

School of Sciences and Engineering

Fall 2023

CSCE-3302: Computer Architecture Course

Project\_1 Milestone\_3 Report

Submitted to:

*Dr. Cherif Salama*

Submitted by:

Name: Fekry Mohamed ID: 900192372  
 Name: Mario Ghaly ID: 900202178

Name: Freddy Amgad ID:

26/11/2023

**Abstract:**

The femtoRV32 project aims to implement a RISC-V processor on the Nexys A7 board, supporting all the RV32I base integer instructions. This implementation features a pipelined architecture with effective hazard handling and utilizes a single memory for both instructions and data. The project includes rigorous testing, covering all 40 instructions and addressing potential hazard scenarios. Additionally, this project supports the integer multiplication and division instructions, along with alternative solutions to mitigate structural hazards.

**Introduction:**

Modern computing systems rely on efficient processor architectures to meet the demands of various applications. The RISC-V instruction set architecture, characterized by its simplicity and modularity, has gained prominence in the field of computer architecture. In this context, the femtoRV32 project seeks to implement a RISC-V processor on the Nexys A7 kit. The project places a strong emphasis on a pipelined architecture, leveraging five stages to execute instructions effectively by introducing an alternative solution to handle the structural hazard. This implementation ensures correct hazard handling, with particular attention to the challenges introduced by using a single-ported memory for both data and instructions. By addressing structural hazards through every-other-cycle instruction issuing, the processor achieves a balance between performance and resource constraints.

**Project Objectives:**

For milestone 3 of the femtoRV32 project, we implemented the RV32I base integer instruction set for all 40 user-level instructions in the unprivileged ISA. The pipelined is supported in this phase by dividing each instruction into 5 cycles with hazard detection unit to eliminate the potential hazard. All the 40 instructions were implemented as specified in the manual except for ECALL and FENCE, which were implemented as no-op instructions, so we did this by outputting the instruction add x0, x0, x0 from the instruction memory in case any of these two instructions is the target instruction. Also EBREAK was interpreted as a halting instruction ending the execution of the program by preventing loading a new instruction.

**The design of the project:**

**Architecture:**

A diagram of a machine

Description automatically generated

**Processor components:**

1. **Control Unit:**

The processor's control unit is the block responsible for generating control signals based on opcode, and function field. Control signals determine various operations within the processor, including ALU operations, memory access, write back operation. Notable control signals include ALUOp, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, Byte, HalfWord, ZeroExtention, PC\_mux\_sel, and WB\_mux\_sel. These signals collectively orchestrate the execution of instructions within the processor. We added additional control signals, such as load byte and load half word operations, that control the load and store operations of the data memory. Furthermore, the control unit is modified to support the rest of instruction types like I-type and J-type.

1. **ALU Control Unit:**

The control unit for the Arithmetic Logic Unit (ALU) generates a 4-bit control signal, "ALU\_Sel" based on inputs such as "ALUOp" and "funct3". The module employs conditional statements to determine the appropriate ALU operation for different instruction formats. We modified the ALU control unit to support the rest of ALU operation especially the R-type and I-type because we implemented only four operations in the lab.

1. **Branch Control Unit:**

The Branch Control Unit uses control signals such as the "Branch" signal, "funct3," and the ALU flags like "sf," "zf," "vf," and "cf" to determine if the branch is taken or not. The module effectively determines whether a branch should be taken based on the specified conditions and flag statuses for all the branching instructions.

1. **ALU:**

The "ALU" is a 32-bit Arithmetic Logic Unit that performs logical and arithmetic operations on two 32-bit inputs. It supports logical operations such as logical AND, OR, XOR. Similarly, it supports arithmetic and shift operations, such as ADD, SLL, and SRL. The ALU is modified to perform the rest of logical and arithmetic operations based on the 4-bit select signal "sel.". Sign, carry, overflow, and zero flags have been added to support controlling the upcoming stages like the branching control.

