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

Title: RISC-V Pipeline Processor

Eraj Zehra (ez08158)

Sher Ali (sa08505)

Fahad Nadeem (sn07558)

30th April 2024

1. **Introduction**

**Objective:**

To build a 5-stage pipelined processor capable of executing any one array sorting algorithm other than the bubble sort.

Basically, you will be converting your single cycle processor to a pipelined one. Normally the instructions you have already implemented should enable you to execute a sorting algorithm program with small additions i.e., you might need to implement the bgt or blt instruction, or something similar, so that you know when you’d need to sdap two values. This would require small modifications to the circuit.

1. **Methodology:**

**Task 1:** Implementing a sorting algorithm in single cycle processor

* 1. **. Task Explanation:**

We choose insertion sort as our sorting algorithm and translated its pseudocode into assembly language using the RISC-V instruction set. Then, we adapted the single-cycle processor (done in lab 11), to execute the code for the sorting algorithm. This involved adjusting the architecture to accommodate the necessary instructions for comparisons, which are crucial for sorting algorithms like insertion sort. Essentially, we modified the processor's design to facilitate the execution of the insertion sort algorithm efficiently.

* **Sorting Algorithm on Venus:**

Here is our Insertion sorting algorithm in assembly:

# Array base address stored in x10

li x10, 0x100

# Elements to be added to array

li x5, 30

li x6, 13

li x7, 9

li x8, 3

li x9, 5

# Storing elements in array

sw x5, 0(x10)

sw x6, 8(x10)

sw x7, 16(x10)

sw x8, 24(x10)

sw x9, 32(x10)

Insertion\_Sort:

li x13, 5 # x13 <- array\_size = 5

li x14, 1 # x14 <- step = 1

Outer\_Loop:

bge x14, x13, Exit\_Outer # When step >= size Exit the outer loop

addi x15, x14, -1 # x15 <- j = step - 1

# Calculating key = array[step]:

slli x16, x14, 3 # x16 = step index offset

add x16, x10, x16 # x16 = address of array[step]

lw x17, 0(x16) # x17 <- key = array[step]

Inner\_Loop:

# Calculating array[j]:

slli x18, x15, 3 # x18 = j index offset)

add x18, x10, x18 # x18 = address of array[j]

lw x19, 0(x18) # x19 <- array[j]

bge x17, x19, Exit\_Inner # When key >= array[j] Exit the inner loop

blt x15, x0, Exit\_Inner # When j < 0 Exit inner loop

# Swapping:

lw x20, 8(x18) # x20 <- temp = array[j+1]

sw x19, 8(x18) # array[j+1] = array[j]

sw x20, 0(x18) # array[j] = temp

addi x15, x15 -1 # --j

j Inner\_Loop

Exit\_Inner:

# array[j + 1] = key;

addi x21, x15, 1 # x21 = j + 1

slli x21, x21, 3 # x21 <- (j+1) index offset

add x21, x21, x10 # x21 <- address of array[j+1]

sw x17, 0(x21) # array[j + 1] = key

addi x14, x14, 1

j Outer\_Loop

Exit\_Outer:

C language psuedocode was converted to Assembly language and then tested on venus ( online simulator ).

For the algorithm (converted in Assembly language). Following is the test case and its result:

**Test Case and Results:**

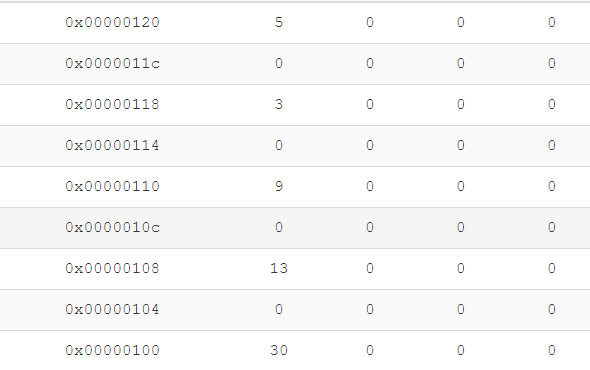


Figure 1: Input Memory for test Case

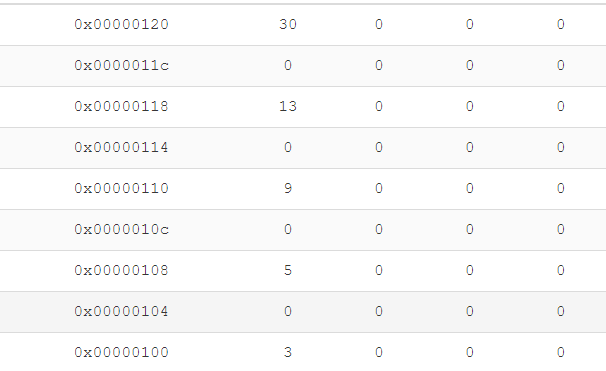


Figure 2: Sorted Output

The platform we chose to test our algorithm only supports 32 bit memory so we cannot use ld sd commands there so we converted our ld sd commands to ld sd ( using word instead of double ). So after testing we have to convert our code back to using double again and then we converted our code to machine language code.

Now we implemented our algorithm on the RISC V single cycle processor we made in the lab. We have uploaded our instruction first in the instruction memory after converting them from assembly to machine language and then upload our base array in the data memory which we had to sort. However, there is some tweak as our assembly code contains only 5 elements to sort but in actual processor that we have added 7 elements in data Memory and then sorted them.

**Code Changes:**

To deal with JAL instruction I have added another mux which is taking two inputs one from adder 1 and one from MemToReg mux and the according selection bit letting the data out to enter in registerFile. The module of Instruction memory is being modified by adding the 32 bits instructions according to the sorting algorithm.

**Results of Vivado:**

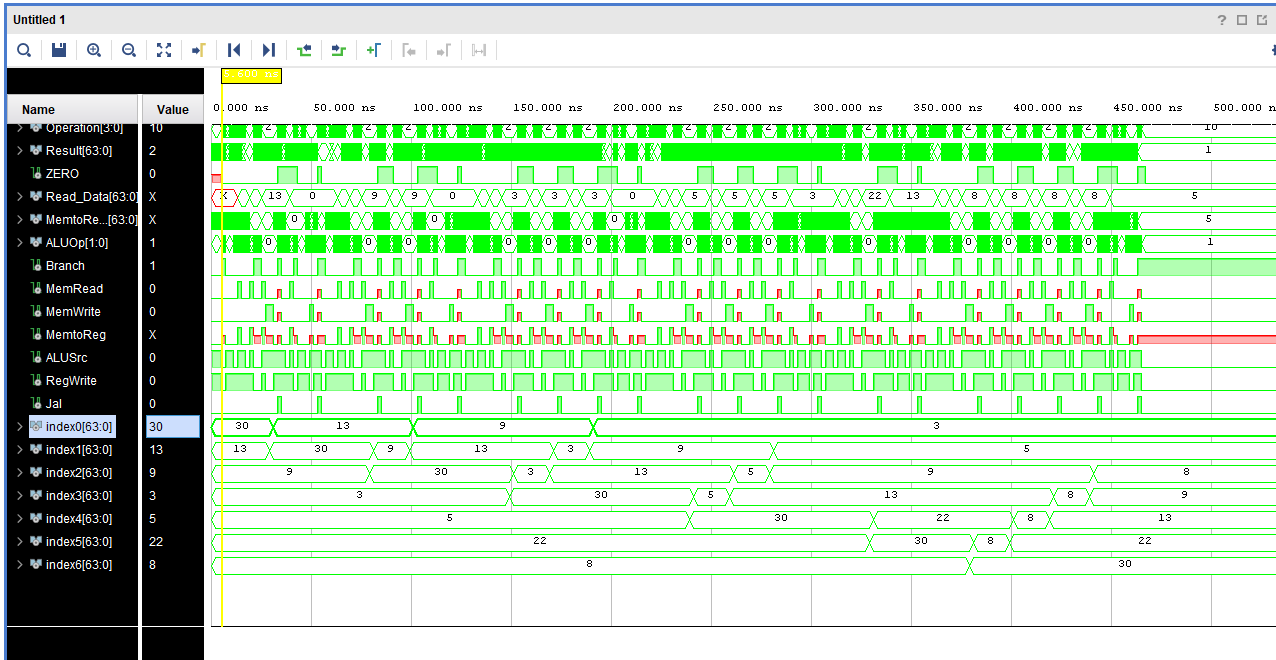


Figure 5: Output of values stored in Memory Array (Sorted)

So, the only change we made was adding extra JAL Mux about which we discussed above.

