EE/CS 371L/330L - Computer Architecture

Instructors - Dr. Owais Talat and Ms. Hafsa Amanullah

Lab Project - RISC V Processor

Muhammad Jazzel Mehmood (mm06886)
Sijjil Khan (sk06900)

Date: 06/05/2022

1 Introduction

1.1 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.

1.2 Sorting Algorithm

We will be using insertion sort to test our processor functionality. Complete Project (GitHub Link)

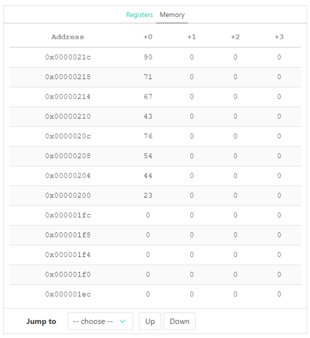
2 Tasks

2.1 Task 1

2.1.1 Sorting Algorithm

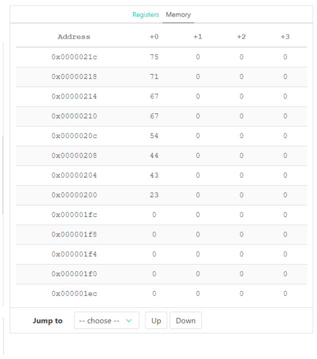
C language psuedocode was converted to Assembly language and then tested on venus (online simulator).
For the algorithm (converted in Assembly language). We tested for different cases where two of them are as follows

Test 1



Address	+0	+1	+2	+3
0x0000021c	90	0	0	0
0x00000218	71	0	0	0
0x00000214	67	0	0	0
0x00000210	43	0	0	0
0x0000020c	76	0	0	0
0x00000208	54	0	0	0
0x00000204	44	0	0	0
0x00000200	23	0	0	0
0x000001fc	0	0	0	0
0x000001f8	0	0	0	0
0x000001f4	0	0	0	0
0x000001f0	0	0	0	0
0x000001ec	0	0	0	0

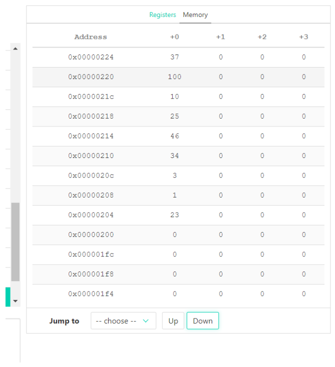
Figure 1: Input Memory for test 1



Address	+0	+1	+2	+3
0x0000021c	75	0	0	0
0x00000218	71	0	0	0
0x00000214	67	0	0	0
0x00000210	67	0	0	0
0x0000020c	54	0	0	0
0x00000208	44	0	0	0
0x00000204	43	0	0	0
0x00000200	23	0	0	0
0x000001fc	0	0	0	0
0x000001f8	0	0	0	0
0x000001f4	0	0	0	0
0x000001f0	0	0	0	0
0x000001ec	0	0	0	0

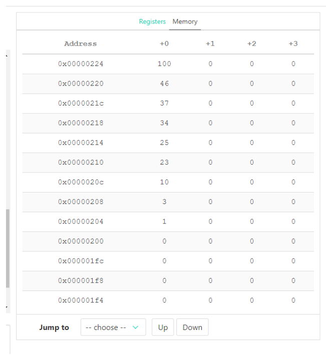
Figure 2: Output Memory for test 1

Test 2



Address	+0	+1	+2	+3
0x00000224	37	0	0	0
0x00000220	100	0	0	0
0x0000021c	10	0	0	0
0x00000218	25	0	0	0
0x00000214	46	0	0	0
0x00000210	34	0	0	0
0x0000020c	3	0	0	0
0x00000208	1	0	0	0
0x00000204	23	0	0	0
0x00000200	0	0	0	0
0x000001fc	0	0	0	0
0x000001f8	0	0	0	0
0x000001f4	0	0	0	0

Figure 3: Input Memory for test 2



Address	+0	+1	+2	+3
0x00000224	100	0	0	0
0x00000220	46	0	0	0
0x0000021c	37	0	0	0
0x00000218	34	0	0	0
0x00000214	25	0	0	0
0x00000210	23	0	0	0
0x0000020c	10	0	0	0
0x00000208	3	0	0	0
0x00000204	1	0	0	0
0x00000200	0	0	0	0
0x000001fc	0	0	0	0
0x000001f8	0	0	0	0
0x000001f4	0	0	0	0

Figure 4: Ouput Memory for test 2

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.

Here is the link to the converted Assembly code and Machine langauge code.

https://github.com/Jazzel/HU-CA-Project/tree/main/Source/Sorting-Algorithm

2.1.2 RISC V Implementation (single cycle)

Now we implemented our algorithm on the RISC V single cycle processor we made in the lab. We did some tweaks in the ALU and instruction memory which are as follows: Firstly we have bgt and blt commands in our algorithm and our processor does not have support for these so we added another module named Branch_module and an Branch input which controls the last mux in the processor.

2.1.3 Changes

So we introduced a bit in ALU which contains the NOT of last bit of ALU Result and then we introduced another module named as Branch_module. We know that if the Zero form ALU Result is 1 then the branch statement is beq and if it is 0 then the statement is bne. So by adding the another variable we can OR it with the Zero to add more functionality. So if both variables are 1 then we have the bgt statement and if both are 0 then we have blt statement. Next we AND these statements to make our branch variable 1 as we have to control the MUX with this variable. So by doing that we finally make our processor work on bgt and blt statements.

2.1.4 Code

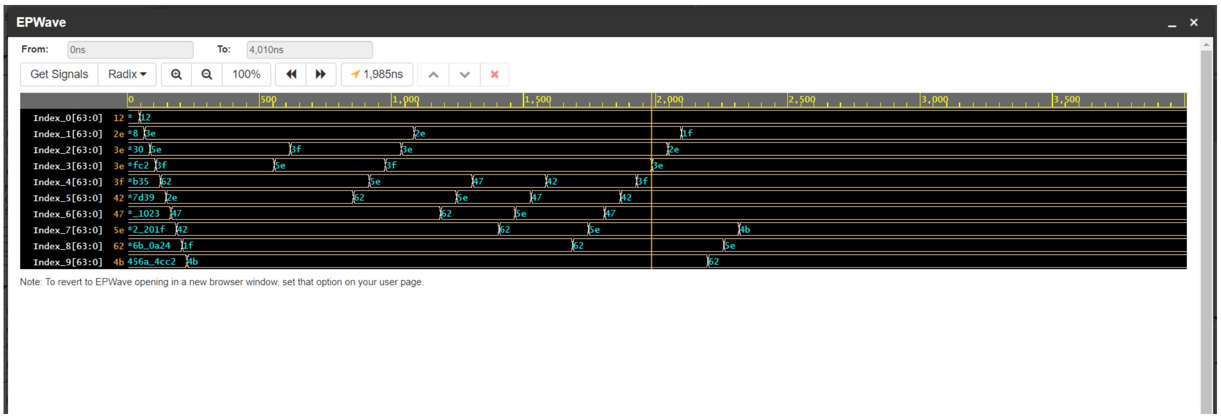


Figure 5: Output of values stored in MemoryArray (Sorted)

https://github.com/Jazzel/HU-CA-Project/tree/main/Source/Single-Cycle-Processor (Code and Implementation of Single Cycle RISC V Processor)

2.1.5 Synopsis

So, the changes were made in the ALU, and the last MUX module (changed the select line for that) and a new module Branch_module was introduced to make our sorting algorithm work.

2.2 Task 2

2.2.1 Testing RISC V Implementation (5-stage/pipelined)

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 to make the processor work.

2.2.2 Changes

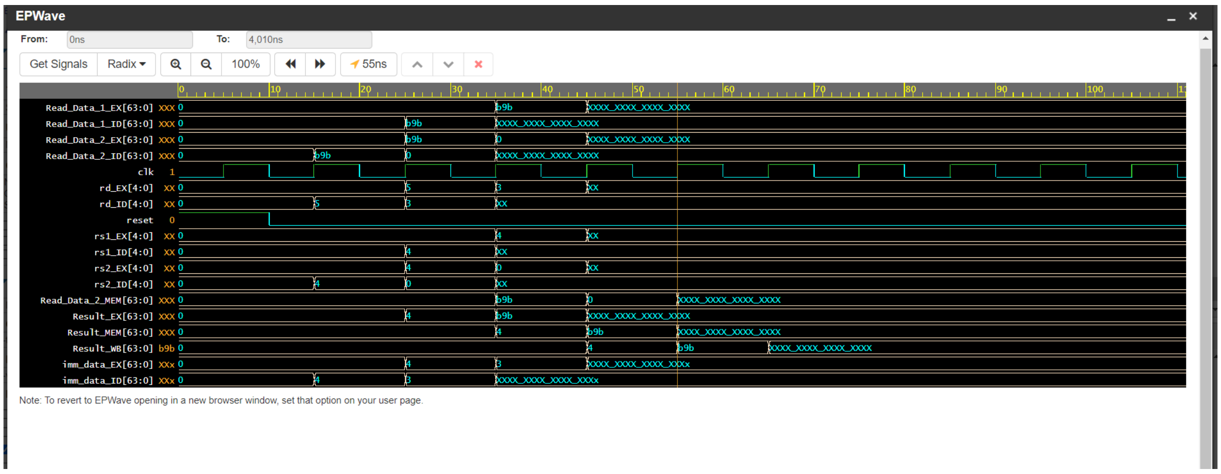
The following modules were added to make staging work:

- IF_ID
- ID_EX
- EX_MEM
- MEM_WB

These are the stages which connect the modules of the processor with the pipelines.

2.2.3 Test cases

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



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Figure 6: Output of testcases add & addi

2.2.4 Code

https://github.com/Jazzel/HU-CA-Project/tree/main/Source/Pipelined-Processor (Code, Code and Implementation of 5-Stage/Pipelined RISC V Processor)

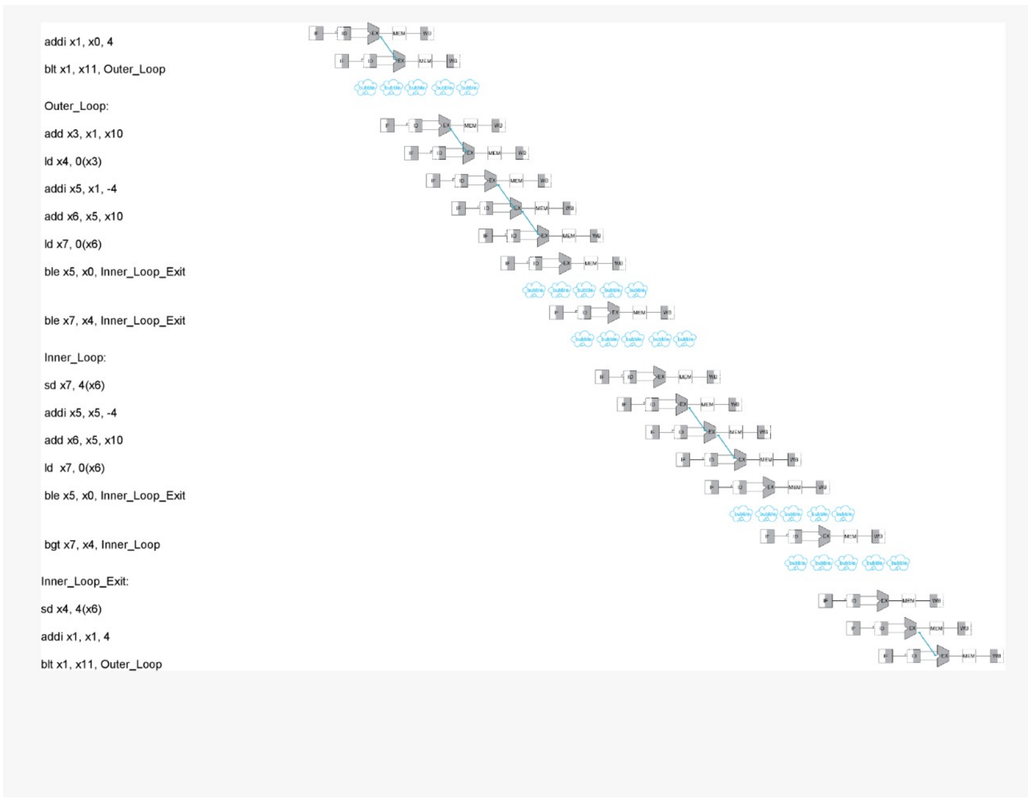
2.2.5 Synopsis

So, after adding the memory management modules or stages we have to change the inputs and outputs of every module to make them work according to the pipeline and memory. So we have to change the modules initialization in the RISC V Processor module (top level module).

