ELE432 Advanced Digital Design Project
Design of a 16-bit RISC Processor Using VHDL

Sümeyra Durak
21528057
sumeyraduraak@gmail.com
Hacettepe University

Umut Utku Koçak
21528335
umututku.kocak@gmail.com
Hacettepe University

Fatma Zülal Kiraz
21428173
kirazulal@gmail.com
Hacettepe University

***Abstract*—The computer software represents instructions that the computer will execute and how to execute defined tasks. The computer software represents instructions that the computer will execute and how to execute defined tasks. Special instructions that the computer can execute are called that computer's instruction set.**

Instruction set must be defined before hardware implementation. In some systems, the instruction set is fairly limited to minimize the physical size needed in the CPU. Therefore, the CPU executes defined instructions very quickly, however, to achieve a task The CPU should deal with a lot of operation. This architectural approach is called a reduced instruction set computer (RISC). [1] This project aims to design and implement a 16-Bit RISC Processor with VHDL, Karansing P. Thakor, by reference to Ankushkumar Pal's published paper on ijet.org.

1. Design Methodology

The way we followed while carrying out the design was as follows We divided the instruction set into 3 parts: arithmetic and logical instructions, conditional and unconditional jumps, and load-store instructions. Then, we detected the dataflow. We created the modules used for each piece to achieve that task only. We tested the Instructions on these modules we created. After that, we combined the modules and tested our entire CPU system to execute all instruction set.

2. Instruction Set

The Instruction Set is exactly the same as the reference design as requested. We started our design by analyzing the instruction set. The opcode is a unique binary code to accomplish an instruction. The CPU decodes opcode and decides what steps to follow to complete the instruction. As in reference design, the first 9 instructions contain arithmetic and logical operations.

	Instruction	Opcode				Operation
1	ADD	0	0	0	0	Rd = Rs1 + Rs2
2	SUB	0	0	0	1	If RS1 > RS2 Then Rd = RS1 – RS2 Else Rd = RS2 – RS1
3	AND	0	0	1	0	Rd = Rs1 & Rs2
4	OR	0	0	1	1	Rd = Rs1 | Rs2
5	NOT	0	1	0	0	Rd = ~ Rs
6	XOR	0	1	0	1	Rd = Rs1 ^ Rs2
7	CMP (Equal	0	1	1	0	If Rs1 = Rs2 Then Equal = 1, else Equal =0 If R1 = 0 Then AZ = 1, else AZ=0 If Rs2 =0 Then BZ =1, else BZ =0 If Rs1> Rs2 Then AGB = 1, else AGB = 0 If Rs1< Rs2 Then ALB = 1, else ALB = 0
8	SHIFT LEFT	0	1	1	1	Rd=Rs1 <<1
9	SHIFT RIGHT	1	0	0	0	Rd=Rs1 >>1
10	LOAD	1	0	0	1	Rd = Mem[Rs1]
11	STORE	1	0	1	0	Mem[Rs1] = Rs2
12	JUMP	1	0	1	1	PC = PC+Offset
13	NOP	1	1	0	0	No operation
14	JZ	1	1	0	1	PC = PC+Offset if Rd == 0
15	JNZ	1	1	1	0	PC = PC+Offset if Rd != 1
16	LOAD 8-BIT IMMEDIATE	1	1	1	1	Rd = 8-bit Immediate

Figure 1. Proposed instruction set

3. Architecture

3.1. Modules

Arithmetic and Logic Unit input and outputs are shown in the Figure-2. This block computes arithmetic or logic instructions between 'inp1' and 'inp2' input ports depending on the 'ALU_op' input port which is controlled by the control unit, and shows the result at 'outp' output port. Also, it shows the 'flags' which represents the comparison between the inputs. 'cmp_s1_s2' output is used for conditional jumps, if the condition is true, it allows the calculated target branch address is written into the PC register. Also, it has an asynchronous reset.

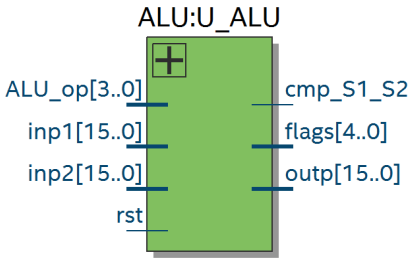

Figure 2. ALU Module

Arithmetic and Logic Unit input and outputs are shown in the Figure-3. This is a parallel register that updates on rising edge of the clock with an asynchronous reset. This block is used for storing the target branch address for conditional jumps.

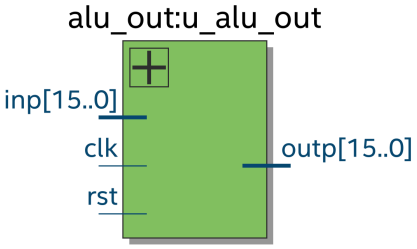

Figure 3. ALU Out Module

Control unit input and outputs are shown in the Figure-4. Control unit organizes the control signal which are used for controlling the data flow between the modules using the opcode information which comes from the fetched instruction. The system is controlled using a positive edge triggered finite state machine shown in the Figure-9. And the output signal changes depending on the state and input port 'opcode' is shown in the Figure-10. Also, it has an asynchronous reset.

Instruction register input and outputs are shown in the Figure-5. . At the rising edge of the clock, when 'IR_wr_en' input port which is the read enable for this register, is high, the module reads 16-bit data from the 'instruction' input port. Then, divides the data and transmits to the output ports that named 'opcodes', 'sel_D', 'sel_S1', 'sel_S2' and 'immediate' continuously. Also, it has an asynchronous reset.

Load store unit input and outputs are shown in the Figure-6. This unit is used for organizing the memory load store operations between the external memory and the general purpose register file.

Program counter register input and outputs are shown in the Figure-7. At the rising edge of the clock, when 'en' input port which is the read enable for this register, is high, the module reads 16-bit data from the 'inp' input port and transmits to the 'outp' output port continuously. Also, it has an asynchronous reset.

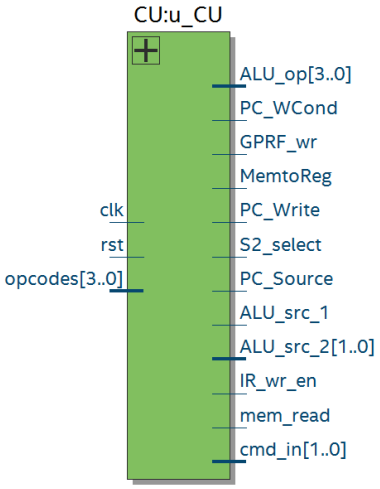

Figure 4. Controller Unit Module

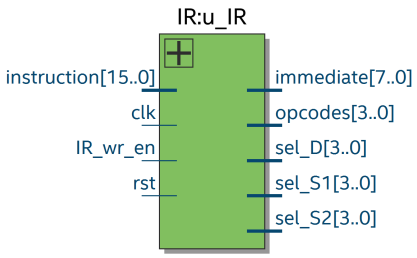

Figure 5. Instruction Register Module

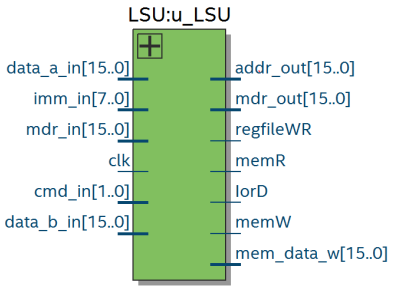

Figure 6. Load-Store Unit Module

General purpose register file input and outputs are shown in the Figure-8. This block is basically a RAM block which can be read out the content of the memory and written in with an enable port using the addresses that are specified by the fetched instruction. Also, it has an asynchronous reset.

