2162 Term Project

The Tomasulo Algorithm Implementation

Assigned: 11/3/2015

Due: 12/15/2015

In this project, you will implement the Tomasulo algorithm with register renaming, ROB, speculative execution (branch prediction + misprediction handling)*, and memory disambiguation technique. You may choose your own programming langue that run in either Windows or Linux, as I will be using those to run your code. Please document your code properly. Failure to do so may result in point deduction. On the due day, you will turn in a report and the source code electronically. I will run some test cases in your code on my machines. Be prepared for email questions from me on compiling and executing your code.

- **Forming a team**

You should form a team of three people. Each member in a group should carry about the same amount of responsibility. Team members will receive individual grade corresponding to his/her responsibilities in the project. Contribution from all team members is key to the success of this project.

- **Processor configuration**

You will implement the following 5 stages: ISSUE, EX, MEM, WB, and COMMIT. The ISSUE stage also includes instruction fetch and decode. The EX stage for loads and stores calculates their effective address (as explained below). Branch resolution is also done in the EX stage. The tasks performed in each stage are in line with what we discussed in class except for the specifications given for the load/store queue and branch unit.

The main components of the processor are: one instruction buffer, architecture register files (integer and floating point), register aliasing table (RAT), reservation stations for each function units, one CDB, one ROB, one branch predictor including the target buffer, and one load/store queue. The ROB fields are defined as explained on page 106 of the text. One point missing in the text is that the "Destination" field of a store instruction records its location in the load/store queue. The details of the load/store queue are given below.

Load/Store queue. All loads and stores are enqueued in their program order. The queue can be viewed as the reservation stations for the memory unit. There is a dedicated adder for effective address calculation. This is done in the EX stage of the load/store instruction. Hence, loads and stores do not occupy the integer ALU. Each entry in the queue contains two fields: address and value (not useful for loads).

Stores write into the memory only in the commit stage. Hence, they are dequeued only when they commit. A load, however, can go to memory whenever the address is calculated, and no forwarding-from-a-store was found. Forwarding-from-a-store is checked in the memory stage. It takes 1 cycle to perform the forwarding if a match is found. If not found, or there are still unresolved memory addresses for stores, then the load accesses the memory for data. Once a load returns from the memory or gets

the value from a previous store, its entry in the load/store queue is cleared. Note that it is correct not to clear this entry, but the queue can quickly fill up, causing structure hazards for future loads/stores.

The details about the branch unit:

Branch unit. Branch instructions are issued into the reservation stations of an integer ALU. We will implement the simplest one-bit predictor for each branch instruction. Use a BTB of 8 entries to store the target. Use the least significant 3 bits of the word address of the PC to index into the BTB. Prediction is done in the first cycle of execution (ISSUE stage). The branch is resolved at the end of the EX stage. Upon a misprediction, actions must be taken to squash wrong instructions. These include: 1. recover the RAT; 2. clear the reservation station's wait-for tag fields for wrong instructions; 3. clear ROB entries that surpass the branch. Assume these actions take one cycle, and fetching from the correct instruction starts in the next cycle. For example, if misprediction is detected in cycle n, correct fetch should start at cycle n+2.

Other hardware units. We will consider the following FUs:

1. Integer adder; unpipelined
2. FP adder; pipelined
3. FP multiplier; pipelined
4. Integer and FP register files, 32 entries each. Integer R0 is hardwired to 0.
5. Memory; single-ported; non-pipelined

The hardware configuration should be fully parameterizable, e.g. the # of ROB entries, the # of reservation stations for each FU, the # of execution cycles for each FU, the # of cycles for memory access etc. should all be the inputs to your simulator. You will simulate a "memory" of 256B (64W) which stores the data for a running testbench. This memory can be as simple as a data array. I will leave the implementation details to you as long as you have your own way of initializing the memory and printing it out finally. During the demo, I will let you initialize your own memory at the address and values I defined.

- **Instruction Set Architecture**

  Data Transfer Instructions

Ld Fa, offset(Ra)	Load a single precision floating point value to Fa
Sd Fa, offset(Ra)	Store a single precision floating point value to memory

  Control Transfer Instructions

Beq Rs, Rt, offset	If Rs==Rt then branch to PC+4+offset<<2
Bne Rs, Rt, offset	If Rs!=Rt then branch to PC+4+offset<<2

  ALU Instructions

Add Rd, Rs, Rt	Rd = Rs + Rt	Integer
Add.d Fd, Fs, Ft	Fd = Fs + Ft	FP
Addi Rt, Rs, immediate	Rt = Rs + immediate	Integer
Sub Rd, Rs, Rt	Rd = Rs - Rt	Integer
Sub.d Fd, Fs, Ft	Fd = Fs – Ft	FP
Mult.d Fd, Fs, Ft	Fd = Fs * Ft, assuming that Fd is enough to	FP

	hold the result	

- **Inputs**

The input to your simulator is a text file containing assembly instructions specified in the above format. The text is case insensitive. The configurations of the hardware units such as RS numbers and EX cycles are also provided through this input file. The registers should be initialized to zero unless specified in the input. Memory addresses are byte addresses. A sample input is shown below. But I'll leave it to you to decide on how to initialize the configurations.

	# of rs	Cycles in EX	Cycles in Mem	# of FUs
Integer adder	2	1		1
FP adder	3	3		1
FP multiplier	2	20		1
Load/store unit	3	1	4	1

ROB entries = 128

R1=10, R2=20, F2=30.1

Mem[4]=1, Mem[8]=2, Mem[12]=3.4

Add.d F1, F2, F3

Ld F4, 8(R1)

Bne R2, R3, -3

- **Outputs**

The output from your simulator should contain information about the final instruction status table including the cycles in each stage, and the results of the executed program including both registers and memory contents (non-zeros), formatted as the following.

ISSUE EX MEM WB COMMIT

Instruction_1
Instruction_2
Instruction_3
…
start cycle-end cycle
loop { Instruction_1, Instruction_2, Instruction_3
…

REGISTER VALUES

R0 R1 R2…
value:
F0 F1 F2…
value:

NON-ZERO MEMORY VALUES

Addresses	Values
addr_1	xxx
addr_2	yyy
addr_3	zzz
…	

- **Test benchmarks**

Write your own test cases. This part should be included in your final report in a clear manner. The quality and the length of test cases will be taken into consideration of your project grade. Your test cases should cover base cases, load/store queue forwarding, structure hazards, branch prediction, misprediction handling, etc. Be creative.

- **Format of report**

Section 1: Instructions on how to run your executable including 1) system requirement; 2) how to compile (command, makefile etc.); 3) name of executable (.exe); 4) how to input; 5) location of output (screen, file)

Section 2: Test benchmarks developed by your group, and their results in the above Output format. They should be developed with progression from easy to hard, and categorized into 1) straight-line base cases where no dependencies exist among instructions; 2) straight-line code where there are dependencies (true and false); 3) straight-line code where there are forwarding among load/store instructions; 4) straight-line code where there are structure hazards in reservation stations and functional unit; 5) simple loop on top of either 1), 2), 3), or 4), or all of them; 6)* demonstration of branch prediction. For example, upon misprediction, instructions from wrong path are fetched.

Section 3: Responsibilities of each individual group member, e.g. portion of code written by him/her, test benchmarks written by him/her, debugging effort, meeting/participation effort, etc.

- **Turn in**

Phase I (50%, due 11pm, 11/24/2015): preliminary version of your report. The requirement of this phase is to finish the skeleton of the code. You must be able to take a complete input file, parse it correctly, and produce output file for the simplest test cases. You should also finish developing of **all test cases**, as outlined above, and hand-derive their outputs. In the report, include a sample input test case, and sample output format (with or without results). This would serve the purpose of debugging for your second phase.

Phase II (50%, 12/15/2015): 1) final version of your report; this version of report only adds to the preliminary version the results produced by your simulator for each test case you've developed. You may include more test cases if you wish. 2) final source code; 3) an email from individual team member with evaluations of other two members and yourself. This evaluation should state clearly each member's responsibilities in the project, and contributions, including amount of work and time devoted. A grade should be given to your partners from your point of view in the following scale:

4 – extremely satisfactory, he/she is the main developer of the project. I would definitely have him/her as my partner again next time

3 – satisfactory, he/she is not the main developer, but still contributed significantly in this project.

2 – borderline, he/she contributed positively to the project, but not enough to ensure to completeness. Someone else (you, or the other member) has to make up for the loss.

1 -- he/she did not really contribute to the project due to (time, etc.). I would probably switch to others next time I need a partner.

- **Tips**

1. Before you start coding, you should first understand fully how the algorithm works. Developing your own test cases helps a lot.
2. Start from small sample code, one instruction at a time. If a single instruction doesn't work, neither does a sequence of code.
3. Vary the input parameters during debugging. For example, change the number of reservation numbers and the FU execution latencies, and see how your program reacts.
4. Write a powerful print_stat() that can generate pretty on-screen display of the reservation station status, instruction status, the register result status, and memory status on every cycle. This can be of great help in debugging.
5. Also design a clear output display for registers and memory contents. This can help improve my impression on your code quality.
6. Divide the work among the group members. When you can distribute the load in a team efficiently, you can improve your time management and the quality of your project.
7. Make plans ahead of time. Conquer the project step by step. For example, instruction parsing, input interface, and output display can all be done first and separately. Leave at least one week for final testing and debugging. Don't wait till last minute.