E3: 231 Digital Systems Design with FPGAs

PROJECT REPORT ON

16-BIT GENERAL PURPOSE RISC PROCESSOR

Submitted by

BASATI SIVAKRISHNA (M. TECH EPD-22976)

KATTA JOHN AKHIL (M. TECH EPD-23021)

12 April 2024

Course Instructor: DEBYAN DAS

Indian Institute of Science, Bangalore

1.Introduction and Motivation:

As we can see that modern hardware is using processors for almost every application. And from the sources we can say that RISC processors are dominating in this domain. So, an efficient processor with low power consumption and with high frequency of operation will make existing hardware more powerful than now.

Our design is on designing multicycle RISC CPU which should be flexible enough to incorporate new assembly instructions and do pipelining if needed in future.

Motivation behind selecting multicycle than pipelined version for our design:

- Lot of microprocessors and controller will be having an IDE support from the vendor(manufacturer) and IDE will support both assembly and C language code editors.
- People choose C or C++ over assembly because of ease of coding and portable.
- If we observe the existing embedded codes for these microcontrollers, we can understand that more than 70% of code will have loops and conditional statements.
- Which results in branch instruction. So pipelined version won't be efficient for the branch instruction since pipeline flush is needed.
- And in multicycle there is no concept of data dependency hazards.

2.Background Study:

Since the selected processor is multicycle version, we need to study and understand about each stage thoroughly. And our main goal is not only about design rather than making it flexible for pipelining, so we should design each stage accordingly.

- Instruction Fetch:
  - In this stage, the processor fetches the next instruction from memory.
  - The program counter (PC) holds the address of the next instruction to be fetched.
  - The instruction is typically fetched from the instruction memory based on the address in the PC.
  - The fetched instruction is then stored in a temporary register for further processing.
- Instruction Decode:
  - In this stage, the fetched instruction is decoded to determine the operation it specifies.
  - The opcode (operation code) of the instruction is extracted.
  - Operand addresses may also be decoded if the instruction involves data manipulation or memory access.
  - Control signals for subsequent stages are generated based on the decoded instruction.
- Instruction Execute:
  - In this stage, the actual operation specified by the instruction is performed.
  - Arithmetic, logic, or memory operations are executed based on the instruction type.
  - Data processing, such as addition, subtraction, bitwise operations, or memory read/write, takes place.
  - For branch instructions, the target address may be calculated based on conditions evaluated in this stage.
- Write back:
  - In this final stage, the results of the executed instruction are written back to the register file.
  - For arithmetic or logic instructions, the result is written to the destination register.
  - For load and store instructions, the data retrieved from memory or registers is written to the destination register or memory respectively.

3.Design Space Exploration and Design Strategies:

ISA DESIGN:

- To design any processor, one should fix what all assembly instructions that processors should support. In our case we have considered total 23 instructions each will have its unique op code.
- Since our design using 16-bit ALU we need to design ISA with respect to 16 bits. And each 16-bit instruction should define properly to get built with less hardware possible.
- We can classify selected instructions into 4 addressing modes and assign register and op code values as shown below.

- REGISTER TYPE

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
OP CODE					REG 1			REG 2			REG 3			XX	

- IMMEDIATE TYPE

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
OP CODE					REG 1			REG 2			IMM VALUE				

- BRANCH TYPE

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
OP CODE					REG 1			XXX			OFFSET VALUE				

- HALT

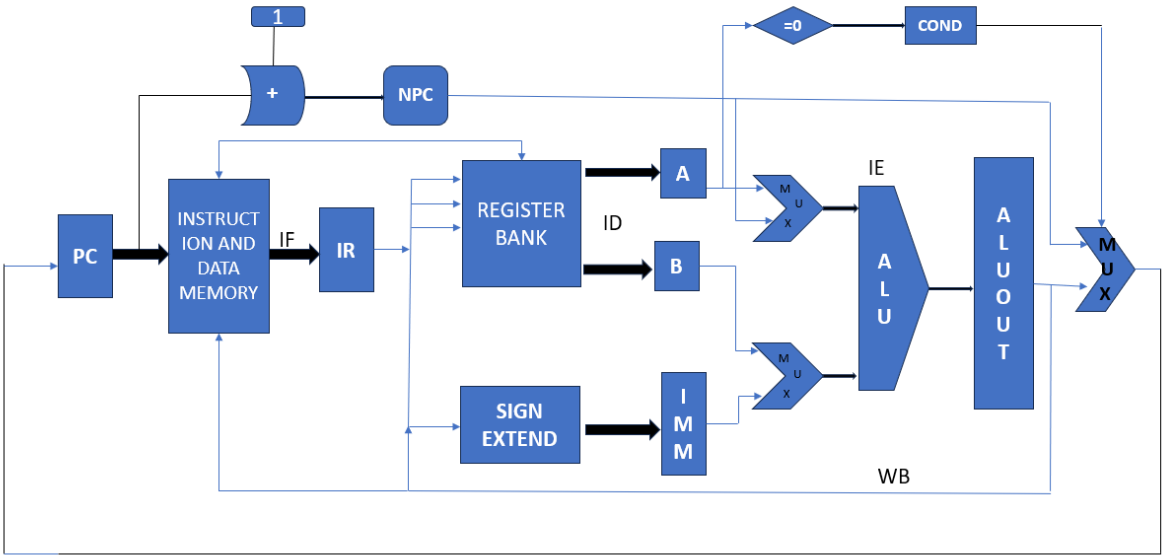
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
OP CODE					XXX			XXX			XXXXX				

