the design will take place was considered and the necessary resources were collected. Afterwards, the behavioral model design phase was carried out and the simulations were completed. After the synthesis step, static timing analysis was performed. In the synthesis phase, after the verification processes were carried out, the implementation step was started. After re-simulation, the FPGA was programmed.

2-Instruction Design

The designed instruction structure consists of an opcode and an operand. Each instruction must have one opcode value. Some instructions do not require an operand value. The dimensions of the designed instructions consist of 8 and 16 bits.

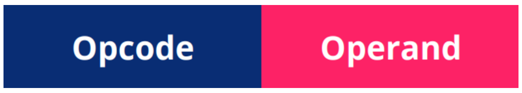


Figure 5 Instruction Structure: The design represents instruction structure occurred from opcode and operand.

While processing instructions, they follow the steps of fetch, decode and execution. Each instruction is processed at different clock cycle times. The reason for this is to shorten the critical path and increase the performance of the processor. Therefore, the processor design is the multiple clock cycle processor design.

The processing of instructions is briefly as follows. With the first initial fetch step, the instruction is read from and fetched from the ROM, which takes one clock cycle. Then, the decoding process takes place according to the instruction and it is understood what the instruction is. It takes a clock cycle in the decode step. The steps so far are the same for all instructions. The execution step is different for each instruction. In this step, the execution step takes place according to the signals that control the hardware coming from the control unit. If data is to be read or written from memory in the instruction, one clock cycle is processed more. This is because memory responds after one clock cycle.

Here is an example of how an instruction works in my design:

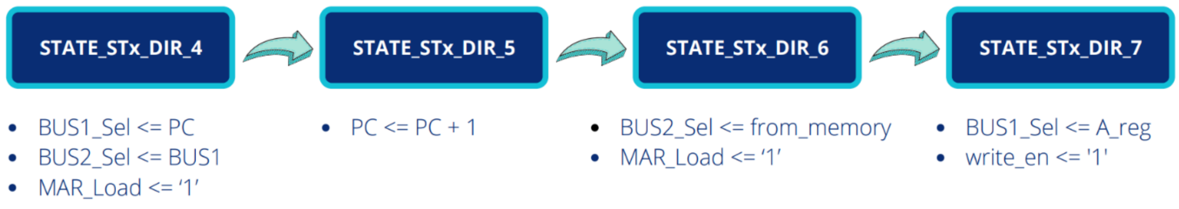
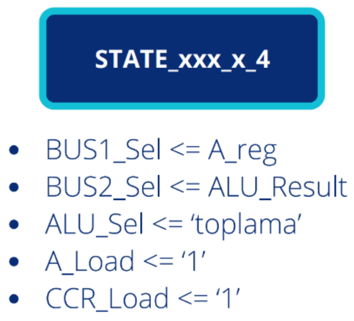


Figure 6 Instruction Execution Steps: The figure represents the flow of execution steps of an instruction.

Above, you can see the execution steps after the fetch and decode steps of the store instruction. The task of this instruction is to save the value in the desired general-purpose register to the direct address found as the operand in the instruction. This instruction is 16 bits.

- After decoding, the PC value incremented in fetch is transferred to the MAR register to reach the direct address (operand). When transferring to the MAR register, the PC is transferred first to BUS1 and then from BUS1 to BUS2. It is also transferred from BUS2 to MAR. (Attention: The MAR value is designed to access the memory address directly.)
- In the next step, the PC value is increased by one for the next instruction and the direct address (operand) data is expected from the memory.
- In the next step, the incoming data is transferred to BUS2 and then to MAR.
- In the last step, the value in the general-purpose register is transferred to BUS1. This data is written from BUS1 directly into the memory address shown by MAR.

Another example is the type of instruction that needs to be processed in the ALU:



STATE_xxx_x_4

- BUS1_Sel <= A_reg
- BUS2_Sel <= ALU_Result
- ALU_Sel <= 'toplama'
- A_Load <= '1'
- CCR_Load <= '1'

Figure 7 Instruction Execution Step In ALU: The figure represents the state of an instruction in ALU.

The type of this instruction is executed in the ALU according to the selected operation. There are two input inputs in the ALU to perform operations. One of them is B general-purpose register and the other is BUS1. (Attention: register A is not directly connected.) The execution after the fetch and decode steps consists of a single step and takes a single clock cycle.

- According to this instruction, which is read with an incremented PC value in the fetch step, the signals from the control unit set the ALU as in the image above. For example, if addition is to be made, the value of A register is transferred to BUS1. The ALU is set according to the aggregation process. The result is written to register A and the CCR register is updated, showing the NZVC states of the result.

2.1- Instruction Table

	Opcode (Mnemonic)	**Opcode (Value)**	**Description**
1	YUKLE_A_SBT	x”86”	Load register A using immediate value
2	YUKLE_A	x"87"	Load register A using direct addressing
3	YUKLE_B_SBT	x"88"	Load register B using immediate value
4	YUKLE_B	x"89"	Load register B using direct addressing
5	LOAD_PC_TO_SP1	x"90"	Load PC value to SP1 register
6	LOAD_SP1_TO_PC	x"91"	Load SP1 value to PC register
7	JUMP_AND_LINK_SP1	x"92"	PC takes value of SP1, and previous PC value is saved in SP1
8	LOAD_SP1_IMM	x"93"	Load immediate value Stack Pointer-1 register
9	LOAD_SP1_DIR	x"94"	Load value to Stack Pointer-1 register from direct address
10	KAYDET_A	x"96"	Store Register A to Memory using direct address
11	KAYDET_B	x"97"	Store Register B to Memory using direct address
12	TOPLA_AB	x"42"	A=A+B
13	CIKAR_AB	x"43"	A=A-B
14	AND_AB	x"44"	A = A and B
15	OR_AB	x"45"	A = A or B
16	ARTTIR_A	x"46"	A=A+1
17	ARTTIR_B	x"47"	B=B+1
18	DUSUR_A	x"48"	A=A-1
19	DUSUR_B	x"49"	B=B-1
20	SRL_A	x"4A"	Logical shift right
21	SLL_A	x"4B"	Logical shift left
22	SRA_A	x"4C"	Arithmatic shift right
23	SLA_A	x"4D"	Arithmatic shift left
24	ROR_A	x"4E"	Rotate right logical
25	ROL_A	x"4F"	Rotate left logical
26	RESET_ALU	x"50"	Reset ALU, clear all signals
27	NOT_A	x"51"	A=~A
28	XOR_AB	x"52"	A = A xor B
28	ATLA	x"20"	Branch always to address provided
29	ATLA_NEGATIFSE	x"21"	Branch to address provided if N=1
30	ATLA_POZITIFSE	x"22"	Branch to Address provided if N=0
31	ATLA_ESITSE_SIFIR	x"23"	Branch to Address provided if Z=1
32	ATLA_DEGILSE_SIFIR	x"24"	Branch to Address provided if Z=0
33	ATLA_OVERFLOW_VARSA	x"25"	Branch to Address provided if V=1
34	ATLA_OVERFLOW_YOKSA	x"26"	Branch to Address provided if V=0
35	ATLA_ELDE_VARSA	x"27"	Branch to Address provided if C=1
36	ATLA_ELDE_YOKSA	x"28"	Branch to Address provided if C=0
37	NOP	x"01"	Stall datapath line during four clock cycle

Table 1 Instruction Table

2.2- Instruction Verification

All instructions will be tested in this section. The test software is written to the ROM with VHDL in the following format:

PC => Opcode or Operand,

- YUKLE_A_SBT

0 => YUKLE_A_SBT,

1 => x"0",

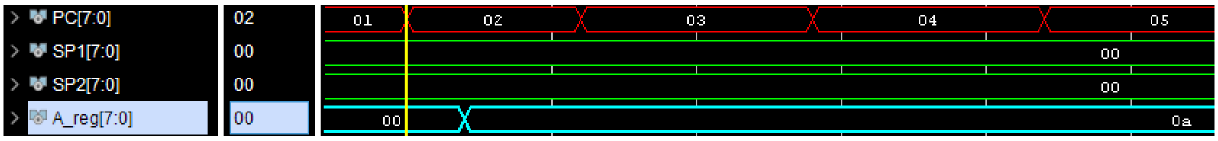


Figure 8 The Waveform Of Load Imm. (A) Instruction: The waveform shows the load instruction works correctly.

After PC value 1, we see that x0A value is loaded into A register.

- YUKLE_A – YUKLE_B

8 => YUKLE_A,

9 => x"81",

10 => YUKLE_B,

11 => x"80",

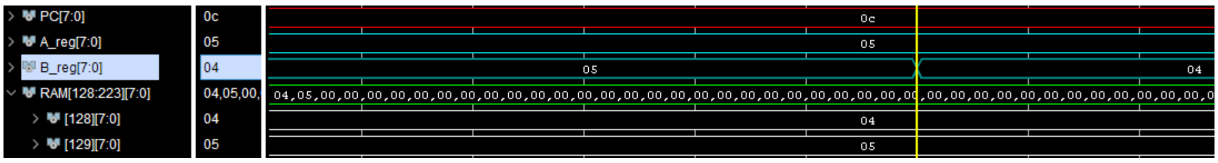


Figure 9 The Waveform Of Load From Addr. Instructions: The waveform shows the load instructions work correctly.

There are values (4 and 5) at the x"80" and x"81"th address in RAM. We see that these values are loaded into register A and register B.

- KAYDET_A

2 => KAYDET_A,

3 => x"80”,

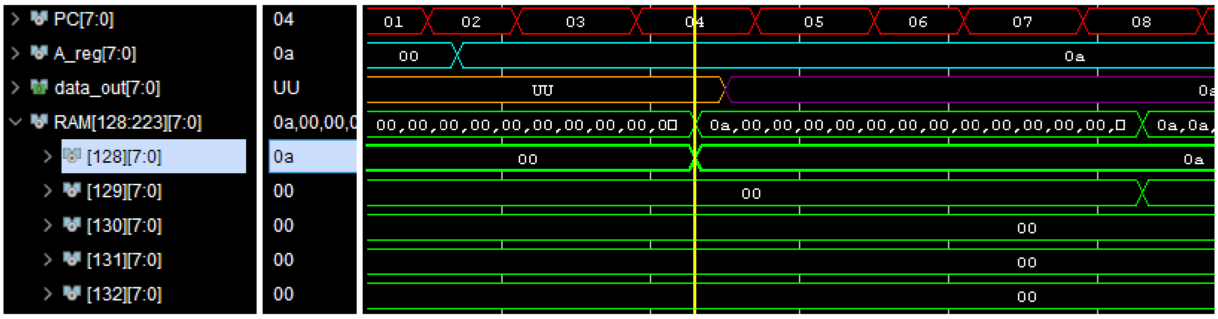


Figure 10 The Waveform Of Store (A) Instruction: The waveform shows the store instruction works correctly.

0x80 is decimal equal to 128. We see that 0x0a value is saved to 128th address in RAM.

- YUKLE_B_SBT

4 => YUKLE_B_SBT,

5 => x"0A",

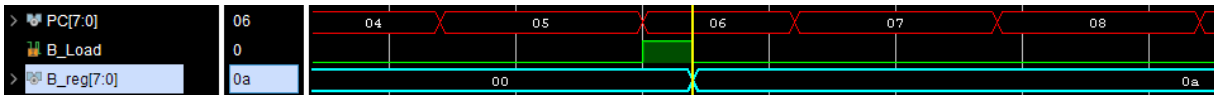


Figure 11 The Waveform Of Load Imm. (B) Instruction: The waveform shows the load instruction works correctly.

After PC value 5, we see that x0A value is loaded into B register.

- KAYDET_B

6 => KAYDET_B,

7 => x"81",

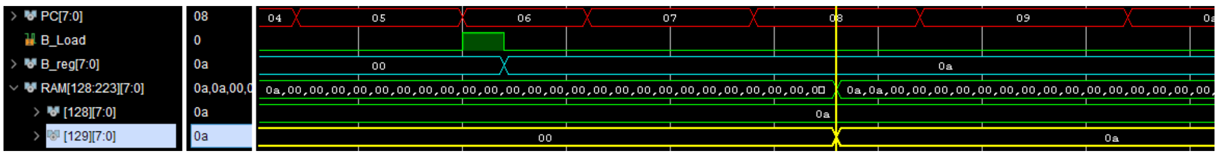

PC[7:0]
B_Load
B_reg[7:0]
RAM[128:223][7:0]
[128][7:0]
[129][7:0]
08
0
0a
0a,0a,00,0
0a
0a
04 05 06 07 08 09
00 0a
0a,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00
0a,0a,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00
0a
00 0a

Figure 12 The Waveform Of Store (B) Instruction: The waveform shows the store instruction works correctly.

0x81 is decimal equal to 129. We see that 0x0a value is saved to 129th address in RAM.

- LOAD_PC_TO_SP1

8 => LOAD_PC_TO_SP1,

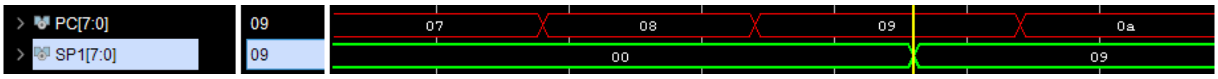


Figure 13 The Waveform Of PC To SP1 Instruction: The waveform shows the stack pointer instruction works correctly.

We see that the new program counter value is saved in SP1.

- LOAD_SP1_IMM

9 => LOAD_SP1_IMM,

10 => x"00",

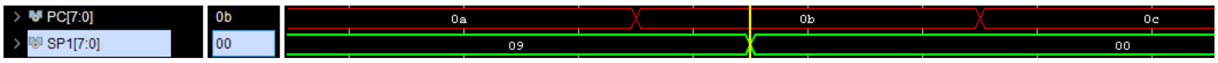


Figure 14 The Waveform Of SP1 Imm. Instruction: The waveform shows the stack pointer instruction works correctly.

We see that the immediate value entered in SP1 is loaded after the PC value is 10.

- LOAD_SP1_TO_PC

11 => LOAD_SP1_TO_PC,

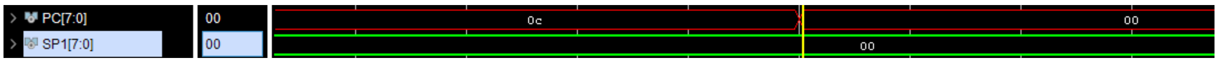


Figure 15 The Waveform Of SP1 To PC Instruction: The waveform shows the stack pointer instruction works correctly.

We see that the SP1 value is registered in the PC register.

- LOAD_SP1_DIR

12 => LOAD_SP1_DIR,

13 => x"80",

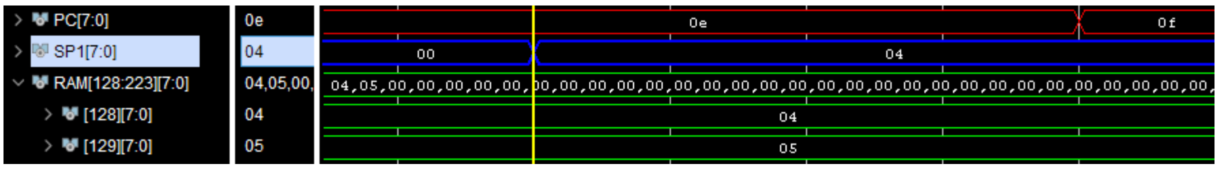


Figure 16 The Waveform Of PC To SP1 Dir. Addr. Instruction: The waveform shows the stack pointer instruction works correctly.

There is 4 at the x"80"th address in RAM. We see that this value is loaded into the SP1 register.

- JUMP_AND_LINK_SP1

12 => JUMP_AND_LINK_SP1,

13 => x"02",

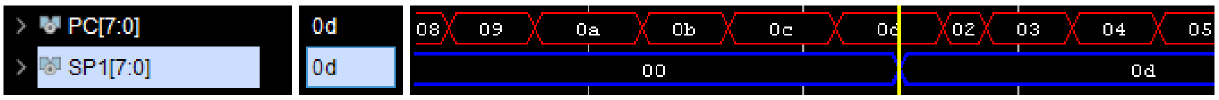


Figure 17 The Waveform Of Jump Instruction: The waveform shows the jump instruction works correctly.

We can see that Current PC value (0x0d) is saved in SP1. We see that the PC value has been updated to 0x02.

