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.

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

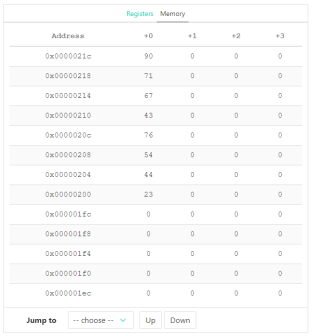

Figure 1: Input Memory for test 1

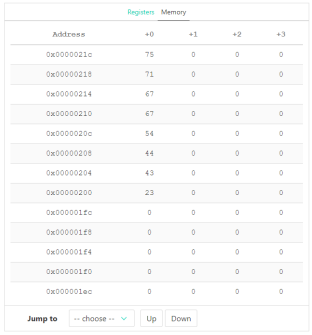

Figure 2: Output Memory for test 1

Test 2

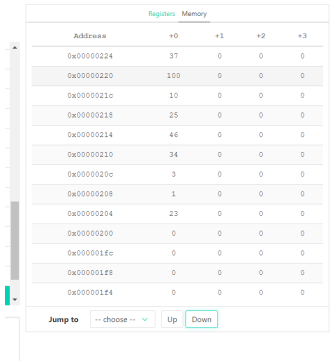

Figure 3: Input Memory for test 2

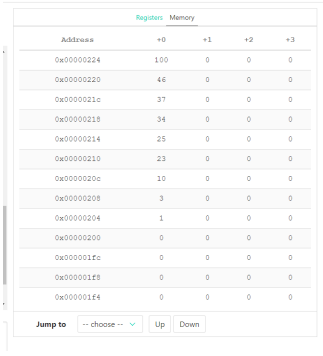

Figure 4: Ouput Memory for test 2

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 we have the converted Assembly code.

—- Sorting : Insertion sort —-
j = 1
addi x1, x0, 4
if j ¡ len(A) then Start the loop
blt x1, x11, Outer_Loop
$Outer_Loop$:
$j + base_address$
$addx3, x1, x10$
$key = A[j]$
$ldx4, 0(x3)$
$i = j - 1$
$addix5, x1, -4$
$i + base_address$
$addx6, x5, x10$
$A[i]$
$ldx7, 0(x6)$
$if i <= 0 then break$
$blex5, x0, Inner_Loop_Exit$

$if A[i] <= key then break$
$ble x7, x4, Inner_Loop_Exit$
$Inner_Loop:$
$A[i+1] = A[i]$
$sd x7, 4(x6)$
$i--$
$addi x5, x5, -4$
$i + base_address$
$add x6, x5, x10$
$A[i]$
$ld x7, 0(x6)$
$if i <= 0 then break$
$ble x5, x0, Inner_Loop_Exit$
$if A[i] > key then continue$
$bgt x7, x4, Inner_Loop$
$Inner_Loop_Exit:$
$A[i+1] = key$
$sd x4, 4(x6)$
$j++$
$addi x1, x1, 4$
$if j < Len(A) then continue$
$blt x1, x11, Outer_Loop$

And here we have the converted machine code.

```
// blt x1 x11 -60
11111100101100001100001011100011;
// addi x1 x1 4
00000000010000001000000010010011;
// sd x4 4(x6)
00000000010000110010001000100011;
// blt x4 x7 -20
11111110011100100100011011100011;
// bge x0 x5 8
00000000010100000101010001100011;
// ld x7 0(x6)
00000000000000011001000111000011;
// add x6 x5 x10
00000000101000101000001100110011;

// addi x5 x5 -4
11111111110000101000001010010011;
// sd x7 4(x6)
00000000011100110010001000100011;
// bge x4 x7 28
00000000011100100101111001100011;
// bge x0 x5 32
00000010010100000101000001100011;
// ld x7 0(x6)
00000000000000011001000111000011;
// add x6 x5 x10
00000000101000101000001100110011;
// addi x5 x1 -4
```

```
11111111110000001000001010010011;
// ld x4 0(x3)
00000000000000011010001000000011;
// add x3 x1 x10
00000000101000001000000110110011;
// blt x1 x10 4
00000000101000001100001001100011;
// addi x1 x0 4
00000000010000000000000010010011;
// sd x9 36(x10)
00000010100101010010001000100011;
// addi x9 x0 75
00000100101100000000010010010011;
// sd x9 32(x10)
00000010100101010010000000100011;
// addi x9 x0 31
00000001111100000000010010010011;
// sd x9 28(x10)
00000000100101010010111000100011;
// addi x9 x0 66
00000100001000000000010010010011;
// sd x9 24(x10)
00000000100101010010110000100011;
// addi x9 x0 71
00000100011100000000010010010011;
// sd x9 20(x10)
00000000100101010010101000100011;
// addi x9 x0 46
00000010111000000000010010010011;
// sd x9 16(x10)
00000000100101010010100000100011;
// addi x9 x0 98
00000110001000000000010010010011;
// sd x9 12(x10)
00000000100101010010011000100011;
// addi x9 x0 63
00000011111100000000010010010011;
// sd x9 4(x10)
00000000100101010010010000100011;
// addi x9 x0 94
00000101111000000000010010010011;
// sd x9 4(x10)
00000000100101010010001000100011;
// addi x9 x0 62
00000011111000000000010010010011;
// sd x9 0(x10)
00000000100101010010000000100011;
// addi x9 x0 18
00000001100010000000000010010011;
// addi x11 x0 40
00000010100000000000010110010011;
// addi x10 x0 0
00000000000000000000010100010011;
```

