CE-321L/CS-330L: Computer Architecture
Pipelined RISC V Processor Ultimate

Musab Kasbati, Naaseh Sajid

19th April 2024

Introduction

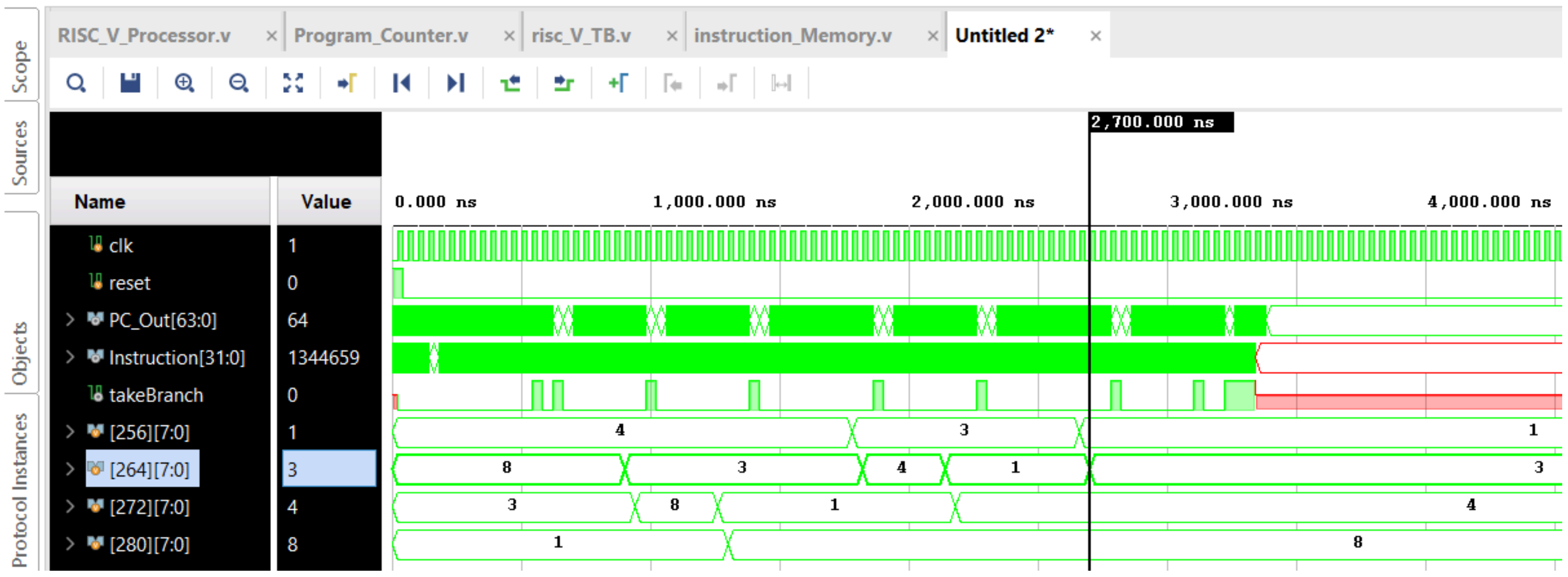
The project focuses on simulating a pipelined RISC V processor and test its functionality on individual types of operations as well as a combination of multiple operations within a larger sorting algorithm. The sorting algorithm of our choice is Bubble Sort.