- CIKAR_AB – TOPLA_AB – AND_AB – OR_AB - ARTTIR_A - ARTTIR_B – DUSUR_A – DUSUR_B

After this part, the instructions to perform the operations in the ALU will be verify. It should be noted that register B is directly connected to input A of ALU and register A is connected to input B of ALU. Statements A and B in the instructions represent the inputs of the ALU.

0 => YUKLE_A_SBT,

1 => x"03",

2 => YUKLE_B_SBT,

3 => x"05",

4 => CIKAR_AB,

5 => TOPLA_AB,

6 => AND_AB,

7 => OR_AB,

8 => ARTTIR_A,

9 => ARTTIR_B,

10 => DUSUR_A,

11 => DUSUR_B,

others => x"00"

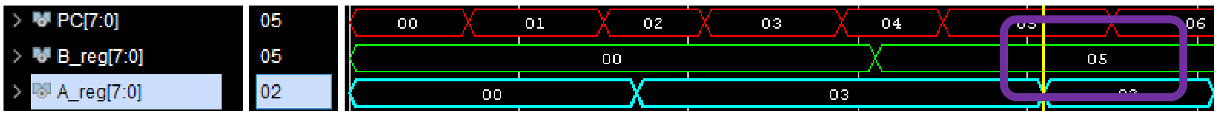


Figure 18 The Waveform Of Subtraction Instruction: The waveform shows the arithmetic instruction works correctly.

In the first stage, we see that the value of x03 is loaded into register A and the value of x05 is loaded into register B. Then we see that x03 is subtracted from x05. We successfully completed the verification of the CIKAR_A instruction by writing the result back to the A register.

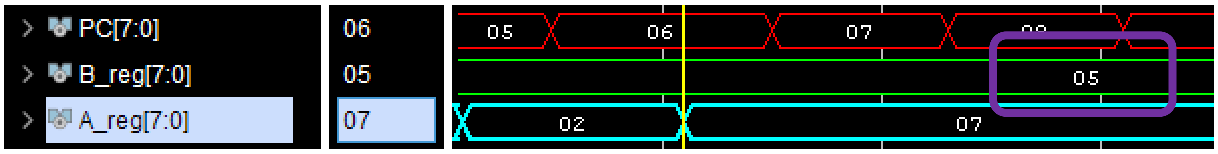


Figure 19 The Waveform Of Summation Instruction: The waveform shows the arithmetic instruction works correctly.

With the TOPLA_AB command, the values in registers A and B were added and written into register A. When the value in register A is x02 and the value in register B is value x05, we see that the result of x07 is written to register A and the operation is completed.

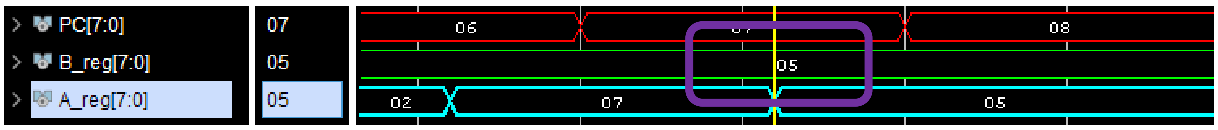


Figure 20 The Waveform Of And Instruction: The waveform shows the arithmetic instruction works correctly.

When the PC value was 6 AND_AB was fetched. After entering the value in register A as x07 and the value in register B in x05 and operation, we see that the result of x05 is rewritten to register A.

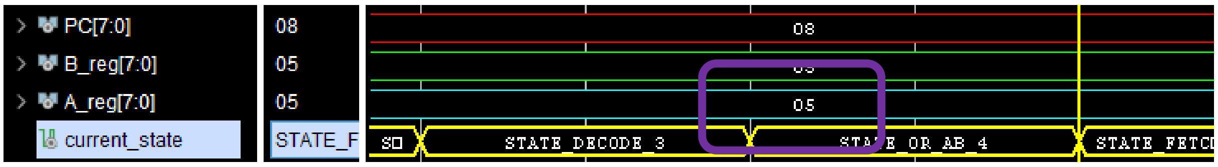


Figure 21 The Waveform Of Or Instruction: The waveform shows the arithmetic instruction works correctly.

With the OR_AB instruction, the x05 value in the A register and the x05 value in the B register enter the or operation and the result is written to the A register as x05 and the verified operation is completed.

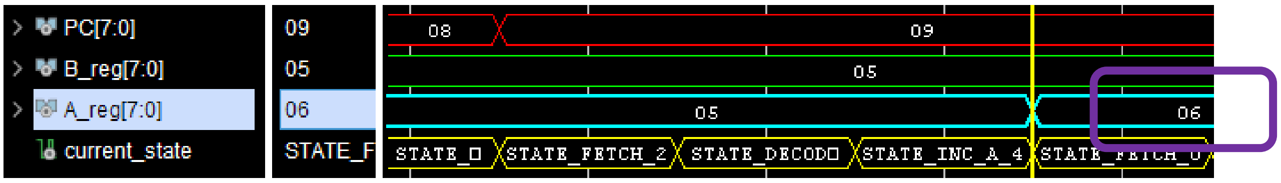


Figure 22 The Waveform Of Inc. (A) Instruction: The waveform shows the arithmetic instruction works correctly.

The ARTTIR_A instruction increments the value of register B by one and writes the result to register A. Since the value of register B is x05, the value of x06 is written to register A as a result.

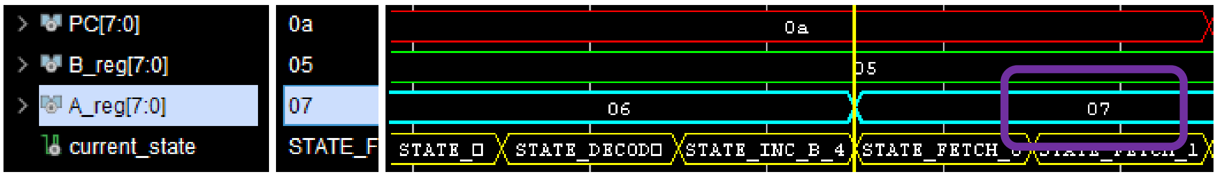


Figure 23 The Waveform Of Inc. (B) Instruction: The waveform shows the arithmetic instruction works correctly.

The ARTTIR_B instruction increments the value of register A by one and writes the result to register A. Since the value of register A is x06, the value of x07 is written to register A as a result.

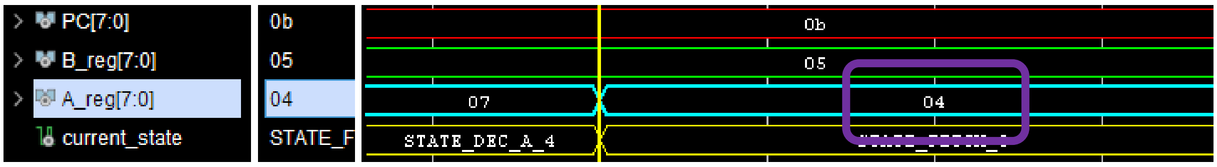


Figure 24 The Waveform Of Dec. (A) Instruction: The waveform shows the arithmetic instruction works correctly.

The DUSUR_A instruction decrements the value of register B by one and writes the result to register A. Since the value of register B is x05, the value of x04 is written to register A as a result.

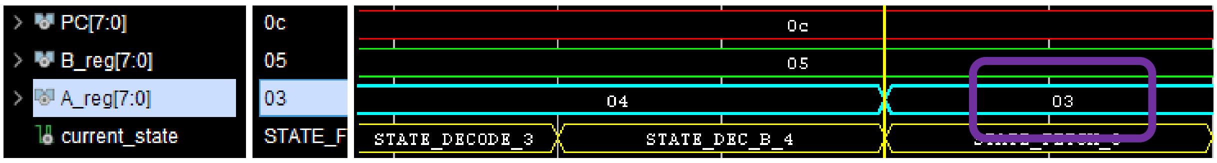


Figure 25 The Waveform Of Dec.(B) Instruction: The waveform shows the arithmetic instruction works correctly.

The DUSUR_B instruction decrements the value of register A by one and writes the result to register A. Since the value of register A is x04, the value of x03 is written to register A as a result.

- SRL_A – SLL_A

0 => YUKLE_A_SBT,

1 => x"01",

2 => YUKLE_B_SBT,

3 => x"02",

4 => SRL_A,

5 => SLL_A,

The statement A in the instructions represents the input A at the input of the ALU. Register B is connected to input A of ALU.

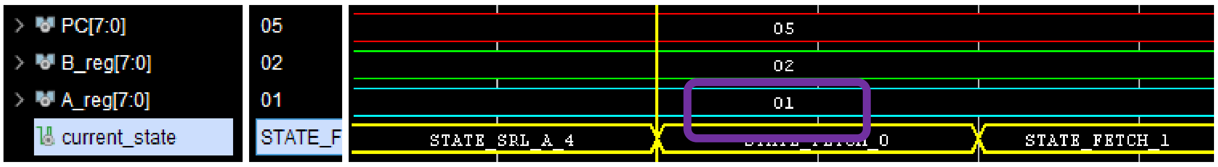


Figure 26 The Waveform Of Shift Right Logical Instruction: The waveform shows the shifting instruction works correctly.

When the value of x02 in register B is shifted to the right by the value of x01 in register A, it becomes x01 and this value is loaded back into register A.

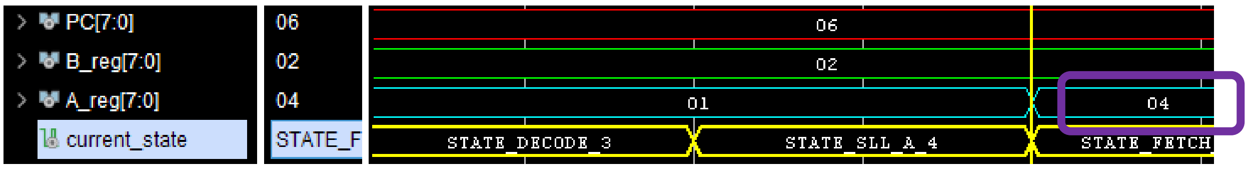


Figure 27 The Waveform Of Shift Left Logical Instruction: The waveform shows the shifting instruction works correctly.

When the value of x02 in register B is shifted to the left by the value of x01 in register A, it becomes x04 and this value is loaded back into register A.

- ROR_A – ROL_B

4 => ROR_A,

5 => ROL_A,

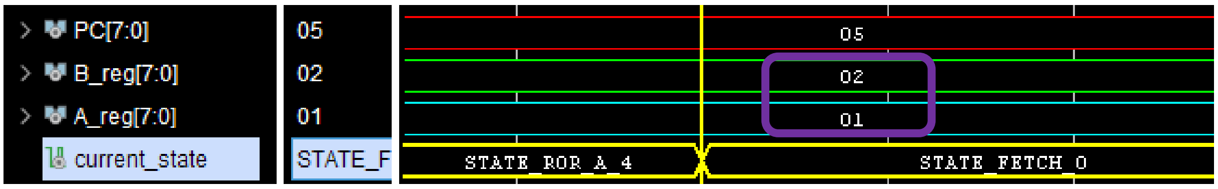


Figure 28 The Waveform Of Rotation To Right Instruction: The waveform shows the rotation instruction works correctly.

When the x02 value of register B is rotated to the right, the result will be x01. We see that the result has been successfully loaded into register A.

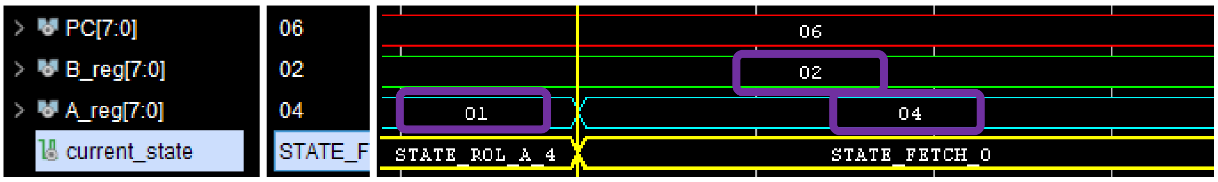


Figure 29 The Waveform Of Rotation To Left Instruction: The waveform shows the rotation instruction works correctly.

When the x02 value of register B is rotated to the left, the result will be x04. We see that the result has been successfully loaded into register A.

- XOR_AB – NOT_A – RESET_ALU

0 => YUKLE_A_SBT,

1 => x"02",

2 => YUKLE_B_SBT,

3 => x"03",

4 => XOR_AB,

5 => NOT_A,

6 => RESET_ALU,

others => x”00”

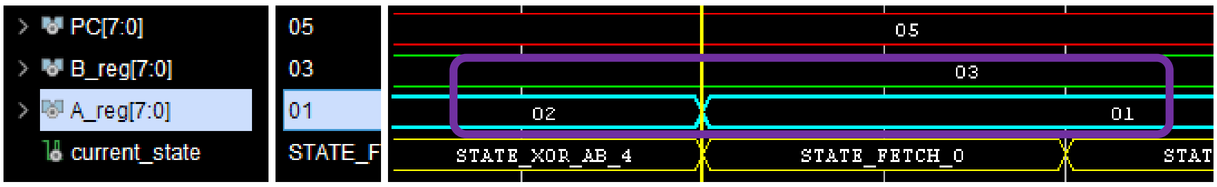


Figure 30 The Waveform Of XOR Instruction: The waveform shows the arithmetic instruction works correctly.

With the XOR_AB operation, the "x02 xor x03" operation was performed and the result of x01 was loaded into register A.

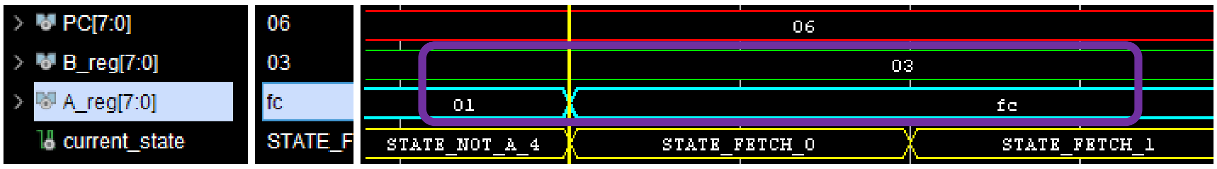


Figure 31 The Waveform Of NOT Instruction: The waveform shows the arithmetic instruction works correctly.

With NOT_A, the value of x03 in register B is logically reversed and the value of xfc is loaded into register A.

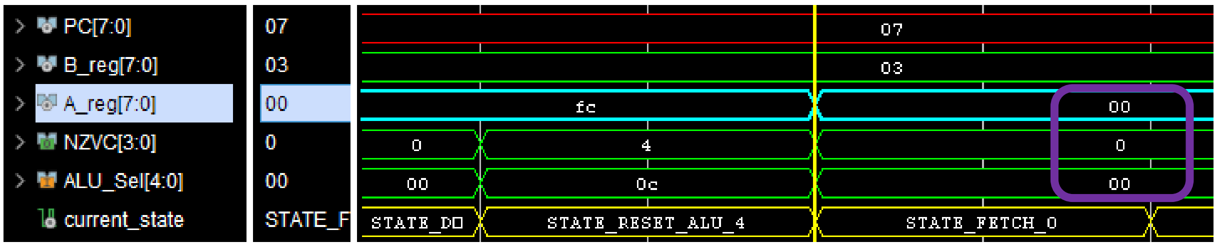


Figure 32 The Waveform Of ALU Reset Instruction: The waveform shows the arithmetic instruction works correctly.

The RESET_ALU instruction resets the ALU signals and output as seen in the image above. At the same time, since the ALU result is conventionally written to the A register, the value of the A register is also reset.

