THE AMERICAN UNIVERSITY IN CAIRO

100 YEARS

CSCE3301 - Computer Architecture

RISC-V Processor

Names: Jana Elfeky, Kareem Elnaghy

Dr. Cherif Salama

Introduction

This project aims to design and implement a processor supporting the RV32I base integer instruction set. The final implementation was tested on the Nexys A7 trainer kit, allowing it to execute various programs by adhering to RISC-V specifications. The project includes handling hazards in a pipelined design, using single-ported memory, and addressing structural hazards with every-other-cycle instruction issuing.

For this milestone, we focused on building a pipelined datapath that supports the RV32I base integer instruction set and uses one single ported memory for both data and instructions. This processor is designed to execute all 42 user-level RV32I instructions, except ECALL, EBREAK, PAUSE, FENCE, and FENCE.TSO, where ECALL is instead interpreted as halting the program and the others as no operations.

We created a block diagram detailing the pipelined datapath, outlining all the components and the interactions between them. After that, we implemented the design using Verilog, of which the structure of the implementation is explained below in detail.

Bonuses

1. We created a program generator using C++ to automatically produce random but valid test instructions for the processor. This generator helps us efficiently test all RV32I instructions by generating diverse instruction sequences while ensuring that memory offsets stay within valid ranges. This way, we could confirm the processor handles different scenarios reliably without causing memory errors.

2. We implemented our RISC-V datapath directly on the Nexys A7 FPGA board. This allowed us to test the functionality of our pipelined processor in a real hardware environment, ensuring that all RV32I instructions were correctly executed. By using the actual FPGA resources, we were able to verify the performance and accuracy of our processor, including proper handling of data hazards and memory accesses. The board's feedback and real-time execution helped us identify any potential issues and fine-tune the implementation.

Structure

The **topModule** module is a top module which connects a pipelined RISC-V CPU with a seven-segment display driver, enabling real-time output display. The CPU processes data using \`clock\`, and its output is sent to LEDs and the display via \`ledSel\` and \`ssdSel\`. A 13-bit wire (\`ssd\`) links the CPU to the display driver, which refreshes segments using \`SSD_clock\` to show updated values on the seven-segment display. The pipelined cpu module implements a datapath as shown in the diagram below.

Pipelined CPU
The \`pipelinedCPU\` module is a pipelined RISC-V processor where each submodule serves a distinct role in processing instructions based on which stage they are in:

1. **Fetching Stage:**
    a. **register #(32) PC:** A 32-bit Program Counter that holds the current instruction address.
    b. **RCA #(32) rca:** An adder that increments the PC by 4 to point to the next instruction.
    c. **register #(96) IF_ID:** Register which stores the information needed from the fetching stage and outputs it to the decoding stage.

2. **Decoding Stage:**
    a. **control_unit CU:** Generates control signals to guide data flow and operation based on the instruction opcode.
    b. **registerFile #(32) RF:** A Register File providing read and write access to CPU registers.
    c. **ImmGen immgen:** Immediate Generator, producing the immediate value from the instruction and shifting the value based on the opcode of the instruction.
    d. **N_bit_mux #(8) controlMUX:** Based on the branchOutcome signal, the mux chooses between the control signals generated by the control unit or it flushes the register and makes the control signals 0’s.
    e. **register #(191) ID_EX:** Register which stores the information needed from the decoding stage and outputs it to the execution stage.