4. Results

4.1. Instruction Waveforms

4.1.1. Arithmetic and Logical Instructions.

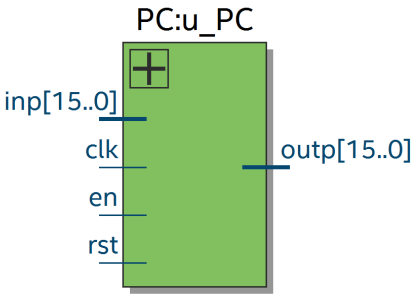

Figure 7. Program Counter Module

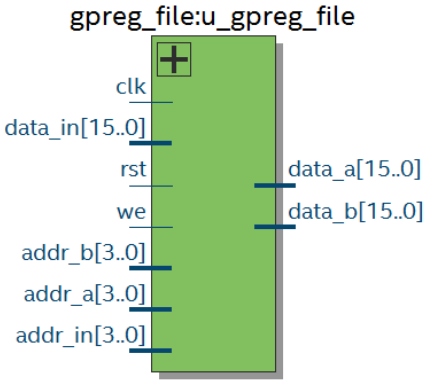

Figure 8. Register File Module

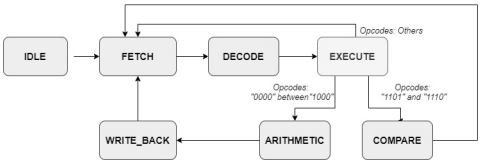

Figure 9. State Diagram of Controller Unit

	IDLE	FETCH	DECODE	EXECUTE Opcodes 1001, 1010, 1111	EXECUTE Opcodes 1011	EXECUTE Opcodes 1101, 1110	EXECUTE Opcodes others	ARITHMETIC	WRITE_BACK	COMPARE
GPRF_wr	0	0	0	0	0	0	0	0	1	0
IR_wr_en	0	1	0	0	0	0	0	0	0	0
S2_select	0	0	0	0	0	1	0	0	0	0
ALU_op	1100	0000	1100	Opcodes	Opcodes	1011	Opcodes	Opcodes	Opcodes	Opcodes
ALU_src_1	0	1	0	0	1	1	0	0	0	0
ALU_src_2	00	01	00	00	10	10	00	00	00	00
PC_WCond	0	0	0	0	0	0	0	0	0	1
PC_Write	0	1	0	0	1	0	0	0	0	0
PC_Source	0	0	0	0	0	1	0	0	0	0
mem_read	0	1	0	0	0	0	0	0	0	0
MemtoReg	1	1	1	1	1	1	1	1	0	1
cmd_in	00	00	00	Opcodes (1 downto 0)	00	00	00	00	00	00

Figure 10. Control signal output according to states and opcodes

It takes 5 cycles to execute arithmetic and logical instructions. As can be seen from the results in the test benches, the values from the R3 and R9 addresses were put into arithmetic and logical operations, and the results were written into address R0 at the end of the next 5 cycles. First, we enable memory from the control unit with Mem_read. Then by enabling the Write into IR with the IR_wr signal coming from the control unit, the 16-bit instruction is written from memory to the IR register..

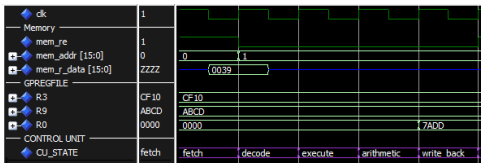

Figure 11. ADD instruction waveform

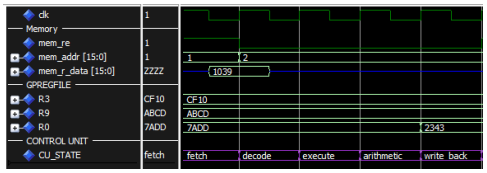

Figure 12. SUB instruction waveform

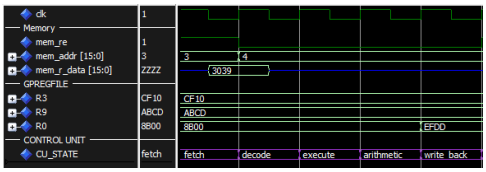

Figure 13. OR instruction waveform

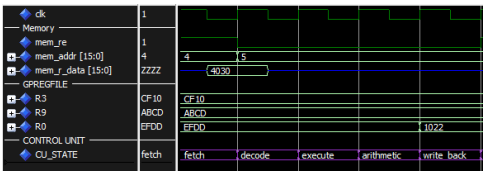

Figure 14. NOT instruction waveform

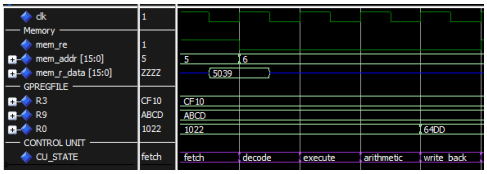

Figure 15. XOR instruction waveform

To illustrate, ALU_Src_1 and ALU_Src_2 select the values that we will put the signals into the operation from GPReg. (With 0-0) At the same time, when fetching the

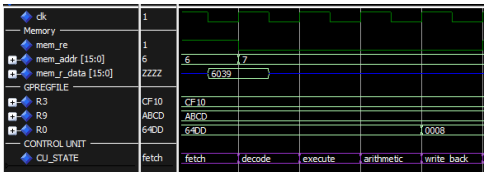

Figure 16. CMP instruction waveform

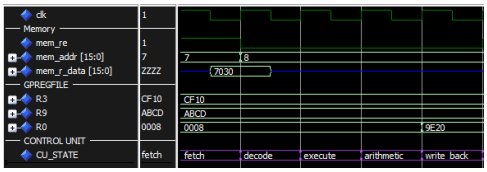

Figure 17. SHIFT LEFT instruction waveform

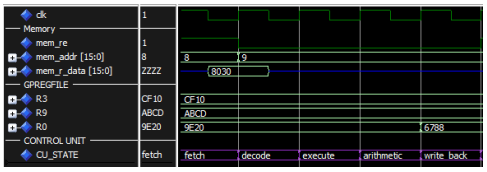

Figure 18. SHIFT RIGHT instruction waveform

instruction, we give the current value of the PC to the input of Mux. With ALU_Src_2 (1), we get the current value of the PC and with ALU_Src_1 (1) we get ”0001” and increase the PC by 1 within ALU. And we get the result of the operation from ALU_OUT and write it in the GPReg file..

4.1.2. Jump (Branching) Instructions.

JZ and JNZ takes 4, unconditional JMP takes 3 cycles. If opcode conditional jump, first, the target address to jump must be calculated. With ALU_Src_2, the immediate address from IR is selected and given to ALU via mux. Then, the current Program Counter value is given to ALU with ALU_Src_1. ALU calculates the target address by adding these two addresses. The calculated target address is kept in ALU_OUT. When executing instruction in ALU, the next address is calculated as usual. If the condition is true, the PC_Source signal selects the target address held in ALU_OUT via mux. If the condition cannot be achieved, PC_source gives 1 incremented PC value in ALU to the PC..

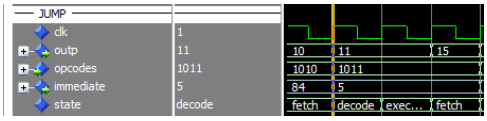

Figure 19. JUMP instruction waveform

In the Unconditional Jump Case, the current PC value is adding with the immediate address from IR. It is given to the PC regardless of a condition..

As can be easily observed from waveforms, the immediate-offset address from the IR is 5, and the opcode is

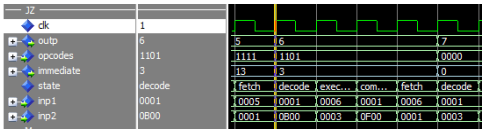

Figure 20. JUMP ZERO instruction waveform

Figure 21. JUMP NOT ZERO instruction waveform

”1011”. Since the PC was 10 during fetch, it jumped to 15 after 3 cycles. The Jump Condition works as we expected..

For the Conditional Jump Case, since the value in the register is not 0, JZ condition was not provided. The Immediate-offset 3 addresses sent by the IR are not added to the PC. Therefore, the CPU continued its cycle by increasing the PC by only 1. In the JNZ instruction case, since the value in address one is not zero the JNZ condition has been met, the Program Counter has jumped from 6 to 9..