2.3 Task 3

2.3.1 Detecting Data Hazards

Following is our Pipelined diagram for the Sorting algorithm.



```
addi x1, x0, 4
blt x1, x11, Outer_Loop
Outer_Loop:
add x3, x1, x10
ld x4, 0(x3)
addi x5, x1, -4
add x6, x5, x10
ld x7, 0(x6)
ble x5, x0, Inner_Loop_Exit
ble x7, x4, Inner_Loop_Exit
Inner_Loop:
sd x7, 4(x6)
addi x5, x5, -4
add x6, x5, x10
ld  x7, 0(x6)
ble x5, x0, Inner_Loop_Exit
bgt x7, x4, Inner_Loop
Inner_Loop_Exit:
sd x4, 4(x6)
addi x1, x1, 4
blt x1, x11, Outer_Loop
```

Figure 7: Input Memory for test 1

2.3.2 Handling Data Hazards

For hazards handling, We will write modules for hazards and stall detection to first identify where we need to forward our circuitry and then we have to add the forward module which will actually forward the statements. Also we can switch those code statements which are not dependent on any other line with the stalls we can make our algorithm work in less number of cycles.

2.3.3 Changes

Here we again updated our previously made processor to work with forwarding and stall logic. So 3 new modules were added.
The modules we added are as follows:

- Forwarding
- Stall
- Mux-control

The stall modules checks for stalls in the processor while running instructions and forwarding module forwards the instructions according to stalls in the given code, Finally the Mux control module manages memory of registers.

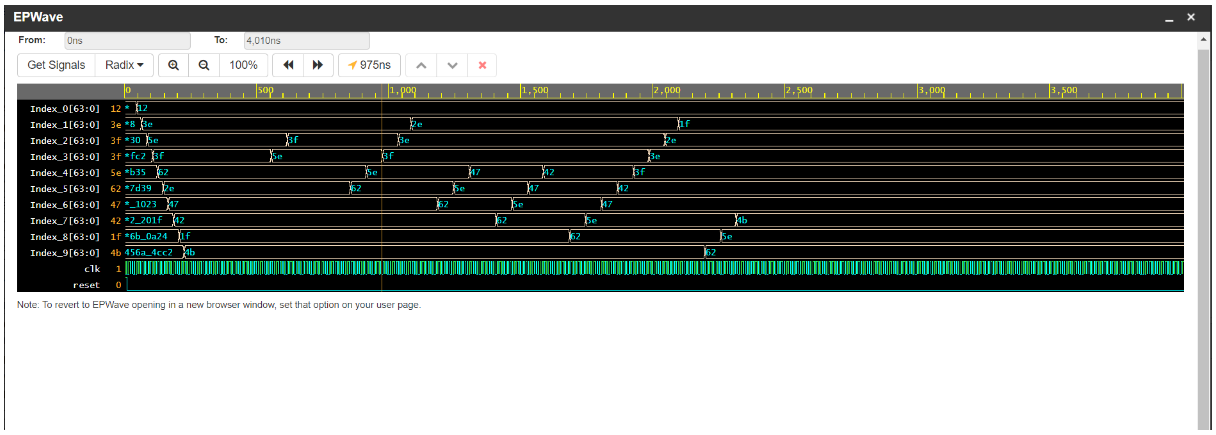


Figure 8: Output of values stored in MemoryArray (Sorted)

2.3.4 Code

https://github.com/Jazzel/HU-CA-Project/tree/main/Source/Pipelined-Processor-with-Stalling-and-Forwarding (Code, Code and Implementation of 5-Stage/Pipelined RISC V Processor with forwarding and stalls logic)

2.3.5 Synopsis

5 stages pipelined processor was working fine but there are stalls in our code due to the fact that the next value of register is required in the next line of code for that the code and environment needs to be sequential and the processor stops for few cycles to get back to the sequence. For that we added forwarding and stalling logic which almost solves this issue and makes our processor work faster.

2.4 Conclusion

From building seperate modules to initialzing everything to make a complete processor it took a lot of brain power and EDA crashing and glitching everytime whenever we need it the most, we learned alot.
According to theory, the pipelined processor works faster than the single cycle processor as it does not wait for the whole instruction to execute completely. It's more like parallel computing or asynchronous programming which is really useful and effective.
Also with better hazards handling we could make our processor work more effectively and faster. As in our case (sorting algorithm) we had a lot of stalls which we could not fix due to the time and complexity issue, But this processor can work more faster with better hazards and stall units. Moreover implementing a real processor through gate level logic really made us love Digital Logic Design more as this course was an extension to DLD course. Looking forward for such courses and projects in the future.

Complete Project (GitHub Link)