

CE-321L/CS-330L: Computer Architecture

5-State Pipelined Processor

Ahmed Khalid, Salman Adnan, Ibrahim Rana

30th April 2024

**Contents:**

1. Introduction
2. Task 1 – Sorting Algorithm (Single Cycled)
   1. Bubble Sort Implementation
   2. Simulation
3. Changes to the Single Cycled Processor
4. **Introduction**

Project Overview: Implementing a pipelined processor for array sorting

The purpose of this project is to implement array sorting with a 5-stage pipelined processor written using the RISC-V assembly language. The software used is Verilog HDL.

Goal:

This project's primary goal is to gradually improve processor architecture. It starts with building a 5-stage pipelined processor and progresses to implementing a sorting algorithm in RISC-V assembly on a single-cycle CPU. The sorting algorithm should be effectively executed by this enhanced processor in a significantly shorter amount of time. Achieving functional implementation is not the only goal; data, control, and structural dangers present in pipelined designs must also be addressed and minimized. This guarantees that the array sorting method will run smoothly.

Project Structure:

* We start by using a traditional single-cycle processor to lay the foundation.
* The architecture is then improved by switching to a five-stage pipeline and optimizing it for faster execution.
* Every phase of the project's development, following the given instructions, will be recorded in the report.

1. **Task 1 – Sorting Algorithm**

Bubble Sort Code: (This is implemented within the Instruction Memory)

module InstructionMemory(

input [63:0] Inst\_Address,

output reg [31:0] Instruction

);

reg [7:0] im [95:0];

initial

begin

im[0] = 8'h63;//blt x22, x8, CompareElements #8

im[1] = 8'h44;

im[2] = 8'h8B;

im[3] = 8'h00;

im[4] = 8'h63;// beq x0, x0, EndBubbleSort #92

im[5] = 8'h0E;

im[6] = 8'h00;

im[7] = 8'h04;

im[8] = 8'h93;//CompareElements:

//addi x23, x22, 0

im[9] = 8'h0B;

im[10] = 8'h0B;

im[11] = 8'h00;

im[12] = 8'h93;//addi x29, x28, 0

im[13] = 8'h0E;

im[14] = 8'h0E;

im[15] = 8'h00;

im[16] = 8'h63;//blt x23, x8, SwapElements #8

im[17] = 8'hC4;

im[18] = 8'h8B;

im[19] = 8'h00;

im[20] = 8'hE3;//beq x0, x0, BubbleSort # -20

im[21] = 8'h06;

im[22] = 8'h00;

im[23] = 8'hFE;

im[24] = 8'h03;//SwapElements:

// lw x12, 0(x28)

im[25] = 8'h36;

im[26] = 8'h0E;

im[27] = 8'h00;

im[28] = 8'h83;//lw x13, 0(x29)

im[29] = 8'hB6;

im[30] = 8'h0E;

im[31] = 8'h00;

im[32] = 8'h63;// blt x12, x13, IncrementIndex #28

im[33] = 8'h4E;

im[34] = 8'hD6;

im[35] = 8'h00;

im[36] = 8'h93;//addi x23, x23, 1

im[37] = 8'h8B;

im[38] = 8'h1B;

im[39] = 8'h00;

im[40] = 8'h93;// addi x29, x29, 8

im[41] = 8'h8E;

im[42] = 8'h8E;

im[43] = 8'h00;

im[44] = 8'hE3;// blt x23, x8, SwapElements # -20

im[45] = 8'hC6;

im[46] = 8'h8B;

im[47] = 8'hFE;

im[48] = 8'h13;//addi x22, x22, 1

im[49] = 8'h0B;

im[50] = 8'h1B;

im[51] = 8'h00;

im[52] = 8'h13;//addi x28, x28, 8

im[53] = 8'h0E;

im[54] = 8'h8E;

im[55] = 8'h00;

im[56] = 8'hE3;//beq x0, x0, BubbleSort

im[57] = 8'h04;

im[58] = 8'h00;

im[59] = 8'hFC;

im[60] = 8'hB3;//IncrementIndex:

// add x5, x12, x0

im[61] = 8'h02;

im[62] = 8'h06;

im[63] = 8'h00;

im[64] = 8'h23;//sw x13, 0(x28)

im[65] = 8'h30;

im[66] = 8'hDE;

im[67] = 8'h00;

im[68] = 8'h23;//sw x5, 0(x29)

im[69] = 8'hB0;

im[70] = 8'h5E;

im[71] = 8'h00;

im[72] = 8'h93;//addi x23, x23, 1

im[73] = 8'h8B;

im[74] = 8'h1B;

im[75] = 8'h00;

im[76] = 8'h93;//addi x29, x29, 8

im[77] = 8'h8E;

im[78] = 8'h8E;

im[79] = 8'h00;

im[80] = 8'hE3;//blt x23, x8, SwapElements

im[81] = 8'hC4;

im[82] = 8'h8B;

im[83] = 8'hFC;

im[84] = 8'h13;// addi x22, x22, 1

im[85] = 8'h0B;

im[86] = 8'h1B;

im[87] = 8'h00;

im[88] = 8'h13;// addi x28, x28, 8

im[89] = 8'h0E;

im[90] = 8'h8E;

im[91] = 8'h00;

im[92] = 8'hE3;//beq x0, x0, BubbleSort

im[93] = 8'h02;

im[94] = 8'h00;

im[95] = 8'hFA;

end//EndBubbleSort:

always @(Inst\_Address)// making one instruction from 4 parts of im (instruction memory)

begin

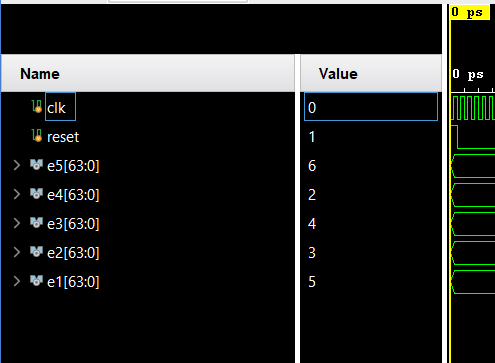
Instruction = {im[Inst\_Address + 3], im[Inst\_Address + 2], im[Inst\_Address + 1], im[Inst\_Address]};

end

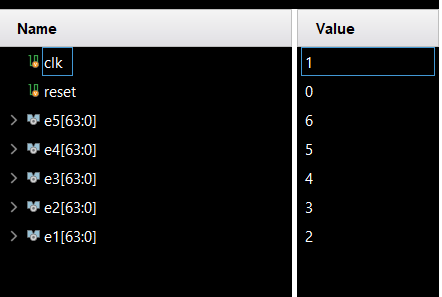
endmodule

**2.2. Simulation**

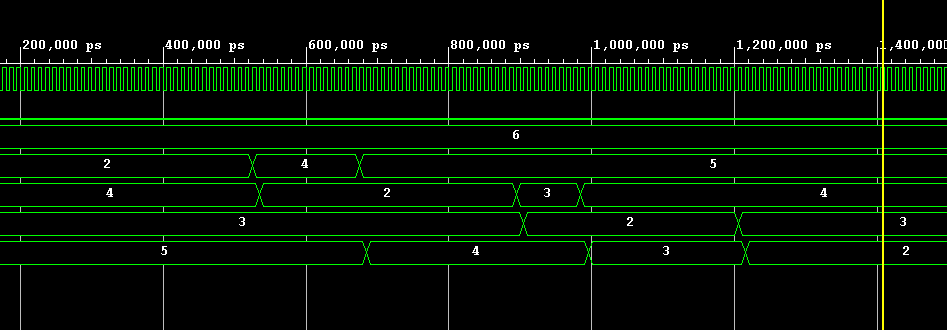
**Intial:**

****

**Final:**



**Process:**

****

1. **Changes To the Single Cycled Processor**

**3.1. ALU**

module ALU64bit(

input [63:0] A,

input [63:0] B,

input [3:0] ALUOp,

output reg [63:0] Result,

output Zero //output for zero result

)

always @(\*) begin

case (ALUOp)

4'b0000: Result = A & B; //AND

4'b0001: Result = A | B; //OR

4'b0010: Result = A + B; //ADD

4'b0110: Result = A - B; //SUB

4'b1100: Result = ~(A | B); //NOR

4'b0100: Result = (A < B) ? 0 : 1; // Lesser than comparison

default: Result = 0; //Default case for unsupported ALUOP values

endcase

end

assign Zero = (Result == 0); // checks if the result is zero.

endmodule

Explanation: We made a couple of changes here, firstly handling branch instructions where when comparing two values if one is smaller we set it to zero, mimicking the BEQ instruction. This was done through the mux with a selection line branch and zero. If branch is zero, we update PC + 4 directly, else if zero is high indicating a condition then the branch target replaces the PC. Executing the branch operation.