- ATLA - ATLA_POZITIFSE - ATLA_NEGATIFSE - NOP

```
0 => YUKLE_A_SBT,
1 => x"04",
2 => YUKLE_B_SBT,
3 => x"02",
4 => CIKAR_AB,
5 => ATLA_POZITIFSE,         --> don't branch
6 => x"00",
7 => ATLA_NEGATIFSE,         --> branch pc will be x0a
8 => x"0a",                  --> in decimal 10
9 => NOP,
10=> NOP,
11=> ATLA,                   --> branch pc will be x0e
12=> x"0e",                  --> in decimal 14
13=> NOP,
14=> NOP,
15=> NOP,
16=> YUKLE_A_SBT,
17=> x"01",
18=> YUKLE_B_SBT,
19=> x"01",
20=> TOPLA_AB,
21=> ATLA_NEGATIFSE,         --> don't branch
22=> x"00",
23=> ATLA_POZITIFSE,         --> branch pc will be x1a
24=> x"1a",                  --> 26 in decimal
25=> NOP,
26=> NOP,
```

27=> ATLA,   -> branch pc will be x00

28=> x"00",

29=> others => x"00"

First, I loaded x04 and x02 values into registers A and B. When x04 is subtracted from x02, the decimal number -2 (negatively) is obtained.

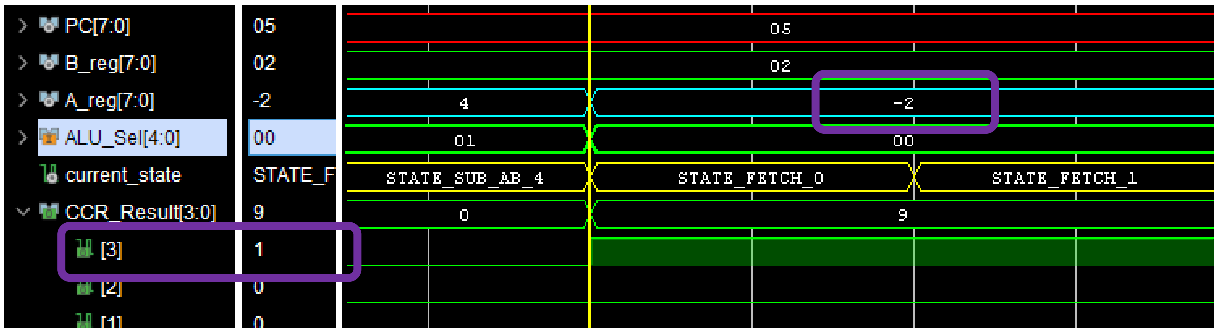


Figure 33 The Waveform Of The State Of CCR: The waveform shows that negative value is detected.

The last bit of the CCR was logic 1, as a negative value was obtained at the end of the operation.

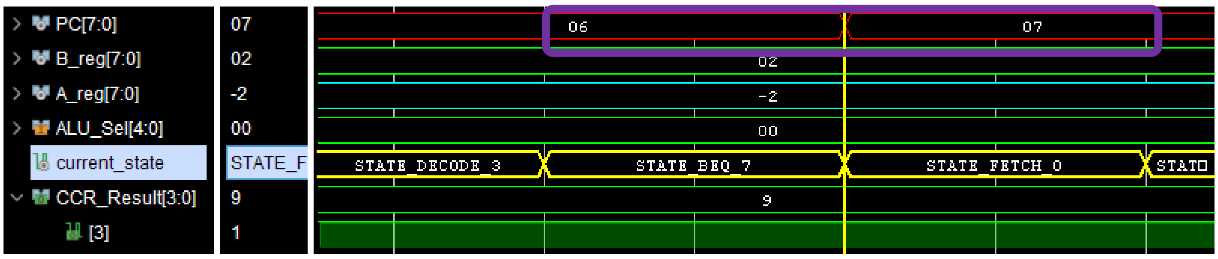


Figure 34 The Waveform Of ATLA_POZITIFSE Instruction: The waveform shows the branch instruction works correctly.

When the PC value comes to the ATLA_POZITIFSE instruction, the branch operation did not occur because the result was negative, and the PC value went directly from x06 to x07 (PC value was not x00).

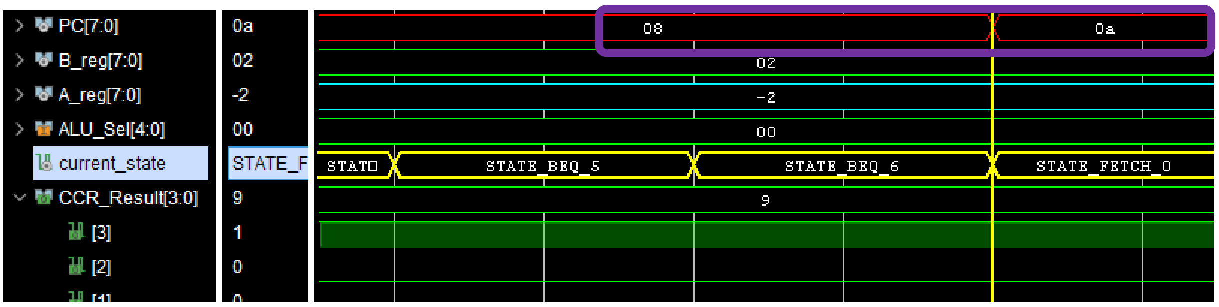


Figure 35 The Waveform Of ATLA_NEGATIFSE Instruction: The waveform shows the branch instruction works correctly.

Because the result was negative, the PC value changed from x08 to x0a (10 in decimal) thanks to the ATLA_NEGATIFSE instruction.

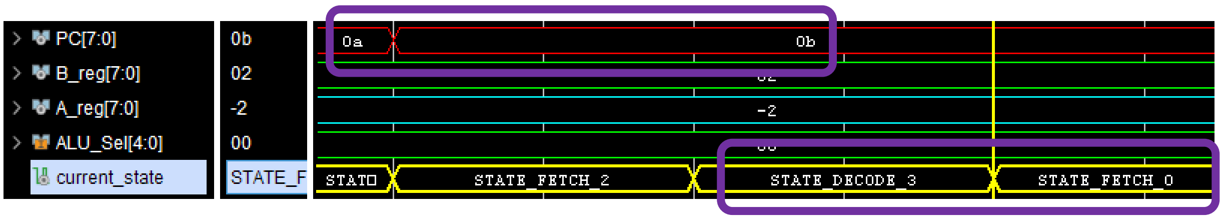


Figure 36 The Waveform Of NOP Instruction: The waveform shows the NOP instruction works correctly.

When the PC value is x0a, it points to the NOP instruction. The NOP instruction makes the processor wait for the fetch and decode time. No execution steps occur. As seen in the picture above, it is seen that after the decode step, the next instruction is passed to the fetch step.

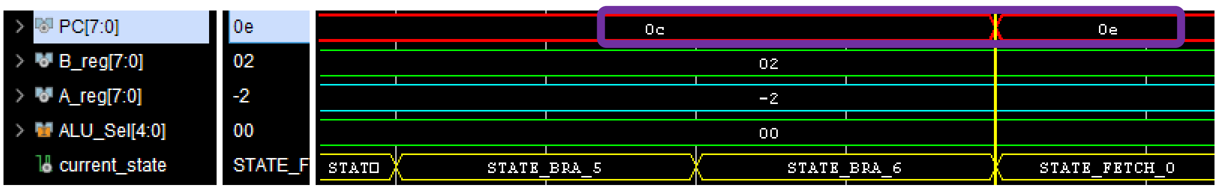


Figure 37 The Waveform Of ATLA Instruction: The waveform shows the branch instruction works correctly.

When the PC value reaches x0b, it points to the ATLA instruction. Then the value of PC increases by one (x0c) and it is understood which address (x0e) to branch to. The PC value is branched from x0c to x0e by the ATLA instruction as seen in the picture above.

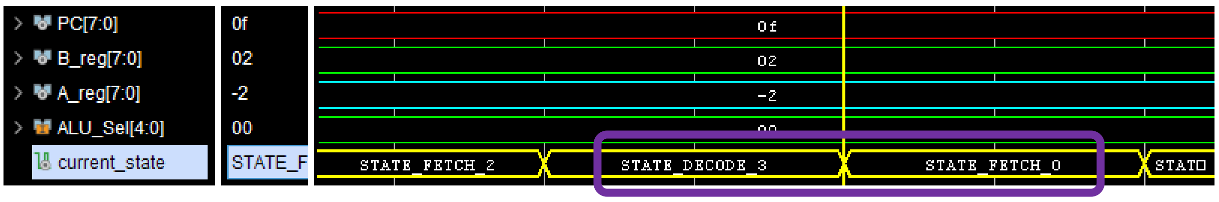


Figure 38 The Waveform Of NOP Instruction: The waveform shows the NOP instruction works correctly.

Afterwards, we see above that the NOP instruction is executed.

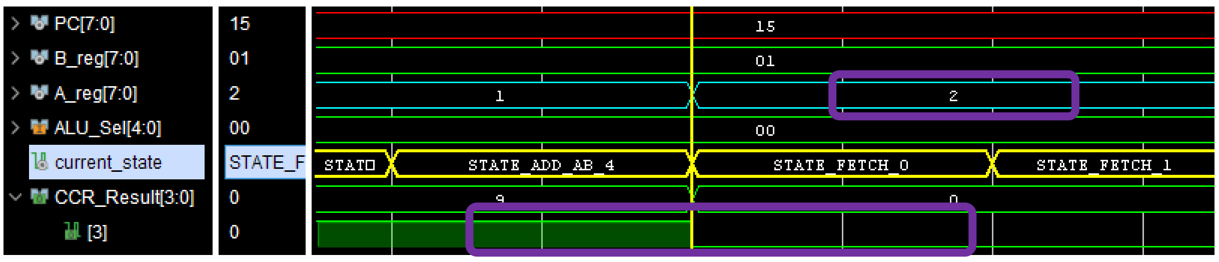


Figure 39 The Waveform Of A&B Register Loaded: The waveform shows that A&B registers are updated again.

Afterwards, new values were loaded into the A and B registers again. The value x01 is loaded into registers A and B and summed. The result was positive.

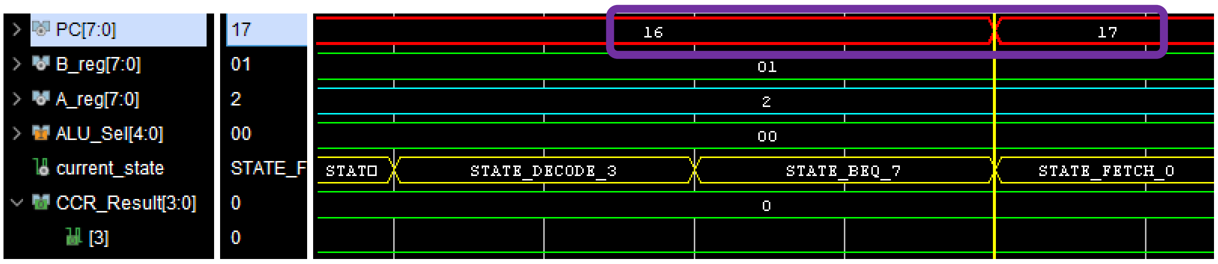


Figure 40 The Waveform Of ATLA_NEGATIFSE Instruction: The waveform shows the branch instruction works correctly.

When the PC value points to the ATLA_NEGATIFSE instruction, the branch operation did not occur because the result was positive.

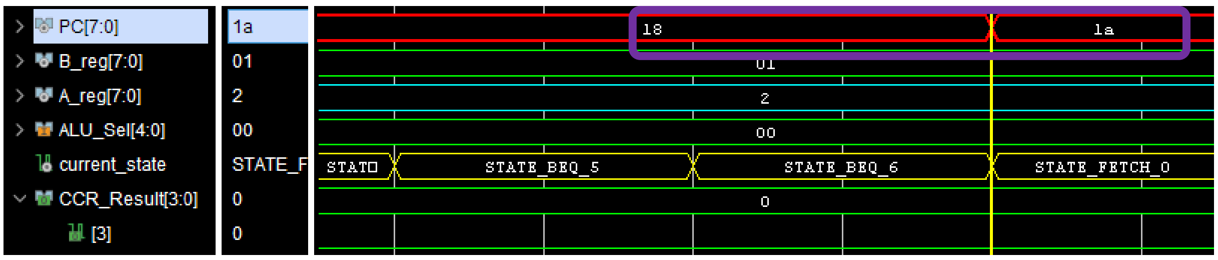


Figure 41 The Waveform Of ATLA_POZITIFSE Instruction: The waveform shows the branch instruction works correctly.

The PC value is branched from x18 to x1a because the result is positive when the PC value points to the ATLA_POZITIFSE instruction.

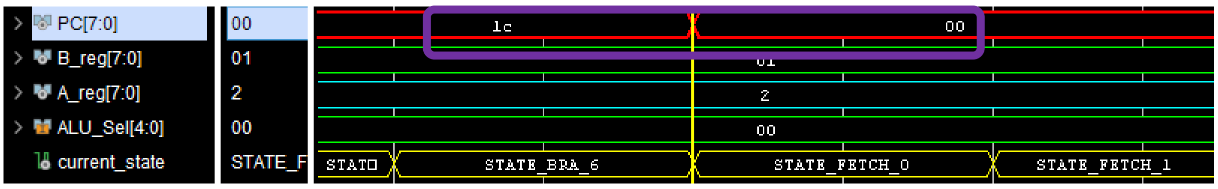


Figure 42 The Waveform Of ATLA Instruction: The waveform shows the branch instruction works correctly.

After the PC value becomes x1c (in 28 decimals), the PC value takes the value x00 indicated by the ATLA instruction.

- ATLA_OVERFLOW_YOKSA - ATLA_OVERFLOW_VARSA - ATLA_ELDE_YOKSA - ATLA_ELDE_VARSA - ATLA_ESITSE_SIFIR - ATLA_DEGILSE_SIFIR

0 => YUKLE_A_SBT,

1 => x"80",

```
2 => YUKLE_B_SBT,
3 => x"80",
4 => TOPLA_AB,
5 => ATLA_OVERFLOW_YOKSA,      --> don't branch
6 => x"00",
7 => ATLA_OVERFLOW_VARSA,      --> branch
8 => x"0a",
9 => NOP,
10=> ATLA_ELDE_YOKSA,          --> don't branch
11=> x"00",
12=> ATLA_ELDE_VARSA,          --> branch
13=> x"0F",
14=> NOP,
15=> ATLA_ESITSE_SIFIR,        --> don't branch
16=> x"02",
17=> ATLA_DEGILSE_SIFIR,
18=> x"00",
others => x"00"
```

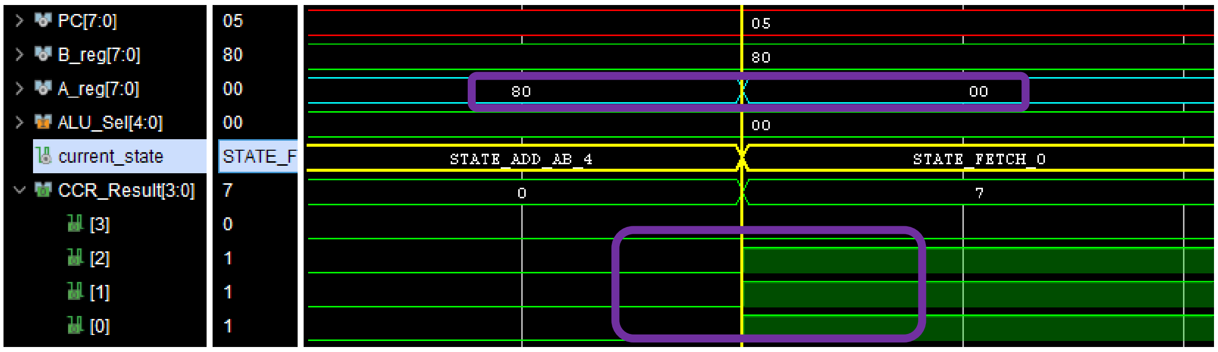


Figure 43 The Waveform After Summation: The waveform shows that carry flag is set after the execution of summation.

After adding the x80 values in the A and B registers, both the overflow and the carry situation occurred. We see that the 1st bit of CCR, which represents the overflow state, and the 0th bit, which represents the cary state, are logic high.

