CSL 718

Architecture of High Performance Computers

Assignment 3

IMPLEMENTATION OF TOMASULO'S ALGORITHM IN TEJAS

Submitted By:

Pallavi Sethi (2014MCS2868)

Harinder Pal (2014MCS2123)

VERSION HISTORY:

Version	Author(s)	Description	Date Completed
1.0	Pallavi Sethi, Harinder Pal	Initial Draft	03/27/2015
2.0	Pallavi Sethi, Harinder Pal	Final Report	04/26/2015

Table of Contents

1. INTRODUCTION

1.1. Scope of the document

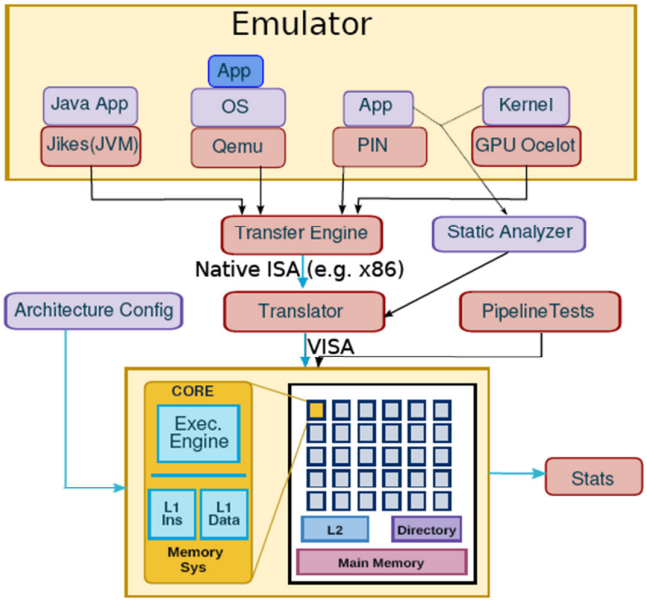
This document explains the design of the strategy to implement Tomasulo's algorithm for simulating "out-of-order" pipeline exectuion in Tejas (*The Efficient Java based Architectural Simulator).*

2. TEJAS

2.1. About Tejas:

Tejas is an open-source, Java-based multicore architectural simulator built by the Srishti research group, IIT Delhi

It is a trace driven simulator which is platform independent & is one of the fastest cycle accurate simulators available in academia.



High Level Architecture of Tejas

2.2. Important Components of Tejas:

I. Emulator
II. Transfer Engine
III. Translator
IV. Pipeline
V. Branch Predictor
VI. Memory System

2.2.1. Emulator:

This is used to collect the traces of the input benchmark. We will be using Intel-pin emulator for our purpose, though it also supports other emulators like Jikes, Qemu, GPUOcelot etc.

Tejas includes *CasualityTool.cpp* which is responsible for collecting the required traces using pin tool.

2.2.2. Transfer Engine:

Traces collected by the emulator are then fed to the translator. This requires a communication mechanism, Tejas supports four types of communication mechanisms – (1) Shared Memory, (2) files, (3) pipes & (4) network sockets.

The package – *emulatorInterface.communication.** packages are used to implement the above mentioned communication channels.

2.2.3. Translator:

The traces are fed to the translator modules which are then converted to a RISC instruction set called VISA (Virtual Instruction Set Architecture). This allows us to have a smaller instruction set compared to CISC x86 & thus can be easily simulated.

The package – *emulatorInterface.translator.** packages are used to implement the translator module in Tejas. We have separate class files for each type of VISA instruction inside the visaHandler package. All these classes implements DynamicInstructionHandler interface & define a 'handle function'.

2.2.4. Pipelines:

We have separate pipelines of in-order & out-of-order execution. All the pipeline stages are implemented as java class files in the pipeline packages.

Both the in-order & out-of-order pipelines implements PipelineInterface, which includes some of the important functions for the execution. One of them is OneCycleOperation.

OneCycleOperation:

This function is called after every cycle & thus includes the tasks to be completed in a cycle. After every call of this function, the clock cycle is incremented by 1. We include the pipeline stages logic call in the function.

2.2.5 Branch Predictor:

The emulator sends the trace of only those instructions that were on correct control path & informs the simulator if a branch was taken or not. The predicted outcome & emulator provided outcome are compared to indicate whether a mispredictions has occurred.

