Elec/Comp 526
Spring 2019
Project 1A

Overview

This project deals with implementing and evaluating a 5-stage pipelined processor for a small subset of the DLX instruction set. The processor follows the description in the lectures.

这个project主要就是用一个5阶段pipelined处理器执行一些DLX的指令集。

This document describes part A of Assignment 1 dealing with handling data and control dependencies in software (PART 1), and adding processor support for Branch Speculation (PART 2). A separate document and support code for part B to handle data dependencies will be posted later. The supporting code for PART 1 and 2 are in the tar files project1A.PART1.tar and project1A.PART2.tar on Canvas.

所支持的指令集合： 1. ALU， 加法， 2. 载入 3.

Processor Instruction Set

The processor supports very few instructions (i) **ALU instructions** ADD and ADDI that add two register operands (RR) or a register operand and a (sign-extended) 16-bit constant (RI); (ii) **LOAD instruction** that reads a memory word into a specified register; (iii) **STORE instruction** that writes a specified register to memory; (iv) Conditional **BRANCH** instruction BNEZ that takes the branch if the value in the specified register is non-zero. In addition there is a **NOP** instruction that does not affect any state and a **HALT** instruction that stops the processor (and the simulation) when it reaches the ID stage of the pipeline. The op codes and mnemonics are specified in the header file **global.h**.

What does BNEZ do?

The conditional branch instruction (**BNEZ**) computes the target address as the sum of the sign-extended offset specified in lower-16 bits of the instruction and the address of the instruction immediately following the branch. **BNEZ** specifies a single register operand *rs* in addition to the offset; the branch is taken if the specified register holds a non-zero value; else the execution continues in-line.

The memory instructions **LOAD** and **STORE** respectively read and write a word of memory. The only memory addressing mode supported is *base+offset* mode: the specified source register (*rs*) is the base address; the specified 16-bit immediate value is sign-extended to 32 bits and added to the base register contents to get the memory address. **LOAD** specifies the destination register *rt* into which the memory word must be loaded, while for the **STORE** instruction register *rt* holds the value to be written to memory. There are 16 32-bit-wide registers in the register file **REG_FILE**[0 ..15].

The instruction formats are described below. Given a 32-bit instruction $[\ I\ ]_{31:0}$

RR instructions (ADD)	**LOAD**	**STORE**
$[\ I\ ]_{31:26}$ -- Op Code	$[\ I\ ]_{31:26}$ -- Op Code	$[\ I\ ]_{31:26}$ -- Op Code
$[\ I\ ]_{25:21}$ -- rs	$[\ I\ ]_{25:21}$ -- rs	$[\ I\ ]_{25:21}$ -- rs
$[\ I\ ]_{20:16}$ -- rt	$[\ I\ ]_{20:16}$ -- rt	$[\ I\ ]_{20:16}$ -- rt
$[\ I\ ]_{15:11}$ -- rd	$[\ I\ ]_{15:0}$ -- Offset	$[\ I\ ]_{15:0}$ -- Offset
$[\ I\ ]_{10:0}$ -- Unused		

RI (ADDI)	BNEZ	NOP,HALT
[I]$_{31:26}$ -- Op Code	[I]$_{31:26}$ -- Op Code	[I]$_{31:26}$ -- Op Code
[I]$_{25:21}$ -- rs	[I]$_{25:21}$ -- rs	[I]$_{25:21}$ -- Unused
[I]$_{20:16}$ -- rt	[I]$_{20:16}$ -- Unused	[I]$_{20:16}$ -- Unused
[I]$_{15:0}$ -- Immediate	[I]$_{15:0}$ -- Offset	[I]$_{15:11}$ -- Unused

The **ADD** instruction adds the contents of registers *rs* and *rt* and places the result in register *rd*. The **ADDI** instruction adds the contents of register *rs* with the sign-extended immediate value specified in the lower-16 bits of the instruction, and places the result in register *rt*.

Pipeline Model

The processor pipeline has five stages: FETCH, ISSUE, EXEC, MEM and WRITE. The functions of each stage are described below. The stages are separated by pipeline registers **PR**[0..4] (the structure of a pipeline register is defined in *global.h*). PR[0] is between the FETCH and ISSUE stages, PR[1] between ISSUE and EXEC stages, and so on. PR[4] is a pipeline register after the WRITE stage and is updated in the code but has no real use (it is never read).

In addition the actual pipeline registers PR[ ] described above, the simulation model defines a set of **Shadow Pipeline** registers **SHADOW_PR**[0.. 4]. This is because we are trying to mimic a synchronous update of all pipeline registers performed by the hardware using our software single-threaded simulator. If you used a language like Verilog you would not need this roundabout approach.

A stage **i** takes its inputs from the pipeline register PR[i-1] to its "left" and stores its computed values in the shadow pipeline register SHADOW_PR[i] to its "right". After all 5 stages have done their computation, a separate synchronization process (*sync.c*) copies the buffered values in SHADOW_PR[i] to PR[i] completing the clock cycle to correctly emulate the hardware.

FETCH: Read an instruction from the Instruction Memory location pointed to by the program counter **PC** and update **PC** to the address of the next instruction to be executed. This is either the current PC value plus 4 or the target address of a branch. The **stallIF** signal will not be used in this assignment (1A).

ISSUE: Decode the instruction and read the source register(s) REGISTER_FILE[rs] and REGISTER_FILE[rt]. The decoded fields and register values are stored in different fields of the shadow pipeline register SHADOW_PR[1]. The **writeback** flag is used to indicate that this instruction will write to a destination register **destRegister** in the WRITE stage. The **control** field is a set of bits indicating the instruction (bit 0, 1, 2, 3, or 4 are set if the instruction is ADD, LOAD, STORE, BNE, or ADDI respectively).

