EE480 Assignment 4: Reversible Pipelined AXA

Implementor's Notes

John Adams, Andrew Crittenden, Alex Elam
Department of Electrical and Computer Engineering
University of Kentucky, Lexington, KY USA
john.adams7@uky.edu, crittenden.andrew@uky.edu, alex.elam@uky.edu

ABSTRACT

The objective of this project was to implement reverse execution as a method of handling errors. When the execution of the code sees an illegal instruction (for this project, it is the misuse of the exchange command), it reverses the execution until there are no more errors detected in which case it begins forward execution again.

1. GENERAL APPROACH

We reused the AIK specification from the last project. The encoding can be found in the axa.ods spreadsheet or in the axa.aik AIK specification. We then decided to use five stages in our pipelined design: Stage 0 Update PC, Stage 1 Fetch Instruction, Stage 2 Read Registers, Stage 3 Read/Write Memory, and Stage 4 ALU & Write Registers. We decided to combine stages 4 and 5 from the last project so that stage 4 would own the halt variable. We wanted to be able to halt instructions easily when handling the fail instruction. We set up vmem0 as the register memory, vmem1 as the text memory, and vmem2 as the data memory. We changed the undo stack size from 16 to 256 registers. We then added the definitions for the 4 signals, SIGILL, SIGTMV, SIGCHK, and SIGLEX. We created two 4-bit registers check and errors to handle the reverse execution. We also created a variable for each stage labeling its direction. For example, when s0fwd is 1 the instruction runs forward, 0 would run backward. In stage 0 we set s0fwd based on if there are any errors, then we increment or decrement the PC depending on whether the processor is running forward or backward. Our processor only checks for the SIGILL instruction. In stage 2 after we fetch the instruction, we check if the operation is an exchange and if it is the correct type. If it is the incorrect type we set the SIGILL bit of errors. We then change the illegal instruction to a NOP.

The definition of the ALU varies from that of assignment 3 due to the added support for reverse execution. To implement this change, we included a definition called DREST that is used to restore the value of s4alu using the value at the top of the undo stack. DREST is used in this manner to define the reverse execution for the lhi, llo, shr, or, and, and dup instructions. The reverse execution of the remaining instructions are defined as follows: The add and sub instructions simply invert the value to be added to s3dst so that in reverse add becomes sub and vice versa. The rol instruction becomes a rotate right. The ex, xhi, and xlo instructions don't change as they are reversible by definition.

If the instruction is jerr mask and it is in forward execution, then we add the mask to the set of exception conditions we are checking for (check variable). If it is running in reverse we remove the mask from check and we remove it from errors. Then if all the errors have been resolved we flush the pipe with NOPs and jump to the address at register $d. The processor will then go into forward execution at that address. Next we added the com instruction. In forward execution com sets check to 0 since we know we no longer need to check for errors. In reverse execution it sets errors to 0 since we know that there are no errors prior to the commit. For the fail instruction in reverse execution it acts as a NOP. In forward execution we need to compare mask and check. If we raise an exception that we were not checking for the machine halts (mask & ~check != 0). Otherwise we set errors to the mask of the bits we were checking for. This will begin reverse execution when stage 0 checks if errors is equal to zero. Our processor now supports reverse execution. Jerr handles the error conditions we are checking for and switches to forward execution when there are no remaining errors. Fail mask reverses the execution when mask sets an error.

2. TESTING

We reused the testbench from the last project to test the ALU instructions. We wrote the test code tests_2 to test the new functionality of reverse execution. First it tests fail and jerr when a SIGTMV is set. It then tests the SIGILL instruction. We catch the illegal exchange instruction. It tests land restoring the last pc from the undo stack. It tests when jerr runs in reverse and needs to continue in reverse instead of jumping.

3. ISSUES

There are multiple op-codes unused by our AXA encoding; executing instructions with these unused op-codes will result in undefined behavior. Data and instruction memory out-of-bounds checking is not performed, but is handled. Our implementation of fail and jerr is not efficient since we add noops for both forward and reverse execution. We only need the noops when the jerr instruction should jump or the fail instruction halts.

This pipelined implementation does not perform speculative execution or value forwarding, and therefore frequently encounters pipeline bubble

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