Many types of branch predictors are implemented in Tejas: alwaysTaken, alwaysNotTaken, GAG, GAP, Bimodal, GShare *etc.*

The package *pipeline.branchpredictor* contains the required code. It contains a separate class file for each type of branch predictor.

2.2.6. Memory System:

Tejas includes a memory hierarchy with a main memory, L2 shared cache, a shared directory & L1 cache (per core). All the memory operations are handled through events.

The package *memorySystem.** includes all the related code.

2.3. Configuring Tejas:

Tejas can be used in multiple configurations, the parameters can be configured through an xml file (config.xml). This file is parsed in the beginning and all the parameters are set.

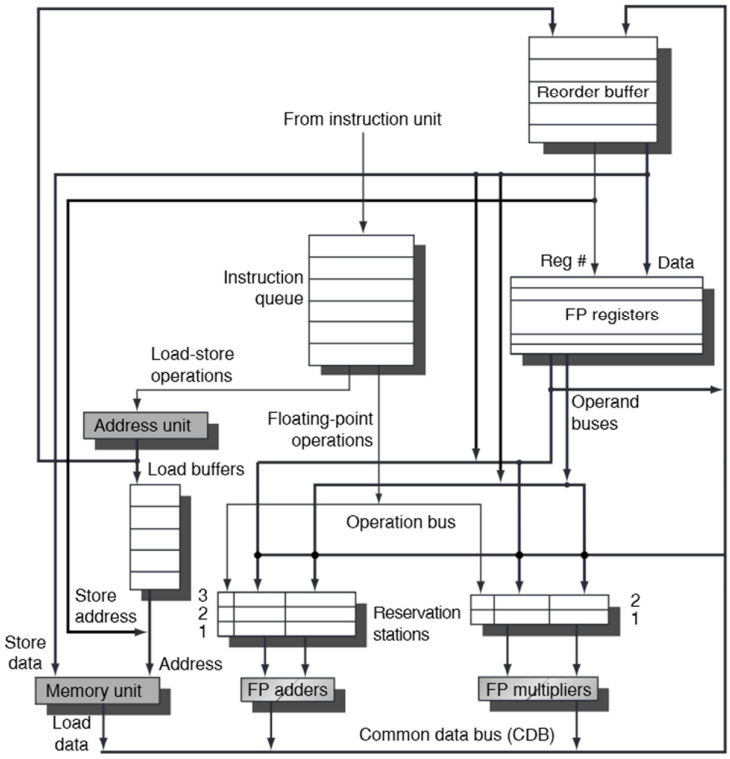
3. TOMASULO

3.1. About Tomasulo's algorithm:

Tomasulo's algorithm is used for dynamic scheduling of instructions to allow out-of-order execution in superscalar processors.

The key points of the algorithm:

- It follows Issue Bound Operand Fetch technique.
- It handles all the data hazards by using register renaming, data forwarding & ROB (Re Order Buffer).
- Register Renaming : It is provided by Reservation stations which buffer the operands of instructions waiting to be issued.
  Basically it eliminates the need to get the operand from register as it buffers the operands in reservation stations as soon as they are available.
- Data Forwarding : A common data bus (CDB) is used to connect the output of Functional Units to Reservation Stations, ROB & register file.
- ROB : This is basically used to implement *speculation*, wherein register file is not updated until an instruction commits. Instructions can be executed out-of-order but will be committed in-order.
- Load/Store Queue: A load/store queue is maintained to eliminate data hazards RAW,WAR and WAW.



Tomasulo with ROB

4. Implementation Details

We have reused most of the Tejas components including Emulator, TransferEngine, Translator, Memory System and Branch Predictor. We have modified the pipeline.multi_issue_inorder package to implement Tomasulo's out-of-order strategy.

4.1. Data Structures used for implementing Tomasulo's Algorithm:

4.1.1. Reservation Stations (Toma_ReservationStation.java)

It contains an array of Toma_ReservationStationEntry, and its size.
The Toma_ReservationStationEntry.java contains the following fields:

- instruction: Reference of the Instruction which has currently occupied Reservation Station entry.
- isBusy: Whether the Reservation Station entry is free or not.
- inst_entryNumber_ROB: Entry Number of Instruction in ROB.
- sourceOperand1_availability/sourceOperand2_availability: For both source operands, If they are available or not. Contains 0 when value is available in value fields. Otherwise contains ROB entry Number of Instruction whose destination Register is the source operand.
- sourceOperand1_value/sourceOperand2_value: Value of source operands if available.
- Address: used to store address in case of Load/Store Instructions.
- IsStartedExecution: Boolean indicating if execution of Instruction has started.
- IsCompletedExecution: Boolean indicating if execution of Instruction has completed.
- TimeToCompleteExecution: Time at which the execution will be complete. It is set using latency of corresponding Functional Unit, when we start execution. Every cycle, we check, if an Instruction has started execution, then current time is compared to this time. If yes, then IsCompletedExecution is set to true.

