Assignment 2: The Making Of An IDIOT Implementor's Notes

Dylan Wright
dylan.wright@uky.edu
Casey O'Kane
casey.okane@uky.edu

***Abstract*—Goal of this assignment involved the implementation of the IDIOT instruction set using the AIK assembler, the Verilog Hardware Design Language and detailed test plan to exhaustively test the different components and logic of the design.**

I. General Approach

Due to the complex nature of this assignment, the general approach first involved forming a top down design, a comprehensive finite state machine (FSM) for the ISA instructions and then a far more specific FSM. Diagrams were created for these designs and they can be found as **Figures 1 through 7** respectively in the **Appendix** of this report.

One key decision that was made over the course of this project included the decision to select a Harvard Architecture as a model for the memory unit rather than a Von Neumann model. The group ultimately decided to go with this method as it would allow them to simultaneously read or write instructions to or from memory.This decision is reflected in the AIK specification that was created `IDIOT_Specv` found in the `IDIOT` directory of the provided tarball).

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 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.

Once the AIK specification was created, skeleton modules `alu.v, control.v, memory.v, processor.v,` and `register_file.v` followed. Initially these files were written with basic functionality and later expanded using a top level approach, by then setting up `processor.v` and the completing the modules that it instantiates. `signals.v` contains many of the signals that would be used to communicate between devices along with a few global constants utilized throughout the Verilog code (the 16 bit register constant WORD for example).

Speaking more to `processor.v`, it instantiates the other necessary modules and then uses one level sensitive and two edge sensitive always blocks to simulate the processor. The specifics of these operations are detailed in the diagrams represented in the **Appendix** of these notes

After completing all of the mentioned modules, associated test benches were constructed as stated in **B. Verilog Modules** of the **Testing** portion of these notes and issues were resolved as they arose.

II. Implementation

This section describes how each module was implemented.

A. ALU

The ALU was implemented as a simple combinational circuit. The module switches on the opcode and uses high level code to implement each operation. This may note produce the minimal solution, but it works. Dup did not need to be implemented as an ALU operation but it was to make the control flow simpler.

B. Register File

The register file was implemented as a 64 word (16 bit) deep memory. It inputs a register address which will be sourced from the control logic in the module which instantiates it. This address is used to select which register to save into. The first 8 registers each receive an initial value.

The read only registers are only read only in the sense that the proccesor trusts the programmer/compiler to not write into a read only register.

C. Memory

The memory is implemented in the same way as the register file but with a deeper address space. Otherwise it is essentially the same.

D. Control

The control module implements a FSM which defines which control signals to set when. This module also can control the bus, and read the instruction register. This module has two primary sections; the combinational and sequential sections. The combinational section determines the next state. The sequential section determines which signals to set to what.

E. Proccesor

The proccesor module instantiates each of the other modules and connects them. It arbitrates access to a bus based on the control signals from the control module.

III. Testing

A. Instruction Set Architecture

In order to test the IDIOT instruction set specification a test framework was implemented. This framework is in the `IDIOT/` directory. The framework consists of the following files:

1) `aik.py`: To automatically test files `aik.py` sends a PUSH request to the AIK cgi program. The returned html page is parsed and each section is output. The `.text` and `.data` sections are sent to stdout and the assembler messages are sent to `stderr`. This method is not ideal, an AIK executable would be preferable. Sample run:

```
$ echo "file.idiot" | ./aik.py
```

2) `diss.py`: To make test results human readable, `diss.py` disassembles a `.out` file (the `.text` and `.data` segment of the output of `aik.py`). The code is converted to binary and displayed in tabular format. Sample run:

```
$ echo "file.out" | ./diss.py
```

3) `test.sh`: This file can be used to test each `.idiot` file in the `progs/` directory. This script runs each file through AIK and compares the output to the expected output. `.text` and `.data` segment expected output should be placed in a file with the same name as the program and a `.expected.out` file extension. Expected assembler messages should be placed in a file with a `.expected.err` extension. The test script will report the number of passed, failed and possibly failed tests. This test framework was adapted from a script provided by Dr. Jaromczyk in the Fall 2015 CS 441G: Compilers course. Sample run:

```
$ ./test
```

B. Verilog Modules

Every Verilog module included in the provided tarball has an associated testbench file, as denoted by the *_tb.v extension that is used for each module. All of the tests can be run using the `test.sh` script in the verilog directory.

