Elec/Comp 526
Spring 2019
Project 2

Overview

This project deals with implementing and evaluating an out-of-order (OOO) dynamically scheduled processor based on Tomasulo's algorithm.

Processor Instruction Set

The processor supports 10 instructions: 4 **ALU**, 2 **Branch**, 2 **MEM**, **NOP** and **HALT**. The ALU instructions have mnemonics **ADDFP**, **SUBFP**, **MULFP** and **INTADD**: their op codes and execution times are specified in the header file. An unconditional branch instruction (**BRANCH**) and a conditional branch (**BNEZ**) are also supported. The target address for a branch is the sum of the 32-bit sign-extended offset specified in the instruction and the address of the instruction immediately following the branch. **BNEZ** takes a single register operand in addition to the offset; the branch is taken if the specified register does not have value zero; else the execution continues in-line. The memory instructions **LOADFP** and **STOREFP** read or write a word of memory. The only memory addressing mode supported is register indirect: the specified source register (Src Reg 1) holds the memory address of the desired word. In addition, **LOADFP** specifies the destination register into which the memory word must be loaded, while **STOREFP** has a second source register (SrcReg 2) that holds the value to be written to memory. The **HALT** instruction stalls the fetch and issue of new instructions while allowing the rest of the pipeline to drain and complete previously issued instructions. There are 32 general-purpose registers that can be used as either floating point or integer registers.

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

ALU instructions

$[I]_{31:26}$ -- Op Code
$[I]_{25:21}$ -- Src Reg 1
$[I]_{20:16}$ -- Src Reg 2
$[I]_{15:11}$ -- Dest Reg
$[I]_{10:0}$ -- Unused

LOAD

$[I]_{31:26}$ -- Op Code
$[I]_{25:21}$ -- Src Reg1
$[I]_{20:16}$ -- Unused
$[I]_{15:11}$ -- Dest Reg
$[I]_{10:0}$ -- Unused

STORE

$[I]_{31:26}$ -- Op Code
$[I]_{25:21}$ -- Src Reg1
$[I]_{20:16}$ -- Src Reg 2
$[I]_{15:11}$ -- Unused
$[I]_{10:0}$ -- Unused

BRANCH

$[I]_{31:26}$ -- Op Code
$[I]_{25:21}$ -- Unused
$[I]_{20:16}$ -- Unused
$[I]_{15:0}$ -- Offset

BNEZ

$[I]_{31:26}$ -- Op Code
$[I]_{25:21}$ -- Src Reg1
$[I]_{20:16}$ -- Unused
$[I]_{15:0}$ -- Offset

NOP,HALT

$[I]_{31:26}$ -- Op Code
$[I]_{25:21}$ -- Unused
$[I]_{20:16}$ -- Unused
$[I]_{15:11}$ -- Unused

Pipeline Model

The processor pipeline has five stages: FETCH, ISSUE, DISPATCH, EXECUTE and WRITE. The functions of each stage are described below.

FETCH: Read an instruction from Instruction Memory and update PC.

If the **stallIF** signal is not asserted (*i.e. FALSE*) the PC, INSTRUCTION and PC4 registers are updated; else all registers remain unchanged for this cycle. If the **branchFlag** signal is not asserted the word at the current PC is read from instruction memory and PC is incremented by 4; otherwise PC is updated to **nextPC** and INSTRUCTION is set to **NOP**. The updated PC is copied to PC4 for use by the issue stage. The signals **stallIF**, **branchFlag** and **nextPC** are set by the ISSUE stage based on the instruction it is currently decoding.

ISSUE: Decode the instruction and stall the FETCH stage if necessary. If the instruction should be issued get a free RS entry (if possible) and fill in its fields. Update the state of the destination register.

The instructions **NOP**, **HALT**, **BRANCH** and **BNEZ** do *not* issue into the Reservation Station but complete their execution in the ISSUE stage. The **NOP** instruction simply delays till the next clock cycle. The **BRANCH** and **BNEZ** instructions compute the branch address in the ISSUE stage itself and provide it to the FETCH stage. The **BNEZ** instruction also checks the outcome of the branch in the ISSUE stage. If a branch is to be taken the **branchFlag** signal is asserted. **BRANCH** is an unconditional branch that is always taken. If the outcome of a conditional branch statement cannot be determined at this time, the **stallIF** signal is asserted. The **HALT** instruction always asserts **stallIF** (it should also set **isHALT** to TRUE for updating statistics).

The remaining instructions require a Reservation Station to issue. If a free reservation station is not available, the **stallIF** signal must be asserted. If a reservation station is available, its fields (see the declaration of a reservation station entry in the header file) need to be filled in according to Tomasulo's algorithm. The **operand1** and **operand2** fields are the values of the source operands; the **op1RDY** and **op2RDY** fields indicate whether the corresponding operands are valid. The **tag1** and **tag2** fields are the tags of the instruction producing a future value *i.e.* an operand whose producer is in flight at this time. Note that **LOAD** has only one source operand and that **STORE** has no destination register.

The **free** field of a reservation station entry should be set to FALSE and the **fu** field should be set with the index of the functional unit chosen to execute that instruction. The destination register is tagged with the instruction by writing to the array **REG_TAG.** The tag of the instruction is the index in the Reservation Station allotted to it. Helper functions **getFU()** and **getFreeRS()** are used to select a functional unit and a free Reservation Station.

DISPATCH: Look for an eligible instruction to dispatch from the Reservation Station to the Function Unit for execution.

An eligible instruction is one that has been issued but not yet been dispatched, whose source operands are available, and whose designated functional unit is free. If an eligible instruction is found, the **busy** flag of the RS entry is set to TRUE and the operand values and instruction tag are copied to the pipeline register between the DISPATCH and EXECUTE stages. This is implemented by the structure **myWork**; this structure communicates operand and tag values to designated functional unit. The signal **workAvail** for that functional unit is asserted to inform the latter that there is a fresh operation to execute.