We have implemented a centralized reservation station for all Functional Units. Its size is configurable. Instructions are issued into Reservation Station from output of Fetch Stage.

4.1.2. Register File (Toma_RegisterFile.java)

The main components of RegisterFile.java include:

- toma_ROBentry []: Array which contains entry number of Instruction in ROB
- isBusy []: Array which holds the status of the value present in register File. An entry being busy will mean that the value present is not valid (yet to be committed)
- Values []: Array of register file values

- registerFileSize: Size of Register File, which is configurable

4.1.3. ROB (Toma_ReOrderBuffer.java)

ROB contains an array of Toma_ROBentry, along with head, tail & the size.
Each Toma_ROBentry class includes the following fields:

- Instruction: Reference of the Instruction which has currently occupied ROB entry.
- isBusy: Status being busy will mean that the entry contains an instruction which is not yet committed.
- isReady: Tells whether the result is computed or not
- resultValue: result value
- destinationRegNumber: holds the register number of the destination register.
- toma_lsqEntry: Reference of LSQ entry, in case of Load/Store Instructions

Instructions are added into ReOrder Buffer at Issue stage, at the same time when they are added to the Reservation Station. So we need to have space in both Reservation Station and ROB in order to Issue the Instruction.

4.1.4. Common Data Bus (CDB)

To implement Common Data Bus, we have created 3 classes:

- Toma_CDB.java: to carry the calculated result after execution completes, to Reservation station and ROB.
- Toma_CDBentry.java: It includes the information, which is sent while sending request to handle CDB event.
- Toma_CDBevent.java: The event which is used when we handle CDB request using events.

The CDB can be multiport the number of ports can be configurable in config.xml

4.1.5. Load/Store Queue (Toma_LSQ.java)

We even need a load & store Queue for the memory operations. The pending load & store instructions (memory address & the value) are stored in these buffers. It is used to eliminate the data hazards like RAW, WAW and WAR.
Toma_LSQ contains an array of Toma_LSQentry, along with head, tail & the size.
Fields in Toma_LSQentry are:

- Toma_LSQEntryType: Type is Load ot Store
- Address: to store address of Load/Store
- isOccupied: boolean indicating whether entry is free or occupied
- isStartedCalculatingAddress: boolean indicating whether the address calculation has started.
- IsAddressCalculated: boolean indicating whether Address calculation is complete
- timeToCompleteAddressCalculation:
- toma_robEntry: Reference of Reservation Station entry, in which current Load/Store Instruction is present.

• toma_Rsentry: Reference of ROB entry, in which current Load/Store Instruction is present.
• IndexInQ: Index in Load/store Queue.

Instructions are added in LSQ in the issue Stage. If LSQ is full, then Instruction cannot be issued and also no Further instruction can be issued because we have to Insert Instructions in ROB in order so that Instructions are committed in order.

4.2. Pipeline Stages to be implemented:

The following are the pipeline stages which we will be simulating as per Tomasulo's algorithm. We will be writing a separate class for each stage as follows.

1) Issue (Toma_issue.java)
2) Execute (Toma_execute.java)
3) WriteResult (Toma_writeResult.java)
4) Commit (Toma_ROB.java extends)

We are reusing Fetch Stage of Multi Issue Inorder pipeline (FetchUnitIn_MII.java) for our pipeline. It fetches Instructions from *'InputToPipeline'* into ifIdLatch.

