EE380-001 Assignment 2: Multicycle Lab Project

Implementor's Notes

PAUL GRUBBS, CARLOS BORGE, and IAN THORNSBURG

This project involved us implementing 4 new MIPS instructions (or MIPS-like instructions as some of them are not part of the MIPS standard) to the provided verilog MIPS simulator. The instructions were a Set Less Than Immediate (slti) instruction, a simplistic 8-bit 'random' number generator (rand8) instruction, a 32-bit population count (ones) instruction and an atomic increment (inc) instruction. We were provided with some hints for the right methods to use to implement them so most of the project boils down to triggering the right components in the right order to implement the more-or-less known procedure to do them, based on our understanding of how this simple CPU is designed and operates.

ACM Reference Format:
Paul Grubbs, Carlos Borge, and Ian Thornsburg. 2025. EE380-001 Assignment 2: Multicycle Lab Project: Implementor's Notes. 1, 1 (February 2025), 1 page. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 GENERAL APPROACH

Our general approach was initially to review the provided macros and two provided instructions to get an idea of how each component worked. We had to do the same with the one provided decoding function which would return an arbitrary value if the provided instruction matched a refrence value for each instruction we had to implement. After reviewing the provided code we traced through the values of each register, making sure that all of the output statements were triggered before the input statements to ensure that we didn't have any errors associated with when values were actually latched from the bus. From there we just had to follow along each step of the process, loading the values we had into the right registers and then running whatever procedures we had to (such as ALU operations or memory reads/writes) to complete the step before moving onto the next one.

The rand8 instruction updates a seed by computing: rd = (13 * rs) % 256 Using the identity 13 = 8 + 4 + 1, the operation is implemented via shift-and-add (rs shifted left by 3 for 8*rs, by 2 for 4*rs, and adding rs) followed by a bitwise AND with 0xFF. The instruction is encoded with OP=0 and FUNCT=1.

The most efficient population count instruction for a machine without a fast mulitplier was implemented. The instruction was tested with the hex value F0F0F0F0 and it gave the correct result of 00000010 or 16 in decimal.

2 ISSUES

One of the biggest issues we faced was a simple unfamiliarity with the simulator and how it was designed. While there were enough references to figure it out eventually, the references available weren't fully comprehensive and it required piecing different bits and pieces together to get a full picture of how things worked. For instance, neither of thw two provided instructions used an immediate value meaning that the only way to figure out how to use that was to go back and look through the list of available 'define' macros and find the "IRimmedout" one.

Another issue was that initial implementations used state numbers above 255, which caused decoding errors due to the 8-bit width of the state register. Renumbering the microcode sequence to use values below 256 resolved this issue.

We also found that, despite the example code not doing so, we had to call the "NEXT" macro even after calling the "UntilMFC" macro. While we don't know why the provided examples do not do this, we found that failing to would result in the code simply looping at whichever state called UntilMFC until reaching the execution limit and stopping.

Finally, we discovered that if you attempted to write to a register that was not initialized it would appear to not work, even though the exact same instruction would work if writing a register that was fully initialized. In other words if a register isn't initialized at the start the program, it can't be used at all later, even when only writing to it.

Authors' address: Paul Grubbs; Carlos Borge; Ian Thornsburg.

XXXX-XXXX/2025/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn