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

A 5-Stage Pipelined Processor for executing one array sorting algorithm

Project Report

Muhammad Areeb Kazmi | mk07202
Muhammad Sulaiman Rokerya | mr07389
Razi Haider | rh06882
5-5-2023

Introduction

This project deals with implementing a RISC-V processor using Verilog HDL on Xilinix Vivado. The main goal of the project is to be able to implement a full-fledged 5-staged pipelined processor that can efficiently execute a one array sorting algorithm.

We will be executing the bubble sort algorithm on our processor. Below are the five objectives we aim to achieve in this project:

- To be able to code the bubble sort algorithm in assembly language using Venus [1].
- To run the machine code of the algorithm on RISC-V single cycle processor we used in our Lab 11.
- To execute the algorithm on a pipelined processor
- To be able to detect and avoid data hazards using the techniques like forwarding
- To compare the performance of the two processors in terms of execution time.

Task 1 – Sorting Algorithm on Single Cycle Processor

C code for Bubble Sort

Below is the code for bubble sort that we used for taking inspiration for the algorithm:

```
void Bubble_Sort(int arr[], int n) {
    int i, j;
    for (i = 0; j < n - 1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Figure 1: C code for bubble sort

Bubble Sort Code for Assembly Language

We used the Venus environment to code for our RISC-V architecture code for Bubble sort; however, please note that the environment uses RISC-V 32-bit architecture while we use 64-bit architecture in our Verilog processor. That is why here, we have loaded words instead of double words.

```
# x22 = i, x23 = j          # x28 has address of i
# x29 has address of j
# let x5 be temp register   # x8 = 5

Loop1:
    blt x22, x8, Loop2  # checking if i has NOT reached the upper bound
    beq x0, x0, Exit

Loop2:                  # nested loop (inner loop)
    addi x23, x22, 0    # j = i
    addi x29, x28, 0    # address of j = address of i
    blt x23, x8, Loop3  # Checking if j < 5
    beq x0, x0, Loop1   # unconditional jump

Loop3:
    lw x12, 0(x28)      # x12 = a[i]
    lw x13, 0(x29)      # x13 = a[j]
    blt x12, x13, IF
    addi x23, x23, 1    # j = j + 1;
    addi x29, x29, 4    # address of j + 4
    blt x23, x8, Loop3  # j < 8
    addi x22, x22, 1    # i = i + 1
    addi x28, x28, 4    # address of i jumped 4 bytes
    beq x0, x0, Loop1

IF:
    add x5, x12, x0     # temp = arr[i]
    sw x13, 0(x28)      # arr[i] = arr[j]
    sw x5, 0(x29)       # arr[j] = temp
    addi x23, x23, 1    # j++
    addi x29, x29, 4    # address of j jumped a word
    blt x23, x8, Loop3  # j < 8
    addi x22, x22, 1    # i++
    addi x28, x28, 4    # address of i jumped a word
    beq x0, x0, Loop1
Exit:
```

Figure 2: Assembly code for Bubble Sort

Changes made to Lab 11 for implementing the Single Cycle Processor

1) Changes made to Control Unit

We first will have to add the SB-type condition in the Control Unit so it can handle the **blt** instruction.

The changes were already made in our lab 11 submission, which can be found on the Repository Link under Lab 11 folder [2].

2) Changes made to ALU Control

We made changes in the following snipped of the ALU Control design file

```
 9    begin
10        case(ALUOp)
11            2'b00: Operation = 4'b0010;
12            2'b01:
13            begin                 // branch instruction types
14             case(Funct[2:0])
15             3'b000:begin         // beq
16                 Operation = 4'b0110;    // subract
17                end
18             3'b100:              // blt
19             begin
20                 Operation = 4'b0100;    // less than operation
21             end
22            endcase
23            end
24
25
26            2'b10:
27            begin
28            case(Funct)
29                4'b0000: Operation = 4'b0010;
30                4'b1000: Operation = 4'b0110;
31                4'b0111: Operation = 4'b0000;
32                4'b0110: Operation = 4'b0001;
33            endcase
34            end
```

Figure 3: Changes made in the ALU Control

Here, we made changes from line 13 onwards till Line 21 where we added the case when the processor will encounter branch instruction type of either **blt** or **beq.**
Since the Control Unit takes as input the Funct (1-bit from funct7, i.e., 30th bit, and 3 bits from funct3, i.e., bits [14:12])
Therefore, the output will be a signal that directly controls the ALU by generating one of he 7 operations in our case.

Here, the case of ALUOp decides whether if the operation should be add(00) for load and store, or to be determined by the funct7 and funct3 fields, 10, 01. Where we added an additional case structure when ALUOp will be 01.

3) Changes made to ALU

We have modified the ALU to add the case when ALUOp will be equal to '0100' for the lesser than condition which compares the two values and returns 0 or 1 as per the condition.
If the first value is less than the second value, 0 is assigned to the Result just like in case of **beq.**

After this, no changes will be required for the additional branch type instruction and only changing the instruction and data memory will be left.