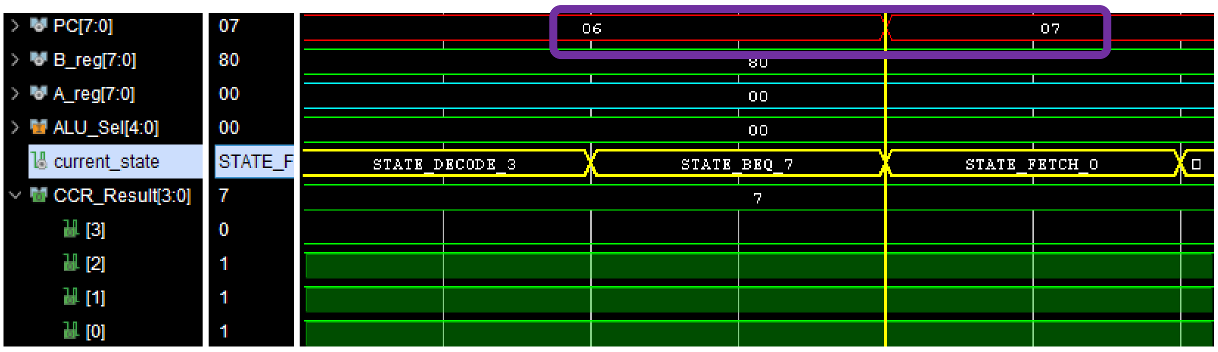


Figure 44 The Waveform Of ATLA_OVERFLOW_YOKSA Instruction: The waveform shows the branch instruction works correctly.

When the PC value points to the ATLA_OVERFLOW_YOKSA instruction, the branch operation did not occur due to the overflow condition. PC value became x07 after x06.

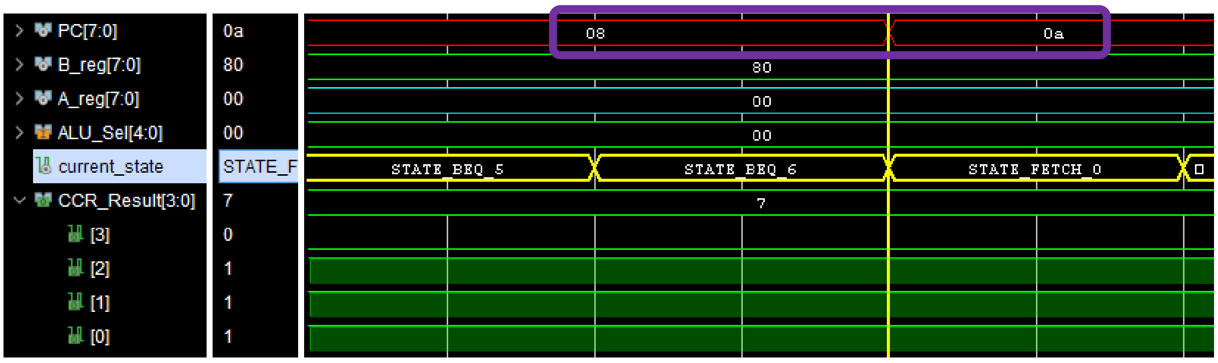


Figure 45 The Waveform Of ATLA_OVERFLOW_VARSA Instruction: The waveform shows the branch instruction works correctly.

When the PC value points to the ATLA_OVERFLOW_VARSA instruction, the branch operation occurred due to the overflow condition. PC value became x0a after x08.

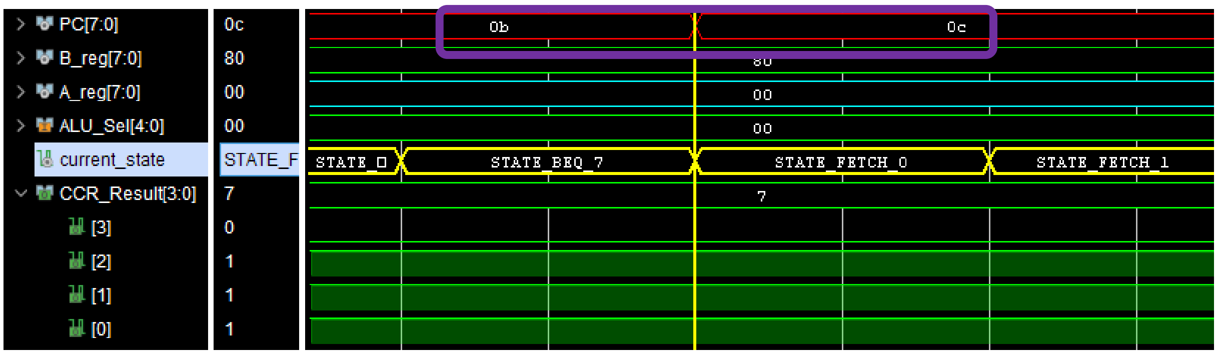


Figure 46 The Waveform Of ATLA_ELDE_YOKSA Instruction: The waveform shows the branch instruction works correctly.

When the PC value points to the ATLA_ELDE_YOKSA instruction, the branch operation did not occur due to the cary state. PC value became x0c after x0b.

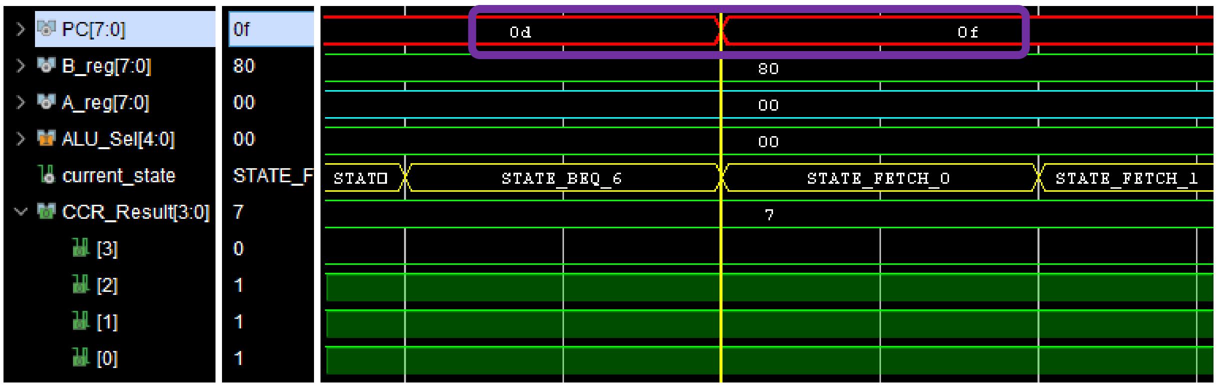


Figure 47 The Waveform Of ATLA_ELDE_VARSA Instruction: The waveform shows the branch instruction works correctly.

When the PC value points to the ATLA_ELDE_VARSA instruction, the branch operation occurred due to the cary state. PC value became x0f after x0d.

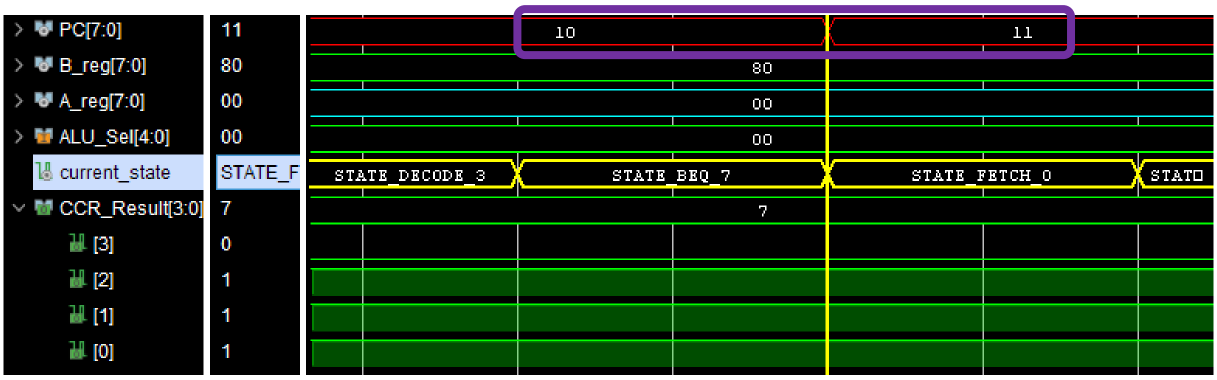


Figure 48 The Waveform Of ATLA_DEGILSE_SIFIR Instruction: The waveform shows the branch instruction works correctly.

The second bit of the CCR represents the zero state. As seen above, the zero state is active. When the PC value points to the ATLA_DEGILSE_SIFIR instruction, the branch operation is not performed, so this branch instruction is verifed.

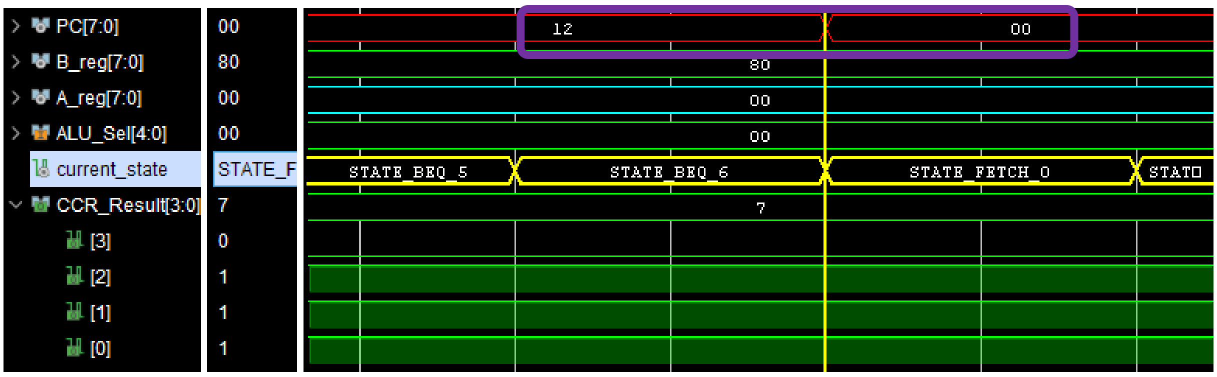


Figure 49 The Waveform Of ATLA_ESITSE_SIFIR Instruction: The waveform shows the branch instruction works correctly.

It should be noted that the PC value in the above simulation graph is hexadecimal. When the PC value shows the ATLA_ESITSE_SIFIR instruction, the branch operation was performed because the zero state was active, and the PC value was branched to x00.

3- Interrupt Controller Unit

controller unit (ICU) is the main interrupt unit that forms the interrupt interface of all peripherals designed to support the interrupt feature. When there is any data flow/data change in the peripheral units depending on user configurations, this unit transmits data to the ICU with related signals, analyzes the incoming data, updates the registers that inform the user, and most importantly, controls the data flow for the interrupt status of the CPU providing their duties.

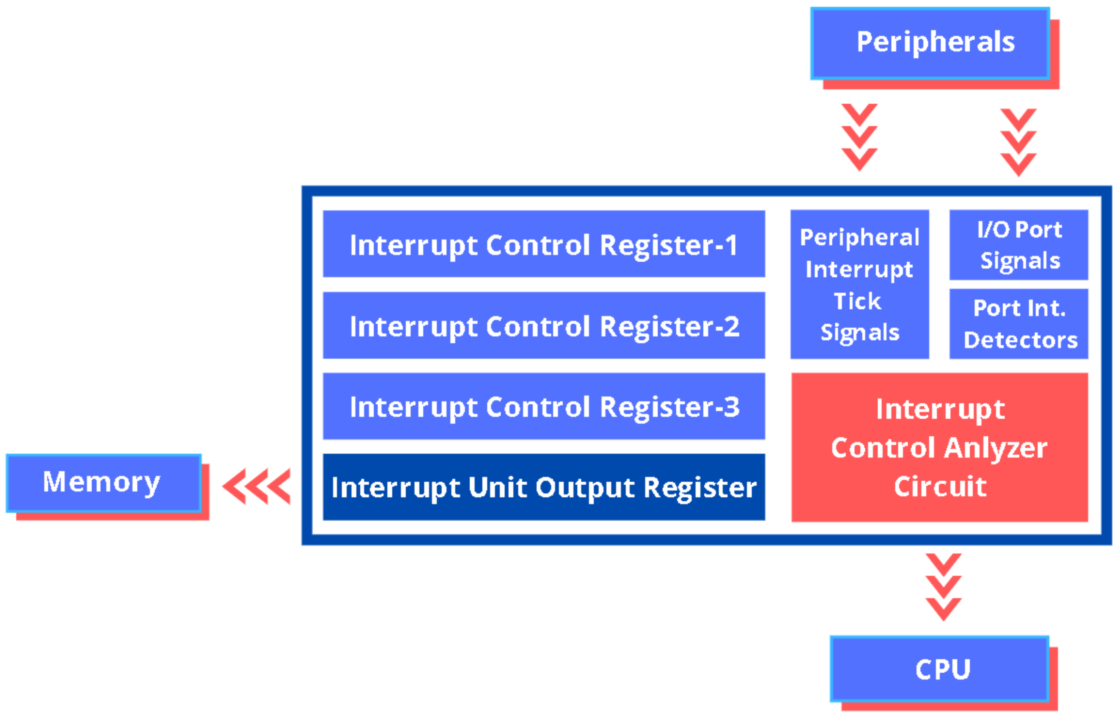

Peripherals
Interrupt Control Register-1
Interrupt Control Register-2
Interrupt Control Register-3
Interrupt Unit Output Register
Peripheral Interrupt Tick Signals
I/O Port Signals
Port Int. Detectors
Interrupt Control Anlyzer Circuit
Memory
CPU

Figure 50 ICU Schematic: The schematic shows ICU design and all unit hierarchical designed in ICU.

There are 4 registers designed as a user interface in the Interrupt controller unit. It has 3 registers designed for short-term recording of input signals from peripheral units. There are two port detectors that detect the changes in the I/O ports.

3.1- Interrupt Controller Unit Registers

3.1.1- Interrupt Control Register-1 (ICx_CR1)

Address: xED

7	6	5	4	3	2	1	0
	reserved		Opt. Per. Int. Enable	TIMER2 Int. Enable	TIMER1 Int. Enable	I/O Port-B Int. Enable	I/O Port-A Int. Enable

Table 2 Interrupt Control Register-1 (ICx_CR1)

Bit 7:5 Reserved, must be kept at reset value.

Bit 4 Optional Peripheral Interrupt Enable

The bit controls the activation of any peripherals that will be added to the design.

0: Deactive optional peripheral interrupt

1: Active optional peripheral interrupt

Bit 3 TIMER2 Interrupt Enable

0: Deactive TIMER2 Interrupt

1: Active TIMER2 Interrupt

Bit 2 TIMER1 Interrupt Enable

0: Deactive TIMER1 Interrupt

1: Active TIMER1 Interrupt

Bit 1 I/O Port-B Interrupt Enable

0: Deactive Port-B Interrupt

1: Active Port-B Interrupt

Bit 0 I/O Port-A Interrupt Enable

0: Deactive Port-A Interrupt

1: Active Port-A Interrupt

3.1.2- Interrupt Control Register-2 (ICx_CR2)

Address: xEE

It is the register that provides interrupt configuration of I/O Port-A pins.

7	6	5	4	3	2	1	0
Pin7 Int. Enable	Pin6 Int. Enable	Pin5 Int. Enable	Pin4 Int. Enable	Pin3 Int. Enable	Pin2 Int. Enable	Pin1 Int. Enable	Pin0 Int. Enable

Table 3 Interrupt Control Register-2 (ICx_CR2)

Bit 7 Pin7 Interrupt Enable

0: Deactive Port-A Pin7 Interrupt

1: Active Port-A Pin7 Interrupt

Bit 6:0	Pins [6:0] Interrupt Enable
	Same as Bit 7.

3.1.3- Interrupt Control Register-3 (ICx_CR3)

Address: xEF

It is the register that provides interrupt configuration of I/O Port-B pins.

7	6	5	4	3	2	1	0
Pin7 Int. Enable	Pin6 Int. Enable	Pin5 Int. Enable	Pin4 Int. Enable	Pin3 Int. Enable	Pin2 Int. Enable	Pin1 Int. Enable	Pin0 Int. Enable

Table 4 Interrupt Control Register-3 (ICx_CR3)

Bit 7	Pin7 Interrupt Enable
	0: Deactive Port-B Pin7 Interrupt
	1: Active Port-B Pin7 Interrupt
Bit 6:0	Pins [6:0] Interrupt Enable
	Same as Bit 7.

3.1.4- Interrupt Unit Output Register (IOR)

Address: xFB

It is the read-only type output register of the interrupt controller unit, which shows the user which interrupts are active on the peripheral unit that has entered/requested the interrupt routine.