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

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.

Results

https://www.edaplayground.com/x/p4vw (Code, Results and Implementation of Single Cycle RISC V Processor)

2.1.3 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.
The modules we added are as follows:

- IF_ID
- ID_EX
- EX_MEM
- MEM_WB

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

Results

https://www.edaplayground.com/x/RZwN (Code, Results and Implementation of 5-Stage/Pipelined RISC V Processor)

2.2.2 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 $\mathrm{RISC}_{V\,P}processormodule(toplevelmodule)$.

2.3 Task 3

2.3.1 Detecting Data Hazards

Following is our Pipelined diagram for the Sorting algorithm.

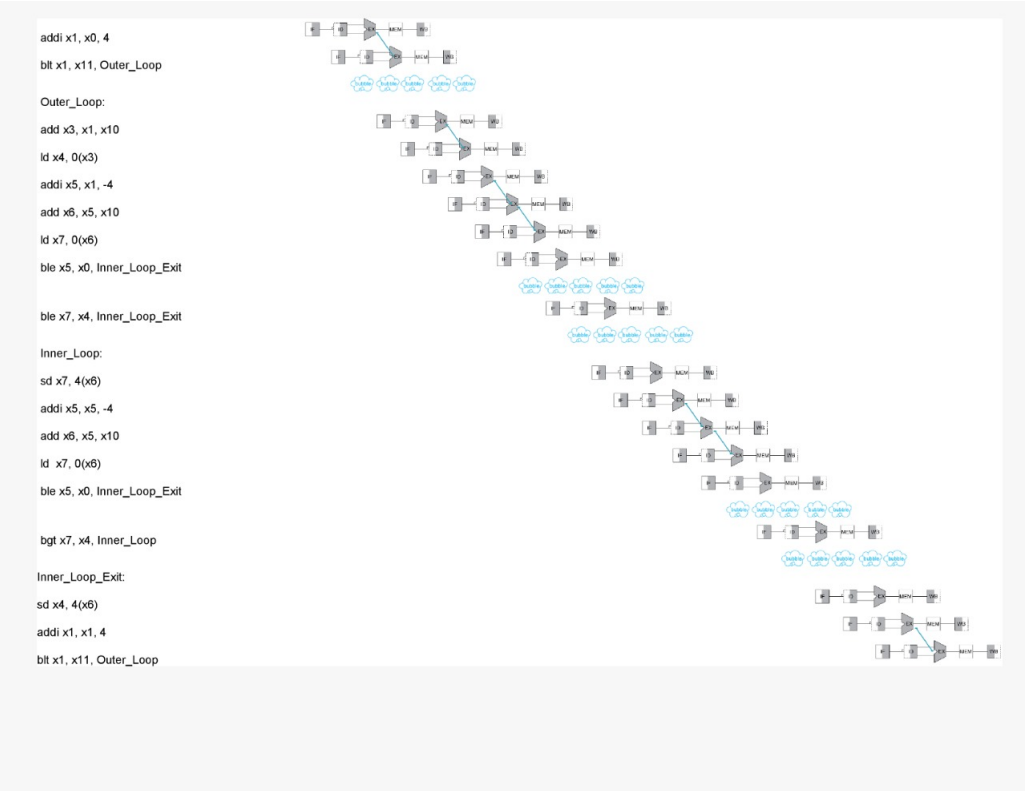

Figure 5: Input Memory for test 1

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.

2.4 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.