Project 2: Tomasulo Algorithm

Maxim Kovalev (mkovalev@andrew.cmu.edu),
Mridula Chappalli Srinivasa (mchappal@andrew.cmu.edu)
Department of Electrical and Computer Engineering
Carnegie Mellon University Silicon Valley Campus
Moffett Field, CA 94043

February 2014

Abstract

In this project we have succesfully implemented Tomasulo Algorithm for an arbitrary number of reservation stations and other parameters, and verified that it works correctly on reference traces.

Contents

1 Introduction

In this project, we have implemented a variation of Tomasulo Algorithm for add and multiply instructions with register operands. This alrogithm in used to

facilitate out-of-order execution by issuing inctuctions to special buffers called reservation stations, and executing instuctions simultaneously or in the order of true data dependencies rather than the order of issuing.

With the default paramaters provided in the assignment, we have been able to achieve the speedup of over 400% compared to in-order execution.

1.1 Sequential version

In order to verify the results, we have made two implementations of an in-order non-pipelined simulator. One was written from scratch in Python, and another was using the same tomasulo_sim.o framework. Both ouputted exactly the same results, which we regard as a good evidence that both of them work correctly.

Since by default Python uses arbitrary-precision arithmetic, we have discovered that even in small trace the values in registers go far beyond the capacity of 32-bit integer. Thus, we concluded that these traces highly depend on overflowing and switched to the 32-bit signed integer implementation provided by NumPy library

2 Implementation

2.1 Data structures

2.1.1 Reservation stations

There are two reservation stations in this simulator - one for the adder, and the other for the multiplier - that are both arrays of reservation station slots. One slot is a structure with two operand fields, two tag fields, an availablity flag, that indicates if this slot is "empty", an "is being executed" flag that indicates whther the instruction in this reservation station has been sent to ALU but not finished yet, and the counter for the number of cycles spent by the instruction in the reservation station to implement the QoS.

2.1.2 Register file

Register file consists of two arrays - register values and tags. Unlike in actuall processors, we don't use a busy bit, but simply indicate that there is the actual value in the register by setting the tag equal to -1.

2.2 Tagging algorithm

In our implementation we use two arrays for adder and multiplier reservation stations, both having indices starting from 0. A tag, however, should be a global indentifier of a reservatoin station, which is why we need to use a reversible algorithm for generating these tags. To do that, we left shift the index in the array by one bit, and if it is the adder reservation station, we write 1 to the least signigifant bit; otherwise we leave 0 there.

2.3 initTomasulo

In this function we first try to read the config file to determine the number of reservation stations. If the file is not found, we use 3 adder reservation stations and 5 multiplier ones. Then we allocate the memory for these stations, mark them as availbale, and then initialize registers to 0.

2.4 issue

In this function, we first try to determine if there is any available reservation station, depending on the type of the instruction. If there are no such stations, this instruction is not issued.

Otherwise, we try to fetch its operands. If its second operand is a constant rather that a register value, we simply load it to the reservation station and set the tag to -1. Otherwise, we check if the tag in the corresponding resister is -1, meaning that the value can be used, it is loaded, and the tag in the reservation station is set to -1. Otherwise, we disregard the value and just copy the value of the tag. The same process is repeated for the first operand.

The value of the counter of the cycles spent in the reservation station is set to 0.

Finally, according to our tagging alrogithm, we calculate the tag for the selected reservation station, and write it to the tag field of the destination register.

2.5 execute

First of all, we use this function to increment the idle time counter for each used reservation station. Even if this function is called multiple times during one cycle, it increments the counters uniformly, thus making them directly proportional to the idle time. This can be done in every function that is guaranteed to be called the same number of times during each cycle, except for issue: in this case, some of the instuctions are not in the reservation stations yet, which is why it does not make sense to increment the counters.

Then, depending on the type of the instuction, we check if there are any slots in the reservation stations ready to be executed (i.e. are not empty, have both tags equal to -1, and are not being already executed), find the one that has been sitting in the reservation station for the longest time, and pushes it to the execute request.

2.6 writeResult

This function implements CDB broadcasts. It iterates over all reservatoin stations and registers, writes the resuling values wehnever there is a tag match, and sets the tag to -1. Then, it determines from which reservation station this instruction came, and marks it as not being executed and available.

2.7 checkDone

This function verifies that all the registers have tags equal to -1, all the reservation stations are empty, and if that is true, it dumps the register values and stops the simulation.

3 Questions

1. **Explain how Tomasulo's algorithm avoids WAW, RAW, WAR hazards. If we made the Instruction issue out-of-order, do any of these hazards now exist?**

   Read after Write dependencies are eliminated by the use of tags. If there is a pending write to the register, it is identified by a tag and when this register is listed as one of the operands in the reservation station, then the associated tag is copied to the reservation station instead of the value, thereby preventing RAW hazards. Also, the results of execution can be transferred to the operands waiting on that particular result in the reservation station and the registers directly through common data bus.(This is achieved through tag broadcast and matching).[1]

   The write after read dependency is taken care of by copying the value in the register operand to the reservation station before it could be overwritten by another instruction.Due to this early copying mechanism, WAR hazards are avoided. [1]

   The write after write dependency is avoided by tag updation i.e the tag associated with a register is updated to a value reflecting the latest instruction to update that register, so that the earlier instruction that also had to write to the same destination register cannot overwrite the latest value.[1]

   If the instructions are issued out of order, there is a possibility of Write After Read hazard, as the instruction at a later stage in the program can write to the register before an earlier instruction can read the operand from that particular register.

2. **How did you generate tags in your implementation? Is this how you would do it in hardware?**

   The number of hardware tags to be generated is decided based on the number of contributers to the common data bus. We treat each tag as a 32-bit integer in our implementation. Since the number of tags is only determined by the number of reservation stations for this particular implementation, we could have used lesser number of bits to store the tag value.

   In addition, as the "not tagged" tag we used -1, which is difficult to implement in hardware. In this case we would use an additional bit, for

distinguishing whether it is the actual value or a tag in the register, and disregard the tag or the value depending on this bit.

Finally, since our simulator is programmed to work with any number of reservation stations, we use a relatively sophisticated technique to address two arrays of reservation stations, whose size is unknown in the compile time. In actual hardware the number of reservation stations is fixed, so it would be possible to employ a much simpler flat addressing mode.

3. **Try increasing the number of slots in your reservation stations and the maximum issue rate. Why does performance improve even though only 1 instruction can begin execution per cycle for a single reservation station? When is it better to increase the maximum instructions issued per cycle? When is it better to increase the number of reservations station slots?**

   The maximum instructions issued per cycle can be increased when there are fewer dependencies between the instructions (the operands of the instructions are available immediately to be executed) and the instructions are executed almost immediately after entering the reservation station(as the operands are already available).

   The number of reservation station slots can be increased if there are dependencies between instructions, i.e. if an instruction waiting on the availability of a particular operand can sit in the reservation station for a long time preventing the instruction that has all its operands ready(an independent instruction) from executing thereby causing unnecessary stalls. If the number of slots in the reservation stations are increased to accommadate these instructions, then the processor performance can be increased by enabling independent instructions to execute immediately.

4. **Although your implementation only had a single reservation station per functional unit, it is possible to have multiple reservation stations with same functionality (e.g., 4 separate reservation stations, each with a multiplier functional unit) to increase overall throughput. How does one choose the right number of reservation stations?**

   If there are large number of arithmetic instructions that utilize the functional units frequently, then by having each reservation slot associate with the appropriate functional unit, it is possible to increase the performance by faster execution of those instructions.Instructions need not stall waiting for the availability of the functional units. However, the cost of implementing the functional units in hardware is not trivial. Thereby the performance increase obtained by increasing the number of functional units should be justifiable wrt to hardware cost incurred.

5. **What changes would need to be made in order to support branches, loads and stores?** We need buffers to store the data loaded from memory and store buffers to be implemented for store instructions. The load

buffer should be able to transfer the operands to the reservation stations while the Store buffer should be able to pull the results immediately after execution(through the common data bus). We need to account for RAW dependencies between load and store instructions.

One way of accounting for branch instructions is to stall the execution until the branches are resolved. We could have a reservation station for the branch instructions which could pull the outcome of the condition operation through common data bus(for conditional branches) and then issue appropriate instructions to the reservation stations.Other possibility is to use a branch prediction mechanism to predict the outcome of a branch and fetch and issue the appropriate instructions to the reservation stations.However, if the branch is mispredicted, then this can nullify the performance gain obtained by the out-of-order execution core.[1]

4 References

[1] P. Shen, John; Lipasti, Mikko *Modern processor design: fundamentals of superscalar processors*. Boston: McGraw-Hill Higher Education. 2005