7	6	5	4	3	2	1	0
Interrupt Flag Bits Of I/O Port Pins			Optional Per. Int. Flag	TIM2 Int. Flag	TIM1 Int. Flag	I/O Port-B Int. Flag	I/O Port-A Int. Flag

Table 5 Interrupt Unit Output Register (IOR)

Bit 7:5	Interrupt Flag Bits Of I/O Port Pins

	000: Interrupt flag is enabled comes from Pin0
	001: Interrupt flag is enabled comes from Pin1
	010: Interrupt flag is enabled comes from Pin2
	011: Interrupt flag is enabled comes from Pin3
	100: Interrupt flag is enabled comes from Pin4
	101: Interrupt flag is enabled comes from Pin5
	110: Interrupt flag is enabled comes from Pin6
	111: Interrupt flag is enabled comes from Pin7
Bit 4	Optional Peripheral Interrupt Flag
	0: Optional peripheral interrupt flag is deactivated
	1: Optional peripheral interrupt flag is activated
Bit 3	TIMER2 Interrupt Flag
	0: Optional peripheral interrupt flag is deactivated
	1: Optional peripheral interrupt flag is activated
Bit 2	TIMER1 Interrupt Flag
	0: Optional peripheral interrupt flag is deactivated
	1: Optional peripheral interrupt flag is activated
Bit 1	I/O Port-A Interrupt Flag
	0: Optional peripheral interrupt flag is deactivated
	1: Optional peripheral interrupt flag is activated
Bit 0	I/O Port-B Peripheral Interrupt Flag
	0: Optional peripheral interrupt flag is deactivated
	1: Optional peripheral interrupt flag is activated

3.2- Interrupt Controller Unit Verification

Verification processes were carried out by activating all peripheral units and sending signals to interrupt triggering inputs at certain periods.

```
IC_Reset_i                  <= '1';

wait for 100ns;

IC_Reset_i                  <= '0';

wait for 100ns;


interrupt_control_reg1_i    <= "00011111"; -- all interrupts of the peripherals are enable

interrupt_control_reg2_i    <= "11111111"; -- all interrupts of port-a pins are enable

interrupt_control_reg3_i    <= "11111111"; -- all interrupts of port-b pins are enable

IC_PortA_Signal_i           <= "00000000"; -- all inputs of port-a are zero

IC_PortB_Signal_i           <= "00000000"; -- all inputs of port-b are zero
```

Yukarıda görüldüğü gibi ICU’nin ilgili register’ları, tüm peripheraller’in interrupt özelliği aktif edilmesi için set edildi. Port-A ve Port-B birimlerinin inputları 0 yapıldı.

- TIMER1 Interrupt Test

```
-- timer1 interrupt

IC_Signal_i             <= "00000100";

wait for 40ns;

IC_Signal_i             <= "00000000";

wait for 120ns;


-- Reset

IC_Reset_i              <= '1';

wait for 80ns;

IC_Reset_i              <= '0';

wait for 40ns;
```

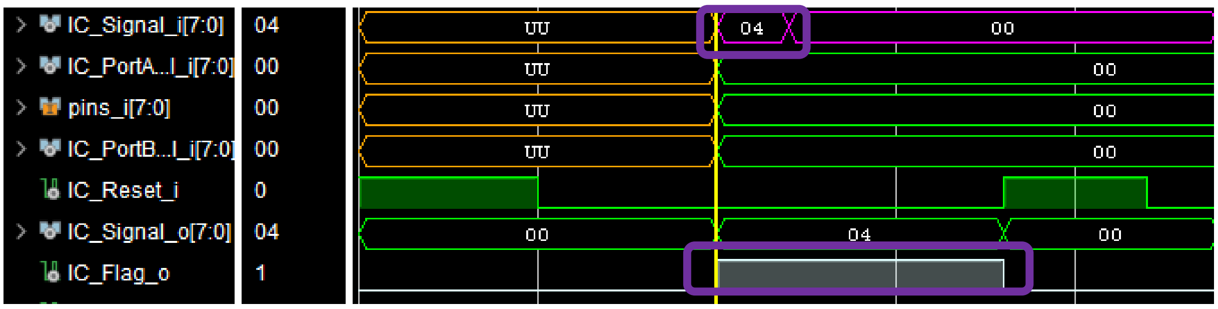


Figure 51 The Waveform Prepared For TIM1 Interrupt: The waveform shows that TIM1 interrupt property works properly.

As seen above, the tick signal from TIMER1 triggered the interrupt unit. As a result, the interrupt flag to the CPU has been set and the ICU has worked correctly for this unit.

- TIMER2 Interrupt Test

```
-- timer2 interrupt

IC_Signal_i                 <= "00001000";

wait for 40ns;

IC_Signal_i                 <= "00000000";

wait for 120ns;


-- Reset

IC_Reset_i                  <= '1';

wait for 80ns;

IC_Reset_i                  <= '0';

wait for 40ns;
```

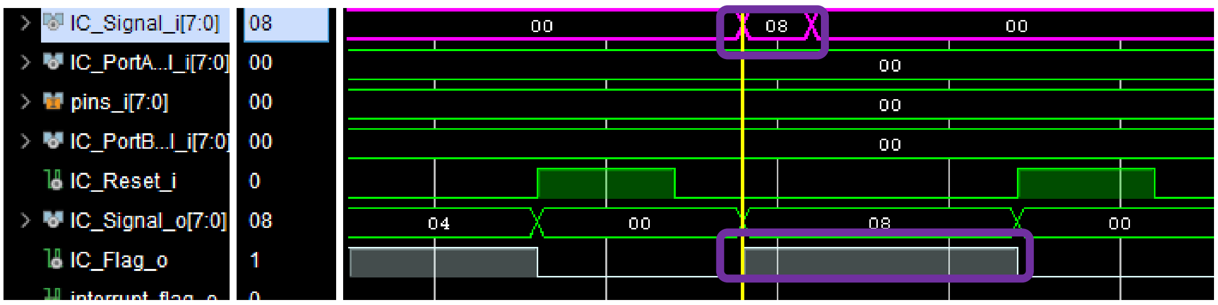


Figure 52 The Waveform Prepared For TIM2 Interrupt: The waveform shows that TIM2 interrupt property works properly.

As seen above, the tick signal from TIMER2 triggered the interrupt unit. As a result, the interrupt flag to the CPU has been set and the ICU has worked correctly for this unit.

- Optional Peripheral Interrupt Test

```
-- optional peripheral interrupt
IC_Signal_i            <= "00010000";
wait for 40ns;
IC_Signal_i            <= "00000000";
IC_Reset_i             <= '0';
wait for 40ns;

-- Reset
IC_Reset_i             <= '1';
wait for 80ns;
IC_Reset_i             <= '0';
wait for 40ns;
```

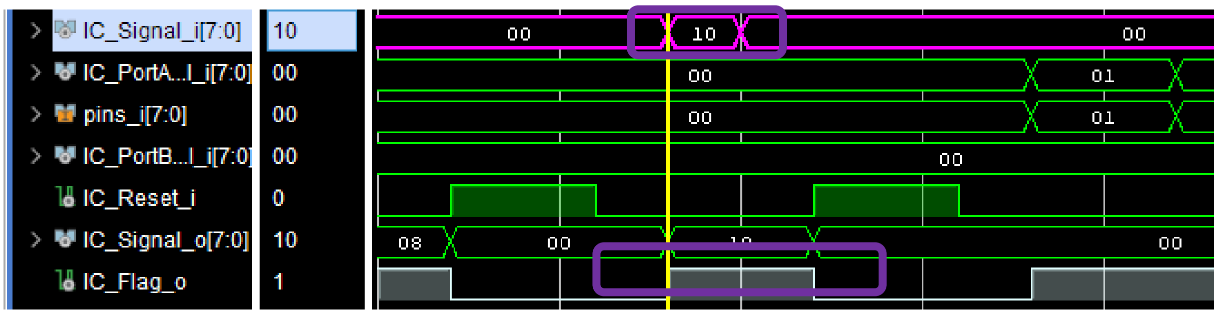


Figure 53 The Waveform Prepared For A Peripheral Interrupt: The waveform shows that optional peripheral interrupt property works properly.

As seen above, the tick signal from a peripheral triggered the interrupt unit. As a result, the interrupt flag to the CPU is set. The ICU for this unit worked correctly.

- I/O Port Interrupt Test

```
-- porta interrupt
IC_PortA_Signal_i      <= "00000001";
wait for 80ns;
IC_PortA_Signal_i      <= "00000000";
wait for 120ns;

-- Reset
```

```
IC_Reset_i              <= '1';

wait for 80ns;

IC_Reset_i              <= '0';

wait for 40ns;


-- portb interrupt

IC_PortB_Signal_i       <= "00000001";

wait for 80ns;

IC_PortB_Signal_i       <= "00000000";

wait for 120ns;
```

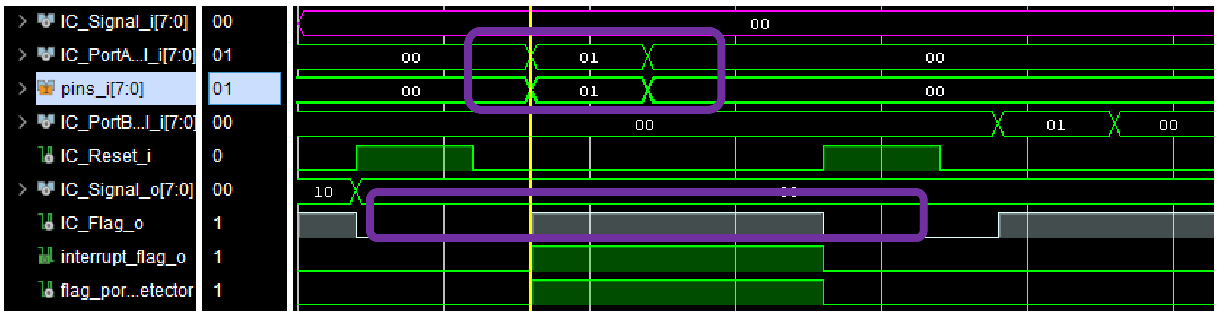


Figure 54 The Waveform Prepared For In. Port-A Interrupt: The waveform shows that Input Port-A interrupt property works properly.

External interrupt was configured for the first input pin of Port-A. Accordingly, when the first pin is set as external, the interrupt flags are set.

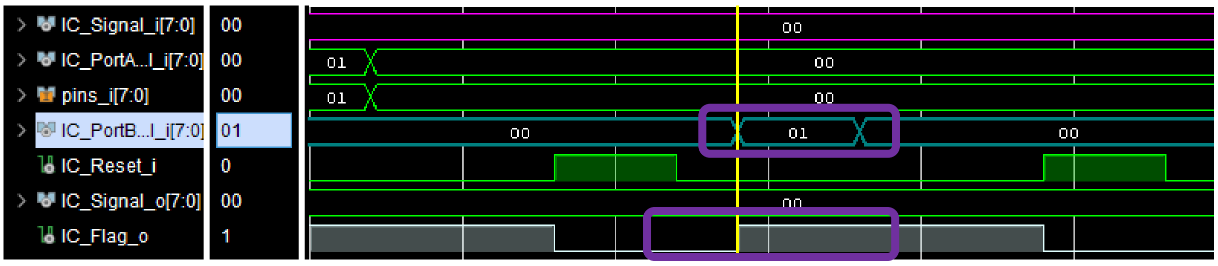


Figure 55 The Waveform Prepared For In. Port-B Interrupt: The waveform shows that Input Port-B interrupt property works properly.

External interrupt was configured for the first input pin of Port-B. Accordingly, when the first pin is set as external, the interrupt flags are set.

4- Timer Module

The designed timer module aims to ensure that it works for two general purposes. The first of these is that it acts as a counter connected to the clock and gives a warning when the set time interval is completed (normal mode), and the second of which is the pulse width modulation feature.

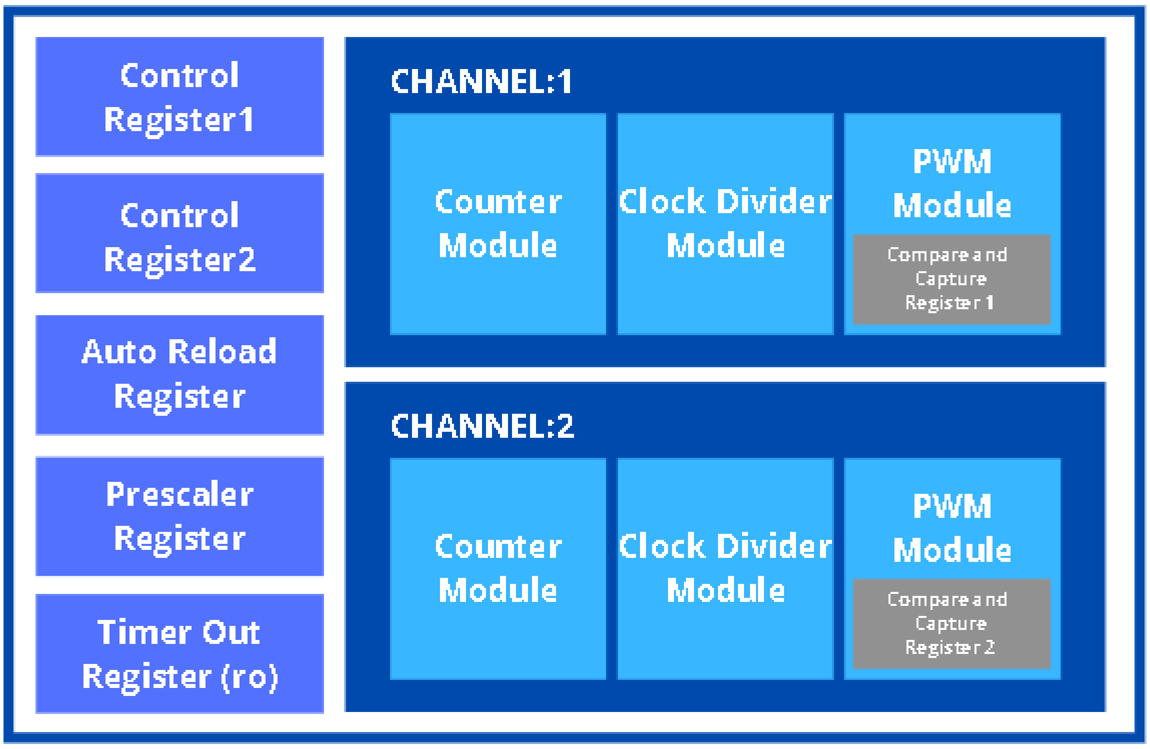


Figure 56 Timer Schematic: The schematic shows Timer design and all unit hierarchical designed in Timer.

As seen above, three separate modules are designed within the timer module. These are the counter, clock divider, and PWM generator modules.

The timer design, which will be integrated into the designed computer system, consists of two channels. It can work in these two channels simultaneously and can work in two different modes.

There are six (6) registers required for the configuration settings in the timer module. However, one register shows the timer status working in read-only mode.

4.1- TIMER Registers

4.1.1- Timer Control Register-1 (TIMx_CR1)

Address: xE2 (TIM1) – xE8 (TIM2)

7	6	5	4	3	2	1	0
	reserved		Counter mode-1	Counter mode-0	CH2 enable	CH1 enable	Timer enable

Table 6 Timer Control Register-1 (TIMx_CR1)

Bit 7:5 Reserved, must be kept at reset value.

Bit 4:3 Counter Modes

00: up (default)

01: down

10: center-aligned

Bit 2 Timer Channel-2 Enable

0: Deactive channel-2 (default)

1:Active channel-2

Bit 1 Timer Channel-1 Enable

0: Deactive channel-1 (default)

1:Active channel-1

Bit 0 Timer Enable

0: Deactive timer (default)

1:Active timer

4.1.2- Timer Control Register-2 (TIMx_CR2)

Address: xE3 (TIM1) – xE9 (TIM2)

7	6	5	4	3	2	1	0
reserved		CH2 PWM channel polarity	CH2 PWM output generation	CH2 mode selection	CH1 PWM channel polarity	CH1 PWM output generation	CH1 mode selection

Table 7 Timer Control Register-2 (TIMx_CR2)

Bit 7:6 Reserved, must be kept at reset value.

