CPE480 Assignment 2: Multi-Cycle Tangled

Implementor's Notes

Gerard (Jed) Mijares
Ben Luckett
Nick Santini
Department of Electrical and Computer Engineering
University of Kentucky, Lexington, KY USA

ABSTRACT

This is an multi-cycled implementation of a processor for the Tangled instruction set.

1. GENERAL APPROACH

Our AIK specification of Tangled is largely based on the sample solution provided by Dr. Dietz, with a few changes. Namely, we added two additional possibilities for the first 4 bits of the instruction: `Start` and `Decode`. `0xd` was available, so we used that for `Start`. However, at this point, we were out of hex numbers for the first 4 bits. So, we combined the instructions that were originally categorized under both `0x7` and `0x8` to only begin with `0x7`, leaving `0x8` available for `Decode`.

Our processor is structured as a state machine. The first step is to use a case statement on the first 4 bits of the instruction. Sometimes, that is sufficient to distinguish which instruction we should use, but if not, we nest a second case statement on the second 4 bits of the instruction to further determine which instruction to use.

Since some instructions will require a second 16-bit word to process, we elect to simply store both of the next two words in the `Decode` state.

As recommended by Dr. Dietz, we make use of Verilog's `` `define ``s to name ranges of bits we frequently use and the OP code for each instruction.

We decided to run our Verilog locally, so a few commands differ from Dr. Dietz's web interface. A Makefile is included to facilitate running the project. To run different assembly programs, the `readmemh` instructions must be changed to the appropriate filenames.

2. TESTING

Our testing consisted of carefully chosen test code that would cover most of the verilog that we wrote. Our testbench includes a dumpfile with dumpvars that include the registers we used, the two states we used s and s2, the instruction register, next instruction, program counter, and halt. Using the dumpfile along with those dumpvars allowed us to debug most of our program with the program GTKWave. With GTKWave, we could see the instructions and values in the order they were being executed, which when following along with the test assembly output from AIK, allowed us to follow the program and ensure our instructions were handled properly. Once we knew each was working correctly, we created a single "testAssembly" program that ran all the instructions and halted early if error was detected. Multiple scenarios were tested for instructions that might have different outputs, such as shifting both left and right.

Testing the Qat instructions is the most simple and straight forward of all of the instructions to test. Because the Qat instructions are required to be trapped by a sys call. In our case every time a Qat instruction is run, the state is changed to 'OPsys which is our trap. In the event that the state changes to that defined by 'OPsys, the system is halted. In order to test this we began with all of the Qat assembly instructions commented out and then one by one we uncommented one Qat instruction at a time. If program halts after the specified Qat instruction then we know that it passed. Non-Qat instructions were added after the Qat instructions during testing to ensure that the program was actually halting and not just reaching the sys call at the end of our assembly instructions. To run this included test program, the `readmemh` command must be changed to use the proper .text and .data files.

3. ISSUES

Qat instructions originally didn't increment the pc when we needed two words. The first 16-bit word was ready, but the second 16-bit part of the instruction was not being loaded because the pc needed to be incremented by 1. This caused all of the the 16-bit Qat instructions to appear to work fine while the 32-bit Qat instructions failed. One thing to mention is that in the process of getting the Qat instructions working, we interpreted the "error" in the specification to be handled smoothly and to continue with the rest of the instructions after a Qat instruction. This involved using the code:

```
$error("missing coprocessor");
```

When a Qat instruction was encountered it would print the error along with the line number while continuing to run the rest of the instructions. This proved to be incredibly useful in that we could keep the Qat instructions while continuing to debug the rest of the instructions. For the final

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

result, we removed the error text and halted every time a Qat instruction was called as per Dr. Dietz's requests.

Other implementation difficulties were the conditional branch instructions. Initially, we tried to `0` extend the 8-bit branch offset to match with the 16-bit pc, but we noticed that this did not perform as expected. We were also trying a ternary if and testing the equality of the specified register with the value `16'b0`. Using this method, it seemed that the ternary operator always chose the same outcome independent of the value stored in the register. The condition was switched to using a unary `or` and unary `nor`, and the condition choice problem was fixed. It was then determined that the zero extension we had been using needed to be switched to sign extension, and then within the branching choice of both `brt` and `brf`, a second ternary operator was added that made a choice based on if the branch was forwards or backwards. Both conditional branches were then tested with and without their condition being fulfilled, an in the forward and reverse directions.