1) `alu_tb.v`: This testbench excises the ALU module defined in `alu.v` by reading in test vectors and checking the ALU's output. The test vectors are in `tests/aluXvector.vmem`, `tests/aluYvector.vmem`, `tests/aluZvector.vmem`, `tests/aluOpvector.vmem`.

2) `memory_tb.v`: This testbench exercises the memory module defined in `memory.v` by setting each cell's value to its address. Next it checks that each cell's value has not changed.

3) `register_file_tb.v`: This testbenc exercises the register file module defined in `register_file.v` with the same methodology as the memory test bench.

4) `proccesor_tb.v`: This testbench exercises the control logic and proccesor connections. It does so by providing a reset and clock to the instantiated proccesor module (defined in `proccesor.v`). The memory module is initialized with a vmem file. By default this is `tests/proccesor/proccesor-custom.vmem`. This contains a custom IDIOT assembly program assembled using `aik`. This program performs a simple loop. After that it executes the thus far ALU instructions, squash, store, and load. Then it ends. Because this program uses every instruction it produces nearly complete coverage. The output of covered for this program is included as Documentation/covreport.txt

The other primary test vector for this testbench is `tests/proccesor/proccesor-test-non-trivial.vmem`. This contains a non trivial program generated by `idiocc` and `aik`. This testbench is also used to run the other proccesor vmem tests by `test.sh`. This test was produced by a slightly modified version of `idiocc`. This version is included in the `IDIOT` directory. The jump to main in this version is fixed to include an underscore.

C. Utilization of GTKWave

GTKWave was used extensively to debug timing issues with the control. It was helpful to see when signals were being changed. To this effect, it was advantageous to write several file filters to make reading waveforms easier. These files are in `verilog/gtkwave-config/`

D. Testing Results

The results of running `test.sh` shows that the design passes the functional test suite (the ALU, register_file, and memory tests). This means that the design modules appear to be correctly implemented. The results of running `proccesor_tb.v` using the custom program test vector. This program loops 10 times. This behavior can be confirmed using GTKWave.

IV. Issues

A. Known Errors

It was noticed that for the non trivial test, the In addition, to the previously described errors it might help to view the following **Notable Workarounds** section.

In the non trivial test, there is a possible error. The program reaches the code generated for exiting the function but instead of returning and reaching the sys instruction (halt), it reaches the `jz $u0, $ra` and then starts at the beginning. This may be due to several things. There is no return in the c program used to generate the test vector, the idiocc compiler may be producing incorrect linkage, or there may be a problem in the design.

Also, it might help to view the following **Notable Workarounds** section of this section, that might be another location were a problem might arise.

B. Notable Workarounds

1) Instruction Register Assignment: In processor.v, when assigning to the instruction register, it was decided that rather than storing the value stored on the Bus, that the value stored in the MDR should be used as the IR only reads from the MDR anyway.

2) Control FSM Timing: The original design for the finite state machine was much smaller than the final one present in the project. In order to ameliorate timing issues, a number of state were expanded into either multiple states or lengthened. This was usually due to a register not latching in time or the bus not having the right value. While this is not an ideal solution to the problem, it does work.