Bit 5 CH2 PWM Channel Polarity Selection

0: logic-high during the duty cycle

1: logic-low during the duty cycle

Bit 4	CH2 PWM output generation 0: Deactive (default) 1:Active
Bit 3	CH2 mode selection 0: Counter mode (default) 1: PWM mode
Bit 2	CH1 PWM Channel Polarity Selection 0: logic-high during the duty cycle 1: logic-low during the duty cycle
Bit 1	CH1 PWM output generation 0: Deactive (default) 1:Active
Bit 0	CH1 mode selection 0: Counter mode (default) 1: PWM mode

4.1.3- Prescaler Register

Address: xE4 (TIM1) – xEA (TIM2)

7	6	5	4	3	2	1	0
Prescaler value							

Table 8 Prescaler Register

Bit 7:0	Prescaler value

It is the value that divides the clock frequency entering the timer module. The clock frequency of the timer is equal to fclk_in/ PSC.

4.1.4- Auto Reload Register

Address: xE5 (TIM1) – xEB (TIM2)

7	6	5	4	3	2	1	0
Auto Reload Value							

Table 9 Auto Reload Register (ARR)

Bit 7:0 Auto reload value

It is the value to be loaded in the actual auto-reload register.

4.1.5- CCR1 Register

Address: xE6 (TIM1) – xEC (TIM2)

7	6	5	4	3	2	1	0
Capture/Compare Value							

Table 10 CCR1 Register

Bit 7:0 Capture Compare Value

CC value is the value to be loaded in the actual capture/compare 1 register designed for CH1 (preload value).

4.1.6- CCR2 Register

Address: xE7 (TIM1)

7	6	5	4	3	2	1	0
Capture/Compare Value							

Table 11 CCR2 Register

Bit 7:0 Capture Compare Value

CC value is the value to be loaded in the actual capture/compare 1 register designed for CH2 (preload value).

4.1.7- Timer Output Register

Address: xF9 (TIM1) – xFA (TIM2)

All bits are read-only.

7	6	5	4	3	2	1	0
Reserved						CH2 System Tick Flag	CH1 System Tick Flag

Table 12 Timer Output Register

Bit 7:2	Reserved, must be kept at reset value.
Bit 1	CH2 system tick flag 0: Configured cycle not completed. 1: Configured cycle completed.
Bit 0	CH1 system tick flag 0: Configured cycle not completed. 1: Configured cycle completed.

4.2- Timer Module Verification

- Counter & Counter Mode Test

a) Up Mode Test:

The values assigned to registers in testbench file (in VHDL):

```
control_reg_1        <= "00000111"; -- “00” up (default) mode
control_reg_2        <= "00011011";

prescaler_reg        <= x"64";     -- 100 in decimal
auto_reload_reg      <= x"64";     -- 100 in decimal

ccr_reg_ch1          <= x"0A";     -- 10 in decimal
```

ccr_reg_ch2 <= x"0A"; -- 10 in decimal

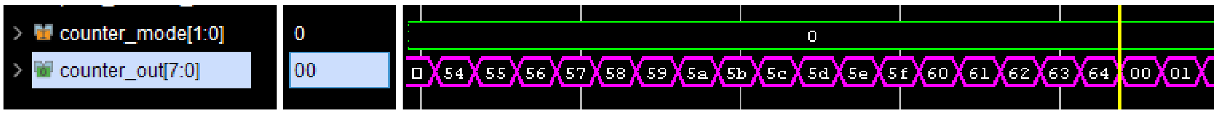


Figure 57 The Waveform Prepared For Counter Up Mode Test: The waveform shows that counting up mode works properly.

When the counter_out signal is followed above, it will be seen that the counter is counting upwards. When the counter value reaches the ARR value (x64), it is reset and the process is complete.

a) Down Mode Test

control_reg_1 <= "00001111"; -- "01" down mode

-- Same register values

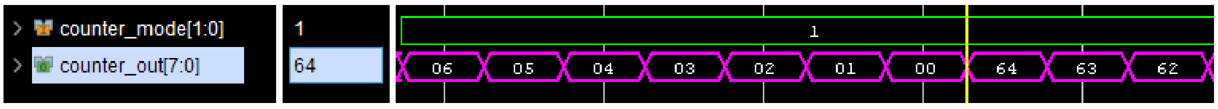


Figure 58 The Waveform Prepared For Down Mode Test: The waveform shows that down mode counting works properly.

When it works in count-down mode, the counter_out signal starts from the ARR value and starts counting down. When the counter_out signal reaches x00, its value becomes x64 again.

b) Center-Aligned Mode

control_reg_1 <= "00010111"; -- "11" center-aligned mode

-- Same register values

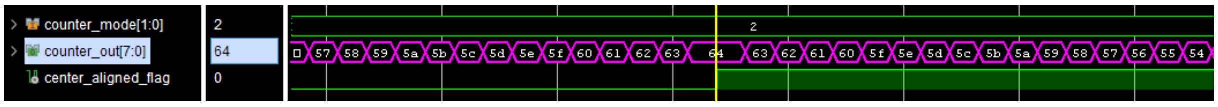

counter_mode[1:0]
counter_out[7:0]
center_aligned_flag
2
64
0
2
0 57 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 64 63 62 61 60 5f 5e 5d 5c 5b 5a 59 58 57 56 55 54

Figure 59 The Waveform Prepared For Center-Aligned Mode: The waveform shows that center-aligned counting mode works properly.

When the timer counter starts to work in this mode, it continues to count from 0 to ARR (x64) in the forward direction. When the counter value reaches ARR (x64), it continues to count down to 0 again.

- Timer Mode Verification

a) Counter Mode/Normal Mode (Default)

The values assigned to registers in testbench file (in VHDL):

```
control_reg_1     <= "00000111";      -- up
control_reg_2     <= "00000000";      -- normal mode for ch1 and ch2


prescaler_reg     <= x"64";           -- 100 in decimal
auto_reload_reg   <= x"64";           -- 100 in decimal


ccr_reg_ch1       <= x"0A";           -- 10 in decimal
ccr_reg_ch2       <= x"0A";           -- 10 in decimal
```

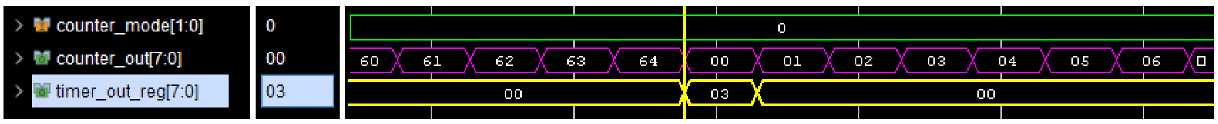


Figure 60 The Waveform Prepared For Normal Mode: The waveform shows that counter mode works properly.

When the specified counter time expires, the timer output register is successfully activated in both channels.

b) PWM Mode (logic-high during duty cycle)

The values assigned to registers in testbench file (in VHDL):

```
control_reg_1     <= "00000111";      -- up
control_reg_2     <= "00011011";      -- pwm mode for ch1 and ch2


prescaler_reg     <= x"64";           -- 100 in decimal
auto_reload_reg   <= x"64";           -- 100 in decimal


ccr_reg_ch1       <= x"0A";           -- 10 in decimal
ccr_reg_ch2       <= x"0A";           -- 10 in decimal
```

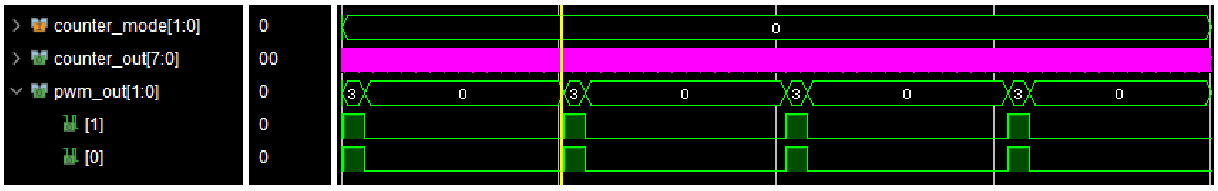


Figure 61 The Waveform Prepared For PWM (High) Mode: The waveform shows that Input logic high pwm mode works properly.

The ARR value is set to 100 while the CCR value is set to 10. Thus, we have produced the PWM wave with a duty cycle rate of 10% at the frequency we want.

c) PWM Mode (logic-low during duty cycle)

The values assigned to registers in testbench file (in VHDL):

```
control_reg_1      <= "00000111";      -- up
control_reg_2      <= "00111111";      -- pwm mode for ch1 and ch2

prescaler_reg      <= x"64";           -- 100 in decimal
auto_reload_reg    <= x"64";           -- 100 in decimal

ccr_reg_ch1        <= x"0A";           -- 10 in decimal
ccr_reg_ch2        <= x"0A";           -- 10 in decimal
```

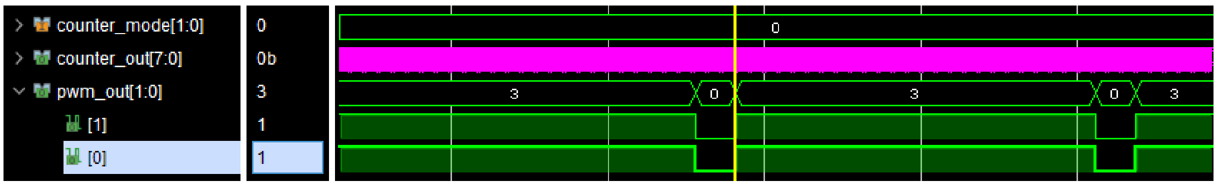


Figure 62 The Waveform Prepared For PWM (Low) Mode: The waveform shows that Input logic low pwm mode works properly.

The ARR value is set to 100 while the CCR value is set to 10. Thus, we have produced the PWM wave with a duty cycle rate of 10% with low polarity at the frequency we want.

5- Elobration-Synthesis-Implementation Stages

Vivado offers us a design image in three stages. The first of these is elaboration design, the other is the design image after the synthesis stage, and finally the design image as a result of place&route. All of these stages have been successfully completed and these design details will be explored in this section.

5.1- Elaboration Of The RTL Design

Elaboration of the RTL design runs RTL linting checks, performs high-level optimizations, infers logic from the RTL and builds design data structures, and optionally applies design constraints. In the default Out-of-Context design flow, we can also have the Vivado include synthesized design checkpoints of IP cores, block designs, DSP modules, or hierarchical blocks into the elaborated design.

Elaborated design of the computer architecture design was produced to examine RTL analysis. After the elaborated design, the computer architecture looks like below:

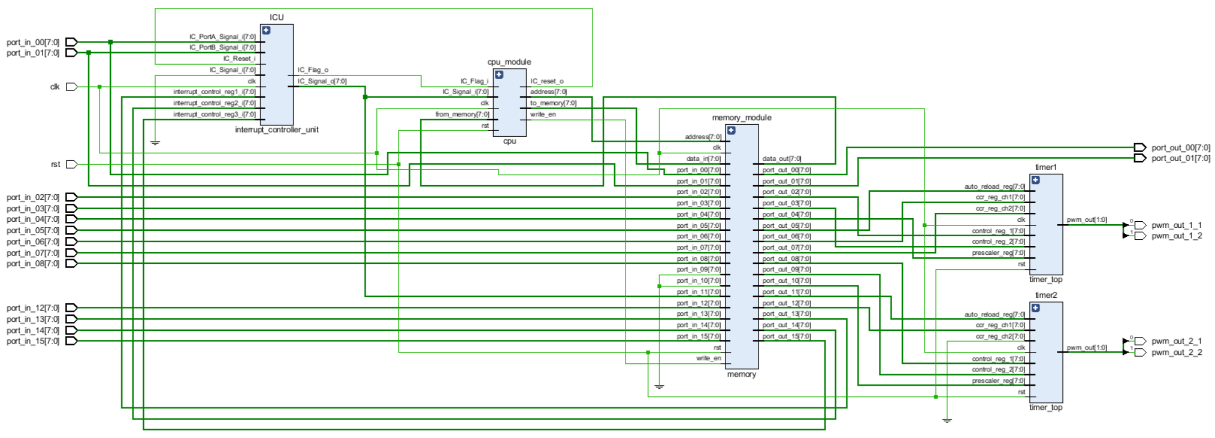


Figure 63 Elaborated Design: The schematic shows that the computer architecture modules and its connections.

CPU, memory, ICU, peripherals (TIM1 etc.) and their connection between each other are clearly visible in the schematic.

5.2- Synthesis

In the synthesis phase, the RTL design of the computer system design is converted to the netlist by synthesis tool in Xilinx Vivado basicly.

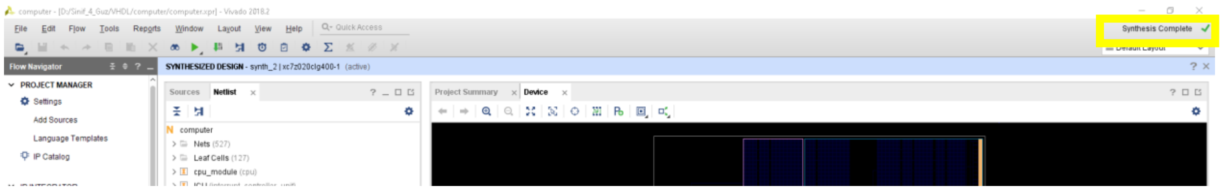


Figure 64 Synthesis Process Completed: The figure shows that synthesis stage is completed successfully.

After starting the synthesis process, as you see above, the shynthesis phase is completed successfully.

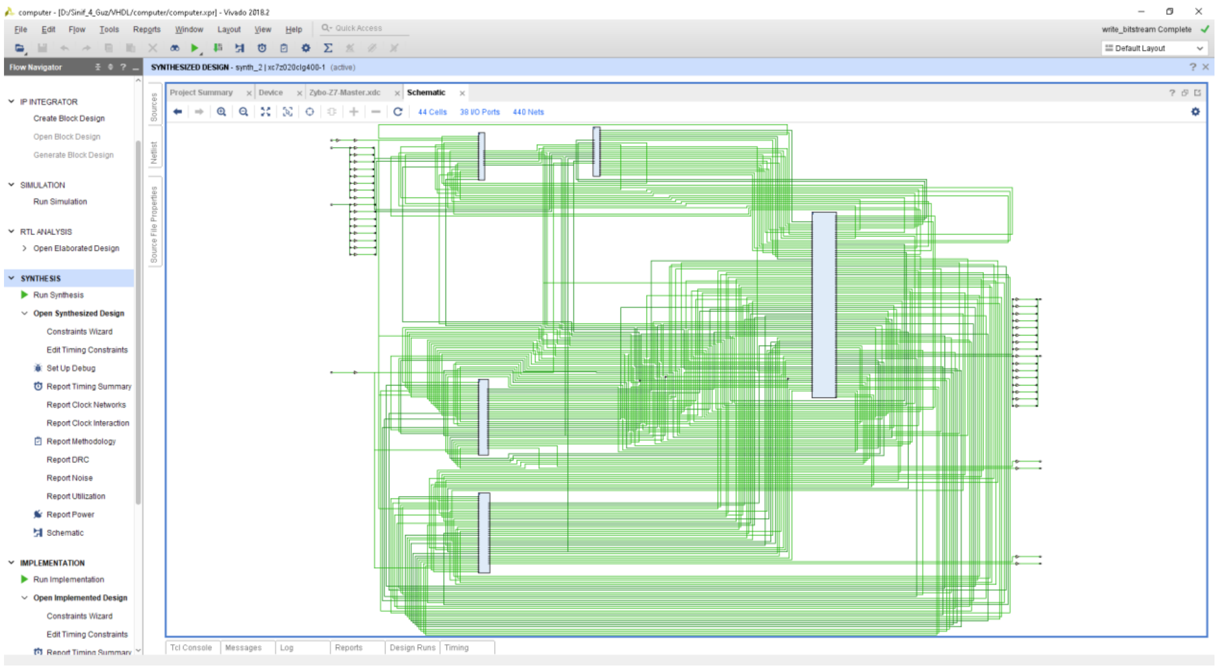


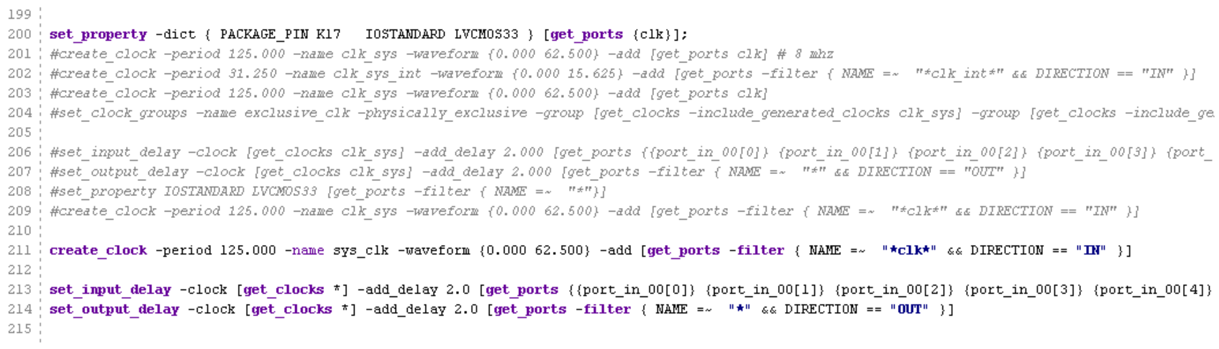
Figure 65 Synthesis Schematic: The schematic shows that that the computer architecture modules and its connections in synthesis stage.