If the instruction is a **HALT** instruction an internal halt signal is asserted and all stages will stop (and the simulation will end).

EXECUTE: The execute stage reads the pipeline register PR[1] to find the operation it needs to perform and the operands it needs. It performs the requested operation and places the results in the Shadow Pipeline Register SHADOW_PR[2].

MEM: This stage performs Load and Store instructions. It reads the address from the **result** field of the pipeline register PR[2] and the data for a store from the **operand2** field (which will be the value of register *rt*) of PR[2], and performs the read (write) from (to) memory. For a LOAD, the value read from memory is placed in the shadow pipeline register SHADOW_PR[3].

The other task performed in the MEM stage is to handle branches. The outcome of the branch (Taken or Not Taken) would have been set in the EXEC stage in pipeline register PR[2]. In the MEM stage, the target address of the branch is computed. If the branch is to be taken the global signal **updatePC** is set to TRUE and the computed target address is assigned to the global signal **nextPC.** These two signals are used by the Fetch stage to determine the next value of the program counter.

WRITE: This stage checks the pipeline register PR[3] to see if the instruction should write to a register (**writeback** flag) and if so updates the destination register with the result.

Workload

Three assembly language programs are provided in the files **program1.c, program2.c** and **program3.c**. Currently they will not perform as expected on the pipelined processor because of data and control hazards. The tasks for PART 1 are described below.

PART 1

a. The code is in the tar file **project1A.PART1.tar** on **Canvas**. Download to a 64-bit Linux system. If the code does not work on your laptop you should use the **CLEAR** cluster.

b. See file **README** in the directory.

c. Compile the program as stated and execute it and observe the trace and the final output. You will see the results are not as expected because of hazards.

d. Change the programs (one at-a-time) to remove data and control hazards. **Do not optimize or change the order of instructions. The only allowed operation is adding NOPs.** You must add the minimum number of NOPs (all zero 32-bit instruction) needed to ensure correctness. Note that the Branch offset and the memory index of instructions will change as you add new instructions, and will need to be corrected.

e. Run each program for **NUM_ITERATIONS** equal to 10, 20, 40, 80 and 100 and record the number of cycles taken in each case. Make sure that the results are as expected.

f. Rewrite the code of each program to minimize the run time by reordering the instructions. You may change the offsets of the existing memory instructions but no new instructions must be used. Don't forget to renumber the instructions and use the correct branch offset.

g. Repeat part **e** for the optimized programs. Plot the results of the optimized and non-optimized versions on the same plot for different values of NUM_ITERATIONS. (One plot for each program).

h. For this part, turn in the plots and your non-optimized but corrected program and your optimized programs on Canvas in a tarred file called PART1.

PART 2

In this part you need to alter the processor to support Branch Speculation. Specifically, in the ISSUE stage we will speculate that a Branch is **Always Taken**. The actual outcome of the branch will be determined later in the MEM stage. The pipeline register structure in **global.h** has been altered by adding new fields to support this speculation. The FETCH stage must take the relevant outputs of the Issue and Mem SHADOW_PR[] registers in determining the next PC value.

The new fields in the Pipeline registers are: **isBranchInstrIssue** and **branchTargetAddressIssue** which are set in the ISSUE Stage, and **isBranchInstrMem**, **conditionMem,** and **inlinePCMem** that are set in the MEM stage.

In the ISSUE stage, after decoding check if the instruction is BNE. If so set the **isBranchInstrIssue** and **branchTargetAddressIssue** fields of SHADOW PR[1]. The first is a flag that needs to set to TRUE if the instruction is BNE and FALSE otherwise. The second is the target address of the branch from where instruction will speculatively continue. These need to be done by completing the stub **checkAndHandleSpeculativeBranch()** in **issue.c.**

In the Mem stage you must also check if the instruction is BNE and set the **isBranchInstrMem** flag of SHADOW_PR[3] appropriately. Also set **conditionMem** flag of SHADOW_PR[3] to TRUE if the branch was correctly speculated (*i.e.* branch should be taken) and FALSE otherwise, and set the **inlinePCMem** field of SHADOW PR[3] to the address of the inline instruction that must be executed if the speculation was incorrect. We will assume this to be the instruction immediately following the BNE instruction and re-fetch that instruction into the pipeline. (This takes care of corner cases as when a non-taken branch immediately follows a non-taken branch). Make all the changes in the stub **handleSpeculativeBranch()** in **mem.c.**

The final change is to be made to the FETCH stage by completing the stub **handleBranchSpeculation()** in **fetch.c**. Check if a Branch instruction is in the MEM stage

(using SHADOW_PR[3]) and if so whether or not the branch was correctly speculated. Squash either 1 or 3 instructions in the appropriate Shadow Pipeline registers based on the speculation result. (You can create a NOP by copying the predefined "dummy" struct into the pipeline or shadow pipeline register you want to squash). If the speculation is correct set the flag **updatePC** to FALSE; else set this flag to TRUE and variable **nextPC** to the inline address computed by MEM.

a. The code is in the tar file **project1A.PART2.tar** on **Canvas**.

b. Make the changes to files fetch.c, issue.c and mem.c to support branch speculation as discussed.

c. Use the optimized codes for each of the programs you developed in PART 1 as the workloads.

d. Run each program for **NUM_ITERATIONS** equal to 10, 20, 40, 80 and 100 and record the number of cycles taken in each case. Make sure that the results are as expected.

e. Plot the results of the optimized programs with and without branch speculation on the same plot for different values of NUM_ITERATIONS. (One plot for each program).

f. For this part, turn in the plots and the three updated source code files on Canvas in a tarred file called PART2.