Assignment 3: A TACKY Pipeline Implementation

Implementor's Notes

Chao-Hsuan Huang, Benedicte Fwelo, Brian Carlson
chu276@uky.edu, benedicte.fwelo@uky.edu, brian.carlson1@uky.edu

ABSTRACT

This assignment is to build a more useable processor. A pipelined version of the TACKY processor. The target is to complete a peek performance of two packed instructions at a time. This project was concerned with creating a pipelined implementation of the multi-cycle processor and memory for the TACKY instruction set. Using an amalgamation of the instruction set encoding specifications from the third assignment, we encoded our instruction set using AIK.

GENERAL APPROACH

To achieve this pipelined processor what we can do is to make a single cycle processor and make it a pipelined version. However, there will be some issue we have to solve. Such as the side effect of overwrite a memory and read from memory. All sort of weirdness should be solved.

The first thing we did was to understand the single cycle design and to figure out how many stages we would need for the single cycle design. It turns out we needed five stages.

The first stage is the "Instruction fetch" stage, this is the stage to fetch instructions to main memory and decide whether the program counter is changing. Basically, we fetch instructions and increase the pc by one.

The second stage is the register read stage. It simply passes the values to the next stage. We also have to check if the opcode here isn't one of "jr" "jz8" "jnz" "jp8", since these are the opcodes for jump to other addresses.

The third stage is the ALU/Memory stage and the fourth stage is the second ALU stage. We sperate the opcode into two different kinds, one is the operations that can be done in one cycle. These things are the integer operations, floating point add, reciprocal, and some memory opcodes like st, lf, and li. Those which can not be done in one cycle will be put in the fourth stage. Such as multiply and second part of divide. As to solve this ALU problem we have two ALU modules one for the first part mentioned above and the other for the second part. The thing we do here depends on the value we get from which ALU. In the second ALU stage, we check if we need to compute the second part or we just need to pass through to the next stage

The last stage is the register write stage, which is just write the result back to register. It is worth noting that every stages needs to check if the processor is halt or not and if the pipeline is clear.

After deciding how many stages we need, we will have to deal with the problems we may encountered. The problem will be pipeline interlocks and value forwarding.

example

```
0 pre 0
1 add 1, add 0
2 add 1, add 0
3 jnz8 $0 5
4 add 1, add 1
5 sys
```

example:

stage 0	stage 1	stage 2	stage 3	stage 4
fetch	read	alu1	alu2	write
pre 0	nop	nop	nop	nop
add 1	pre 0	nop	nop	nop
add 1	add 1	pre 0	nop	nop
jnz8 $0 5	add 1	add 1	pre 0	nop
add 1	jnz8 $0 5	add 1	add 1	pre 0
sys	add 1	jnz8 $0 5	add 1	add 1
sys	nop	nop	jnz8 $0 5	add 1
halt	sys	nop	nop	jnz8 $0 5
halt	halt	sys	nop	add 1
halt	halt	halt	sys	nop