Appendix

A Top Down Design

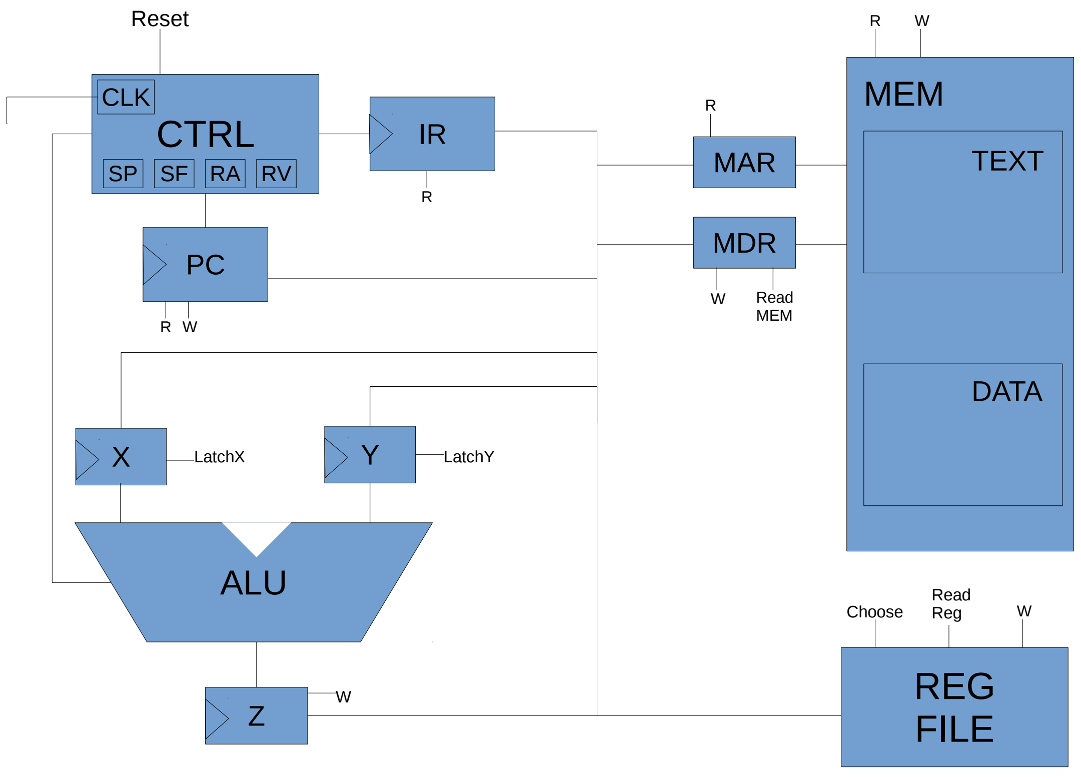

Figure 1: Top Down Design

B General FSM

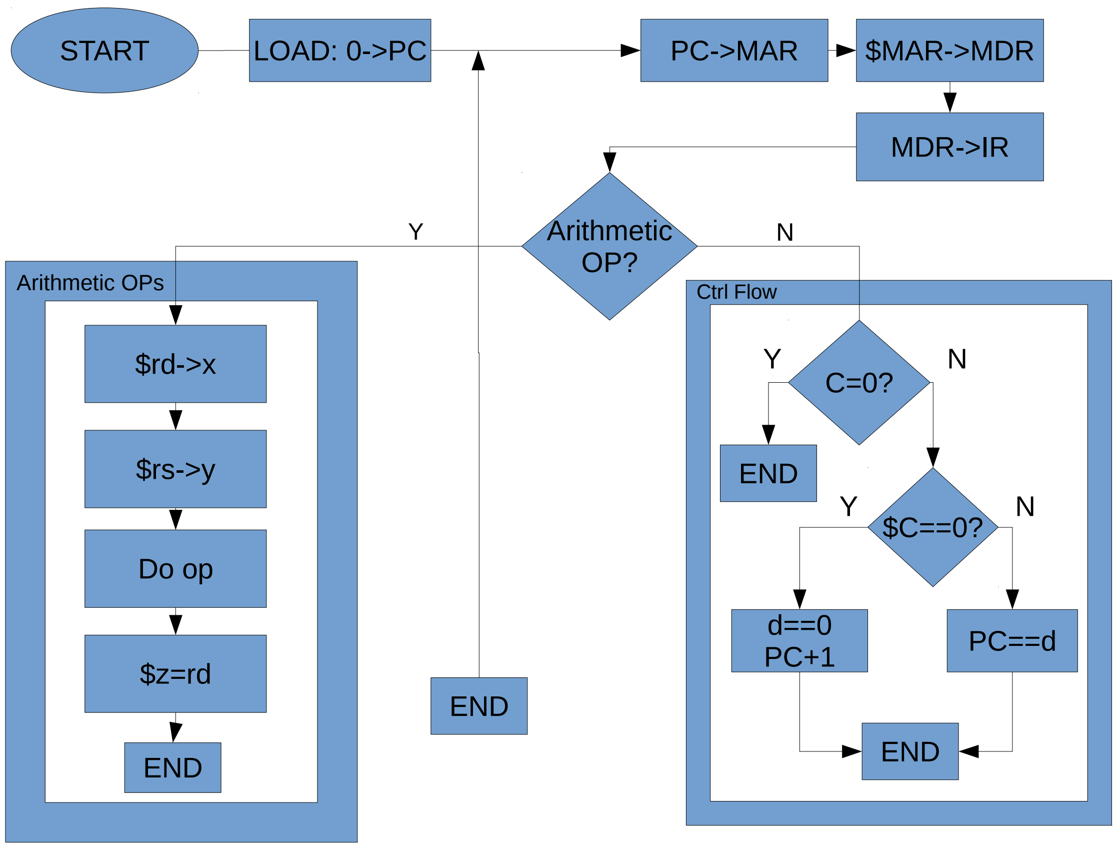

