Project 2: Implementor's Notes

Alex Burkhart Robert McGillivray

***Abstract*—The aim of this project is to create a working hardware description of a processor in Verilog. An assembler specification was written to provide a starting point to writing the modules for the hardware. As the project progressed, the assembler specification superficially changed as design decisions were made, but the core of the assembler specification remained true to what was originally written. Before writing code, a state machine state diagram, a block diagram for control paths, and a block diagram for data paths were made. These provided the initial top-down design used to guide how the code was developed. Using the assembler specification and all diagrams, the modules for each piece of hardware were written. The IDIOCC was used to generate assembly which was then run through the assembler and tested on the implementation of the microprocessor. The processor uses 16 bit words, has 18 instructions, and will include floating point operations in the future.**

I. Implementor's Notes

A. Assembler Specification

The following will outline the assembler specification for our processor design. Each instruction is loaded into 16 bits total with the op-code in the first 4 bits and one or two registers specified in the next 12 bits (6 bits per register). The the op-codes for instructions and a brief explanation for each is as follows:

1) SYS - Operating system call - Halts processor
2) JZ - Jump if zero to address in specified register
3) SZ - Skip next instruction if value is zero
4) ADD - Add values located in specified registers
5) AND - Bitwise and values located in specified registers
6) ANY - Bitwise any value located in specified register
7) OR - Bitwise or values located in specified registers
8) SHR - Bitshift right the value located in the specified register by 1
9) XOR - Bitwise xor values located in specified registers
10) DUP - Copy the value located in a register to another register
11) LD - Load data from RAM to the register file
12) ST - Store data from the register file in RAM
13) LI - Load an immediate value into a reg
14) ADDF - Add 2 floating point values located in the specified registers
15) F2I - Convert a float to an integer
16) I2F - Convert an integer to a float
17) INVF - Compute the inverse floating point number
18) MULF - Floating point multiply

For the purposes of Project 2, the floating point operations are not implemented. Any floating point instruction will be encoded as SYS, halting the processor.

The complete assembler specification for this project is given below in Figure 1.

Fig. 1. Assembler Specification

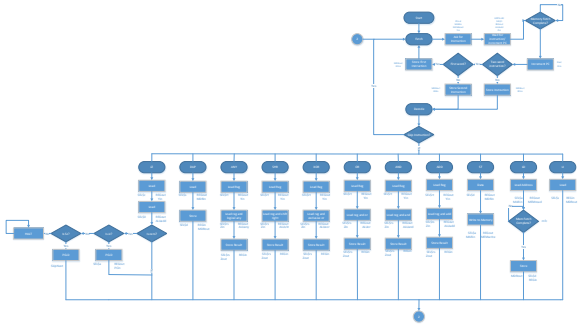
```
jz $c,$a := 0:4 c:6 a:6
sz $c    := 0:4 c:6 0:6
sys      := 0:16
add $d,$s := 1:4 d:6 s:6
and $d,$s := 2:4 d:6 s:6
any $d,$s := 3:4 d:6 s:6
or  $d,$s := 4:4 d:6 s:6
shr $d,$s := 5:4 d:6 s:6
xor $d,$s := 6:4 d:6 s:6
dup $d,$s := 7:4 d:6 s:6
ld  $d,$a := 8:4 d:6 a:6
st  $d,$a := 9:4 d:6 a:6
li  $d,im := 10:4 d:6 0:6 im:16
addf $d,$s := 0:16 ;11:4 d:6 s:6
f2i $d,$s := 0:16 ;12:4 d:6 s:6
i2f $d,$s := 0:16 ;13:4 d:6 s:6
invf $d,$s := 0:16 ;14:4 d:6 s:6
mulf $d,$s := 0:16 ;15:4 d:6 s:6
.const {zero    one     sign    all     sp      fp      ra      rv
        u0      u1      u2      u3      u4      u5      u6      u7
        u8      u9      u10     u11     u12     u13     u14     u15
        u16     u17     u18     u19     u20     u21     u22     u23
        u24     u25     u26     u27     u28     u29     u30     u31
        u32     u33     u34     u35     u36     u37     u38     u39
        u40     u41     u42     u43     u44     u45     u46     u47
        u48     u49     u50     u51     u52     u53     u54     u55 }
.segment .text 16 0x10000 0 .VMEM
.segment .data 16 0x10000 0 .VMEM
.const 0 .lowfirst
```

It can be seen that the first three instructions are encoded similarly (as in the op-codes are the same. The reasoning for this is provided in the next section.

B. State Machine Flow Chart, Data Paths, Control Paths

The state machine for our processor is completely outlined below in Figure 2.

Fig. 2. State machine's state diagram.



The state machine which controls the processor consists of 23 states which are responsible for directing the execution of all instructions. For one word instructions (all but LI), the controller loads the instruction into the instruction register and then executes the instruction. If the instruction is two words long, the state machine recognizes the two-word instruction op-code from the first instruction fetch and loads the second word into the MDR. Since the only two word instruction is LI, there will be no conflicting states between two word instructions. In order to fit 18 instructions into a bit field of size 4, the instructions JZ, SZ, and SYS were consolidated to a single path in the state diagram. This was done because these three instructions perform a similar function (not to mention the assignment specified combining these specific instructions).

C. Implementation Decisions

This section will discuss the various reasons for the architecture, design, and implementation decisions. The first major decision was to not use a separate control line for each control signal. As the number of control signals increased, it became less and less feasible to have named wires going to and from each module. Instead, one control line of width 64 was implemented. The control line's control signals are as follows:

- controls[0] = irin;
- controls[1] = irout;
- controls[2] = pcin;
- controls[3] = pcout;
- controls[4] = mdrin;
- controls[5] = mdrout;
- controls[6] = marin;
- controls[7] = marout;
- controls[8] = yin;
- controls[9] = yout;
- controls[10] = zin;
- controls[11] = zout;
- controls[12] = regin;
- controls[13] = regout;
- controls[14:19] = selreg;
- controls[20] = memread;
- controls[21] = memwrite;
- controls[22] = outputenable;
- controls[23:26] = aluop;
- controls[27] = mfc;
- controls[28:63] = X (not used);

The controls line is of the type [0:63] instead of [63:0]. This was done for no reason other than to try it.

Each register used will output its current value or HiZ on the rising edge of every clock cycle. Also, each register will latch values from the bus on the falling edge of every clock cycle. This was implemented using two always blocks for both ends of a clock cycle. The reason for doing this was to limit the already large number of clock cycles this multi-cycle machine will take to execute a program.

A Von Neumann architecture was selected because it seemed easier to implement. The only "difficulty" would be making sure the RAM is partitioned for instruction memory and data memory. This "difficulty" is overcome by specifying how the RAM gets set during simulation. AIK will output the RAM file in exactly the way it needs to be for the machine.

D. Test Bench Decisions, Methods, and Coverage

The test bench developed does several things. It checks the validity of each instruction (whether or not the instruction did what it was supposed to do) and it has a good coverage of testing. Using the coverage analyzer, the test bench developed has 100% line coverage and 100% finite state machine coverage. This means that the test bench covers each state and every path to and from each state. It also touches every single line of code, meaning there are no untested, nor unused, lines of code in the project. After testing, the outputs from the register file and RAM are compared to what the actual expected values are. For this project and test, the values all matched. To determine whether or not the program passed, for every error a $display will show what the error was.

Note that for running this project, the entirety of our RAM is compressed to a single 65536*16 bit wire, which increases the execution time substantially. If the program looks like it freezes, just wait and it will eventually finish.

E. Issues

There are no known bugs or issues in the code