**3.2. ALUControl**

module ALU\_Control(

input [1:0] ALUop,

input [3:0] Funct,

output reg [3:0]Operation

);

always @(\*)

begin

case(ALUop)

2'b00:

begin

Operation = 4'b0010;

end

2'b01:

begin

case(Funct[2:0])

3'b000: // BEQ

begin

Operation = 4'b0110; // SUBTRACT

end

3'b100: // BLT

begin

Operation = 4'b0100; // < THAN

end

endcase

end

2'b10:

begin

case(Funct)

4'b0000:

begin

Operation = 4'b0010;

end

4'b1000:

begin

Operation = 4'b0110;

end

4'b0111:

begin

Operation = 4'b0000;

end

4'b0110:

begin

Operation = 4'b0001;

end

endcase

end

endcase

end

endmodule

Explanation:

Two new inputs have been added to the improved ALU Control module: a 2-bit ALUOp control field and the Func Field. With this update, the module can now dynamically generate a 4-bit ALU Control input, which specifies what the ALU must do. The Func Field provides more precise operation selection by combining elements from the funct7 and funct3 fields.   
  
The ALU performs addition when ALUOp is set to "00," which denotes load and store instructions. But in the case of ALUOp values of "10" or "01," the particular operation is determined by the encoding present in the instruction's funct7 and funct3 fields.

When ALUOp is set to "01," indicating a branch-type instruction, an exception takes place. To handle the particular operation related to branch instructions in this circumstance, the ALU Control unit uses a custom case structure.  
  
To put it simply, the updated ALU Control unit takes the combined values of Func and ALUOp to dynamically determine the ALU operation. This means that different types of instructions, like load, store, branch, and others, can be accommodated, which improves the functionality and flexibility of the processor architecture.

**3.3. Control Unit**

`timescale 1ns / 1ps

module Control\_Unit(

input [6:0]Opcode,

output reg Branch,

output reg MemRead,

output reg MemtoReg,

output reg [1:0]ALUop,

output reg MemWrite,

output reg ALUSrc,

output reg RegWrite

);

always @(\*)

begin

//this is for R-type instruction

if(Opcode == 7'b0110011)

begin

Branch = 0;

MemRead = 0;

MemtoReg = 0;

ALUop = 10;

MemWrite = 0;

ALUSrc = 0;

RegWrite = 1;

end

//this is for I-type instruction(ld)

else if(Opcode == 7'b0000011)

begin

Branch = 0;

MemRead = 1;

MemtoReg = 0;

ALUop = 00;

MemWrite = 1;

ALUSrc = 1;

RegWrite = 1;

end

//this is for S-type instruction

else if(Opcode == 7'b0100011)

begin

Branch = 0;

MemRead = 0;

MemtoReg = 1'bX;

ALUop = 00;

MemWrite = 1;

ALUSrc = 1;

RegWrite = 0;

end

//this is for SB-type instruction

else if(Opcode == 7'b1100011)

begin

Branch = 1;

MemRead = 0;

MemtoReg = 1'bX;

ALUop = 01;

MemWrite = 0;

ALUSrc = 0;

RegWrite = 0;

end

//this is an “addi” instruction

else if(Opcode == 7'b0010011)

begin

Branch = 0;

MemRead = 0;

MemtoReg = 1'bX;

ALUop = 00;

MemWrite = 0;

ALUSrc = 1;

RegWrite = 1;

end

// Default case

else

begin

Branch = 1'b0;

MemRead = 1'b0;

MemtoReg = 1'b0;

ALUop = 2'b00;

MemWrite = 1'b0;

ALUSrc = 1'b0;

RegWrite = 1'b0;

end

end

endmodule

Explanation:

Mostly remains unchanged, the control unit itself is taking opcodes which are important for signalling the separate control signals. The default case was adjusted as well alongside the fact that the BLT and BEQ instructions are identical.