Figure 2: General FSM

C Top Down FSM

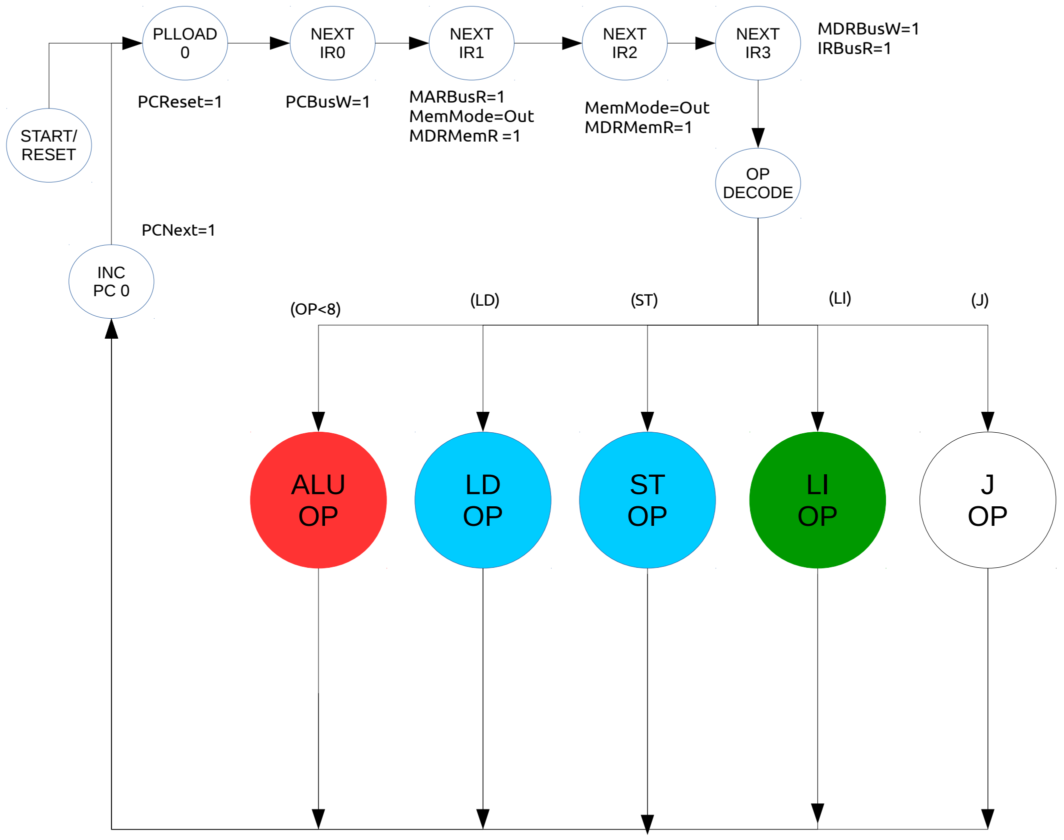

Figure 3: Top Down FSM

D ALU FSM

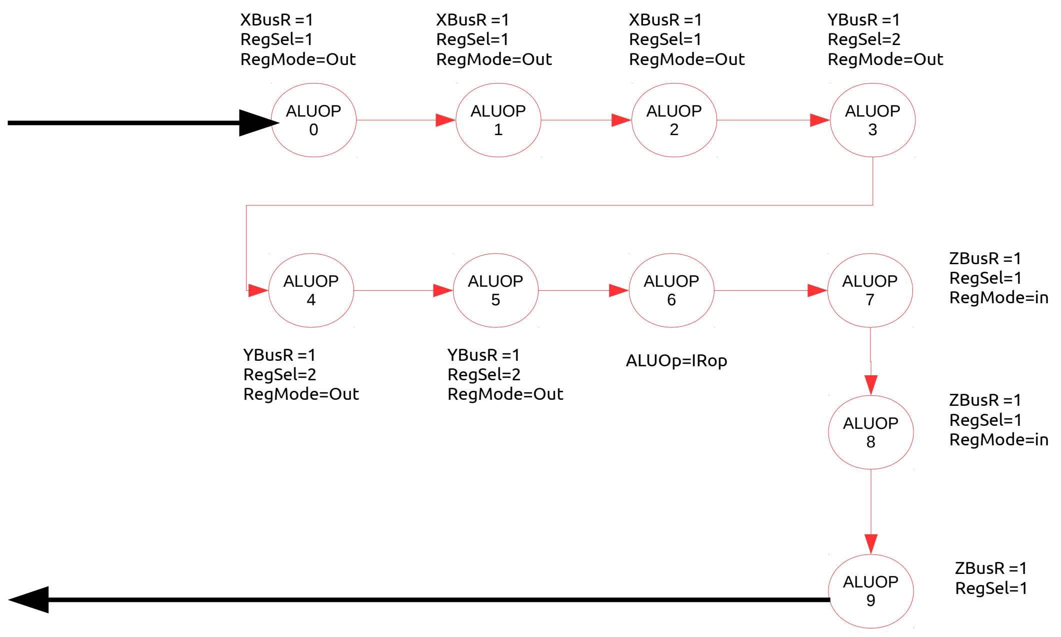