Methodology

- **Task 1:** In this section, we made use of Kvakil online simulator to generate a working assembly version of a Bubble Sort algorithm. This naturally included instructions that were not catered for in earlier labs such (including blt, slli). Then we updated the Instruction Memory to include the code for Bubble Sort followed by successfully testing it on our single cycle processor.
- **Task 2:** We created different modules to implement pipelining methodology. This mainly included creating pipes that would store the value of registers and control signals temporarily. We then implemented a forwarding module to correctly handle how register values and control signals are forwarded throughout.
- **Task 3:** We then created a module for Hazard detection (load-use hazard), by checking the values in the various pipes during execution. If a hazard was detected, a stall was enforced. In the case of branching, we went with the logic that a branch is assumed to not be taken. If then the branch was calculated to be taken, the instructions currently in the pipeline were flushed, a blank/stall instruction was passed for 1-cycle and the PC value was updated i.e. the jump was made. The rest of the execution proceeded as normal. The branching was verified by testing it on the original code for bubble sort.
- **Task 4:** We compared the running time of the different processors on the same bubble sort code. Note that we gave the Single Cycle Processor a longer clock cycle (40ns), as opposed to the shorter (10 ns) clock cycle of the Pipelined Processor. This was to account for the fact that since instructions are executed in stages in a pipelined processor, the time per stage is notably less than the time for complete execution. We kept a factor of 4 times quicker clock cycles to stay consistent with the architecture example we have discussed throughout the course.
- The changes we made depended on different tasks. For example, as we shifted from Task 1 to Task 2, we had to change our top level program in such a way that it can handle pipelining. Similarly, we had to change it again to account for forwarding. This included developing different wires to handle in-between values, making pipes, and manipulating control signals according to use cases. In many cases, we added new inputs and outputs to older pipelines for ease of implementation and robustness.

Results

Task 1:



Name	Value
clk	1
reset	0
PC_Out[63:0]	64
Instruction[31:0]	1344659
takeBranch	0
[256][7:0]	1
[264][7:0]	3
[272][7:0]	4
[280][7:0]	8

Explanation: The bottom 4 lines are the registers in memory where our array is stored. We start with an initial input of. 4, 8, 3, 1. Over time the appropriate substitutions take place (8<->3, 8<->1, 4<->3, 4<->1, 3<->1). These are the same swaps as we would expect from the bubble sort algorithm for this input, therefore, our single cycle processor and our code is working as intended. The clock cycles where a branch was taken can be seen from the takeBranch line. Moreover, we see that the array is first sorted at 2,700.000 ns.

Task 2:
The following snippet is for the instructions below:
1. addi x2 x0 50
2. add x3 x2 x2
3. add x4 x3 x2
4. add x4 x2 x2