1. **Memory:**

In this milestone, we used a single memory for both data and instructions. The memory is single ported and byte addressable. We modified the data memory to be byte addressable to allow loading and storing one byte or half words. It is also modified to allow zero extension for loading byte and half word. To handle the structural hazards, we stall the fetching of instruction during the memory stage of any load/store instructions. If the EX/Mem. ReadData/WriteData is asserted, the memory take the address from the ALU\_output. Otherwise, it take the PC output to fetch the following instruction.

1. **Forwarding Unit:**

The Forwarding Unit resolves data hazards by forwarding data from the output of the ALU or memory stage to the input of the ALU of the following instruction. This ensures that data produced by previous instructions can be used immediately by subsequent instructions, mitigating the need to stall the pipeline. The Forwarding Unit monitors data dependencies between instructions and efficiently routes data to the appropriate stages based on the hazard detection signals.

1. **Hazard Detection Unit:**

The Hazard Detection Unit is responsible for identifying potential hazards within the pipeline and signaling the need for corrective actions. It monitors load-use dependencies, and the structural hazards that requires the program to stall for one cycle. The Hazard Detection Unit works in tandem with the forwarding unit to identify situations where instruction execution might be affected due to data dependencies or control flow changes. When a hazard is detected, the Hazard Detection Unit generates control signals to stall the pipeline by setting the control signals of the instruction to zero and stop incrementing the PC value.

1. **Register File:**

The Register File contains 32-bit registers for reading and storing the data during the instructions execution to increase the program speed. It takes inputs such as read addresses, write address, and write data. Its outputs include data read from specified registers by the input addresses.

9. **Immediate Generator:**

The Immediate Generator takes a 32-bit instruction and produces a 32-bit immediate value based on the instruction opcode (OPCODE).

**Test cases:**we have used three test cases to validate the functionality of the different instruction. These test cases are commented in the DataMem.v file attached with this report if you need to verify the provided output.

* **Test\_1:**
* initial begin
* mem[0]=32'b000000000000\_00000\_010\_00001\_0000011 ; //lw x1, 0(x0)
* mem[1]=32'b000000000100\_00000\_010\_00010\_0000011 ; //lw x2, 4(x0)
* mem[2]=32'b000000001000\_00000\_010\_00011\_0000011 ; //lw x3, 8(x0)
* mem[3]=32'b0000000\_00010\_00001\_110\_00100\_0110011 ; //or x4, x1, x2
* mem[4]=32'b0\_000000\_00011\_00100\_000\_0100\_0\_1100011; //beq x4, x3,4
* mem[5]=32'b0000000\_00010\_00001\_000\_00011\_0110011 ; //add x3, x1,x2
* mem[6]=32'b0000000\_00010\_00011\_000\_00101\_0110011 ; //add x5, x3,x2
* mem[7]=32'b0000000\_00101\_00000\_010\_01100\_0100011; //sw x5, 12(x0)
* mem[8]=32'b000000001100\_00000\_010\_00110\_0000011 ; //lw x6, 12(x0)
* mem[9]=32'b0000000\_00001\_00110\_111\_00111\_0110011 ; //and x7, x6,x1
* mem[10]=32'b0100000\_00010\_00001\_000\_01000\_0110011 ; //sub x8,x1,x2
* mem[11]=32'b0000000\_00010\_00001\_000\_00000\_0110011 ; //add x0,x1,x2
* mem[12]=32'b0000000\_00001\_00000\_000\_01001\_0110011 ;//add x9, x0,x1
* mem[13]=32'b00000000010000000000000010010011; //addi x1, x0, 4
* mem[14]=32'b000000000100\_00001\_000\_00001\_0010011; //addi x1, x1, 4
* mem[15] = 32'b00000000000000000001000010110111; //lui x1, 1
* mem[16] = 32'b00000000000000001100000100010111; //auipc x2, 12
* mem[17] = 32'b00000000000100000000000100010011; //addi x2, x0, 1 //x2 = 1
* mem[18] = 32'b000000000011\_00010\_001\_00100\_0010011; //slli x4,x2,3 //x4 = 8
* mem[19] = 32'b0000000\_00010\_00100\_101\_00100\_0110011;//srl x4,x4,x2 //x2 = 4
* mem[20] = 32'b00000000000000000000000001110011; //ECALL
* mem[21] = 32'b00001111111100000000000000001111; //FENCE
* mem[22] = 32'b00000000000100000000000001110011; //EBREAK
* mem[23] = 32'b00000000000100010100000110110011; //xor x3, x2, x1
* mem[24] = 32'b01000000000100011101000110010011; //srai x3, x3, 1
* end
* **The output:**