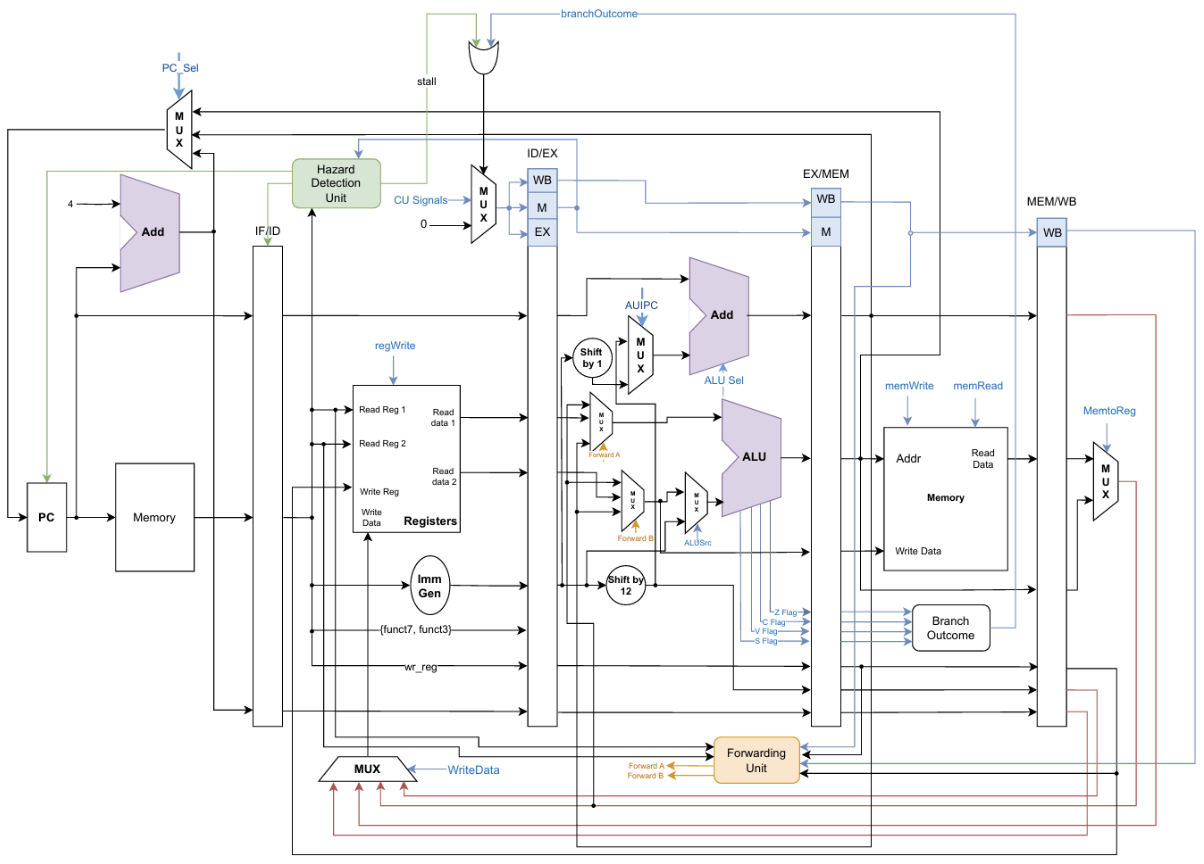
3. **Execution Stage:**
    a. **N_bit_mux_4x1 #(32) aluMUXA:** A multiplexer selecting the first input to the ALU based on whether or not the instruction is forwarded. This decision is made by the forwarding unit signal forwardA which we will discuss later.
    b. **N_bit_mux_4x1 #(32) aluMUXB:** A multiplexer selecting the second input to the ALU based on whether or not the instruction is forwarded. This decision is made by the forwarding unit signal forwardB which we will discuss later.
    c. **N_bit_mux #(32) aluMUX:** A multiplexer selecting between register data and the immediate value as input to the ALU.
    d. **ALU_control aluCU:** Determines the ALU operation type based on control signals and instruction type.
    e. **N_bit_ALU #(32) alu:** Executes arithmetic and logical operations.
    f. **register #(174) EX_MEM:** Register which stores the information needed from the execution stage and outputs it to the memory stage.

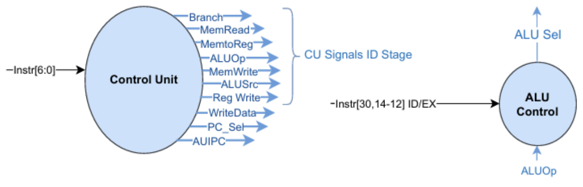
4. **Memory Stage:**
    a. **Memory:** Single-ported memory to fetch instruction on the positive edge of the clock and the data on the negative edge of the clock.
    b. **branchOutcome branch:** Produces a branchOutcome signal based on whether or not the branch will be taken.
    c. **register #(71) MEM_WB:** Register which stores the information needed from the memory stage and outputs it to the write-back stage.

5. **Write-Back Stage:**
    a. **N_bit_mux #(32) aluDataMUX:** MUX choosing between ALU result and data memory output.
    b. **N_bit_mux_4x1 writeDataMUX:** MUX choosing what value will be written to the register based on which instruction is executed.

