EE480 Assignment 3: Quicker Logick

Implementor's Notes

Owen Tanner Wilkerson, Justin Moore, and Nick Early
Department of Electrical and Computer Engineering
University of Kentucky, Lexington, KY USA
tanner.wilkerson@uky.edu

ABSTRACT

This project's main goal was to implement a pipelined processor that executes the Logick assembly language. Our team's implementation includes 4 stages which have the following generalized descriptions: instruction fetching, register reading, ALU processing, and register/memory writing. This following document will give a deeper briefing regarding our approach for this implementation.

1. GENERAL APPROACH

The basic grouping of this system within verilog is composed of three separate modules: decode, ALU, and processor; with processor being the largest and primary module of this solution. In this description of our general approach, we will step through each of these modules in order to describe the purpose and overall implementation thought process for each.

1.1 Module: Decode

The decode module's role is essentially as the name hints; to decode the instruction the gain some additional knowledge on the needed approach to a certain instruction. This "approach" mentioned is simply whether or not a given instruction needs to write to a register. So, the Decode module consists of a case statement that looks for the opcodes being grabbed by the instruction fetch that correspond to instructions that are known to not write any results to a register. Therefore, this module simply communicates two things: passes through the opcode for a given instruction and the address of the destination register where the results of the given instruction are to be written to, this address is communicated by a variable called "regdst" within the code. For the instruction that do not write any information to a register, the opcode of the instruction and a value of 0 for the destination register is communicated by this module. All other instructions decoded by this modules are defaulted to copying the destination register to the "regdst" variable and passing through the opcode. This simple information is used in few mechanics external to this module. One of these mechanics is obviously communicating whether to expect any information to write to a register after the ALU processing step. Additionally, the register write information is also used in the value forwarding mechanic, which will be emphasized within the "Module: Processor" section later in this document.

1.2 Module: ALU

This module is nothing out of the ordinary for the standard idea of an arithmetic logic unit. Given some opcode, optional information needed for extended opcodes, and two inputs, the ALU module simply executes the corresponding arithmetic operation. For this implementation, we put the ALU in charge of the computations for the following instructions: ad, an, eo, mi, no, or, sr, and co. Any other operation based instructions that were not just stated are handled by a secondary always block within the "Processor" module that integrates with this greater ALU module. The only implementation specific concept seen within this ALU module is just that our default case, which executes when unknown opcodes are encountered, resorts to a pass through of the "a" input. These default cases are theoretically never hit except during some intermediate cycles that do not utilize the ALU's output, but it is still best practice to attach a default to all case statement within verilog. Finally, it may be important to note that this implementation has the ALU as the owner of the condition code registers.

1.3 Module: Processor - Stages

The "Processor" Module is the primary module and certainly the most complex module seen in this implementation. This module encapsulates the flow of control logic, intermediate buffer filling, value forwarding logic, and instruction squashing logic. Let us begin with the flow of control and intermediate buffer filling. This processor implements the standard "owner computes" mechanic; therefore, every stages, with the exception of the final stage who does not have the need for a buffer, only writes to its proceeding communication buffer. These buffers are used to communicate to all following stages down the pipeline. These pipeline stages are as follows: stage0(instruction fetch and PC refreshing), stage1(register reading), stage2(ALU operations), followed by a final stage3 (register writing).

Stage0 is a fundamental instruction fetch implemented inside of a clocked always block. Upon every clock tick, this stage accesses the instruction register, fills its communication buffer, and refreshes to PC with an externally computed value, more on this value later.

ACM ISBN N.A..
DOI: N.A.

