**A Better Assembler**

Graham Scates | Byron Giles | Joseph Weisbrod

College of Electrical and Computer Engineering

University of Kentucky

Lexington, KY

[Blgile2@g.uky.edu](mailto:Blgile2@g.uky.edu)

***Abstract----* Implement an instruction set based on the Instruction Definition in Our Target(IDIOT) specifications. Design and simulate the hardware needed in Verilog.**

***Keywords---- assembler, AIK, IDIOT***

1. **Introduction**

The IDIOT instruction set specification was designed by Dr. Hank Dietz in order to better understand the process of multi-cycle hardware design. In order to realize this design, the instruction set must first be encoded in assembly. The Assembler in Kentucky (AIK), also designed by Dr. Dietz, is the platform from which IDIOT will be encoded.

Following the encoding, the hardware for the instructions are then built in Verilog. This includes processor and memory.

Finally, a coverage plan will be implemented to assure that each instruction is tested at least once, and that memory is able to be accessed.

1. **Specifications**

The IDIOT instruction set is a simple design, everything is 16 bit. This includes memory, addresses, memory locations, registers, integers, floating-point values, and instructions. Main memory is addressed by 16-bit word addresses; and adding 1 to the address will give you the next one. For example, primitive data type such e.g. *int, short, char and float* are all the same size.

For integer values, there are the following instructions:

* ALU
  + ADD
  + AND
  + ANY
  + OR
  + XOR
  + DUP
* Jump Zero
* Load
* Load 16-biit Immediate
* Skip if Zero
* Shift Right
* Store
* System Call

Followed by the floating point instructions

* ADDF
* Float to integer
* Integer-to-Float
* Approximate Inverse Float
* Multiply Float

At this time, the floating point extensions for this instruction set will not be full realized, instead, reaching these instructions will function as a system halt.

Lastly, there are 63 registers in this implementation. 55 of which are free to use while the first 8 our reserved for special functions:

|  |  |  |  |
| --- | --- | --- | --- |
| Reg # | Reg Name | Writable? | Use |
| $0 | $zero | Const | ZERO; constant 0x0000 |
| $1 | $one | Const | ONE; constant 0x0001 |
| $2 | $sign | Const | the SIGN bit; constant 0x8000 |
| $3 | $all | Const | ALL bits; constant 0xffff or -1 |
| $4 | $sp | RW | the Stack Pointer |
| $5 | $fp | RW | the Frame Pointer |
| $6 | $ra | RW | the Return Address |
| $7 | $rv | RW | the Return Value |

1. **Assembler Specifications**

Next step is to encode the parameters for the registers and instructions in assembly. This also entails using tailoring the output in a manner that is readable in Verilog. To do this, we revisit AIK.

Given the limitations on the size of a word, the opcodes will occupy the first bit, followed by 3 bits of space for the registers, leaving two 6 bit addresses to use for the registers for each instruction, with the exception of li. Begin by defining a register:

.reg $d, $s := .this:4 d:6 s:6

For all the instructions that follow the same format, AIK allows the aliasing of definitions:

.alias .reg 0 add and any or shr xor dup jz ld st

Instructions that have unique patterns, such as squash zero, system call, and li have to be defined implicitly.

sz $c := 7:4 c:6 1:6

sys := 7:4 0:12

li $d, v := 10:4 d:6 0:6 v:16

Even though, floats are not being implemented in this stage in the design, it’s still important to define their bit schema.

Finally, we must define the 63 registers and the output type for the assembler. This can be found in the assembly specifications.

1. **Verilog Hardware Design**

Before designing the modules, it’s important to define the opcodes that will be used:

`define ADD 4'b0000

`define AND 4'b0001

`define ANY 4'b0010

`define OR 4'b0011

`define SHR 4'b0100

`define XOR 4'b0101

`define DUP 4'b0110

`define JZ 4'b0111

`define LD 4'b1000

`define ST 4'b1001

`define LI 4'b1010

`define ADDF 4'b1011

`define F2I 4'b1100

`define I2F 4'b1101

`define INVF 4'b1110

`define MULF 4'b1111

Actuating the instruction set, in our design, requires, the following modules:

* Controller
* Master clock
* IR
* PC
* REGFILE
* MEM
* ALU

**Controller**: This module is where the bulk of the process control signal is handled.

**Master clock:** The centralized clock that used throughout the controller that simulates a cclock clock cycle for every clock tick.

**IR:** Stores the instructions, for example the hex values that are passed by the AIK assembler. These values are being read into memory, using a MEMREAD. These values are stored in a text file called intructions.txt, this is to be attached with the project TAR file. There are three states that the IR can be in. The first is the read state, this gives out the instructions for the PC. The next is the set Address state, this allows the state to be read in as a temporary variable. The last state is a set VAL, which for testing purposes allows you to set the IR value.

**PC:** Program counter, has four different states. One utilized by jump instructions. One, which is used to proceed to next instruction. One, that allows us to increment the program counter by two for a skip instruction. Lastly, a state that allows the user to read out the current PC value.

**REG:** The structure for the register file. It also has two dedicated buses that allow you to read in values concurrently so that can be written to the REG in a single clock cycle.

**MEM:** Initialized externally, MEM has a similar structure to REG for the exception it only has one bus from which to read and write

**ALU:** Where combinatorial logic operations are performed, the ALU includes ADD, AND, OR, XOR, SHR, ANY, JUMP0.

1. **Test Plan**

The main focus of the test plan is to get coverage. Instead of testing every value from every bit position, this entails running through each instruction at least once, with the exception of the Jump 0 instruction.

Starting with all the ALU operations, the assembler generated the following code from our specifications.

@0000

0209

1209

2209

3209

4209

5209

6209

The most significant bit being the opcode for the instruction and the rest being the registers used to execute the instruction. The results were interpreted using GTK wave, these figures can be found attached with the TAR.

Next is to test the branching instructions.

  li $u2, 0x000A

  ld $u3, $u2

  jz $one, $u0

  sz $one

  sz $zero

  add $u0, $u1

  st $u0, $u2

  jz $zero, $u0

this tests load immediate, load, both branches for jump zero and skip zero, and finally the store instruction.

Lastly, we tested the system call, which at this point effectively acts as a halt, after a clock cycle of updating the program counter.

//generated by AIK version 20070512  
@0000  
7000  
//end

This ensures that every instruction and data path is tested at least once.

1. **Conclusion**

This has been an exercise in single cycle design. Because this is the first CPU design that any of us have encountered, it may not be optimized for efficiency. However, it is able to execute all of the instructions that have been laid out in the specifications with the exception of handling floats. It’s hard to tell how much of this we can re-use for the pipeline design, but this provides a good foundation on which to work from.