**Task 2**: Modifying the single cycle processor to implement a pipeline RISC-V processor

**. Task Explanation:**

Here we updated our previously made processor to work with pipelines. So, 5-stages were added in the processor and we have to add 4 new modules which are; IF/ID, ID/EX, EX/MEM, and MEM/WB. Each module represents a stage in the instruction execution process: fetch, decode, execute, memory, and write back. These stages pass data and instructions to each other using pipeline registers. We added a forwarding unit, which helps to resolve issues like data hazards, where instructions depend on data from previous stages.

* **Code Changes:**

The following modules were added to make staging work:

* IF ID
* ID EX
* EX MEM
* MEM WB
* Forwarding
* Two 3x1 Forwarding Mux

These are the stages which connect the modules of the processor with the pipelines. Also we have added extra module of Forwarding on we have we tested certain instruction which were data Hazards and this forwarding unit is resolving them perfectly.

* **Test case and results:**

Instruction memory was updated to run add and addi commands to test the pipeline stages and we can see from Figure 6 that the memory modules are passing registers to each other and the stages are working perfectly.



Figure 6: Output of two instructions

Here It can be seen that the three instructions that we have added in the instruction file which do contain data hazards as the Rd of first instruction is required by Rs2 of second instruction and forwarding unit is giving the 2 as out in forward\_B mux and then Rd of 2nd instruction is required by Rs1 of third instruction and forwarding unit is giving 2 out as input to Forward\_A Mux.

**Problem:**

So, the only problem we have faced here is that PC is correctly fetching instruction from address 0, then 4 and then 8. But in the 4th cycle it is starting from again 0 address of instruction MEM and then working all fine like going from 0 to so on perfectly. We tried our best to figure out this issue but couldn’t resolve it.

**Task 3:** Detecting data Hazards

**. Task Explanation:**

Hazard detection unit is implemented but is not working correctly. Also, since we continued this task from TASK 2 the same issue which we discussed above in TASK 2 faced here.

* **Code Changes:**

We added stall in PC, Control Unit, IFID and when the stall signal is one the values do not change.

**Task 4:** Comparing Performance

single cycle processor took 700 ns.

pipeline in task 2 took total 70 ns for 3 instructions means that each instruction took 23.33 ns. (no stalls but there is forwarding)

So, each stage will take 4.66 ns as there are 5 stages.

There are total 23 instructions in our sorting algorithm hence, 23.33 + 23\*4.66 = 130.51 ns

Speed up: 700 / 130.51 = **5.36**

1. **Challenges Faced**

The problem we have faced, in task 2, is that PC is correctly fetching instruction from address 0, then 4 and then 8. But in the 4th cycle it is starting from again 0 address of instruction MEM and then working all fine like going from 0 to so on perfectly. We tried our best to figure out this issue but couldn’t resolve it. And we were unable to make our task 3 working correctly as the issue was carried forward due to task 2.

1. **Designing**

We made adjustments to the processor to make sure it could run the insertion sort algorithm. And then we tested everything using simulations to check that it all worked correctly.

1. **Task Distribution**

**Sher Ali:** Implemented the single Cycle Processor and tested it on insertion sort algorithm (task 1)

**Eraj Zehra**: Implemented two of the pipeline registers (IF/ID, ID/EXE), along with the implementation of forwarding unit (task 2)

**Fahad Nadeem:** Implemented two of the pipeline registers (EXE/MEM, MEM/WB), without forwarding, in task 2

1. **Conclusion**

Our project was 50% successful and 50% not as we tried our best to implement task 3 but we failed due to some of the errors. But we successfully, implemented task 1 and task 2.

1. **Link to code**: We have separately submitted the code file in the code submission tab on lms.