Next, stage1 acquires values corresponding to the register addresses that were communicated by the stage0 buffer. Stage1 then fills its buffer with all the data given to it by stage0 along with these freshly accessed values to accompany the register addresses. Stage2 is where ALU module integration and the addition of some instruction handling not covered by the ALU module takes place. The always block fundamentally just passes through the given needed information for register writing along with the outcome of any needed arithmetic operations. Yet, the line where the operation output gets assigned, seen here:

stage2Value <= ((stage1op == 'OPsi) ? ((stage1dstValue 8)|({stage1src,stage1Tsrc}&8'b11111111)) : ((stage1op == 'OPli) ? stage1srcValue : ((stage1op == 'OPlo && stage1dst != 0 && stage1dst != 1 && stage1Tsrc == 2) ? mainmem[stage1srcValue] : ALUResult)));

can look a little confusing, but it is quite simple. If the instruction current in stage 2 is an instruction handled by the ALU module, then this line simply assigns the output value of the ALU module to the corresponding stage2 buffer value. Yet, this line also adds some functionality for instruction not seen within the ALU. These instruction were not included within the ALU module for a variety of reason, such as needing access to an output that no other instruction needs access to or possibly it just made better sense to have the instruction housed within the main module of the program for ease of debugging and code legibility. There instructions include si, li, and lo. Therefore, that line of code is nested if's that are looking for and handling instructions si, li, and lo before requiring that the ALU module handles the instruction at hand. Below that line, is can be seen that this always block also handles the command st and sy on their own line.

Finally, an optional stage is present for the writing of information to registers. Calling this stage "optional" maybe be an abuse of terminology, but the term "optional" is only used to describe the nature that this stage is only executed if the instruction at hand has information to write to a register. Viewing this always block, it can be seen that there is not much to explain regarding its implementation.

1.4 Module: Processor - Additional components

Also within the Processor module, a few other always blocks can be seen. These additional components include value computing, value forwarding, and instruction squashing logic. The value computing and forwarding are handled by the always blocks seen setting srcValue, TsrcValue, and dstValue. The value forward logic is as simply as searching for address dependencies down the pipeline by examining reach stages buffers. If a dependency is found by finding a matching register address for src, Tsrc, or dst somewhere down the pipeline, then the given value is assigned to the output of the stage were the dependency was found. If no dependency is found, the a simple register file access is performed. These value compute blocks are the primary workers behind the register read stage, stage1.

Lastly, the squashing logic seen in the always blocks assigning rrsquash and isSquash. This is quite simple, by default, all branches fall through. Then, once reaching stage1, if the condition codes, coming from the ALU, state that the branch was indeed taken, then squashing is triggered. It can be seen that rrsquash is certainly redundant code, but was left in just in case more advanced squashing logic needs to be added in the future.

2. TESTING AND ISSUES

2.1 Testing

All testing function are contained in the *Testing.txt* file. This file contains multiple small blocks that aim to test the logic correctness of the operations defined in our assembly code. Each one has its own derived process for testing, however many of the operations have similar structure. For example, many of the operations tested from the ALU will place test operands into the operation to be tested, get a result from the operation, and use compare to see if the result matches a predetermined correct answer that will be stored in memory. If the correct answer was not loaded, a branch is not taken and the instructions fall through to a sy call, which halts the operation. In order to test branch, the constant condition registers t and f were used to examine the behavior of the branch operation and its effect on the program counter.

For additional tasting assistance, the Vivado Development environment was used for tracing and general debugging.

2.2 Known Issues

For the most part, this assignment went much better than the last. Yet, there are some issues to report. First, my testing of load and store seems to result in unsuccessful tests, BUT when examined separately both the ld and st commands seem to behave as expected. Yet, when used together, for example a program like the one seen is *Testing.txt* that performs an ld then a st followed by a co to test for correctness, the ld and st seem to behave perfectly fine when examining system traces, but the co have issues. Yet, co never has any issue in any other tested instance. Therefore, either my testing program for load and store together is incorrect, or a part of the code is becoming asynchronous because of the load store together. Of course, it is possible that the error is not that complex and is hidden by a guise of simplicity.

Additionally, there is a risk of malfunctioning condition code forwarding. We attempted to simulated conditions were this could happen, but forwarding and squashing seemed to behave as expected. Yet, there is a certain degree of poor confidence in this handling, but this uncertainty could always be triggered by the complexity of a project like this one.