Elec/Comp 526
Spring 2016
Project 2

Overview

Project 2 deals with implementing and evaluating an out-of-order 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 fie. An unconditional branch instruction (**BRANCH**) and a conditional branch (**BNEZ**) are 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 producer instruction for an operand in case the its values are not ready at the time of issue. Note that LOAD has only one source operand and that STORE has no destination register.

The free field of 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) are woken 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 pipeline register between the EXECUTE and the WRITE stages. 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 it 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.

Assignment

1. **Preliminaries**
    a. The programs are on **CLEAR** in the directory **~pjv/elec526/project2**
    b. Make a subdirectory **elec526/project2** in your **CLEAR** home directory and do all work for this part in that directory.
    c. Copy the files into your working directory.
    d. See file **README** for a description of the other files.

2. The executable file **demo** outputs a detailed trace of 10 iterations of PROGRAM 1. Study the trace to see how the system is expected to behave and reconcile with the provided source code.

3. Complete the coding of **issue.c** and **write.c** as described above. Comments in the code stubs provide additional guidance.

4. Compile, execute and debug your code until you are satisfied. You can set the TRACE and DEBUG flags set in **global.h** to help you debug.

EXPERIMENTS

PROGRAM 1

- Use default values in the files. In **utils.c** make PROGRAM1 the program to be executed (comment out PROGRAM 2 in the loadProgram(**)** function) and set the value of NUM_ITERATIONS in **global.h** to 60. This sets the initial value of register R8 that controls the number of iterations of the loop. In **global.h** set LOADFP_CYCLES = 2, NUM_RESERVATION_STATIONS = 1, and NUM_COPIES = 1**.**

- **Step S1a**: Keep LOADFP_CYCLES = 2**.** Vary from 1 to 128 (in powers of 2) NUM_RESERVATION_STATIONS. Record the number of instructions retired, the times at which execution and retirement completed, and the number of RFFull stall cycles.

- **Step S1b**: Change LOADFP_CYCLES to 16 and repeat step **S1a**.

- On the same plot (plot S1) graph the IPC (the reciprocal of the average number of cycles per retired instruction) against the number of Reservation Stations for both **S1a** and **S1b**. Use base-2 logarithm of the number of RS for the x axis of your plot. Explain the shape of the plot for S1a and the plot for S1b. Why are they so different?

- **Step S2a**: 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 S1. Next repeat this step for NUM_RESERVATION_STATIONS = 64.

- On the same plot (plot S2A) graph the CPI (average number of cycles per retired instruction) against the number of FU copies, for both values of NUM_RESERVATION_STATIONS. (You should set the y axis to be between 0 and 5 so that small perturbations in CPI values are masked.) Explain the plot.

- **Step S2b**: Change LOADFP_CYCLES to 16 and repeat step S2a. Create plot S2B using the data of this step. Explain the plot.

PROGRAM 2

- Repeat Step S1 and S2 (all parts) 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 Owlspace by the due date.