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, 3 pages. https://doi.org/N.A

Bit-Pattern					Op Code	Category	
0	0	0	0	0	NOP		
0	0	0	0	1	pre	Pre	1 Op/Word
0	0	0	1	0	sys	System Call	
0	0	0	1	1	jp8	Immediate Jumps	
0	0	1	0	0	jz8		
0	0	1	0	1	jnz8		
0	0	1	1	0	cf8	Immediate Reg. Load	
0	0	1	1	1	ci8		
0	1	0	0	0	NOP		
0	1	0	0	1	jr	Jump	2 Op/Word
0	1	0	1	0	st	Store	
0	1	0	1	1	NOP		
0	1	1	0	0	a2r	Reg. Swap	2 Op/Word
0	1	1	0	1	r2a		
0	1	1	1	0	li	Load	
0	1	1	1	1	lf		
1	0	0	0	0	NOP		
1	0	0	0	1			
1	0	0	1	0			
1	0	0	1	1			
1	0	1	0	0	sh	Arithmetic	2 Op/Word
1	0	1	0	1	slt		
1	0	1	1	0	NOP		
1	0	1	1	1	cvt	Arithmetic	2 Op/Word
1	1	0	0	0	add		
1	1	0	0	1	sub		
1	1	0	1	0	div		
1	1	0	1	1	mul		
1	1	1	0	0	not	Bitwise	
1	1	1	0	1	xor		
1	1	1	1	0	and		
1	1	1	1	1	or		

Figure 1: TACKY Encoding

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.

1.1 Stage 0: Instruction Fetch

In our Instruction Fetch stage, an instruction is fetched from memory and neccesary stalls are detected and initiated if needed. 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 to 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

N.A, N.A.

ACM ISBN N.A…$0.00
https://doi.org/N.A

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. The instruction is passed on to the next stage, as well.

1.3 Stage 2: ALU / Data Memory

In our ALU/MEM stage, we have two ALUs. There are two goals for these ALUs: to perform arithmetic operations on the register values and to load from and store to memory. Each receive the instruction word, and the values of the registers it would be concerned about. If the instruction word had an immediate, it was fed to the first ALU and the second ALU does nothing. 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. All inputs to this stage are transferred to the next stage. An intermediate "out value" is also transferred to the next stage.

1.4 Stage 3: ALU2

In our ALU 2 stage, we finish up the operations that we didnâĂŹt finish in the previous stage. Again, there are two ALUs, and we receive the same things as input as the first stage of ALUs. In addition, we receive the intermediate value. This intermediate value is going to be the output for most instructions. 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. Again, we pass the instruction word, the register values, and the alu generated values to the next stage.

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. The registers and accumulators that the fetched instruction reads from are compared to the registers and accumulators that are written to in the following three stages. This is achieved using special decoder modules that look at the instruction and output the register numbers that are being read/written or don't care values if not applicable. If a common register is being read from and written to across stages, it is considererd a dependency. Four case statements in the Instruction Fetch stage check for any matches between registers written to in the other stages and the register being read from in the Instruction Fetch stage. Once matches are found, their stage locations are compared. Stages that are farther away from the Register Writeback stage require more NOPs to be padded. Therefore, the number of NOPs to pad is the number related to whichever dependency is farthest from the Writeback Register. This remains true for instances in which multiple dependencies happen across stages. To stall, we output nops for the next X clock cycles and freeze the pc. X is dependent on which stage we caught the dependency. 3 for Reg Read stage, 2 for ALU/MEM stage, and 1 for Writeback stage. If multiple dependencies occur across stages, the largest number of NOPs is used. 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.

1.7 Halting

To handle the sys halting properly, we detected the instruction at the Instruction fetch stage. After this, NOPs were pushed into the pipeline following the sys instruction to avoid subsequent instructions from executing, while allowing ones still in the pipeline to complete. Once sys reaches the Write Back stage, the halt flag on the proccessor is set, which causes the clk signal to stop.

2 TESTING

Our test cases are included in a separate file called "testcases.txt". Test cases were written to cover as many memory, jump, and integer arithmetic instructions as possible. Operations involving floating point were left out, since we are not responsible for making sure that the floating point modules operate correctly. The instructions not tested outside of floating point is jr, li, and st. The is set up to first test immediate register loads and jumps, followed by every other instruction fitting the criteria mentioned earlier. This is done by having sys operations after conditional jumps. These conditional jumps are used to test registers which hold the results of some operation, if the jump is taking, the sys opertion is skipped, else the sys operation is run and causes the processor to halt.

To test, 4 different vmem files need to be used to test the design properly. Register values belong in VMEM0, instruction memory belongs in VMEM1, data memory belongs in VMEM2, and the floating point lookup table belongs in VMEM3. There are preassembled values included in the "TACKY.test" file which can be used for testing.

3 ISSUES

During testing, our processor could properly read and parse instructions until we reached a jump instruction followed by a sys instruction. We supplied the first stage with a jump flag and the pc to jump to, but because it was followed by a sys, our processor parsed the sys instead of jumping. This was likely due to the fact that the actual padding of NOPs was performed at the Register Read stage instead of at the Instruction fetch stage, which caused the next instruction to be visible to the components which decided halting. As a result, testing for whether the sys operation had instructions executing after it became difficult, but from what we've seen this appears to not be the case. Other instructions appear to operate normally.