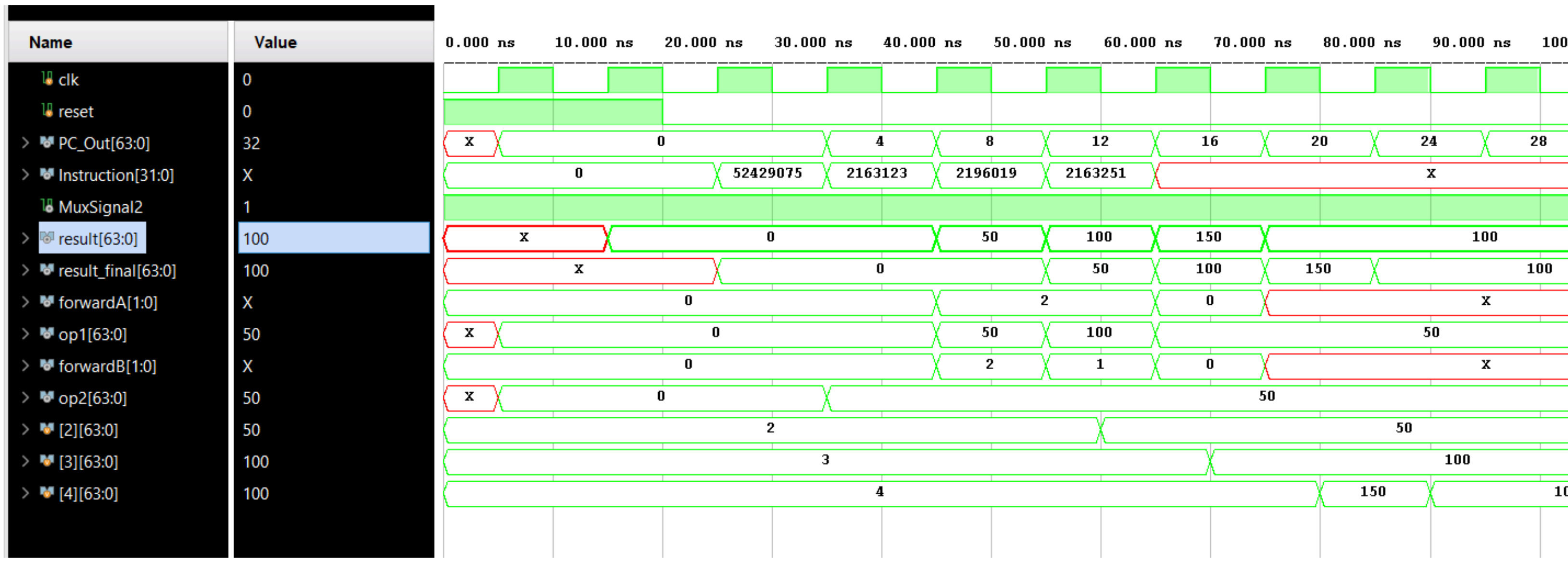


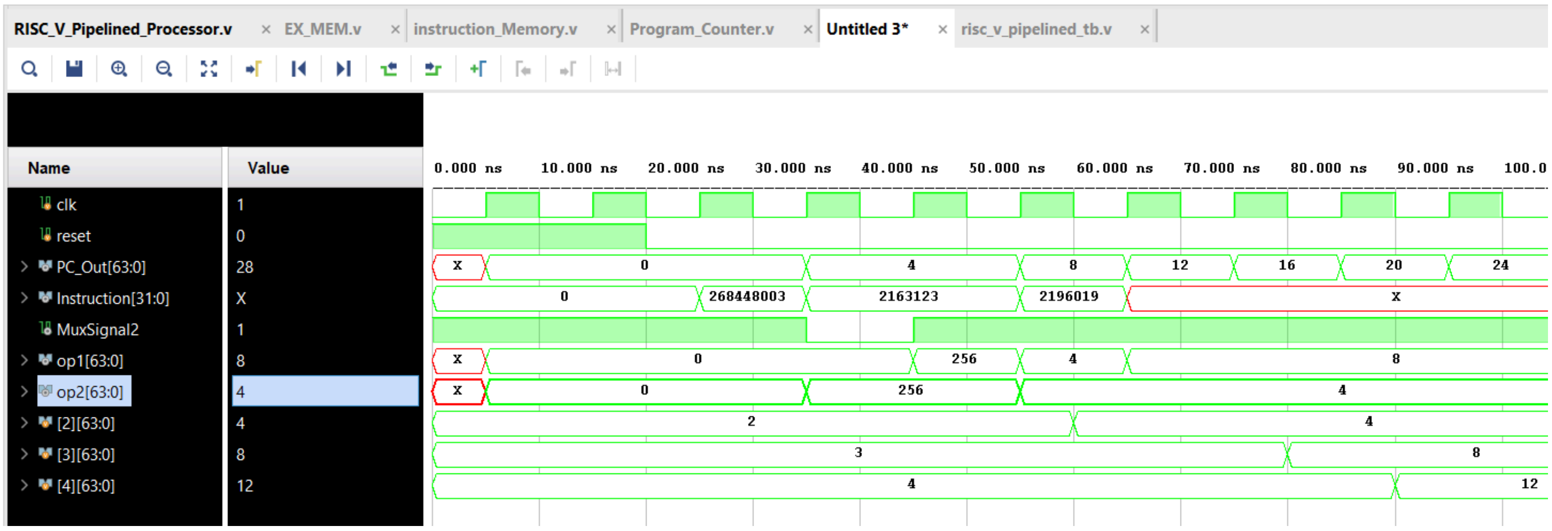
Figure 2: Forward A and Forward B are initially 0. Forward A and Forward B take the value 2 for instruction 2 since both operands are in the prior ALU result. Forward B then takes on the value 1 for instruction 3 because it requires x2 which is in Data Memory, meanwhile ForwardA is 2 since it requires the most recently generated ALU result. They both revert back to 0 for instruction 4, since by then x2 is already written back to memory.