```
6
7       output reg [63:0] Result,
8       output ZERO
9   );
10
11  localparam [3:0]
12  AND = 4'b0000,
13  OR  = 4'b0001,
14  ADD = 4'b0010,
15  Sub = 4'b0110,
16  NOR = 4'b1100;
17
18  assign ZERO = (Result == 0);
19
20  always @ (ALUOp, a, b)
21  begin
22      case (ALUOp)
23          AND: Result = a & b;
24          OR:  Result = a | b;
25          ADD: Result = a + b;
26          Sub: Result = a - b;
27          NOR: Result = ~(a | b);
28          4'b0100: Result = (a < b) ? 0 : 1;     // Lesser than
29      endcase
30  end
31
```

Figure 4: Code snipped containing changes made to the ALU

4) Changes made to Instruction Memory

In the instruction memory module, we had to store all the instructions so each instruction of the assembly code was converted into 32 bit binary address, which would be passed to inst_mem as 8 bits.

The instruction memory was filled with the machine code for instruction. We did not attach the code snipped because of the abundance of lines. Please refet to the Instruction Memory design module on our Github.

5) Changes made to Register File

We removed the part where we assigned values to our registers randomly, and we assigned the value 5 to register **x8**, so it carries the size or the number of the elements that need to be sorted using the bubble sort algorithm.
The rest of the registers are assigned 0 initially.

```
module RegisterFile
  (  input [63:0]WriteData,
     input [4:0]RS1,
   input [4:0]RS2,
   input [4:0]RD,
   input RegWrite, clk, reset,
   output reg [63:0]ReadData1,
   output reg [63:0]ReadData2
  );

  reg[63:0] Registers [31:0];    //initialing registers
  integer i;
  initial
    begin
      for (i = 0; i < 32; i = i + 1)
      begin
        Registers[i] = 0;       // other registers initialized with 0
      end
      Registers[8] = 5;         // x8 = 5, that is the value of the number of elements needed to be sorted