**3.4 Data Memory**

module Data\_Memory(

input clk,

input MemWrite,

input MemRead,

input [63:0] Mem\_Addr,

input [63:0] Write\_Data,

output reg [63:0] Read\_Data,

//

output [63:0] index0,

output [63:0] index1,

output [63:0] index2,

output [63:0] index3,

output [63:0] index4

);

reg [7:0] DataMemory [63:0];

integer i;

initial

begin

for (i=0; i<64; i=i+1)

begin

DataMemory[i] = 0;

end

DataMemory[0] = 8'd5;

DataMemory[8] = 8'd2;

DataMemory[16] = 8'd1;

DataMemory[24] = 8'd7;

DataMemory[32] = 8'd4;

end

assign index0 = {DataMemory[7],DataMemory[6],DataMemory[5],DataMemory[4],DataMemory[3],DataMemory[2],DataMemory[1],DataMemory[0]};

assign index1 = {DataMemory[15],DataMemory[14],DataMemory[13],DataMemory[12],DataMemory[11],DataMemory[10],DataMemory[9],DataMemory[8]};

assign index2 = {DataMemory[23],DataMemory[22],DataMemory[21],DataMemory[20],DataMemory[19],DataMemory[18],DataMemory[17],DataMemory[16]};

assign index3 = {DataMemory[31],DataMemory[30],DataMemory[29],DataMemory[28],DataMemory[27],DataMemory[26],DataMemory[25],DataMemory[24]};

assign index4 = {DataMemory[39],DataMemory[38],DataMemory[37],DataMemory[36],DataMemory[35],DataMemory[34],DataMemory[33],DataMemory[32]};

always @ (\*)

begin

if (MemRead)

Read\_Data = {DataMemory[Mem\_Addr+7],DataMemory[Mem\_Addr+6],DataMemory[Mem\_Addr+5],DataMemory[Mem\_Addr+4],DataMemory[Mem\_Addr+3],DataMemory[Mem\_Addr+2],DataMemory[Mem\_Addr+1],DataMemory[Mem\_Addr]};

end

always @ (posedge clk)

begin

if (MemWrite)

begin

DataMemory[Mem\_Addr] = Write\_Data[7:0];

DataMemory[Mem\_Addr+1] = Write\_Data[15:8];

DataMemory[Mem\_Addr+2] = Write\_Data[23:16];

DataMemory[Mem\_Addr+3] = Write\_Data[31:24];

DataMemory[Mem\_Addr+4] = Write\_Data[39:32];

DataMemory[Mem\_Addr+5] = Write\_Data[47:40];

DataMemory[Mem\_Addr+6] = Write\_Data[55:48];

DataMemory[Mem\_Addr+7] = Write\_Data[63:56];

end

end

endmodule

**3.5. Instruction Memory**:

Updated Code already mentioned in task 1.

1. **Task 2 – Pipelining**
2. **Task 3 – Hazard Detection**
3. **Task 4**
4. **Challenges**

There were a multitude of challenges, keeping code consistent between multiple code bases proved to be a challenge even with the use of GitHub. Many changes needed to be made between previous port configurations. Figuring out the conversion from the assembly code was another issue. Between task 2 and task 3, figuring out pipelining and configuring the code was also an issue between transitioning from the original base.

1. **Task Division**

Ahmed Khalid – Task 1, Task 4, Report

Salman Adnan – Task 2, 3

Ibrahim Rana – Task 2, 3

1. **Conclusion**

The project was particularly difficult because it required extensive debugging of the modules as well as the code. Over the course of the project, we encountered many obstacles, but we persevered because we learned from our mistakes, corrected them, and eventually created a pipelined, multi-cycle processor. Theoretically speaking, this novel design offers greater efficiency over its single-cycle equivalent, indicating the possibility of substantial improvements in CPU architecture. And was overall a fantastic learning experience that has helped shape our understanding of how processing works.

1. **Github Link**

https://github.com/Reckon1ng/RISC\_V-Pipelined-Processor