And supported instructions with the opcode is given below:

```
31
32  parameter ADD = 5'd0, SUB = 5'd1, MUL = 5'd2, AND = 5'd3,
33  OR = 5'd4, INV = 5'd5, LSL = 5'd6, LSR = 5'd7, DEC = 5'd8,
34  INC = 5'd9, MOV = 5'd10, SLT = 5'd11, ADDI = 5'd12, SUBI = 5'd13,
35  SLTI = 5'd14, MOVI = 5'd15, BNEQ = 5'd16, BEQ = 5'd17, BEQZ = 5'd18,
36  BNEQZ = 5'd19, LD = 5'd20, ST = 5'd21, HLT = 5'd22;
37
```

Multistage design:

As defined in background study each stage is defined in Verilog code according to the architecture drawn (shown below).



Instruction Flow:

- According to type of instruction (addressing mode) each op code will route its data to the stages.
- For example, register addressing mode will be storing its source operands in A and B registers in decode stage and result will be written back into destination register.

- Similar load and store instructions' offset address will be calculated from immediate and register values. And written back into register or memory location.

Clock gating:

- In Verilog, clock gating is a technique used to reduce power consumption in digital circuits by selectively disabling the clock signal to certain portions of the circuit when they are not actively performing computations or operations.
- We are using this strategy to disable all stages when running the one stage. Since our design is multicycle only, we don't need other stages to work while one stage is running.
- We are implementing it by using a **stage** variable. So respective hardware only will get powered when it got that enable pin high.
- At each stage we are making next stage's enable signal as high after getting done with that stage.
- By implementing this method, we got overall power consumption as 0.07Watts.

4. Implementation Challenges:

1. While doing behavioural simulation we have seen the expected results, but after implementing it on FPGA, the outcome is not proper enough.
    - This happened because we have not closed all the if else statements and did not enclose default statements for the case instructions. So, the synthesis file generated latches in between which corrupted output,
    - It got solved by introducing else statements for ever if statement.
2. Since our design is risc processor there is no defined output for this. So, synthesis tool generated no hardware considering there is no hardware.
    - This problem got solved by assigning output (ALU OUT) to the top module.

5. TESTING AND RESULTS

- The following assembly code (to find maximum of an array) has executed in the designed processor and results has printed on the simulation waveforms.

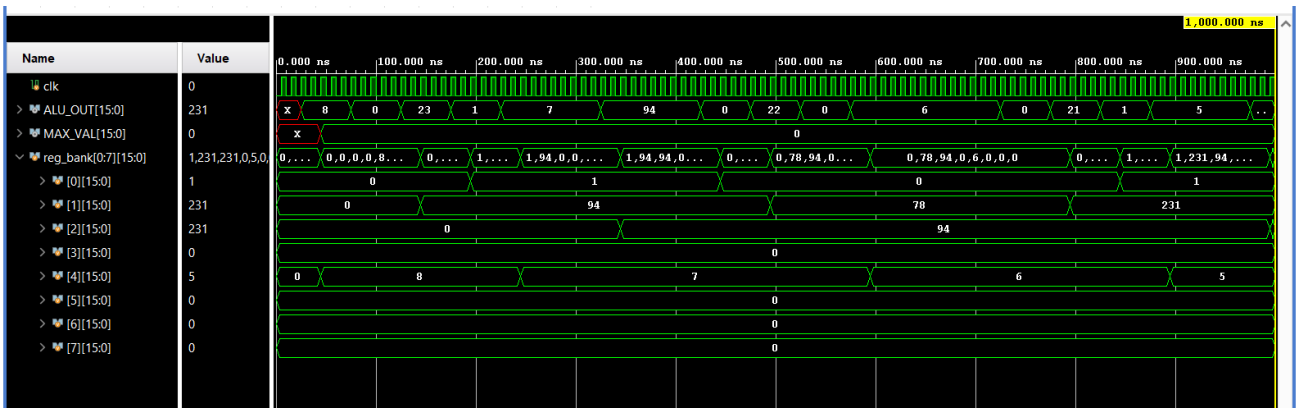
PROGRAM:

```
memory[0] <= 16'b0111_1000_1000_1000; //MOVI R4, #8
memory[1] <= 16'b0111_1000_0000_0000; //MOVI R0, #0    :loop1
memory[2] <= 16'b1010_0001_1000_1111; //LD R1, OS(R4)   OS = 15
memory[3] <= 16'b0101_1010_0010_0000; //SLT R0, R2, R1
memory[4] <= 16'b0100_0100_0001_0000; //DEC R4, R4
memory[5] <= 16'b1001_0000_0000_0010; //BEQZ R0 loop2
memory[6] <= 16'b0101_0001_0000_1000; //MOV R2, R1
memory[7] <= 16'b1001_1100_0001_1010; //BNEQZ R4, loop1   :loop2
memory[8] <= 16'b1011_0000_0000_0000; //Halt
```

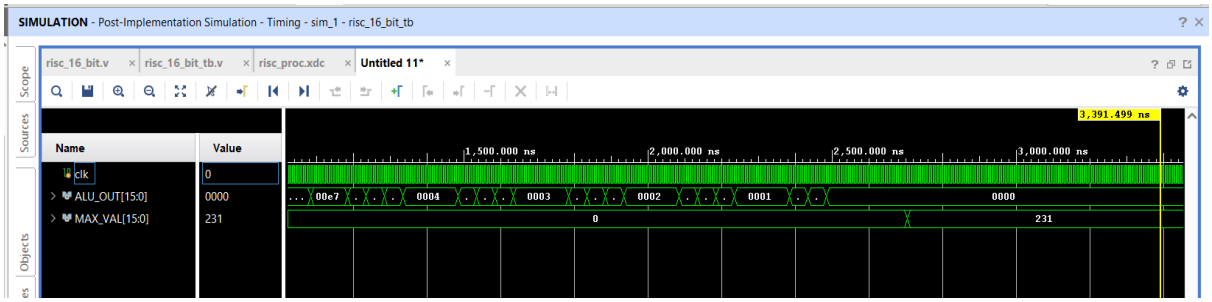
USER DATA:

```
//user data
memory[23] <= 16'd94;
memory[22] <= 16'd78;
memory[21] <= 16'd231; // Maximum of array
memory[20] <= 16'd123;
memory[19] <= 16'd9;
memory[18] <= 16'd14;
memory[17] <= 16'd121;
memory[16] <= 16'd0;
```

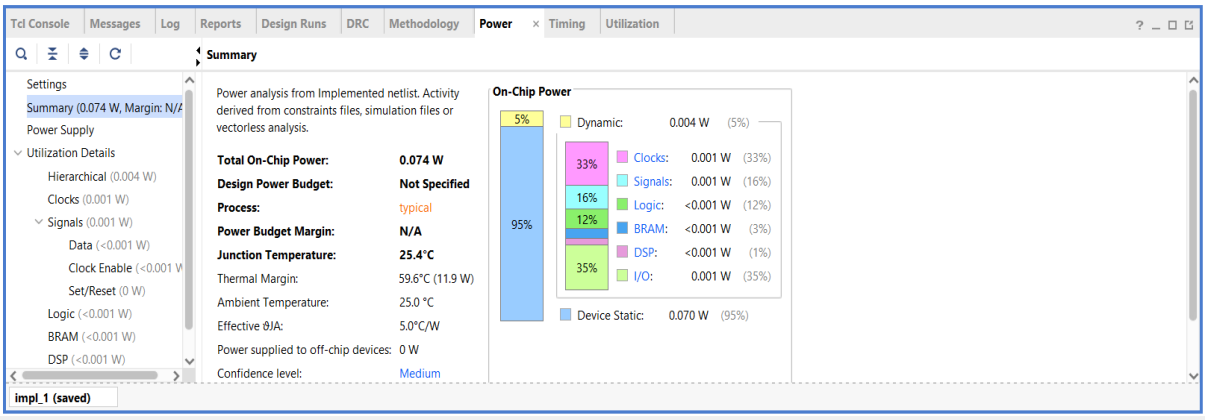
Behavioural simulation waveforms for the above program:



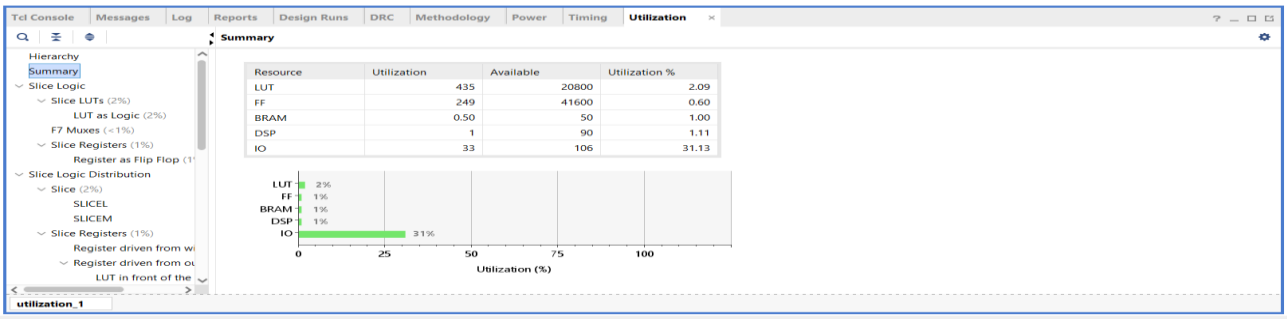
Post implementation synthesis waveforms:



Power Report:



Resource Utilization Report:



6.References:

- SINGLE CYCLE RISC PROCESSOR DESIGN ON FPGA.
- HARDWARE MODELLING USING FPGA.
- VIVADO FORUM PAGE.
- VERILOG TUTORIALS.