    end

```

Figure 5: Changes made to the RegisterFile.v

We have attached the code above in the figure where changes were made, the rest of the code remains the same for the *RegisterFile.v*

6) Changes made to Data Memory

The data memory was initialized with the values we needed to be kept in the array.
We will be passing the values: 2, 1, 3, 0, 4
If our program and processor works correctly, the sorted values in descending order will be 4, 3, 2, 1, and 0

Results of Task 1

We were successfully able to implement task 1 on a single cycle processor and the expected results for sorting were met as shown below:

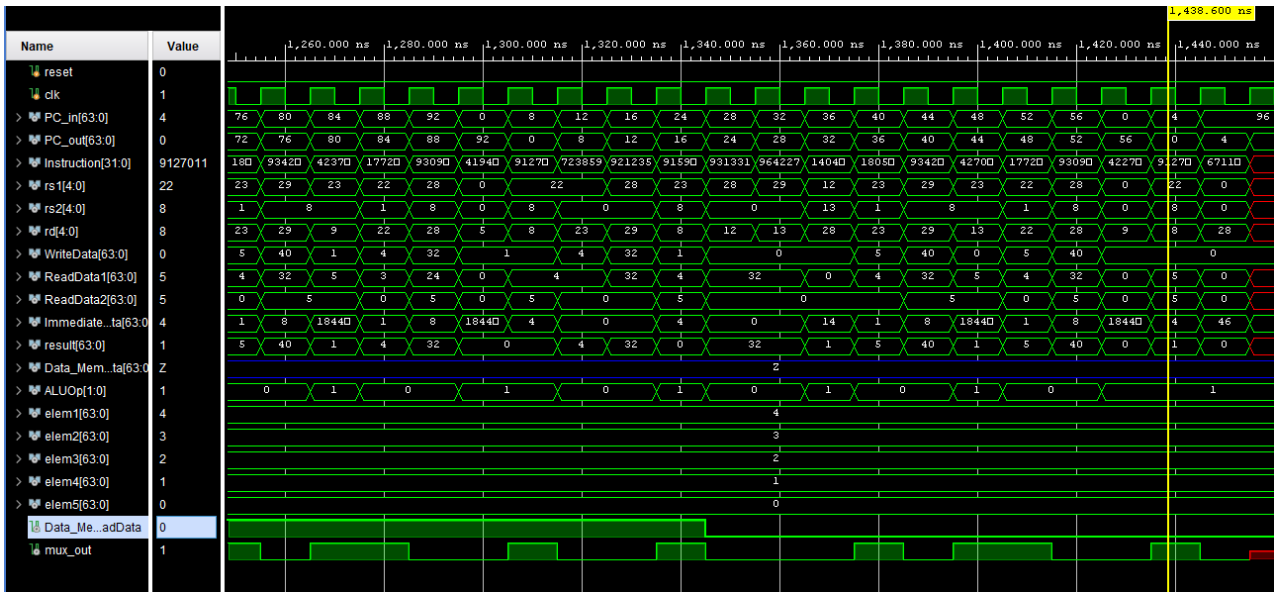

Figure 6: Waveform showing sorted numbers

As we can see in the elem1 till elem5, the numbers have been sorted to their correct positions in 1450 ns.

Task 2-3 – Pipelined Processor Implementation

In this task, we would implement the 5 stage pipelined processor containing the following pipeline registers:

- **IF/ID:** Instruction Fetch / Instruction Decode
- **ID/EX:** Instruction Decode/ Execute
- **EX/MEM:** Execute / Memory Access
- **MEM/WB:** Memory / Write Back

This above mentioned pipeline registers can be seen in the figure below showing an overall picture.

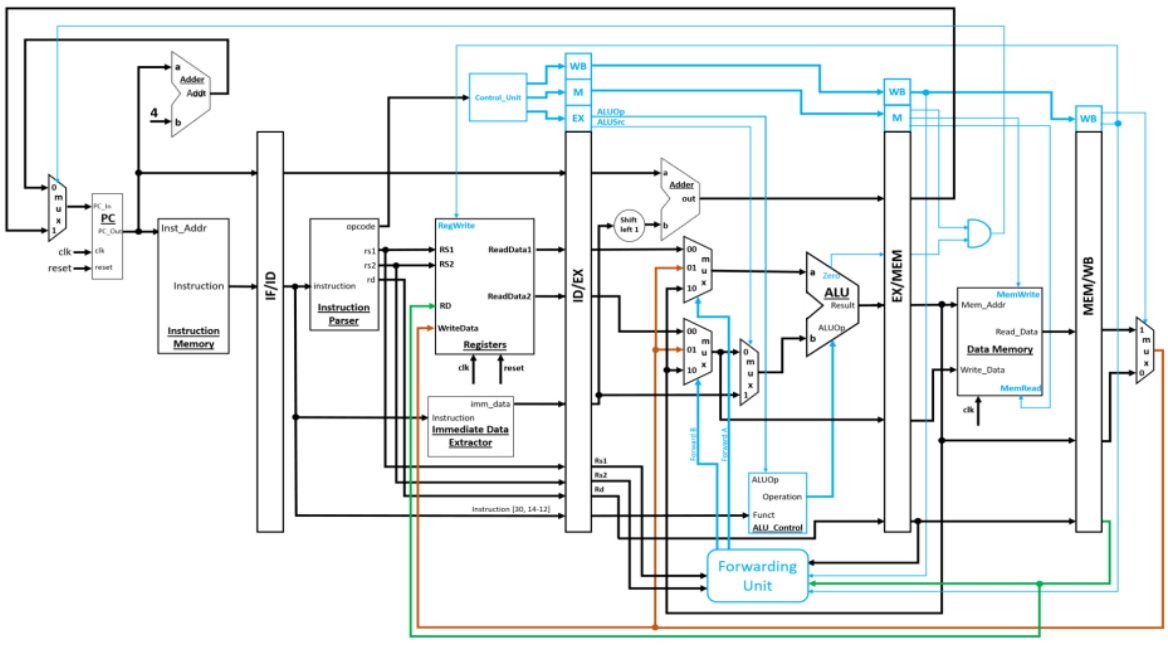

Figure 7: Pipelined Processor with Hazard Detection and Forwarding Units

The program code can be accessed on the repository [2]. However, we will give an overview of each of the design module added.

The IF/ID stage will handle the jump address in the event that the previous instruction was a branch instruction along with the flushing which makes sure that the right instructions are executed from the branch target address.

The ID/EX pipeline register enables us to decide using the signals from MEM/WB register whether it is a read or write operation, where in the case of a flush or stall, the contents are cleared, while otherwise, the contents from the previous stage are stored here.

The EX/MEM pipeline register does a similar operation based on the values of reset and flush.

The MEM/WB pipeline register sends the written values to the Instruction Decode stage where the contents of the registers would be written back.
We were able to implement the forwarding unit using the table below and the conditions:

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Figure 8: The Control values for the Forwarding Unit

Hazard / Stall Detection Unit:

The hazard detection unit will be detecting hazards by checking whether the memory read option is being performed along with the destination register for the current instruction matching the source register for the previous instruction. In such case, the stall flag will be high.

