IEEE
CAIRO UNIVERSITY STUDENT BRANCH
IEEE
CUSB

(Final Project)

Digital Workshop
2024 / 2025

IEEE

Single Cycle RV-32I Processor

Introduction

In this project, you are required to implement a 32-bit single-cycle microarchitecture RISC-V processor based on Harvard Architecture. The single-cycle microarchitecture executes an entire instruction in one cycle. In other words, instruction fetch, instruction decode, execute, write back, and program counter update occurs within a single clock cycle.

Objective

Referring to figure one, you are required to write the RTL Verilog files for all sub-modules of the RISC-V processor (e.g. Register File, Instruction Memory, etc.). Then, implementing the top module of the RISC-V processor. Finally, you will configure this processor on Cyclone® IV FPGA device.

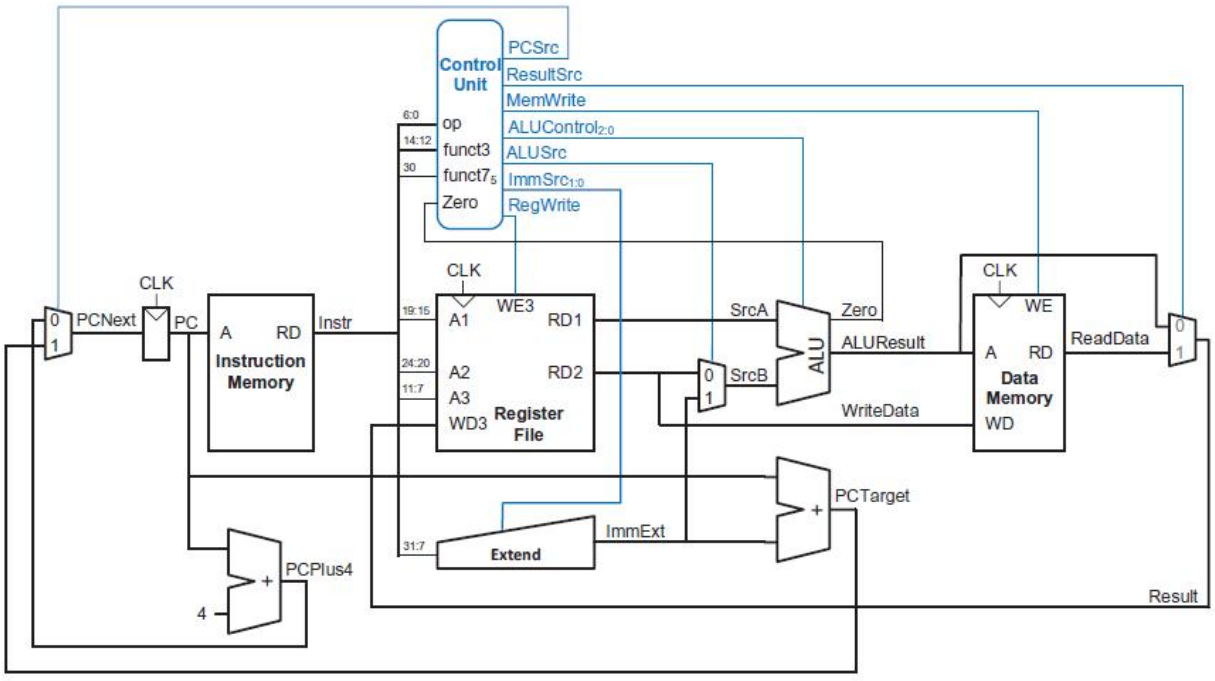
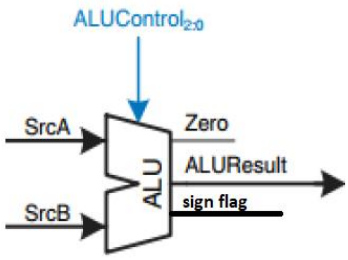


Figure 1: Complete single-cycle RISC-V processor

Main Modules

1. ALU

An Arithmetic/Logical Unit (ALU) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations. The ALU forms the heart of most computer systems. The 3-bit ALUControl signal specifies the operation. The ALU generates a **32-bit ALUResult, a Zero flag** that indicates whether ALUResult == 0, **and a sign flag** that indicates ALU result sign (ALUResult [31]). Thefollowing table lists the specified functions that our ALU can perform.