1) Issue: In this stage, we need to issue Instructions from ifIdLatch.
Before doing this, we will check 3 conditions:
a) Whether the **Reservation Station** has free space for new incoming Instruction.
b) Whether the **Re-Order Buffer (ROB)** has a free space.
Once these Conditions are satisfied, we put the Instruction in both Reservation Station and ROB
c) Whether the **LSQ** has free space. This condition is only for Load/Store Instructions.
If any of these conditions are not satisfied, we cannot issue the instruction. Also, we cannot issue any further instructions because We must have instructions in ROB in order so as to commit in order.
Adding Instruction in Reservation station: First, we set instruction reference and ROB reference in the entry. Then check both source Operands are available in Register file or not. This is checked using isBusy field of Register File. If No, then *SourceOperand availability* is set to zero and its *SourceOperand value* is set. Other appropriate fields are also set.
Adding Instruction in ROB: First, we set instruction reference and LSQ reference in the entry. If destination register is not null, then set ROB entry number of that register in the Register File. Then set the destination register number in ROB entry.
Adding Instruction in LSQ: Load/Store Instructions are added to LSQ with all appropriate fields.

2) Execute: In this stage, we actually execute the instruction, once the operands are available in the Reservation station and FU is free.
We check 3 conditions before executing (iterate over all entries in reservation station):
a) Both source operands are available: check Source operand availability is zero for both operands. If no, then continue and check for next Instruction in reservation station.
b)Instruction has not yet started execution: If completed execution, then continue and check for next Instruction in reservation station. If started execution, then check if time

has come to complete its execution. If not yet started, then go to next step. For Load/store Instructions started execution means started calculating address.
c) Functional Unit is available: We request FU, if it is available we get latency of that FU and set timeToCompleteExecution as Current Time + Latency .And then setStartedExecution(true). In case of Load, after address calculation, we issue '*cache read*' request to read from the memory if no previous store instruction with same address is in the LSQ. When this '*cache read*' event is handled, isExecutionComplete is set to true.

3) **Write Result:** Once the execution is complete, the Result is sent through the common data bus to the Re Order Buffer and Reservation stations. In this stage, we iterate over all entries in reservation station and do as follows:

a) check if completed execution: If yes, then issue request to CDB through event, else continue with next entry in Reservation Station.
b) check Operation type: If operation type of Instruction is *invalid, nop, store, Jump or branch*, then we do not issue request to CDB because these instructions do not have any destination value set after execution, which is to be sent through CDB to ROB and Reservation Station
c) CDB request handle: When CDB request is handled, we need to check in whole reservation station if some source operand is waiting for the destination register whose value is now available and sent through CDB. All such source operands are then set available. Instruction from Reservation station is removed and ready field of ROB is set as true. The instruction can now be committed.

4) **Commit:** Completed Instructions are committed in order and the results are written back to the Register file. The following steps are followed:

a) Check if Instruction at head of ROB is ready. If not, then do nothing.
b) If ready, then we have different logic of commit for branch, non-branch and store instructions.
c) Store: To write the data to memory, send '*cache write'* event to L1 Cache. After this, we assume that memory write is complete as it is buffered and will complete afterwards when darinevents function is called. Now, Remove entry from ROB and clear the ROB entry.
d)Branch: To check branch misprediction, If compare isBranchTaken and branchPrediction Result. If branch is mispredicted, then we have to stall the pipeline for some time (Branch msiprediction Penalty time). For this, we send an branch misprediction event and set *'barnch misprediction* stall' to true. After this set the destination register value and free the ROB entry. In all stages, we have a check for *'barnch misprediction* stall'. If it is true, nothing happens. When this event is handled, *'barnch misprediction* stall' is set to true. Therefore, Till this event is not handled, pipeline will be stalled.
We do not need to flush the ROB and fetch Branch destination as it is written in the algorithm because we are only simulating it and we already have correct sequence of instructions.
e)Non Branch: Simply commit, set the destination register value and free the ROB entry.

We commit as many instructions as possible in one cycle.

Configuration File (config.xml): We have added some configurable parameters required for Tomasulo's algorithm, which are as follows:

- rs_buffersize: Size of Reservation Station
- rob_buffersize: Size of ReOrder Buffer
- Toma_LSQSize: Size of Load/ Store Queue
- Toma_CDBLatency: Latency of Common Data Bus
- Toma_CDBPortType: Port Type of Common Data Bus
- Toma_CDBAccessPorts: Number of Ports in Common Data Bus
- Toma_CDBPortOccupancy: Occupancy of Common Data Bus

5. TESTING:

We have used pipelineTests.java for implementing the following test cases.

