Assignment 3: A Faster IDIOT

Implementor's Notes

Dylan Wright, Zachary Davis and Kristina Shaffer
University of Kentucky Electrical Engineering
EE480 - Spring 2016
Dylan.wright@uky.edu, zrda222@g.uky.edu, Kristina14as@gmail.com

Abstract— The primary objective of this assignment involved the implementation of a multi-stage pipeline IDIOT instruction set using the AIK assembler, the Verilog Design Language and a detailed test bench to implement full coverage of the different modules and logic of the design.

I. INTRODUCTION

This document describes the fourth Project in the EE480: Advanced Computer Architecture course. Assignment 3: A Faster IDIOT is a group project with a primary objective of creating a four or more stage pipelined Verilog implementation of IDIOTs non-floating-point instruction set architecture. This project implementation includes designing and implementing the encoding of the IDIOT instruction set along with using the AIK assembler to output the encoding, which is then used as the input of a single-cycle implementation of the IDIOT processor and memory. The project is to also include a test plan which ensures thorough test coverage of the design.

II. GENERAL APPROACH

Due to the complexity of this project and the multiple different modules that needed to be designed and implemented, the first step for our project began with the formation of a top down design. A top level diagram (seen in **Figure 1**) was created to represent the flow of data and the integration of each module into the overall design.

After constructing the top down design, the AIK specification for the project was formed as *IDIOT_Spec*. This specification primarily involved specifying IDIOT ISA instructions using the conventions discussed in class, for example all floating point instructions are actually system calls (or HALTs) which can be implemented as a special case of the jz instruction.

The modules for this project were divided conceptually into 3 separate stages for the pipeline implementation.

- Stage 0: Instruction Fetch stage included the PC counter and the *InstructionMemory* module.
- Stage 1: Register Read stage which included the *RegisterFile* module.
- Stage 2: ALU/Memory stage which included the ALU module as well as the *DataMemory* module.

In between each of the 3 stages there existed a pipeline register which would contain signals and information (opcodes, destinations, etc.) which would be used in different stages of the processor.

- Pipeline Register 1-2: which contained a write back signal, 4 bit opcode, 6 bit destination, and write not read signal
- Pipeline Register 2-3: which contained a write back signal, 6 bit destination, 6 bit data, and a 4 bit opcode

The included signals were to indicate whether the operation was a read or write to memory to determine how the processor would handle the operation. The different stages that the operations would go through depended on the operations themselves.

1: All Instructions (excluding ld, st, jz, sz, and sys)

Each of these operation were to go directly from the program counter to Instruction Memory to read Register File to ALU to write Register File, with the program counter incrementing by 1.

2: Load/Store

The operations for load and store were treated like the other ALU operations with the exception that they substituted the Data Memory for the ALU. The store instruction specifically was to write to memory rather than read from memory, therefore, its read/write signals was the only one that differed from the other operations.

3. Jump Zero

The jump zero instruction was to put the source register's value into the program counter if the destination register held the value zero.

4. Skip Zero

This operation was to increment the Program Counter by 2 if the destination register held the value zero which would simply skip the first bit position.

5. Load Immediate

The *li* instruction was implemented by inserting a bubble into the pipe and storing the destination address to be used in the next cycle. By setting the op code for the next cycle to

OPdup the immediate value could be sent through the pipe as if it were a regular instruction. This approach also simplified handling forwarding by making the instruction no different from a dup whose source data value was the same as the immediate. The *sz* instruction does not handle squashing a *li* instruction.

III. Encoding Instructions

A. Instruction Specification

Since the IDIOT Instruction Set Architecture possesses 18 different instructions, we began by creating encodings for all 18 instructions. We decided to use and implement Professor Dietz's encodings/AIK specifications in the Assignment 2 solutions that were given. Five of the 18 instructions were floating-point instructions and although they were not implemented within this project, their encodings were still included. Each instruction was designed as one 16-bit word long, with the exception of `li`, which were actually two 16-bit words long. Four bits were used for the opcode and six bits were used for each register. These numbers of bits were implemented while knowing that the IDIOT has 64 "user-visible registers", which each need 6 bits, leaving 4 remaining to use for the opcode. The only issue here is that all 18 instructions will not be able to be distinguished individually in those 4 bits. Knowing that all of the control flow instructions (`jz`, `sz`, and `sys`) can be written using `jz`, they all share a single opcode.