Also, we can examine the schematic of the computer design after the synthesis phase. But it becomes seen little complex. We can look at hierarchical design files and elaborated design to understand computer architecture easily.

5.2.1- Static Timming Analyze

While designing the computer architecture, it was designed in accordance with a single clock generator. This is because using more than one clock generator increases power consumption and increases cost rates. At the same time, using multiple clock generators will lead to a more complex design in terms of synthesis and implementation. Because the clocks will drive the whole system and a more complex design will be created as a result of the place&route process.

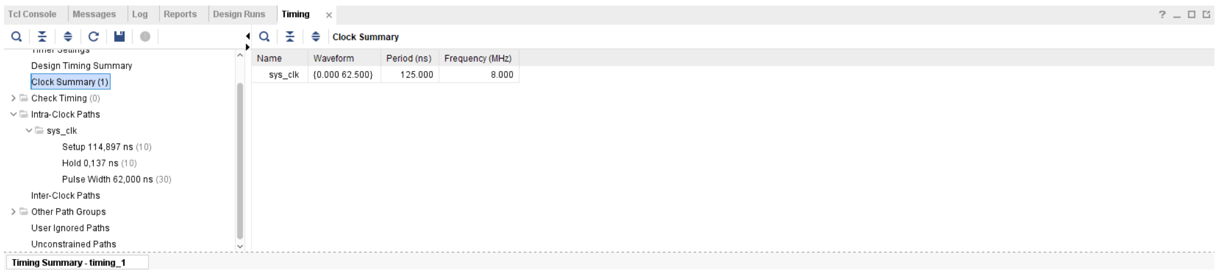
Since the target operating speed for this computer design is 8 MHz, I configured the constraint file and the design according to this clock frequency.



```
set_property -dict { PACKAGE_PIN K17   IOSTANDARD LVCMOS33 } [get_ports {clk}];
#create_clock -period 125.000 -name clk_sys -waveform {0.000 62.500} -add [get_ports clk] # 8 mhz
#create_clock -period 31.250 -name clk_sys_int -waveform {0.000 15.625} -add [get_ports -filter { NAME =~  "*clk_int*" && DIRECTION == "IN" }]
#create_clock -period 125.000 -name clk_sys -waveform {0.000 62.500} -add [get_ports clk]
#set_clock_groups -name exclusive_clk -physically_exclusive -group [get_clocks -include_generated_clocks clk_sys] -group [get_clocks -include_ge

#set_input_delay -clock [get_clocks clk_sys] -add_delay 2.000 [get_ports {{port_in_00[0]} {port_in_00[1]} {port_in_00[2]} {port_in_00[3]} {port_
#set_output_delay -clock [get_clocks clk_sys] -add_delay 2.000 [get_ports -filter { NAME =~  "*" && DIRECTION == "OUT" }]
#set_property IOSTANDARD LVCMOS33 [get_ports -filter { NAME =~  "*"}]
#create_clock -period 125.000 -name clk_sys -waveform {0.000 62.500} -add [get_ports -filter { NAME =~  "*clk*" && DIRECTION == "IN" }]

create_clock -period 125.000 -name sys_clk -waveform {0.000 62.500} -add [get_ports -filter { NAME =~  "*clk*" && DIRECTION == "IN" }]

set_input_delay -clock [get_clocks *] -add_delay 2.0 [get_ports {{port_in_00[0]} {port_in_00[1]} {port_in_00[2]} {port_in_00[3]} {port_in_00[4]}
set_output_delay -clock [get_clocks *] -add_delay 2.0 [get_ports -filter { NAME =~  "*" && DIRECTION == "OUT" }]
```

Figure 66 Clock Part In Constraint File: The figure shows that clock generated and delay configuration for Zybo Z20.

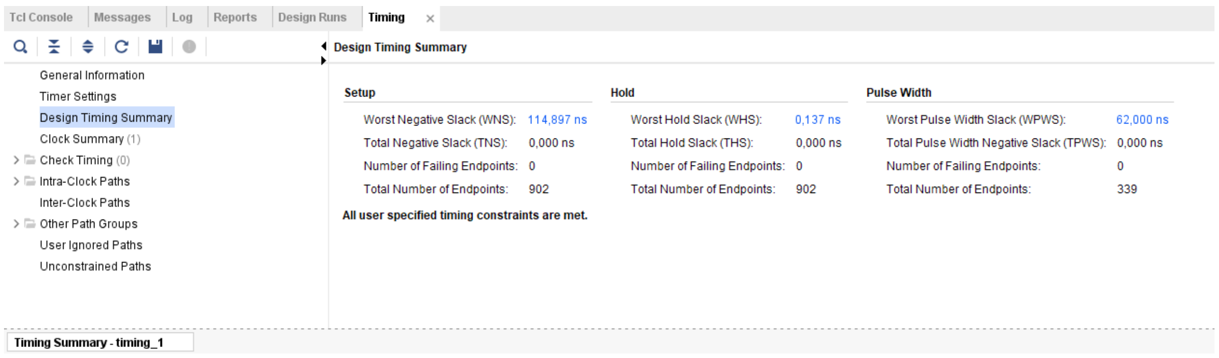
As seen in the image above, the clock configuration is set for a period of 125 ns and 50% highside and 50% lowside clock is generated from the K17 pin of Zybo Z20. Thus, 8MHz frequency clock generator is ready. At the same time, a 2 ns delay has been added to the inputs and outputs to prevent practical errors.



Name	Waveform	Period (ns)	Frequency (MHz)
sys_clk	{0.000 62.500}	125.000	8.000

Figure 67 Clock Summary: The figure shows the clock frequency occurred after synthesis stage.

As seen in the figure above, after the clock generator was created, the clock frequency was successfully set to 8 MHz.



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	114,897 ns	Worst Hold Slack (WHS):	0,137 ns	Worst Pulse Width Slack (WPWS):	62,000 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	902	Total Number of Endpoints:	902	Total Number of Endpoints:	339

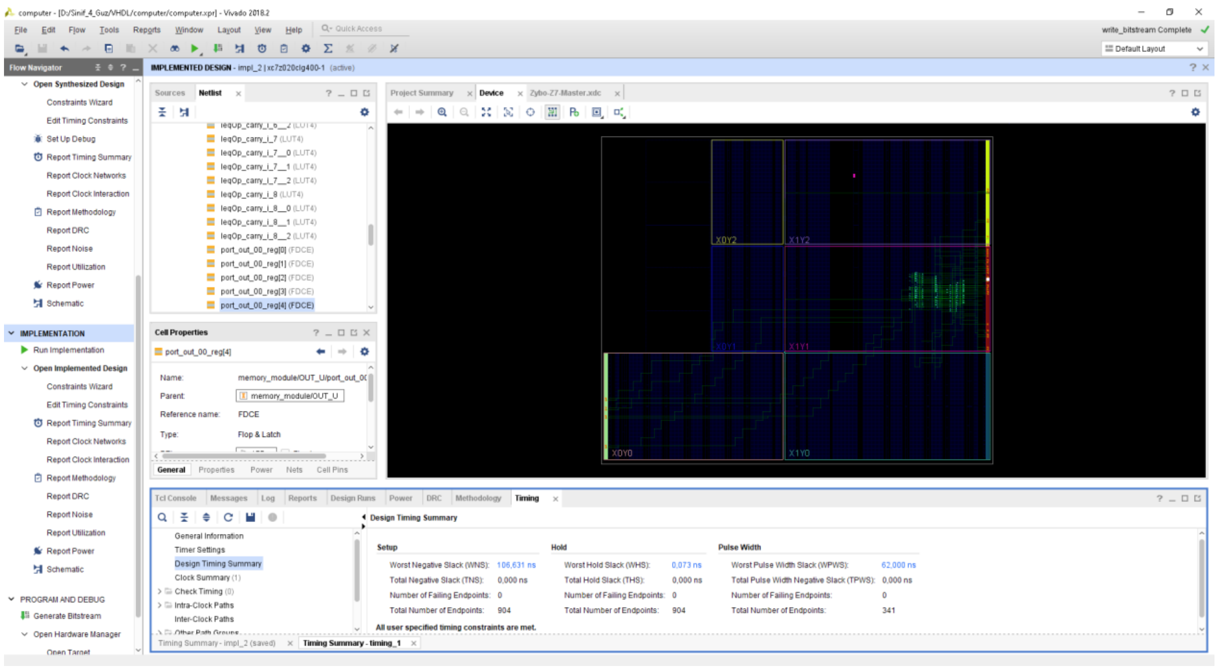
All user specified timing constraints are met.

Figure 68 Design Timing Summary: The figure shows the timing results occurred after synthesis stage.

As seen in the figure above, we have seen worst negative slack +114,897ns and worst hold slack +0.137 ns. The 125 ns clock frequency I have set is more than enough for this system. With this information, we also understand that the maximum speed at which the processor will operate is approximately 165 MHz according to synthesis results.

5.3- Implementation

Vivado's implementation tool provides placement & routing and bitstream for Xilinx Devices from the netlist created after the synthesis phase. The implementation phase is the previous phase of the FPGA programming phase in the FPGA design flow. It is the final stage of the design process.



computer - [D:/Sinif_4_Guz/VHDL/computer/computer.xpr] - Vivado 2018.2

IMPLEMENTED DESIGN - impl_2 | xc7z020clg400-1 (active)

Cell Properties: port_out_00_reg[4]

Name:	memory_module/OUT_U/port_out_00
Parent:	memory_module/OUT_U
Reference name:	FDCE
Type:	Flop & Latch

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	106,631 ns	Worst Hold Slack (WHS):	0,073 ns	Worst Pulse Width Slack (WPWS):	62,000 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	904	Total Number of Endpoints:	904	Total Number of Endpoints:	341

All user specified timing constraints are met.

Figure 69 Implementation Design: The figure shows the implemented design and the timing results occurred after implementation stage.

As seen in the picture above, the designed computer architecture has been successfully implemented on the PL side of the Zybo Z20. Also, according to timing summary, worst negative slack is +106,631 ns and worst hold slack is +0.071 ns. These values are fine for this implemented design.

6- Demonstration

I prepared a demonstration algorithm and system to show the capabilities of my computer system. The purpose of the algorithm is to drive the DC motor at certain speeds according to the mode entered by the user and to notify the user with output components such as seven-segment display/LEDs while driving. However, an emergency mode has been created with the interrupt mode. Emergency mode puts the processor in interrupt mode and stops the motor when the specified button is pressed. When the same button is pressed again, the engine starts from where it left off.

6.1- Demonstration Algorithm Flowchart

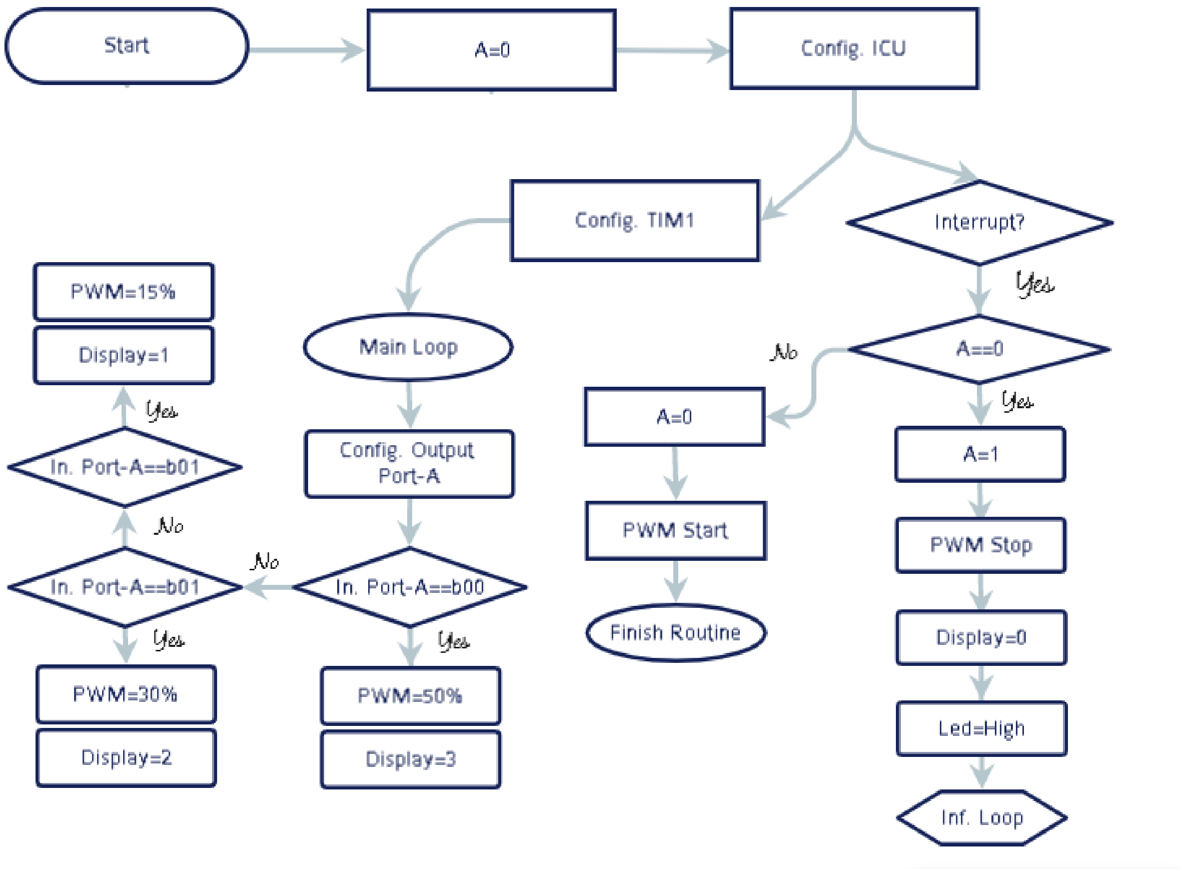


Figure 70 The Flowchart for Demonstration Algorithm: The diagram shows the flowchart prepared to control a dc motor with some cases.

The algorithm works by following the steps below:

a) When the program starts, a variable is created and has zero initialize the value to use in interrupt serial routine for toggling. In the first interrupt, PWM generation will stop and after that, following another interrupt, PWM generation will start again (toggling process).
b) Interrupt Controller Unit (ICU) will be configured. After the configuration of ICU, the interrupt serial routine is activated thanks to ICU.
c) Timer1 will be configured for PWM.
d) After all configurations, the main loop will start. There are three modes as I mention. In main loops, the mode value will be read from input port-a. The value “00” – “01” – “10” in binary

respectively represents fast-speed mode, mid-speed mode, and low-speed mode. In each stage, the seven-segment display will be updated according to modes.

e) In interrupt serial routine, as an addition to part ‘a’, timer configuration, led and the display will be updated.

6.2- Demonstration Algorithm Simulation/Test

In this section, I gonna mention about the tests of modes and interrupt for the demonstration algorithm.

6.2.1- Modes In Demo Algorithm

There are three modes to control the DC motor. I prepared a test bench to make a simulation for the demonstration algorithm. You must look first two bits of port_in_00 (Port-A) to understand motor modes according to the test bench. The period is set as 500us for intervals of the steps/modes.

port_in_00 <= "00000000"; → 00: Fast Speed Mode

wait for 500us;

port_in_00 <= "00000001"; → 01: Mid Speed Mode

wait for 500us;

port_in_00 <= "00000010"; → 10: Low Speed Mode

wait for 500us;

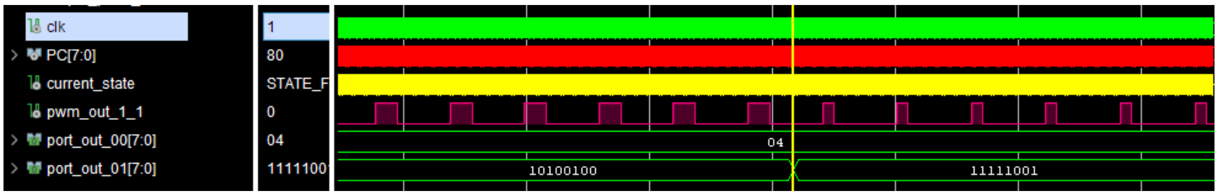


Figure 71 The Waveform Demo Algorithm for Modes-1: The waveform shows that mid speed and slow speed modes work properly.

As you see above, there are two modes (mid 30% duty and slow 15% level speed). And port_out_01 (Port-B) was set for anode seven segment display. PWM modes work for two modes successfully.

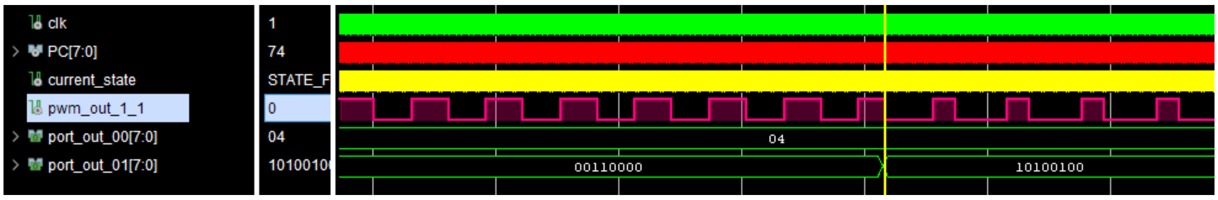


Figure 72 The Waveform for Demo Algorithm Modes-2: The waveform shows that mid speed and fast speed modes work properly.

As you see above, there are two modes (fast 50% duty and mid 30% level speed). And port_out_01 (Port-B) was set for anode seven segment display. PWM modes work for two modes successfully.

6.2.2- Interrupts In Demo Algorithm

I had previously configured the last bit of Port-A for the interrupt and developed the test bench algorithm below for testing.

```
port_in_00 <= "00000000";


wait for 500us;


port_in_00 <= "10000000"; → Interrupt

wait for clock_period*5;

port_in_00 <= "00000000";


wait for 500us;


port_in_00 <= "10000000"; → Interrupt

wait for clock_period*5;

port_in_00 <= "00000000";


wait for 500us;


port_in_00 <= "10000000"; → Interrupt

wait for clock_period*5;

port_in_00 <= "00000000";
```

I put the simulation of the processor in an interrupt state three times. Interrupt states have 500us delays between them. According to the test bench algorithm above, the PWM output will be given properly first. The PWM output generation will stop after the first interrupt state. In the next interrupt, PWM output generation will work as in the first case, and thus it will continue by enabling PWM toggle with the interrupt.

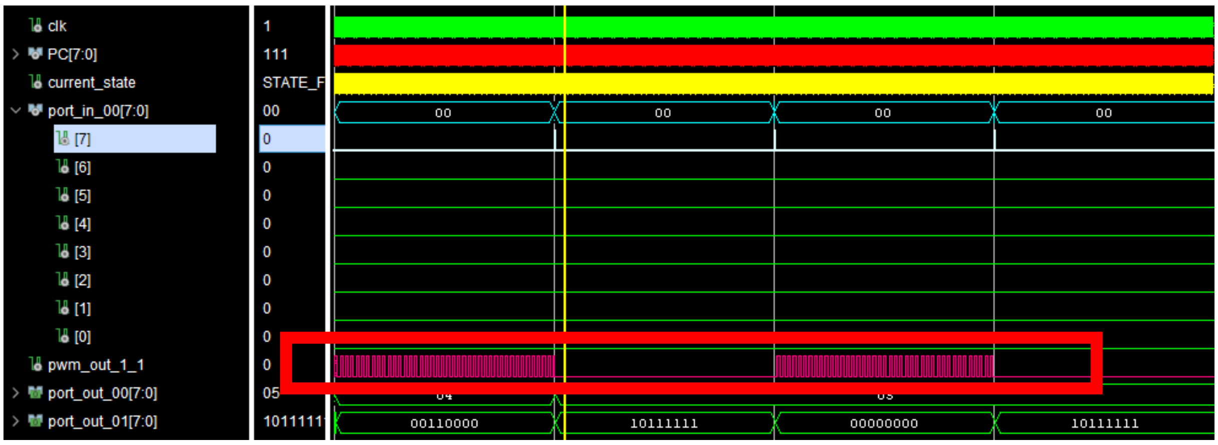


Figure 73 The Waveform for Output of Demo Algorithm: The waveform shows that demo algorithm works properly.

As seen above, depending on the last bit of the input port-a, the CPU enters the interrupt serial routine, and the PWM output changes accordingly.

If we look in more detail:

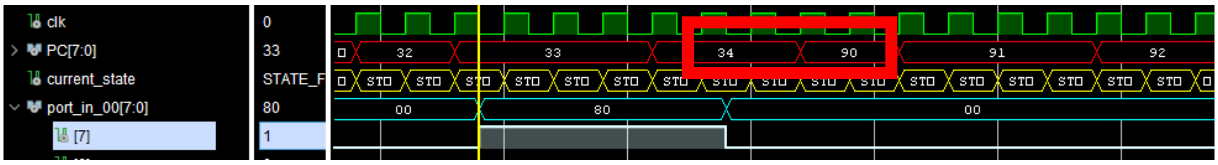


Figure 74 The Waveform of Interrupt Handle in Demo: The waveform indicates that the algorithm brings the PC value to the interrupt serial routine at the memory address.

The program counter keeps increasing normally. When the interrupt is entered, it goes to the interrupt serial routine address (90 in decimal) at the memory address. Be careful! The address to be registered to SP in the first interrupt is 34 in decimal.

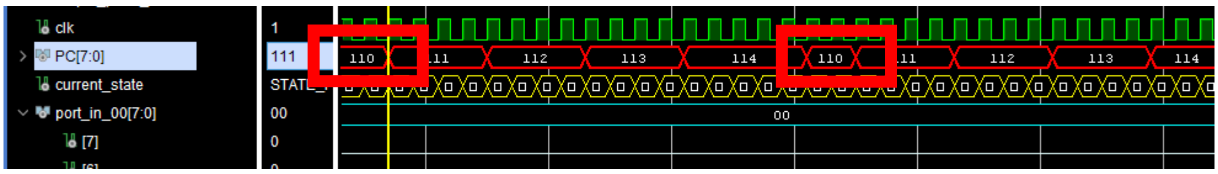


Figure 75 The Waveform of Infinite Loop in Demo: Waveform shows that the code is successfully inserted into the infinite loop.

Then this state will wait in an infinite loop until the next interrupt state. As you can see above, the PC value goes from 110 to 110 over and over.

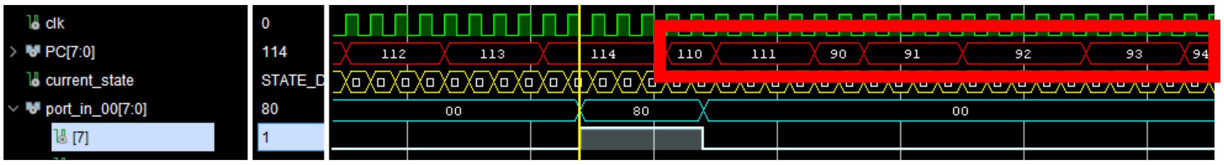


Figure 76 The Waveform of Exit Interrupt Serial Routine in Demo: Waveform indicates PC value exit from interrupt serial routine after its previous value kept in SP2 is brought to PC.

After 500us, the interrupt is entered again and the query is started in the interrupt routine. PC value exits from interrupt serial routine.

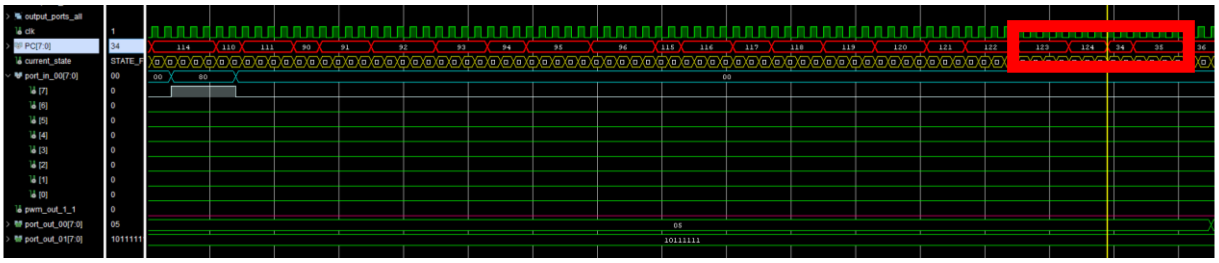


Figure 77 The Waveform of Continuity Of Demo Algorithm: Waveform shows PC value back to main loop.

When the second interrupt state is reached, when PC points to 96, the query takes place and the PC value exits the infinite loop. PC points to 115 and increments through the interrupt serial routine. When the last address of the interrupt serial routine is reached, the PC value becomes 34 again and our test is completed successfully.

6.3- Demonstration Environment

It is a simulation / test environment in which various hardware is prepared to show that the computer architecture is working. This environment shows that the algorithm described above works in the designed computer architecture.

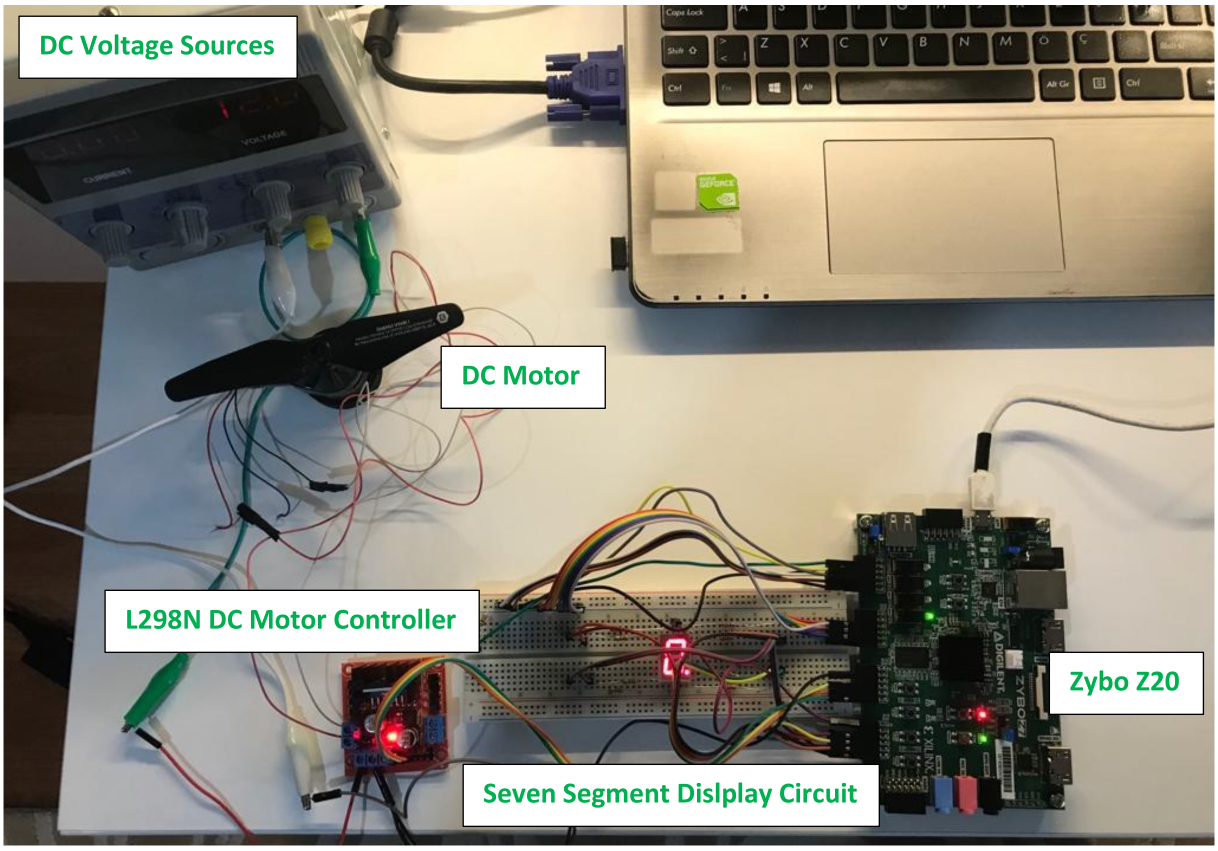


Figure 78 Demonstration Environment: It is a simulation environment figure consisting of FPGA, motor controller, DC motor, seven segment display and DC voltage source.

As seen in the picture above, this demonstration environment consists of Zybo Z20 SoC board, DC motor, DC motor controller, seven segment display and DC voltage source.

The computer architecture is implemented on the PL side of the Zybo Z20. Seven segment display and dc motor controller prepared with Zybo Z20 are directly connected. If the DC motor and DC voltage source are directly connected to the DC motor controller, the environment has been prepared and it has been observed that the computer is working properly.

You can click for the video showing this simulation environment and computer architecture working.

7- Conclusion

As a result, I realized the RTL design of a synthesizable 8-bit RISC computer architecture and successfully completed all the verification processes. After the test and simulation phase, I performed the synthesis of the computer I designed. I prepared the constraint file required during the implementation phase and implemented the Zybo Z20 Zynq device. I observed the operating performance and accuracy of the processor with the dc motor control algorithm that I had prepared before.

The general organization of the computer architecture I designed consists of CPU, memory organization system and peripheral units. The CPU is designed in such a way that the instructions, which consist of an 8-bit opcode and an 8-bit operand, and previously shown in the instruction table, can be computed. Memory system, on the other hand, is a 256-byte memory structure consisting of RAM, ROM and output that gives results according to the address pointed by the relevant address data. Peripheral units consist of ICU and timer units. The ICU is the control system that interacts directly with the CPU required for interrupt situations.

During the project process, I learned computer architectures and organization, the critical points to be considered while designing RTL, the use of IP blocks, and how to perform constraint designs of various hardware and peripherals. However, as a result of my research, I had the opportunity to improve myself on RISC-V.

After all this process, I successfully designed this computer with 8-bit RISC architecture and implemented it on the FPGA board. I observed that after I wrote the algorithm I wanted in assembly language and installed it inside the computer, it worked.

8- References

[1] Ian Grout, in Digital Systems Design with FPGAs and CPLDs, 2008

[2] David A. Patterson - John L. Hennessy, Computer Organization and Design: The Hardware/ Software Interface, Fifth Edition, 2014

[3] Brock J. LaMeres, Introduction to Logic Circuits & Logic Design with VHDL, 2019

[4] Elmustafa Sayed Ali Ahmed Mohamed, Principles of Computer System, 2015

[5] Jim Plusquellic, "CMSC 611: Advanced Computer Architecture", last view: 02.04.2022, http://ece-research.unm.edu/jimp/611/

[6] Sajjad Ahmed et al. "IBTIDA: Fully open-source ASIC implementation of Chisel- generated System on a Chip". In: September 2021, TechRxiv.

[7] Dr. Shun Yan Cheung, Emory College Of Arts And Sciences, Departmant Of Mathematics And Computer Science, Computer Science Courses, 355 Course Documents, http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/

[8] Vinay Reddy Narayana, "Micro-Architecture Design of RISC-V Microprocessor Using VHDL". In: 2017, International Journal of Electrical, Electronics and Data Communication.

[9] Computer Organization and Design RISC-V Edition The Hardware Software Interface, Second Edition, David A. Patterson, John L. Hennessy. In: 31 Dec. 2020

[10] Studies in Microprocessor Design Donald Alpcrt Technical Report NO. 232, June 1982