```
module hazard_detection_unit
  (
    input Memread,
    input [31:0] inst,
    input [4:0] Rd,
    output reg stall
  );

  initial
    begin
      stall = 1'b0;         // no stall initially
    end

  always @(*)
    begin
      if (Memread == 1'b1 && ((Rd == inst[19:15]) || (Rd == inst[24:20])))         // only stall when conditions met
        stall = 1'b1;
      else
        stall = 1'b0;                                                              // else no stall
    end
endmodule
```

Figure 9: Hazard detection code snippet

Flushing Unit:

The flush module will make sure that the pipeline is flushed to prevent the execution of invalid instructions. This is checked by checking the branch signal being high.

```
module pipeline_flush
  (
    input branch,
    output reg flush
  );

  initial
    begin
      flush = 1'b0;          // flush initially 0
    end

  always @(*)
    begin
      if (branch == 1'b1)    // if brnach is high
        flush = 1'b1;        // flush is high
      else
        flush = 1'b0;
    end

endmodule
```

Figure 10: Pipeline Flushing Design Module

Results – Task 2 and 3

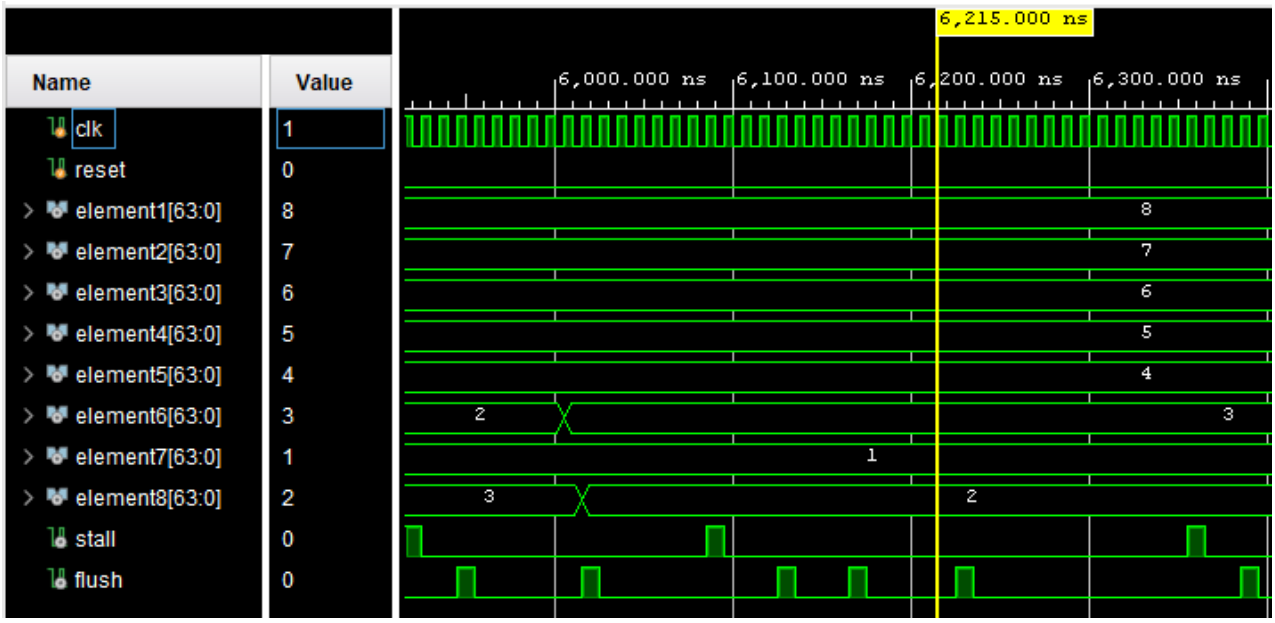

Figure 11: Closeup on waveform showing sorted data

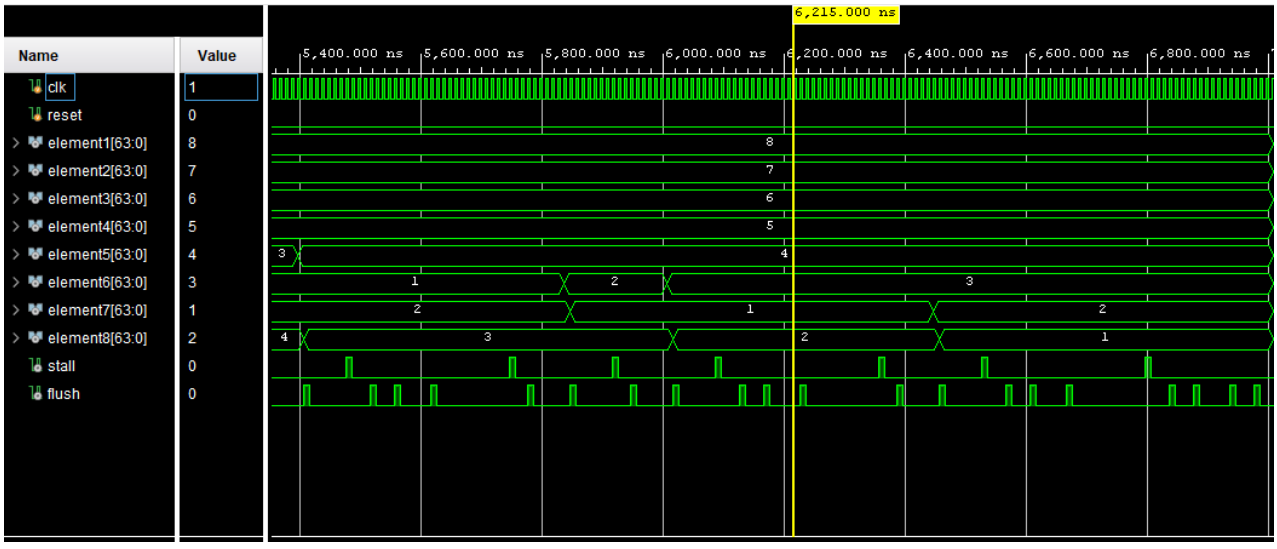

Figure 12: Overall waveform zoomed out

We passed values from 1 till 8 in ascending order, while our final results were sorted in descending order as per expectations along with showing wherever stalls and flushes were taken.

Task 4 – Performance Comparison