B. The Assembler

After coding the instruction specifications, they're functionality was implemented by building the AIK specification of the assembler, which essentially created an executable documentation of the instruction encoding. The approach taken listed one pattern per instruction then specified the output from the assembler. We designed our memory with separate code and data memories.

IV. Implementation

A. Program Counter

The program counter holds the values of the instruction being executed and increment the stored value by one. It was to take in the inputs from addr and for each of the operations, it would increment by 1 with the exception of the sz instruction where the PC would increment by 2.

B. Instruction Memory.

The Instruction Memory was implemented as a 64 word (16 bit) deep memory. It inputs a16-bit register address which will be sourced from program counter. The instruction is then collected from memory and with the instructions as the input, the instruction memory passes on the bits of the instruction to the outputs such that the top 4 most significant bits are put in the opcode output, the next 6 bits are placed in the destination register and the bottom 6 bits are put in the source register.

C. Register File

The register file was implemented as a 64 word (16 bit) deep memory. The register file inputs three different 6-bit address registers: *addr_s* and *addr_d* which are 2 of the outputs from the *Instruction Memory* along with *addr_i* which will be sourced from the pipeline register between stages 2 and 3. The register file also receives a signal (write) from the write back stage telling it when to latch the write back value.

D. ALU

The ALU was implemented as a simple combinational logic circuit which was to decide what to do with the contents of the two 16-bit input registers which were obtained as the outputs of the *RegisterFile* module that were fed through each of their own MUX's to treat the inputs accordingly based on if the operation needed to read or write to memory. The ALU module switches on the opcode and uses high level code to determine the logic to carry out for the implementation of each operation. The output is a 16-bit value that goes to the result register. The default operation was set to the Dup instruction which simply set the result register to the value of the X input value.

E. Data Memory

The Data Memory module took in two 16-bit inputs *addr* and *data_i* which were the outputs from the two multiplexors which followed the *Register File* module, along with a write not read signal. The only operation that would receive a 1 for this signal was a store instruction. If the write not read signal was a 1 (indicating a write) then the memory at the index of the *addr* would be written as the input *data_i* (storing this value into memory) then the output of this module would become the *data_i* values. Otherwise, if the signal indicated a read from memory, the output *data_o*, would become the values of the memory at the index of *addr* (mem[addr]).

F. Memories

There are 2 kinds of memories used in the processor, which are the synchronous RAM and the register file. The synchronous RAM memory uses the address of R0, which is 6 bits, and the contents of the source register, which is 16 bits, as the main ports. The output enables and read/write bits are also used within the RAM.

Second, the register file takes in the address from the FSM and the data immediate from the MDR, then outputs the data_out, which goes into a multiplexer that gets sent to the ALU. The load/store signal is also used in the register file.

V. Test Plan

The test plan proved to be the most difficult piece of the project. Although we will walk through the test plan here, please refer to the testbench that implements the plan for more details. Although this project does not require us to worry about implementation test issues and "design for testability", we designed a testbench to make us more confident that the design is logically and functionally correct. Our main goal was to make sure that the test cases cover about 100% of all the statements. The use of self-testing programs simplified testing. The test bench relies on the test program being loaded into the text section (Instruction memory) and data section (Data memory). The bench then supplies a clock to exercise the processor. This approach relies on the test program testing

that the proper values for operations are produced and correctness is determined if the program halts at the correct location (determined from the final PC value).

For this project the group reused and extended the test framework created by Team 5 (of which Dylan Wright was a member). The framework uses several scripts to make the use of the cgi forms unnecessary. This framework allowed to the group to produce test programs more easily by simplifying the development environment. Files in the IDIOT/ directory with an .idiot extension are IDIOT assembly language programs and files with a .text.vmem or .data.vmem extension are memory images designated for the text and data sections of memory respectively.

The main test program is called prog_test.idiot. This program tests the alu operations, jump operations, and li operation. This program's memory images are currently the ones in pipe.v..

VI. Issues

A. *Load Immediate (Li)*

Li instructions do not work if there are two in a row. Also jumps do not squash these instructions properly. This is because of an issue with the squash logic. This issue could be resolved by increasing the complexity of the jump logic.

B. *Forwarding*

Forwarding works in cases where there is 1 dependency but there are situations where a second dependency cause dependency not to be resolved. This occurs due to a problem in the forwarding unit where it recognizes dependencies between the register read and alu stages but not between the alu and write back stages

References

[1] Dietz, Hank. Icarus Verilog Simulator CGI Interface. http://super.ece.engr.uky.edu:8088/cgi-bin/iver.cgi

Figure 1: Top level design