ALUControl	Function
A + B	000
A SHL B	001
A - B	010
A XOR B	100
A SHR B	101
A OR B	110
A AND B	111

Note:

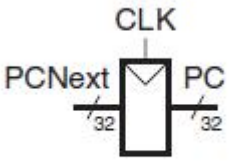
- **ALU** is combinational block that doesn't use clock signal.
- **Don't forget** to make the default of ALU result to be 0.

2.Program Counter

2.1. Program Counter Register

To fetch the instructions from the instruction memory, we need a **pointer** to keep track of the address of the current instruction for this task we use the program counter. The program counter is simply a **32-bit register** that has the address of the current instruction at its output and the address of the next instruction at its input. Firstly, you need to implement this register and then the logic that calculates the address of the next instruction. The program counter has four inputs a **32-bit word** which is the next address, the **clock** signal, **asynchronous reset**, and a **load** signal (always high except for the **HLT** instruction). And have one **32-bit output PC**.

The following truth table describes the behavior of this register.



areset	load	clk	PC
0	**x**	**x**	**0**
1	**0**	**Posedge**	**PC**
1	**1**	**Posedge**	**PCNext**
1	**x**	**Not posedge**	**PC**

2.2. Next PC calculation logic

Now we need to implement the logic that calculates the address of the next instruction. Normally the address of the next instruction is **PC + 4**, but in the case of a **taken branch** the address of the next instruction is **PC + target**. The circuit that handles this consists of 3 elements, a **2x1 multiplexer** and **two binary adders** that have the current value of PC at one of its operands as you can see in figure one. This circuit has two inputs a **32-bit** number **ImmExt** that is coming from the **sign extend unit** (target address for branch instructions) and **PCSrc** signal to select the right source for next PC. And has one **32-bit** output which is the **next PC**.

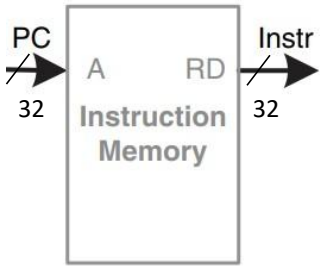
The following truth table describes the behavior of this circuit.

PCSrc	PCNext
0	**PC + 4**
1	**PC + ImmExt**

Note: you can implement the **PC register** and the **calculation circuit** in one module if you want.

3. Instruction memory

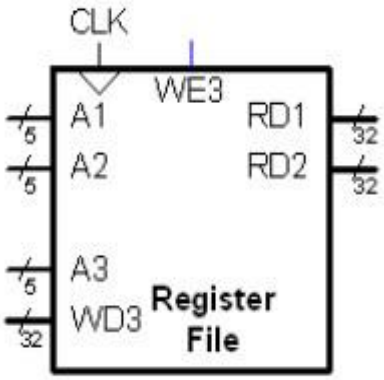
- The instruction memory has a single read port.
- It takes a 32-bit instruction address input, A, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, RD.
- The PC is simply connected to the address input of the instruction memory.
- The instruction memory reads out, or fetches, the 32-bit instruction, labeled Instr.
- Our instruction memory is a Read Only Memory (ROM) that holds the program that your CPU will execute.
- The ROM Memory has width = 32 bits and depth = 64 entries.
- Instructions is read asynchronously.