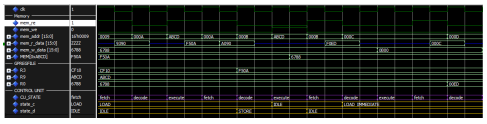

Figure 22. LOAD , STORE and LOAD IMMEDIATE instructions waveform

4.1.3. Memory Access Instructions.

4.2. Example Program Waveform

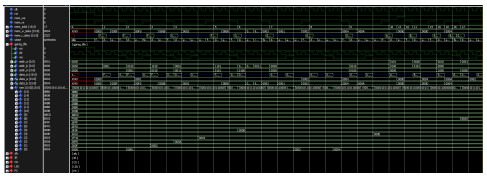

Figure 23. Example Program Waveform

Load, Store and Load Immediate takes 4 cycles. However while at the write_back phase of the control unit, writing to Register File and fetching new instruction happens simultaneously. So we can consider that it act as a small pipeline..

```
function init_ram
    return memory_t is
    variable tmp : memory_t := (others => (others => '0'));
begin
    -- CODE_SEGMENT
    tmp(0) := encode(LI_OP,AD(R0),x"01");
    tmp(1) := encode(LI_OP,AD(R1),x"02");
    tmp(2) := encode(LI_OP,AD(R2),x"0A");
    tmp(3) := encode(ADD_OP,AD(R3),AD(R0),AD(R1));
    tmp(4) := encode(LI_OP,AD(R5),x"0D");
    tmp(5) := encode(JZ_OP,AD(R4),x"03");
    tmp(6) := (others => '0');
    tmp(7) := (others => '0');
    tmp(8) := encode(ADD_OP,AD(R4),AD(R0),AD(R2));
    tmp(9) := encode(STORE_OP,AD(R5),AD(R4));
    tmp(10) := encode(JUMP_OP,x"05");
    tmp(11) := (others => '0');
    tmp(12) := (others => '0');
    tmp(13) := (others => '0');
    tmp(14) := (others => '0');
    tmp(15) := encode(LOAD_OP,AD(R8),AD(R2));
    return tmp;
end init_ram;

-- Declare the RAM signal and specify a default value.  Quartus Prime
-- will create a memory initialization file (.mif) based on the
-- default value.
signal ram : memory_t := init_ram;
```

Figure 24. Example Program Memory Initialization

5. Conclusion

As can be observed from Figure 26, according to post-synthesis, the maximum frequency was measured as about 180 MHz. Also, utilization results have shown in the Figure 25

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins
1	▲ |CPU|	314 (2)	74 (0)	512	0	52	0
1	|ALU:U_ALU|	216 (216)	0 (0)	0	0	0	0
2	|CU:u_CU|	12 (12)	7 (7)	0	0	0	0
3	|IR:u_IR|	0 (0)	16 (16)	0	0	0	0
4	|LSU:u_LSU|	7 (7)	19 (19)	0	0	0	0
5	|PC:u_PC|	0 (0)	16 (16)	0	0	0	0
6	|alu_out:u_alu_out|	0 (0)	16 (16)	0	0	0	0
7	▷ |gpreg_file:u_gpreg_file|	1 (1)	0 (0)	512	0	0	0
8	|mux2:u_mux2_1|	4 (4)	0 (0)	0	0	0	0
9	|mux2:u_mux2_2|	16 (16)	0 (0)	0	0	0	0
10	|mux2:u_mux2_5|	16 (16)	0 (0)	0	0	0	0
11	|mux2:u_mux2_6|	16 (16)	0 (0)	0	0	0	0
12	|mux3:u_mux3_3|	24 (24)	0 (0)	0	0	0	0

Figure 25. Utilization of 16 BIT RISC Processor

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	181.62 MHz	180.02 MHz	clk	limit due to minimum period restriction (tmin)

Figure 26. Post synthesis timing analysis maximum frequency

The desired tasks of the project have been successfully completed. The exact schematic was designed and all specified instructions are executed with the designed CPU. A small program has been implemented and shown in Figure 24 The demands of the project have succeeded. The project works were carried out with online meetings during the pandemic. Tasks were distributed to each group member balanced throughout the project.

Project files can be found in [3]

References

[1] *THAKOR, K.; PAL, Ankushkumar. Design of a 16-bit RISC Processor UsingVHDL. International Journal of Engineering Research and Technology (IJERT), 2017, 6.4.*,

[2] *LAMERES, Brock J. Introduction to logic circuits logic design with VHDL. Springer, 2019.*

[3] $RISC16_BITProcesserReporsitory, [Online]Available : https://github.com/uukocak/risc16\_vhdl$