Task 3:

Test Cases:

<u>Load-use hazard:</u>

1. ld x2 0x100(x0)
2. add x3 x2 x2
3. add x4 x3 x2

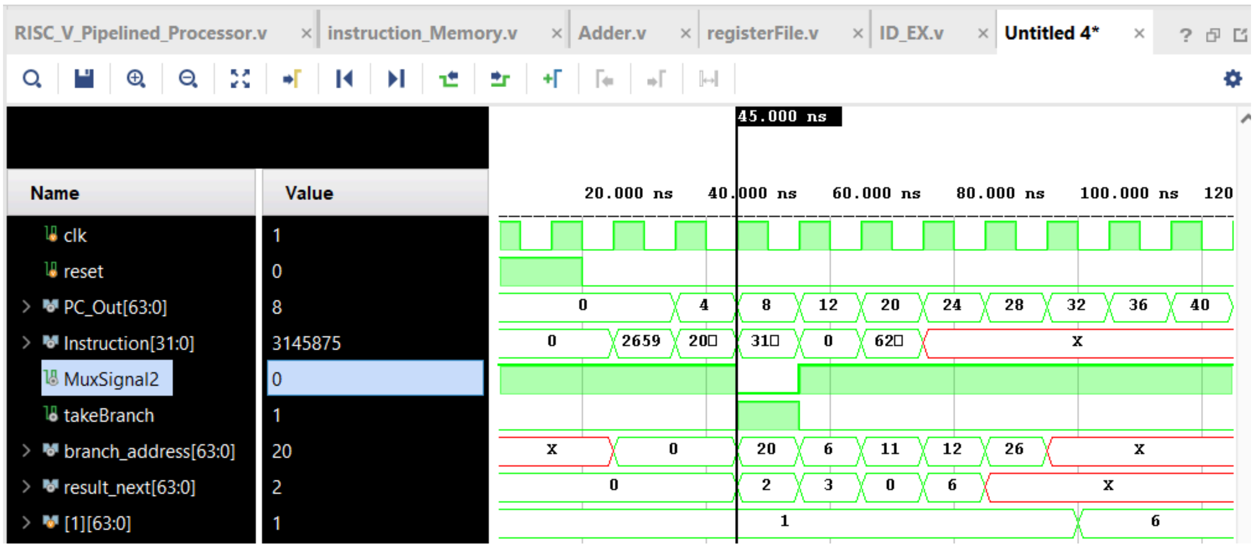


Name	Value
clk	1
reset	0
PC_Out[63:0]	28
Instruction[31:0]	X
MuxSignal2	1
op1[63:0]	8
op2[63:0]	4
[2][63:0]	4
[3][63:0]	8
[4][63:0]	12

Figure 3a. Load-Use Hazard

Explanation:
The bottom 3 lines represent the registers 2, 3, 4. op1 and op2 are operands for ALU calculation after forwarding. We can see that the clock cycle for the 2nd cycle (PC = 4) is longer than those subsequent. Moreover, MuxSignal2 is 0 at the start of the 2nd cycle, this is because at this point, instruction 2 reaches ID stage and the hazard is detected so MuxSignal2 is set to 0 to stall the cycle until the first instruction is ready to provide the value. We see that by the time we reach the 3rd instruction, the loaded value (4) is available in op1 and op2 for further computation. Moreover, we see that no further stall happens for instruction 3 as value is already available.

<u>Control Hazard:</u>
1. beq x0, x0, end
2. addi x1, x0, 2
3. addi x1, x0, 3
4. addi x1, x0, 4
5. addi x1, x0, 5
end:
6. addi x1, x0, 6