NOTE: instruction memory is word aligned (you need only to give it address [31:2])

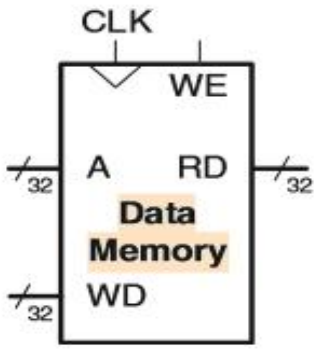
4. Register File

- The Register File contains the 32-bit registers.
- The register file has two read output ports (RD1 and RD2) and a single input write port (WD3), RD1 and RD2 are read with no respect to the clock edge.
- The register file is read asynchronously and written synchronously at the rising edge of the clock.
- The register file supports simultaneous read and writes. The register file has width = 32 bits and depth = 32 entries supports simultaneous read and writes.
- The register file has active low asynchronous reset signal.
- A1 is the register address from which the data are read through the output port RD1. Whereas A2 is corresponding to the register address of output port RD2.



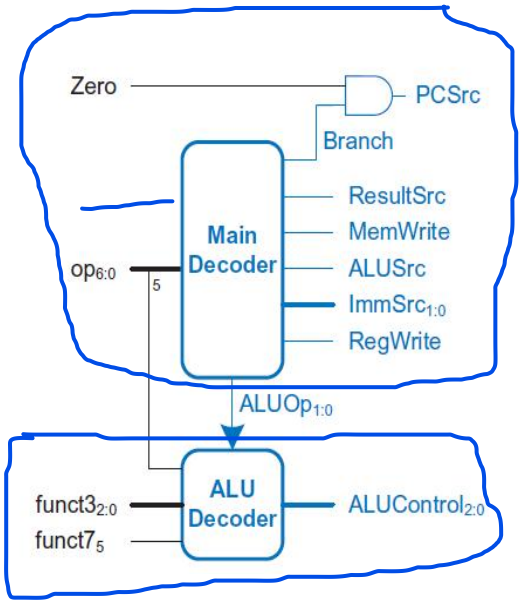
5. Data Memory

- It has a single read/write port.
- If its write enable, WE, is asserted, then it writes data WD into address A on the rising edge of the clock.
- It reads are asynchronous while writes are synchronous to the rising edge of the "clk" signal.
- The Word width of the data memory is 32-bits to match the datapath width. The data memory contains 64 entries.
- RD is read with no respect to the clock edge.
- A is the memory address from which the data are read through the output port RD.



6.Control Unit

The control unit computes the control signals based on the opcode and funct3, funct7 fields ofthe instruction, Instr14:12 and Instr30 respectively. Most of the control information comes from the opcode, but R-type instructions and I-type instructions also use the funct3 and funct7 fields to determine the ALU operation. Thus, we will simplify our design by factoring the control unit into two blocks of combinational logic, as shown in the figure below.



ALU Decoder truth table

ALUOP	$funct_3$	$\{op_5, funct_7\}$	ALUcontrol	Instruction
00	XXX	XX	000(add)	lw,sw
01	000	XX	010(sub)	beq
	001	XX	010(sub)	bnq
	100	XX	010(sub)	blt
10	000	00,01,10	000(add)	add
	000	11	010(sub)	subtract
	001	XX	001(shift left)	SHL
	100	XX	100(XOR)	XOR
	101	XX	101(shift right)	SHR
	110	XX	110(OR)	OR
	111	XX	111(AND)	AND
Default	XXX	XX	000	_____

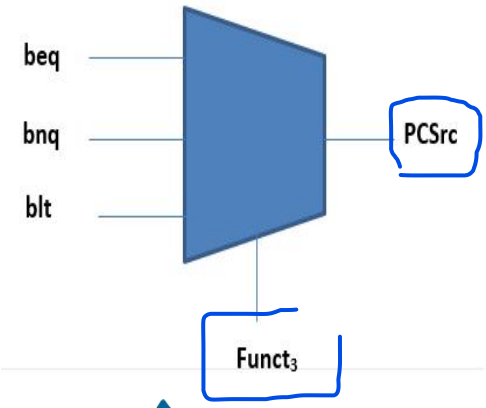
7.Main decoder truth table

Note: All values are written in binary

Opcode	RegWrite	$ImmSrc_{1:0}$	ALUSrc	MemWrite	ResultSrc	Branch	$ALUOp_{1:0}$
loadWord = 7'b000_0011	1	00	1	0	1	0	00
storeWord = 7'b010_0011	0	01	1	1	X	0	00
R-Type = 7'b011_0011	1	XX	0	0	0	0	10
I-type = 7'b001_0011	1	00	1	0	0	0	10
Branch-instructions = 7'b1100011	0	10	0	0	X	1	01
Default	0	00	0	0	0	0	0

