TACKY Pipelined Processor: Implementor's Notes

James Gallagher
Matthew Miller
Adrian Carideo
James.Gallagher@uky.edu
mtmi233@uky.edu
adrian.carideo@uky.edu
Lexington, Kentucky

ABSTRACT

This assignment was primarily focused on converting our previous multi-cycle processor into a proper pipelined processor. This was done to improve the efficiency of our processor. We decided to base our code off of Dr. Dietz's sample solution for assignment 2. We mixed a lot of the ideas from Dr. Dietz's solution to try to improve some of our code. Despite it being 'mostly untested,' Dr. Dietz's code seemed like the best option for a starting point.

1 GENERAL APPROACH

Our general approach was to create 5 pipeline stages, that consisted of instruction fetch, register read, ALU/Memory, ALU/Arithmetic, and reg. write. These stages are generated in several separate always blocks. These always blocks will pass certain registers between stages, these registers are: instruction register(ir), Operating registers(OpRegA & OpRegB), and Accumulator registers(AccRegA & AccRegB). Each stage owns an instance of all of these registers. for example, to pass the instruction from stage 1 to stage 2, the code would be, ir[2] <= ir[1]. Below is an explanation of the intended functionality of each of the phases.

- Stage 0: This is the instruction fetch stage. It loads the current instruction from memory and then increments the program counter, given that all dependencies are satisfied. If a jump instruction is currently further along in the pipeline, stage 0 will pause until the pc is set by the jump instruction.
- Stage 1: This is the read register stage. It reads data from the registers encoded in the instruction into OpRegA and OpRegB. It also passes along the accumulator values and the pre register.
- Stage 2: This is the first ALU stage. this stage is responsible for performing frecip, and to perform any instruction involving loading or storing to/from data memory. Frecip is done in this stage because the fdiv op takes 2 clock cycles to compute as it is a combination of frecip and fmul. The interaction with data memory is done in this stage to better use this stage as frecip is called very infrequently.
- Stage 3: This is the main ALU stage. This stage is responsible for performing all operations not performed in the previous stage. This stage is very similar to the previous assignment's ALU function, only with some minor changes to improve efficiency. This stage is a much more conventional ALU than the previous stage. This stage was based off of Dr. Dietz's sample solution provided for assignment 2. It was modified to fix some bugs and allow it to fit into the pipelined design of assignment 3.
- Stage 4: This is the register write stage. Essentially, it performs a write wherever is necessary based on the instruction. For example, aritmetic operations will write to the register file, while jump instructions will change the PC. All calculations have already been completed, this stage just routes the result to the correct location.

2 PIPELINE DEPENDENCIES

Per spec, TACKY does not worry about intra-instruction word dependencies; they are allowed to execute in parallel. This resolves many dependency issues on its own. As well, our implementation never executes instruction words out of order, again, solving many dependency issues (simplicity at the cost of some efficiency). The remaining dependencies we had to tackle were:

- Next Stage Ready: We had to make sure that an instruction couldn't move on to the next stage before that stage is "empty," overwriting stored values before they can be used. To this end, we specified a waitN flag for stages N = 1-4, all used in the stage N-1 to prevent this. Every stage except 4 will no-op until the stage ahead of it is done, signaled by the waitN flag being false. In the case of stages 2 and 3, which run with 2 always blocks, a separate wait flag had to be created for each always block.
- Write After Read: We had to make sure that if a register was to be read after a write that existed within the pipeline, that the read would not happen until the write was completed. We implemented this by creating an array of registers to track when the corresponding ISA defined register had a pending write. We also implemented two separate registers outside of this array to track pending writes to the Program Counter and Pre registers. We enabled this by using helper functions/tasks to set the pending write flags as each instruction is fetched in stage 0, prevent the sending of an instruction out of stage 0 if it reads from a register that has a pending write, and to then reset the flags of the registers as they are written to in stage 4. In order to increase pipeline efficiency, at the cost of code/hardware, these helper functions only check and alter the flags of the registers that appear in the instruction word, instead of just halting all actions until a pending write is complete. This way, non-dependent instructions can traverse the pipeline without delay.

3 IMPLEMENTATION DETAILS

This section is responsible for explaining some of the code at a somewhat abstract level, with the intention of allowing the grader to better understand our code

- Stage 0: Given that all dependencies are satisfied, the whole instruction from memory at location PC is loaded into ir and the PC is incremented. If a jump were to be calculated in the previous instruction, the PC for this instruction would be changed in the previous stage 4.
- Stage 1: Stage 1's behavior is trivial– it scans the ir and loads in the data from the registers into OpReg's. These will be passed through the program along with the values from the accumulators and the pre register. Keep in mind there is no logic preventing this stage from loading junk values into an OpReg. The later instructions look at the opcodes to determine whether the value stored in an OpReg is actually necessary for the instruction.
- Stage 2: The code consists of two always blocks that first check for any dependency flags. of there are none, there is a case statement to check the two opcode fields for expected ops (li, lf, st, or fdiv) simultaneously. If the op is not one of the ones performed in this stage, the code simply moves the special registers into 'the ready position' that allows the next stage to receive the instruction/data.
- Stage 3: The code is again composed of two always blocks that first check for dependencies, if none exist, the program moves to a case statement that contains all of the other operations. It should be noted that the first case statement contains all of the macros and non-packed instructions in addition to the packed instructions. After the operations are complete, the program moves the values into the intermediate registers to be used by the next stage.
- Stage 4: Every instruction is finalized here. For arithmetic operations the register file is updated. If data memory needs to be written into, it is done so here. The program counter for the next instruction can be written here if a jump needs to occur. This is the stage that relieves all dependency issues. It is divided up into two always blocks, to handle packed instructions simultaneously.

4 TESTING

As recommended in the assignment, our testing program is based upon our test programs from the previous assignment. The biggest tweaks were to remove all intra-instruction word dependencies, and to reorder some of the test cases in order to create unique pipeline situations which would increase code coverage. Until we can have our code compile, though, this coverage and unique situation optimization is entirely theoretical.

5 ISSUES

Currently our project does not compile. We ran into several problems when attempting to get this project to work, but it should be noted that all of the phases worked independently except for stage 3: main ALU. This stage had trouble properly processing the floating point numbers. Despite all of the stages working when tested separately, we couldn't get all of the stages to work together.

Some of the General problems we ran into when developing this project were trying to understand the ideal dependency structure in order to make the implementation easier. In the end, we went with the dependency plan stated above. Another problem we ran into was that the ALU required 2 always blocks for each stage in the ALU stages(stages 2 & 3), in order to process each opcode field simultaneously. This made the implementation and testing of the ALU stages much more difficult. To fix this, we were planning on creating very specific test cases in cooperation with a more complex testbench in order to isolate the problem in our code.