1. **toma_test_simple_noInst:** In this test case, we do not have any instruction. There is only invalid instruction in inputToPipeline.
2. **toma_test_simple_nop:** In this test case, we have only 1 nop instruction.
3. **toma_test_simple_intALU:** In this test case, we have only 1 intALU instruction.
4. **toma_test_simple_mov:** In this test case, we have only 1 mov instruction.
5. **toma_test_simple_FloatDIV:** In this test case, we have only 1 FloatDIV instruction.
6. **toma_test_simple_xchg:** In this test case, we have only 1 xchg instruction.
7. **toma_test_dependency**: In this test case, we have 2 instructions. Second Instruction depends on first. We verified that second instruction waits for first to complete execution before itself starting execution.
8. **toma_test_no_dependency:** In this test case, we have 2 instructions. Second Instruction does not depend on first.
9. **toma_test_out_of_order:** In this test case, we have 3 instructions. Second Instruction does not depend on first. Third is not dependent. We verified that while second is waiting for operand, third started execution.
10. **toma_test_dependency_xchg:** In this test case, we have 2 instructions. Second is xchg Instruction, which depend on first.
11. **toma_test_dependency_xchg_2:** In this test case, we have 4 instructions. Second, third and fourth instruction depend on first. Third instruction is xchg.
12. **toma_test_dependency_int_float:** In this test case, we have 2 instructions. One is intALU and second is FloatALU. Second seems to depend on first because register number is same but actually they are not dependent as register files are different.
13. **toma_test_mov_imm:** In this test case, we have only 1 move immediate instruction.
14. **MinimumDataDependencies:** In this test case , we have 100 instructions, a sequence of intALU instructions that have no data dependencies
15. **maximumDataDependencies:**In this test case , we have 100 instructions, a sequence of intALU instructions, with (i+1)th instruction dependent on ith.

16. **StructuralHazards:** In this test case , we have 100 instructions, a sequence of floatDiv instructions, with no data dependencies

17. **toma_test_simple_load:** In this test case, we have only 1 load instruction.

18. **toma_test_simple_store:** In this test case, we have only 1 store instruction.

19. **toma_test_dependency_load:** In this test case, we have 2 instructions. Second IntALU Instruction depends on first Load instruction. We verified that second instruction waits for first to complete execution before itself starting execution.

20. **toma_test_dependency_store:** In this test case, we have 2 instructions. Second store Instruction depends on first IntALU instruction. We verified that second instruction waits for first to complete execution before itself starting execution.

21. **toma_test_dependency_load_store:** In this test case, we have 4 load/store instructions, which have dependencies.

22. **toma_test_simple_jump:** In this test case, we have only 1 jump instruction.

23. **toma_test_simple_branch:** In this test case, we have only 1 branch instruction.

24. **toma_test_mispredicted_branch:** In this test case, we have 2 instructions. First is mispredicted branch. We verified that pipeline stalls when branch is mispredicted.

25. **toma_test_predicted_branch:** In this test case, we have 2 instructions. First is correctly predicted branch. We verified that pipeline does not stall.

Configuration: RS size = 2, ROB Size = 2, CDB access ports = 2, main Memory Latency = 200

S.No.	Test No.	Test Case	Number of Instructions	Cycles
1	11	toma_test_simple_noInst	0	7
2	12	toma_test_simple_nop	1	605
3	13	toma_test_simple_intALU	1	610
4	14	toma_test_simple_mov	1	610
5	15	toma_test_simple_floatDIV	1	633
6	16	toma_test_simple_xchg	1	608
7	21	toma_test_dependency	2	617
8	22	toma_test_no_dependency	2	613
9	23	toma_test_out_of_order	3	617
10	24	toma_test_dependency_xchg	2	617
11	25	toma_test_dependency_xchg_2	4	624
12	26	toma_test_dependency_int_float	2	612
13	27	toma_test_mov_imm	1	610
14	0	minimumDataDependencies	100	907
15	1	maximumDataDependencies	100	1303
16	2	structuralHazards	100	1821
17	3	renameTest	100	3603
18	17	toma_test_simple_load	1	1225
19	18	toma_test_simple_store	1	606
20	28	toma_test_dependency_load	2	1232
21	29	toma_test_dependency_store	2	613

22	30	toma_test_dependency_load_store	4	1234
23	19	toma_test_simple_jump	1	605
24	20	toma_test_simple_branch	1	605
25	31	toma_test_mispredicted_branch	2	614
26	32	toma_test_predicted_branch	2	610

6. REFERENCES:

http://www.cse.iitd.ac.in/tejas/