Figure 4: Detailed FSM for ALU Operations

E LD ST FSM

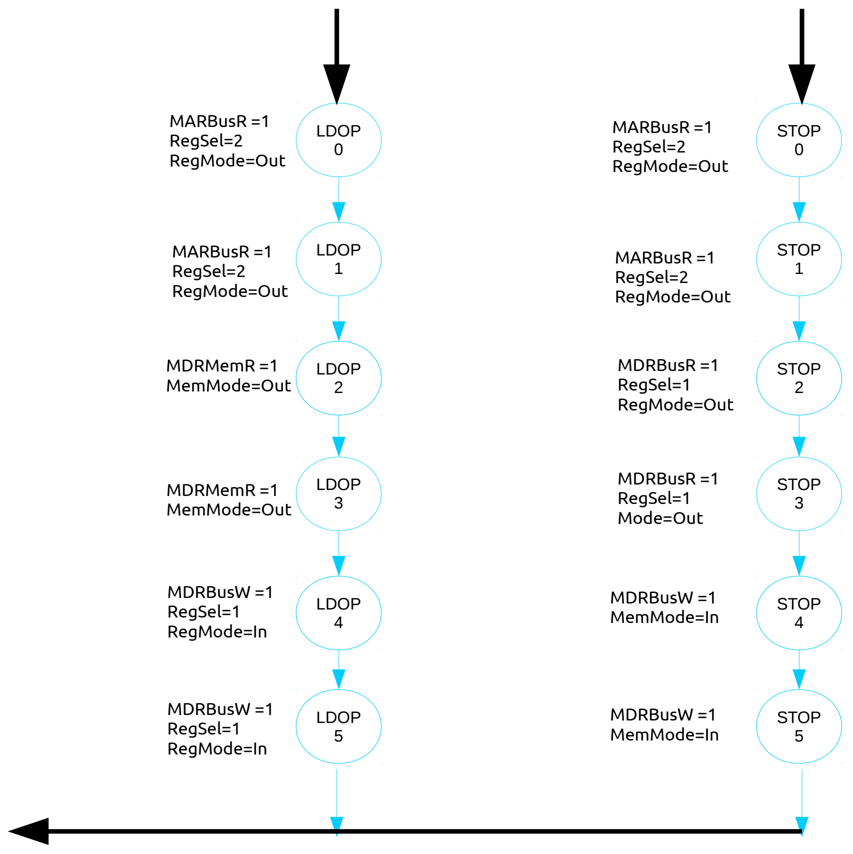

Figure 5: Detailed FSM for Load/Store Operations