**A screenshot of a computer

Description automatically generated**

* **Test\_2:**
* //test\_case\_2 --shifting
* initial begin
* mem[0] = 32'b00000000000100000000000010010011; //addi x1, x0, 1
* //x2 = 1
* mem[1] = 32'b00000000000100000000000100010011; //addi x2, x0, 1
* //x2 = 1
* mem[2] = 32'b00000000000100010001000100010011; //slli x2, x2, 1
* //x2 = 2
* mem[3] = 32'b00000000000100010101000100010011; //srli x2, x2, 1
* //x2 = 1
* mem[4] = 32'b00000000001000010001000100110011; //sll x2, x2, x2
* //x2 = 2
* mem[5] = 32'b00000000001000010001000100110011; //sll x2, x2, x2
* //x2 = 8
* mem[6] = 32'b01000000000100010101000100110011; //sra x2, x1, x2
* //x2 = 4
* end
* **The output:**

**A screenshot of a computer

Description automatically generated**

* **Test\_3:**
* //test\_case\_3 --loading and branching
* initial begin
* mem[0]=32'b000000001100\_00000\_010\_00001\_0000011 ; //lw x1, 12(x0) //9abcdeff
* mem[1]=32'b000000001100\_00000\_000\_00001\_0000011 ; //lb x1, 13(x0) //ffffffff
* mem[2]=32'b000000001100\_00000\_100\_00001\_0000011 ; //lbu x1, 13(x0) //000000ff
* mem[3]=32'b000000001100\_00000\_001\_00001\_0000011 ; //lh x1, 12(x0) //ffffdeff
* mem[4]=32'b000000001100\_00000\_101\_00010\_0000011 ; //lhu x2, 12(x0) //0000deff
* mem[5]=32'b0\_000000\_00010\_00001\_000\_0100\_0\_1100011; //beq x1,x2,4
* //false
* mem[6]=32'b0\_000000\_00010\_00001\_110\_0100\_0\_1100011; //bltu x1,x2,4
* //false
* mem[7]=32'b0\_000000\_00010\_00001\_001\_0100\_0\_1100011; //bne x1,x2,4
* //true
* mem[8]=32'b0000000\_00010\_00001\_110\_00100\_0110011 ;//or x4,x1,x2
* //skipped
* mem[9]=32'b0\_000000\_00010\_00001\_111\_0100\_0\_1100011;//bgeu x1,x2,4
* #true
* end
* **The output:**

**A screenshot of a computer

Description automatically generated**

**Faced difficulties:**

We did not face many difficulties in this project. The main one was recognizing the optimum and simplest way to implement the additional branch and load instruction. However, Dr. Cherif Salama provided a supporting lecture for the project that addressed this issue that helped us implement the branch control block.

**Conclusion:**

In conclusion, for Milestone #3 of the femtoRV32 project, we successfully implemented the RV32I base integer instruction set, covering all 40 user-level instructions within the unprivileged ISA. The implemented design features a pipelined architecture, divided into five stages, and addresses potential hazards through a Hazard Detection Unit and a Forwarding Unit. Throughout the implementation, we have adhered to the principles of the RISC-V instruction set architecture, emphasizing simplicity, modularity, and effective hazard handling. Rigorous testing has been conducted, covering all 40 instructions and addressing potential hazard scenarios. Additionally, the processor supports integer multiplication and division instructions, showcasing its versatility.