EE480 Assignment 3: AXA Pipeline

Implementor's Notes

Peyton Moore, Michael Probst, Alex Schuster
Electrical and Computer Engineering Student
University of Kentucky, Lexington, KY USA
Peyton.Moore1@uky.edu, M.Probst@uky.edu, Alexander.Schuster@uky.edu

ABSTRACT

This is a pipeline processor design that using the AXA instruction set created by Dr. Hank Dietz. The instruction set is design to have forward and backward instruction sets. For simplicity sake, this processor design is only designed using the forward instruction set and contains the UNDO buffer for reverse instruction sets.

1. GENERAL APPROACH

This processor design was created following the AXA assembler instruction set provided by Dr. Hank Dietz. All instructions are implemented with the forward instruction only and push and pull to the UNDO buffer needed for reverse instructions.

The pipeline design is based on a multi-cycle design done previously For the design of the pipeline, there will be five stages. Stage 1 is an instruction fetch. Stage 2 is a register read. Stage 3 is a data memory access. Stage 4 is an ALU operation and the register write.

Stage 1 will grab the instruction from memory and pass it along to stage 2. Stage 2 will grab register values. Stage 3 will access data memory if needed. Stage 4 will execute the instruction and write to the destination register when needed.

The pipeline design is a design that can handle multiple instructions in a more efficient manner than a multicycle design. It is designed to be able to operate multiple stages in a single clock cycle. A pipeline can also halt the processor in any stage.

2. STAGE 1, FETCHING

Stage 1 is responsible for fetching the instruction from instruction memory, determining how to increment the pc, and handling values sent to the undo buffer if a land instruction was detected previously. Stage 1 is laid out to only fetch an instruction from memory if it detects that stage 2 (register read) is ready to receive that instruction, it does this through the âĂIJwaitâĂİ conditional. If it passes this than the instruction is then passed on to the second stage of the pipe. There are three Boolean variables to detect if there was a jump, branch, or land. These are set to true, or 1, in the 4th stage of the pipe for the previous instruction. The 1st stage detects if these were set or not and will handle the pc incrementation accordingly. As for the land, when it was detected in the previous stage it will push the value of the pc to the undo stack buffer and then increment the index of said stack- this is also handled through a Boolean variable.

3. STAGE 2,REGISTER READ

Stage 2 is responsible for handling register reading of the pipeline. This involves reading the src value of a register that is sent with an instruction. The stage checks data dependencies between stage 1 and this stage incase stage 1 tries to send an instruction that uses the registers of the current instruction before it is executed. This is done by using functions that checks the register addresses. This stage also stores the destination register in the undo buffer if the following instructions are given: **lhi,llo,or,dup,and,shr**. When the registers are read from the instruction is passed to Stage 3.

4. STAGE 3, DATA MEMORY

Stage 3 is responsible for handling the data memory accessing of the pipeline. If an address to a memory is sent with the instruction. This stage will handle accessing the data. If there is no address sent, then Stage 3 will pass on the src type found in Stage 1 or Stage 2 depending on the type via a mux to Stage 4. If the data contains info for a jump, that is fed back to stage 1.

5. STAGE 4,EXECUTE AND WRITE

Stage 4 of the pipeline will handle ALU operations and writing to the register file. This stage will perform its function when once stage 3 has completed its most recent operation, which is controlled using ir3. An opcode is retrieved from ir3 which determines what instruction needs to be executed. If an instruction is a branch or jump, the branch or jump flag is set so that stage 1 can be prompted to handle it. All instructions that write to a register load the result of their computation into the res register. If there are no WAW dependencies at the end of computation, the destination register will be set to the value of res, otherwise, it will wait until there are no data dependencies.

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

6. UNDO BUFFER

The UNDO buffer is a function of reversibility that allows the processor to reverse an instruction. The UNDO buffer can handle sixteen 16 bit registers. This design can only push to the UNDO buffer.

7. TESTING

The testing strategy focused on maximizing line coverage while ensuring complete functionality of the processor. Therefore, each instruction of the AXA instruction set was put through the processor to ensure that each instruction was working properly. Additionally, instructions that would have WAR and WAW dependencies were input to test the handling of those data dependencies. Since the undo buffer is only being pushed to and not popped from, it is quickly filled so multiple test programs were made to ensure that the undo buffer was not overflowed. Each instruction source, destination, and result was printed for checking to make sure the processor was correctly operating. The input of the UNDO buffer was checked when applicable to ensure correct values were written.

8. ISSUES

One noteworthy issue is that our UNDO buffer can only handle 16 instructions at once. We do not pop out anything from the UNDO buffer.
Successive jumps/branches cause bugs where the target of the following branch is executed after the path of the first jump/branch is terminated.

9. REFERENCES

This implemented used PinKY from EE480 Fall 2018 pipeline as a reference.
This can be found at
http://aggregate.org/EE480/pipepinky.html