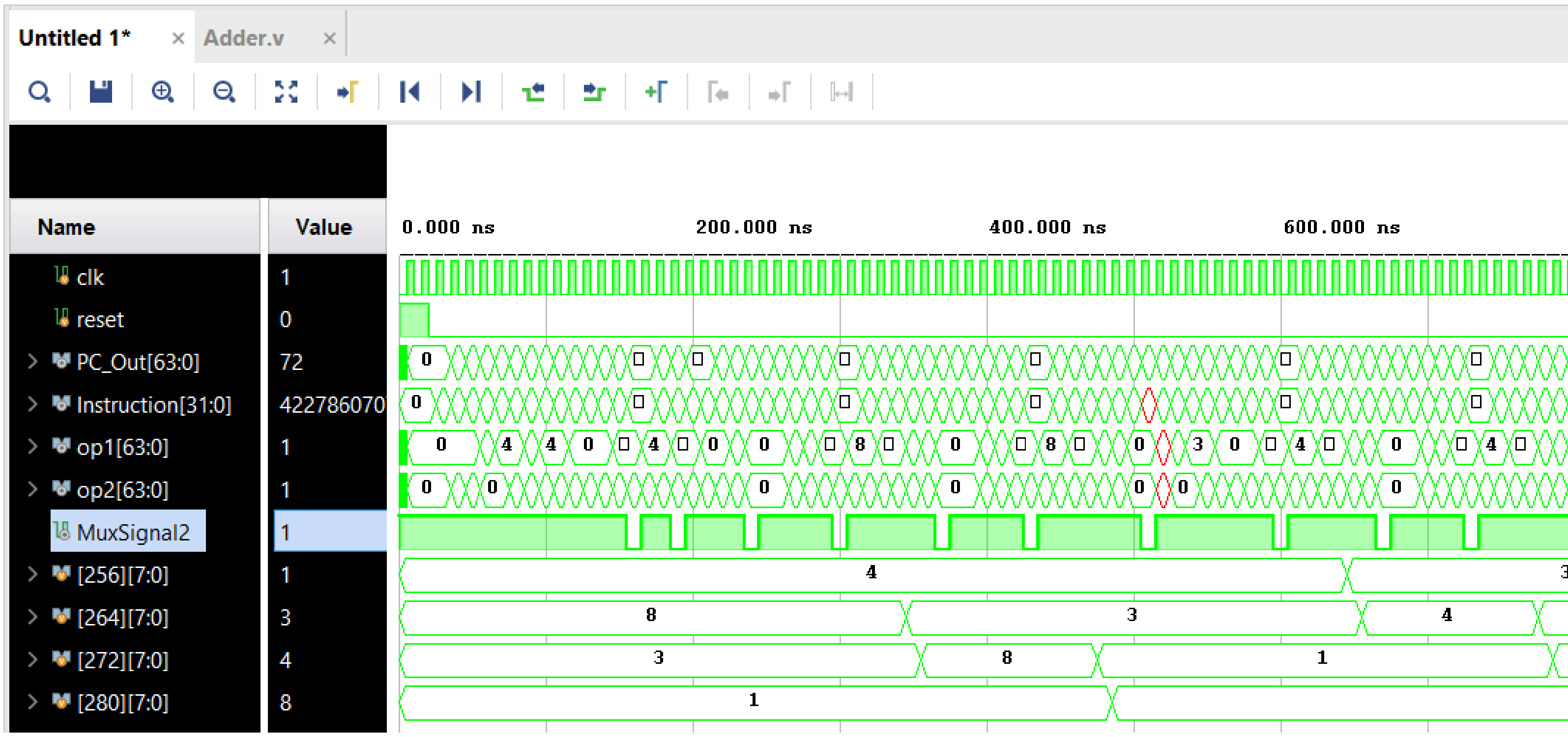
Name	Value
clk	1
reset	0
PC_Out[63:0]	8
Instruction[31:0]	3145875
MuxSignal2	0
takeBranch	1
branch_address[63:0]	20
result_next[63:0]	2
[1][63:0]	1

Figure 3b. Control Hazard

Explanation:
The branch is instruction 1 (PC = 0). We see that although the branch is to be taken, it is not immediately taken as the result of the comparison needs to be calculated. This takes until the 3rd instruction where we see takeBranch is 1 and MuxSignal2 is 0. After which, we still need one more stall cycle to set up our next instruction. After this, our PC jumps to 20 i.e. instruction 6.

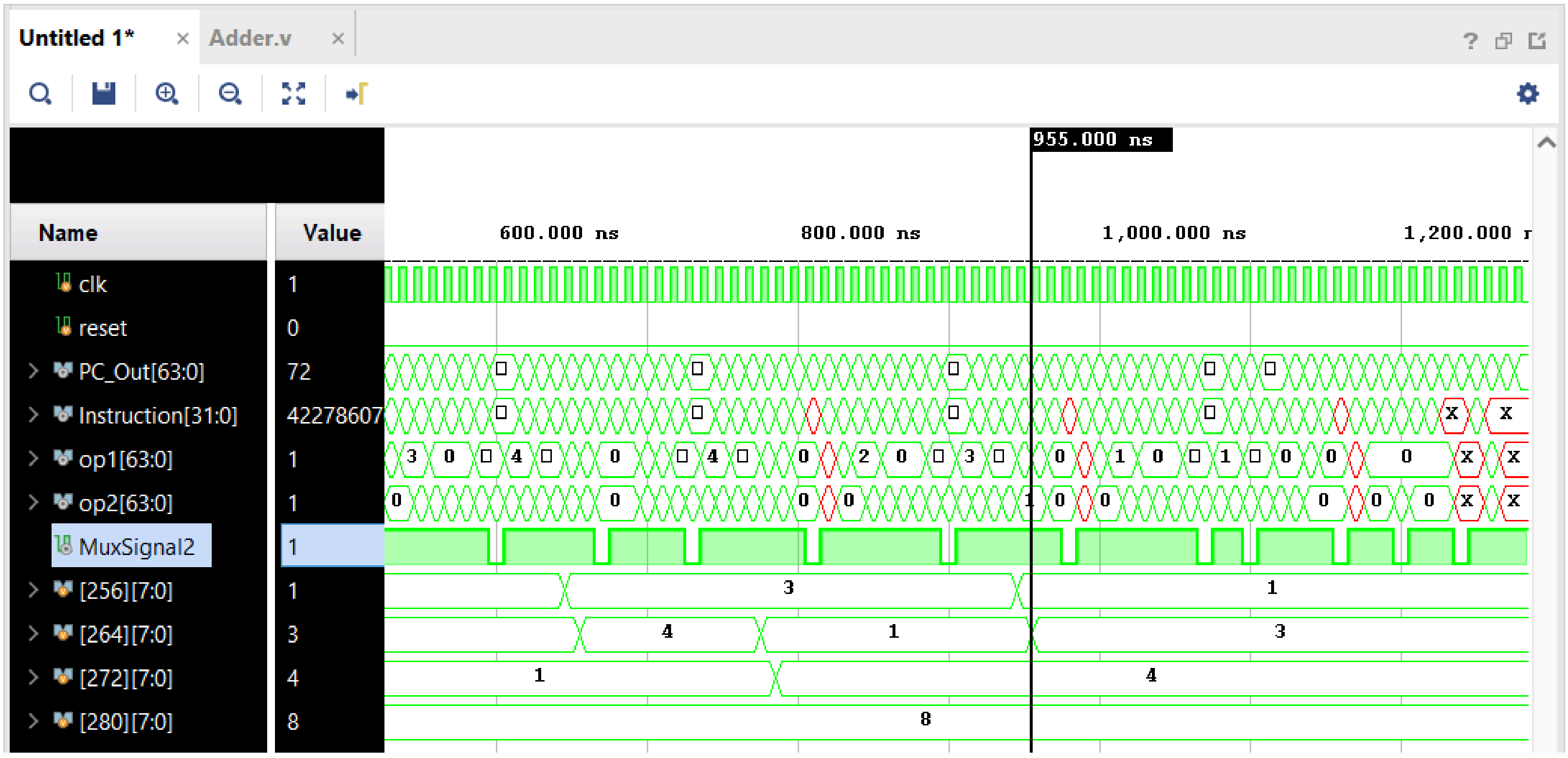
We see in the line result_next that there were potential values to be assigned to register 1 (the bottom line), however, we note that none of these were assigned, only 6, which is the result of the instruction we branch to. This shows that the pipeline is flushed of all previous results.