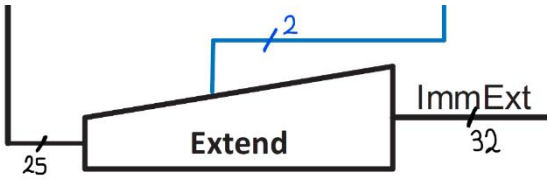
Hints:

- In the control unit figure above, to implement beq instruction we will ANDing zero flag with branch signal to branch when the 2 operands given is zero and to implement other branches instruction:
  - AND (~zero flag) with branch signal (branch not equal (bnq))
  - AND (sign flag) with branch signal (branch less than(blt))
  - Use case statement to select one of this operations according to the instruction given. (**Hint**: MUX (case statement) selection will be $funct_3$)



Small modules

1. Sign extend



Sign extension simply copies the sign bit (most significant bit) of a short input (16 bits) into all the upper bits of the longer output (32 bits).

→example

- Inst [15:0] = **0**000 0000 0011 1101

  ImmExt = **0000 0000 0000 0000 0**000 0000 0011 1101

- Inst [15:0] = **1**000 0000 0011 1101

Table 7.1 ***ImmSrc*** **encoding**

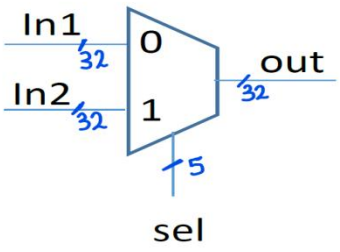
ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate

ImmExt = **1111 1111 1111 1111 1**000 0000 0011 1101

Here in our design, we use extender at different modes according to the type of the instruction to extract immediate value correctly and the following table shows that:

2. MUX

You are required to implement a 2X1 MUX that has 32-bit input and output.



Final results and simulations

After finishing your design take the following program and read it in your instruction memory then simulate your design. Also, you have a snapshot of the right simulation results, make sure that your results are identical to the delivered results here.

FIBONACCI series

Machine code

```
00004033
00000093
00100113
00100193
00100213
00000293
00a00313
00000393
00418c63
00110133
404181b3
00229393
0023a023
00420a63
002080b3
004181b3
00229393
0013a023
00128293
fc62cae3
00000000
```

C-code

```
#include <iostream>
using namespace std;
int main()
  { int x = 0;
   int y = 1;
   int sel = 1;
   for (int i = 0; i < 10; i++)
   {
      if (sel == 1)
        { x = x + y;
         sel = 2;
       cout << x << endl;  }
        else
       {  y = x + y;
         sel = 1;
      cout << y << endl;}
  }
}
```

Assembly code

```
0:  xor  x0,x0,x0      # reference register, always = 0
4:  addi x1,x0,0       # data register (containing one of the last two values in FIBONACCI series)
8:  addi x2,x0,1       # data register (containing the other value)
12: addi x3,x0,1       # selector (select the oldest register from R1 and R2 that had been refreshed)
16: addi x4,x0,1       # always constant and = 1
20: addi x5,x0,0       # loop counter
24: addi x6,x0,10      # total number of loops
28: addi x7,x0,0       # address of the data memory that will receive the latest evaluated result at
loop:
32: beq  x3,x4,eq      # you can understand the code well from the attached c-code
36: add  x2,x2,x1
40: sub  x3,x3,x4
44: slli x7,x5,2
48: sw   x2,0(x7)
52: beq  x4,x4,endloop
eq:
56: add  x1,x1,x2
60: add  x3,x3,x4
64: slli x7,x5,2
68: sw   x1,0(x7)
endloop:
72: addi x5,x5,1
76: blt  x5,x6,loop
80: halt               # you can make it to prevent PC from increasing after the code had been finished
                       # But it is optional
```

Simulation Result