6. **Other Units:**
    a. **forwardingUnit FU:** Generates forwardA and forwardB signals to control the forwarding mechanism from the memory stage.

Datapath


branchOutcome
stall
PC_Sel
M U X
Hazard Detection Unit
CU Signals
0
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
4
Add
IF/ID
AUIPC
Add
Shift by 1
ALU Sel
regWrite
Read Reg 1
Read Reg 2
Write Reg
Write Data
Read data 1
Read data 2
Registers
Forward A
Forward B
ALUSrc
ALU
memWrite
memRead
MemtoReg
Addr
Read Data
Memory
Write Data
PC
Memory
Imm Gen
Shift by 12
{funct7, funct3}
wr_reg
Z Flag
C Flag
V Flag
S Flag
Branch Outcome
Forwarding Unit
MUX
WriteData


Instr[6:0]
Control Unit
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
Reg Write
WriteData
PC_Sel
AUIPC
CU Signals ID Stage
Instr[30,14-12] ID/EX
ALU Control
ALU Sel
ALUOp

Key components of the pipelined datapath for the RISC-V processor include:

1. **Memory**: Single port memory which holds the program instructions and register values, with the clock directing which object is fetched.
2. **Program Counter (PC)**: Determines the current instruction address. After each instruction is executed, the PC increments to fetch the next instruction or halts if ECALL is encountered.
3. **Control Unit**: Decodes the fetched instruction and generates control signals to direct the flow of data, select ALU operations, and coordinate memory and register interactions.
4. **ALU (Arithmetic Logic Unit)**: Performs arithmetic and logical operations based on the control unit's signals. Inputs come from the register file or the immediate generator, depending on the instruction type.
5. **ALU Control**: Receives signal from the control unit and further decodes it to specify the exact operation the ALU should perform.
6. **Immediate Generator**: Extracts and sign-extends immediate values from the instruction for operations needing constants, such as load/store and branch instructions.
7. **Register File**: Stores all general-purpose registers. It reads the required registers for an instruction and writes back results after ALU or memory operations.
8. **Stage Registers:** Store all information from the current stage needed by the next stage of the pipeline.
9. **Multiplexers (MUXes)**: These guide data flow by selecting between inputs (e.g., PC + 4 or branch target for the next PC, or ALU vs. immediate input to the ALU).

Test Cases

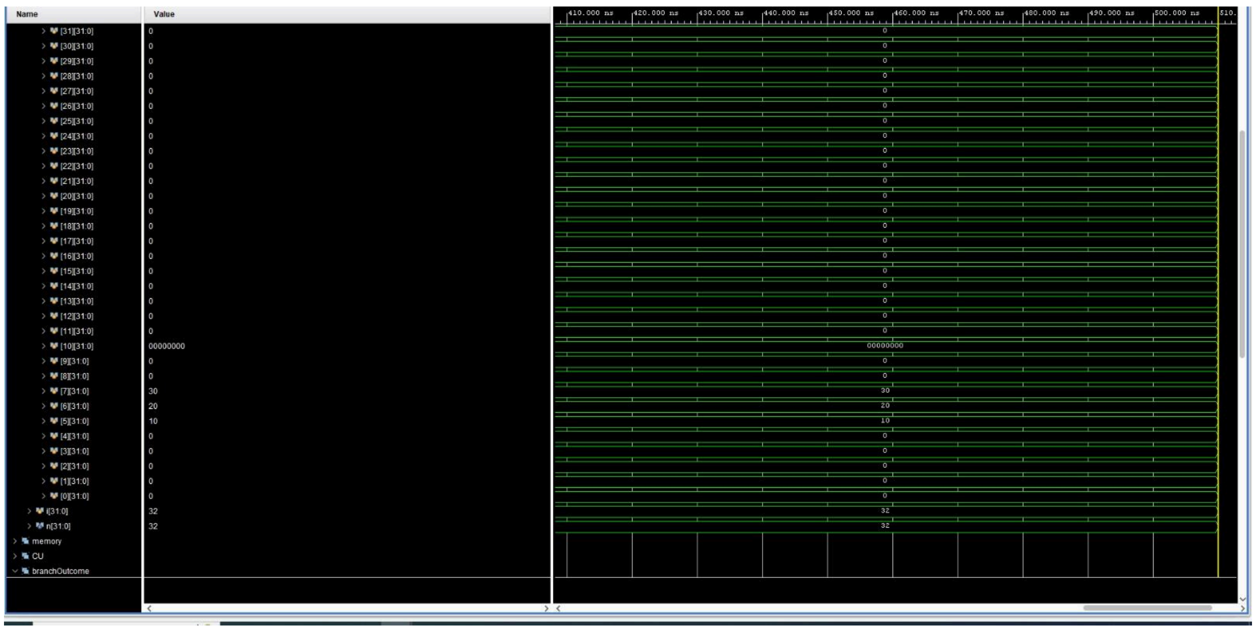
Test Case 1:

Program:

```
00a00293 //li x5, 10
01400313 //li x6, 20
006283b3 //add x7, x5, x6
0013f413 //andi x8, x7, 1
00041463 //bne x8, x0, 8
0074a023 //sw x7, 0(x9)
00000073 //ecall
```

Expected final output:

x5 = 10
x6 = 20
x7 = 30
x8 = 0
x9 = 30



```
Name              Value
  [31][31:0]      0
  [30][31:0]      0
  [29][31:0]      0
  [28][31:0]      0
  [27][31:0]      0
  [26][31:0]      0
  [25][31:0]      0
  [24][31:0]      0
  [23][31:0]      0
  [22][31:0]      0
  [21][31:0]      0
  [20][31:0]      0
  [19][31:0]      0
  [18][31:0]      0
  [17][31:0]      0
  [16][31:0]      0
  [15][31:0]      0
  [14][31:0]      0
  [13][31:0]      0
  [12][31:0]      0
  [11][31:0]      0
  [10][31:0]      00000000
  [9][31:0]       0
  [8][31:0]       0
  [7][31:0]       30
  [6][31:0]       20
  [5][31:0]       10
  [4][31:0]       0
  [3][31:0]       0
  [2][31:0]       0
  [1][31:0]       0
  [0][31:0]       0
 i[31:0]          32
 n[31:0]          32
memory
CU
branchOutcome
```

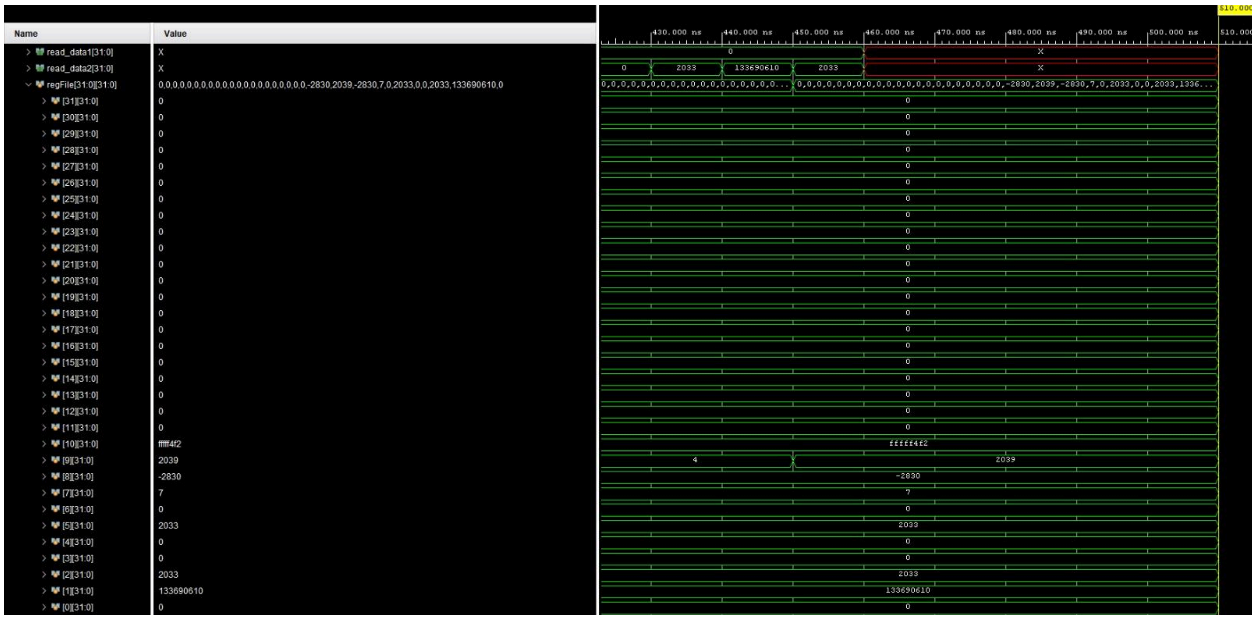
Test Case 2:

Program:

07f7f0b7 //lui x1, 32639
4f208093 //addi x1, x1, 1266
7f100113 //addi x2, x0, 2033
00400493 //addi x9, x0, 4
00102023 //sw x1, 0(x0)
0024a023 //sw x2, 0(x9)
00402283 //lw x5, 4(x0)
00600303 //lb x6, 6(x0)
00504383 //lbu x7, 5(x0)
00001403 //lh x8, 0(x0)
00205483 //lhu x9, 2(x0)
00101123 //sh x1, 2(x0)
00201503 //lh x10, 2(x0)

Expected final output:

x1 = 133690610
x2 = 2033
x9 = 4
x5 = 2033
x6 = 0
x7 = 7
x8 = -2830
x9 = 2039
x10 = 4294964466



Name	Value
read_data1[31:0]	X
read_data2[31:0]	X
regFile[31:0][31:0]	0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-2830,2039,-2830,7,0,2033,0,0,2033,133690610,0
[31][31:0]	0
[30][31:0]	0
[29][31:0]	0
[28][31:0]	0
[27][31:0]	0
[26][31:0]	0
[25][31:0]	0
[24][31:0]	0
[23][31:0]	0
[22][31:0]	0
[21][31:0]	0
[20][31:0]	0
[19][31:0]	0
[18][31:0]	0
[17][31:0]	0
[16][31:0]	0
[15][31:0]	0
[14][31:0]	0
[13][31:0]	0
[12][31:0]	0
[11][31:0]	0
[10][31:0]	fffff4f2
[9][31:0]	2039
[8][31:0]	-2830
[7][31:0]	7
[6][31:0]	0
[5][31:0]	2033
[4][31:0]	0
[3][31:0]	0
[2][31:0]	2033
[1][31:0]	133690610
[0][31:0]	0

Test Case 3: (generated by random program generator)

Initial data values:

mem [2053] = 4
mem [2056] = 1000
mem [2057] = 200
mem [2058] = 10
mem [2063] = 20
mem [2069] = 1500
mem [2078] = 40

Program:

```
0c902a23 //sw x9, 212(x0)
03668a17 //auipc x20, 13928
00f00fb3 //add x31, x0, x15
0a02ae23 //sw x0, 188(x5)
40af0d33 //sub x26, x30, x10
1c898623 //sb x8, 460(x19)
0252d1b7 //lui x3, 9517
ec0a8223 //sb x0, -316(x21)
1200096f //jal x18, 288
0125d717 //auipc x14, 4701
```

Expected final output:

mem [2260] = 200
mem [2068] = 425984
mem [2079] = 20
mem [2240] = 0
mem [2074] = 30
mem [2056] = 232
mem [2051] = 184320
mem [3232] = 0
mem [2066] = 36
PC = 320

Problems & Solutions

While working on this project, we ran into multiple issues:

- We had a problem with the single-ported memory due to incorrect wire declarations, which caused incorrect data access during the instruction fetch and memory stages.
    - Solution: We identified and corrected the wire declarations to ensure data flowed correctly through the memory pipeline.

- We also had a problem with the jump instruction, where the branch outcome was not evaluated as expected, leading to incorrect jump behavior.
    - Solution: We revised the branch condition logic to correctly calculate the branch outcome, allowing accurate execution of jump instructions.

Release Notes

This pipelined RISC-V processor strictly adheres to the RV32I specification, with no additional assumptions or deviations from the standard. All RV32I instructions have been implemented and thoroughly verified, ensuring correct execution and state updates across registers and memory in a pipelined fashion. Certain instructions, specifically `EBREAK`, `PAUSE`, `FENCE`, and `FENCE.TSO`, are currently defined as no-operations (NOPs). The development and testing processes revealed no issues, resulting in a fully operational and reliable implementation.