The single cycle processor was much faster than the pipelined version because a single instruction was being executed in a single cycle. That meant that the clock frequency overall was much lesser than in Task 2-3. The example of total time taken can be taken where it took 1450 ns for our single cycle processor to execute and sort.

However, in Task 3, it took around 7000ns to sort the values. This is due to the limitation of a larger data set.

Challenges Encountered

1) Making sense of the theoretical aspect of pipelining: One major setback was the relatively weaker understanding of pipelining and techniques like stalling and flushing as compared to the practical labs we did. As a result, we had to resort to lecture slides to code and hard code the values.
2) Hazard detection: Even though the lecture slides were good, we faced incorrect results due to the incorrect execution of instructions in the pipeline
3) Time Management and Conflicts: Due to the examinations and other projects, it was hard to collaborate altogether. Ultimately, we had to work remotely.

Task Distribution

1) Task 0 and 1: Sulaiman Abdullah and Razi Haider
2) Task 2 and 3: Muhammad Areeb Kazmi

The deficiencies in our project were mainly the lack of theoretical understanding of why certain things were taking place, for example the reason for flush and stall to exist separately. We hope to revisit the project with reviewing the theoretical aspect as well to strengthen our understanding.

References

[1] Venus environment link: https://venus.kvakil.me/
[2] Github implementation: https://github.com/MuhammadAreebKazmi2/CA-Project-VHDL