EXECUTE: The execute stage is made up of multiple functional units. There are NUM_FU types of functional units and each functional unit has NUM_COPIES copies. All functional units have unique ids between 0 and (NUM_FU * NUM_COPIES -1): the ids of FUs of the same type differ by NUM_FU.

All functional units that receive new operations to execute (**workAvail** for that unit is TRUE) wake up. The unit performs the operation on the operands (available from the fields of **myWork**) and delays for the number of cycles required for that operation. When it completes its operation (several cycles later depending on the delay) it needs to write the result to the EXECUTE/WRITE pipeline register. Each FU has an individual structure (called **resultData**) in this pipeline register. A handshake is required to make sure that any earlier result previously produced by this functional unit have been consumed by the WRITE stage. If the previous result produced by the FU has not yet been processed by the WRITE stage (signal **resultReady** will be TRUE) the FU waits till that flag is set to FALSE before updating **resultData**. When **resultReady** is FALSE the result of the operation is copied to **resultData** along with the tag of the producing instruction and the signal **resultReady** is again set to TRUE.

WRITE: Select a completed instruction from some EXECUTE/WRITE pipeline register and broadcast its result and tag on the CDB. Perform the required actions on the RS and Register File in response to the broadcast.

One functional unit is chosen from among those asserting the **resultReady** signal. If none of the functional units have a result ready, do nothing in this clock cycle. Otherwise, get the result and tag from the **resultData** output by the FU and broadcast the result and tag values to the Reservation Stations and Register File. The functions **CDBUpdateRS** and **CDBUpdateREGFILE** perform the actions of the Reservation Station and Register File respectively in response to the broadcast. The first function copies the broadcast data value into all waiting reservation station source register entries with a matching tag. The RS entry for the completing instruction is set to *free* and its status is set to *not busy*. The second function copies the broadcast value into the register with a matching tag (if any) and sets the register tag to indicate the value is stable. The function unit is made available for the next operation by setting **isFree** to TRUE and **resultReady** is set to FALSE.

Workload

Two assembly language programs (PROGRAM 1 and PROGRAM 2) are provided in the file **utils.c**. Work through the programs and predict the output you expect in each case. One of the programs should be commented out at any time.

The file **utils.c** contains code to initialize the microarchitecture structures on a **reset**, to initialize the registers and memory for the two programs, and a number of helper functions. The file **display.c** implements an additional display stage that is invoked once per cycle. It contains instrumentation code to collect statistics and is necessary for correct termination. You can add any code to the function **do_display** to show data structures that you are interested in during debugging.

Preliminaries

a. The code is in the tar file **project2.tar** on **Canvas**

b. See file **README** for a description of the other files.

c. The output file **demo1** outputs a detailed trace of **10** iterations of PROGRAM 1 with the default parameters in **global.h**. Study the trace to see how the system is expected to behave and reconcile it with the provided source code.

d. The output file **demo2** repeats step c *except* that it uses *2 reservation stations*. That is, the constant **NUM_RESERVATION_STATIONS** is changed from 1 to 2 in **global.h**. Compare the outputs in the two cases. In particular, observe the sequence of register writes done in the two execution traces. You can do this easily by examining the lines which do a register write by searching for the phrase "**Writing to Register**".

e. For the actual experiments change the number of iterations to **60** and the maximum simulation time to **5000.0** by setting **NUM_ITERATIONS** and **MAX_SIMULATION_TIME** in **global.h**.

Assignment Details

a. Complete the code in **issue.c** and **write.c** as described in the specifications. Comments in the code stubs provide additional guidance.

b. Compile, execute and debug your code until you are satisfied.

Experiments

PROGRAM 1

- In **global.h** set the value of **NUM_ITERATIONS** to 60 and **MAX_SIMULATION_TIME** to 5000.0. Also make sure **NUM_COPIES** is 1 and **NUM_RESERVATION_STATIONS** is set to 1**.**

- **Step 1A**: **LOADFP_CYCLES = 2.** Vary **NUM_RESERVATION_STATIONS** from **1 to 128** (in **powers of 2**). Record the **number of instructions retired**, the **times of execution completion** and **retirement completion**, and the **number of RSFull stall cycles**.

- **Step 1B**: Change **LOADFP_CYCLES to 16** and repeat step 1A.

- On the same plot (**Plot S1**) graph the **IPC** (the average number of instructions retired per cycle) against the **number of Reservation Stations** for both steps 1A and 1B. Use base-2 logarithm of the number of RS for the x axis of your plot. Explain the shape of the plot for both steps. **Why are they so different**?

- **Step 2A**: For this experiment, fix the value of **LOADFP_CYCLES = 2** and **NUM_RESERVATION_STATIONS = 4**. Vary **NUM_COPIES** from **1 to 5** (in steps of 1) and record the same data as in Step 1A. Next repeat this step for **NUM_RESERVATION_STATIONS = 64**.

- On the same plot (**Plot S2A**) graph the **IPC** against the **number of FU copies**, for **both** values of **NUM_RESERVATION_STATIONS**. Explain the plot.

- **Step 2B**: Change **LOADFP_CYCLES** to **16** and repeat step 2A. Create **Plot S2B** using the data of this step. Explain the plot.

PROGRAM 2

- Repeat all the experiments and plots done above for PROGRAM 2.

Explain briefly but precisely the differences in the behavior of programs 1 and 2.

SUBMISSION: Submit a total of six plots, explanations and discussions, and the source code for your completed **issue.c** and **write.c** on Canvas by the due date.