CPE480 Assignment #3

Tyler Burkett
tbbu225@uky.edu
University of Kentucky
Lexington, Kentucky

Jarren Tay
jarrentay@uky.edu
University of Kentucky
Lexington, Kentucky

Evan Jones
sejo238@uky.edu
University of Kentucky
Lexington, Kentucky

ABSTRACT

This project is TACKY, a twin accumulator processor that interprets 16 bit instruction words with up to 2 instructions per instruction word. TACKY is pipelined into 5 stages, so it can process up to two instructions per clock cycle. For the sake of simplicity, this hardware handles dependencies

CCS CONCEPTS

• **Computer systems organization → Pipeline computing**; *Very long instruction word*; Reduced instruction set computing.

KEYWORDS

Pipeline, VLIW, RISC Instruction Set, TACKY, accumulator-based architecture

ACM Reference Format:
Tyler Burkett, Jarren Tay, and Evan Jones. 2019. CPE480 Assignment #3 . In ,. ACM, New York, NY, USA, 2 pages. https://doi.org/N.A

1 GENERAL APPROACH

TACKY uses a Very Long Instruction Word (VLIW) that can have one or two instructions in the word. These VLIWs are only 16 bits, so for words that have two instructions, 8 bits are used to define each instruction. 5 bits indicate the operation to perform. 3 bits indicate one of the registers to use. The position of the instruction (whether it appears first or second in the word) determines the second implicit register that is used.

TACKY will also be able to process both integer and floating point 16-bit instructions. To allow for int and float differentiation, our registers are tagged: 0 for integer and 1 for floating point. This means that each register is actually 17 bits. Because our VLIW is only 16 bits, we can only interpret 8-bit constants. To work up to 16 bits, we have an instruction called âĂIJpreâĂİ that is used to load the first half of a 16-bit constant in. This value is then prepended to the immediates of other instructions that take 8 bits.

To process our instructions, we implemented a five-stage pipeline. The stages are as follows: Instruction Fetch, Register Read, ALU/MEM, ALU 2, Register Writeback. In between each stage, we have registers which take the outputs of one stage and temporarily stores it for the next stage. These stages respect the principle of "owner computes"; how registers are written to are controlled by only one stage; other stages which need to change the value of a register not controlled by it set flags to indicate to the owning stage that the value needs to change.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
N.A, N.A.
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN N.A...$0.00
https://doi.org/N.A

1.1 Stage 0: Instruction Fetch

In our Instruction Fetch stage, we fetch the instruction from memory, decide whether we need to stall our pipeline, and detect and initiate the halting procedure. The instruction is determined by a PC register, which get its next value from either an incrementer or a value given to it by the write back stage (mentioned further below), which it selects based on both a jump flag and if the processor is stalling for a dependency. This value is used to access the instruction memory, obtaining the next instruction that is written to a buffer for the next stage. This stage also utilizes logic attached to the registers of subsequent stages and the newly decoded instruction to decide the number of NOPs to push through the next stage in order to resolve issues of dependencies and flow control. These are detailed more in the "Dependencies and Jump Handling" section.

1.2 Stage 1: Register Read

In our Register Read stage, we check the instructions being used in the operations we need to perform and fetch the values of the appropriate registers. In addition, we also perform the pre operation. The register values fetched are determined by the registers indicated in the instruction. The accumulator registers are always fetched, as this is a common occurence, but whether it is used is left to the subsequent stages. The pre instruction is completed in this stage, as the value to store is an immediate value which is available as soon as the instruction reaches this stage. This helps eliminate the pre register as a potential source of dependencies for instrucitons which use an immediate value. Immediate values are also formed in this stage and passed through subsequent stages as needed. The writeback stage is the only stage which direclty uses these values, but it is still passed to the stages between to keep the value and respective instruction in the same stage and avoid potential timing issues.

1.3 Stage 2: ALU / Data Memory

In our ALU/MEM stage, we begin performing the memory and arithmetic instructions weâĂŹve been provided using the register values we received from the Register Read stage. Most instructions will finish in this stage, with the exceptions of load float, load int, and floating-point divide. Load float and load int will read the 16-bit value from memory. Floating point divide will read from the reciprocal lookup table.

1.4 Stage 3: ALU2

In our ALU 2 stage, we finish up the operations that we didnâĂŹt finish in the previous stage. For load float, we prepend a 1 to the value we read to signify that it is a float. For load int, we prepend a 0 instead. For floating point divide, we multiply the value from the lookup table by the accumulator.

1.5 Stage 1: Write Back

In our Writeback stage, the results from the previous stages are collected and selectively stored based on the current instruction. These values include ALU results, data memory results from li and lf instrucitons, and an immediate value determined in the Register Read stage. This stage also handles the logic for determining if a conditional jump is taken or not. Register stores are written directly to the register file, where as values to be written to pc are stored in a buffer register. A flag is set in the event of a jump to allow the Instruction Fetch stage to determine whether the pc register needs to be incremented or set to the value this stage wrote to the buffer register.

1.6 Dependencies and Jump Handling

Because our processor is pipelined, the effects of a previous instruction may not have taken effect when we begin processing the next one. For example, if two sequential instructions both read and modify the same register, the second instruction would read the old register value before the first instruction would update it. To solve this issue, we need to stall stages of our pipeline.

To check whether we need to stall, we first look at the instruction we are about to process, and we determine which registers are being read for this type of instruction. Then, we look at the instructions being processed at the next three stages in order. If those instructions are modifying a register that we need to read, we'll need to stall. To stall, we output nops for the next X clock cycles and freeze the pc. X is dependent on how which stage we caught the dependency. 3 for Reg Read stage, 2 for ALU/MEM stage, and 1 for Writeback stage. Because we freeze the pc for this time, we will execute the correct instruction after getting through the nops. In addition to register dependencies, we also stall if we detect a jump instruction. For jumps, we stall until the instruction gets to the Writeback stage, so that if we need to jump, it will modify the pc.

2 TESTING

No testing was done because there was nothing to test.

3 ISSUES

There were no issues because there was nothing to have an issue with.