Bubble Sort:



Name	Value
clk	1
reset	0
PC_Out[63:0]	72
Instruction[31:0]	422786070
op1[63:0]	1
op2[63:0]	1
MuxSignal2	1
[256][7:0]	1
[264][7:0]	3
[272][7:0]	4
[280][7:0]	8

Figure 4a. First half of sorting



Name	Value
clk	1
reset	0
PC_Out[63:0]	72
Instruction[31:0]	42278607
op1[63:0]	1
op2[63:0]	1
MuxSignal2	1
[256][7:0]	1
[264][7:0]	3
[272][7:0]	4
[280][7:0]	8

Figure4b. Second half of sorting

Explanation: The bottom 4 lines are the registers in memory where our array is stored. We start with the same initial input, i.e. 4, 8, 3, 1. Overtime the appropriate substitutions take place (8<->3, 8<->1, 4<->3, 4<->1, 3<->1). The load-use hazards in our code are handled by stalls which are marked by a ☐ in the value of PC_Out. Moreover, the Mux_Signal2 denotes where the pipeline is flushed i.e. load-hazard stalling and branching. Our code has two loops, so the outer loop/jump can be noted by when the Instruction value becomes X (red) as it was close to the end of the program, so the code ends by the time the branch is evaluated.
Moreover, we see that the array is first sorted at 955.000 ns.

Task 4:
We note that the Single Cycle Processor took 2,700 ns to sort the data, while the Pipelined Processor took only 955 ns. This is a speedup factor of nearly 3 times. The cause of this speedup is that each clock cycle in the Pipelined Processor was 1/4th the duration, so even though the Pipelined processor needed more cycles, its overall execution time was reduced.

We note that the theoretical limit on this factor for our architecture would be a speedup of 4 times, as for some programs, the number of cycles in the pipelined version is barely greater than the number of cycles in a single-cycle processor.

The reason our specific example is unable to reach this ideal is a result of stalls and branches. There are cases in our code where a value is loaded just before use. This causes a load-use hazard which necessitates a stall in our architecture, which is essentially a lost clock cycle. Moreover, whenever we branch, as it takes 2 cycles to determine the result of the branch condition and 1 more cycle for taking the branch, this means that for each branch that is taken, we lose out on 3 clock cycles of work.

The performance of our processor can be improved in many ways e.g. Code Optimization to avoid hazards, better branch predictors, carrying out branch evaluation earlier to minimize lost clock cycles.

Challenges

- The main challenge we faced was in Task 2 where we had to figure out a way to implement branching in which we were having troubles with correct data forwarding. For example, initially the wrong output was being taken in MEM forwarding which led to undefined behaviour. This bug was found through repeated testing and simulations of data across different instructions. There were some misdiagnosis, for example, assuming the cause of errors to be undefined values, however, modifying those have perhaps made our processor more robust.
- For Task 3, there was no concrete implementation provided. The example in the book modified the original architecture significantly by moving the comparison module forward. To figure out how pipelining would work in our case required a systematic approach of breaking the problem down and implementing each action necessary to flush the pipeline and move it forward.

Task Division

- Musab worked on the overall project ensuring that everything is implemented properly in the top level module throughout all three tasks. Additionally, he worked on testing as well.
- Naaseh worked on implementing individual modules in between tasks such as the muxes, forwarding unit, and hazarding unit. Additionally, he also worked on generating the Bubble Sort algorithm.

Conclusions

- The project was a **major** success. We were able to get the ideal output and build a pipelined processor that we believe is robust for any program on our instruction set.
- The success of this project greatly depended on our consistency by working on this project from day 1 and striving to do a task each week + the teamwork that we employed, splitting up tasks that could be done in parallel, e.g. developing the module for the next pipe while the other connects the wires for the previous one.

References (CA_L7_Unit4_Slides) **-** Dr. Farhan Khan + Project Guidelines

Appendices

Musab1Blaser/CA_Project_CPU: Pipelined CPU comparison with Single-Cycle CPU for Bubble Sort (github.com).