CSL 718

Architecture of High Performance Computers

Assignment 3 (Part1)

DESIGN DOCUMENT (V1.0)

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

Table of Contents

1. INTRODUCTION

1.1. Scope of the document

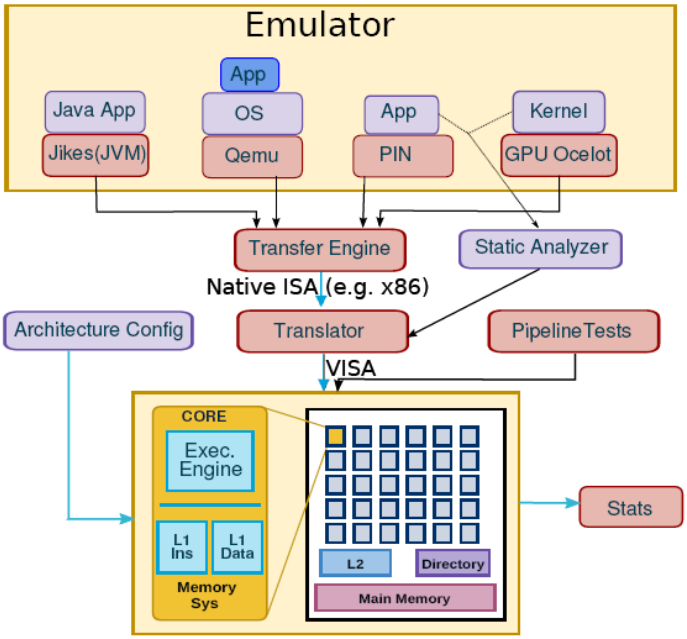
This document explains the design of the strategy to implement Tomasulo's algorithm for simulating "out-of-order" pipeline exectuion in Tejas (*The Efficient Java based Architectural Simulator).*

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.

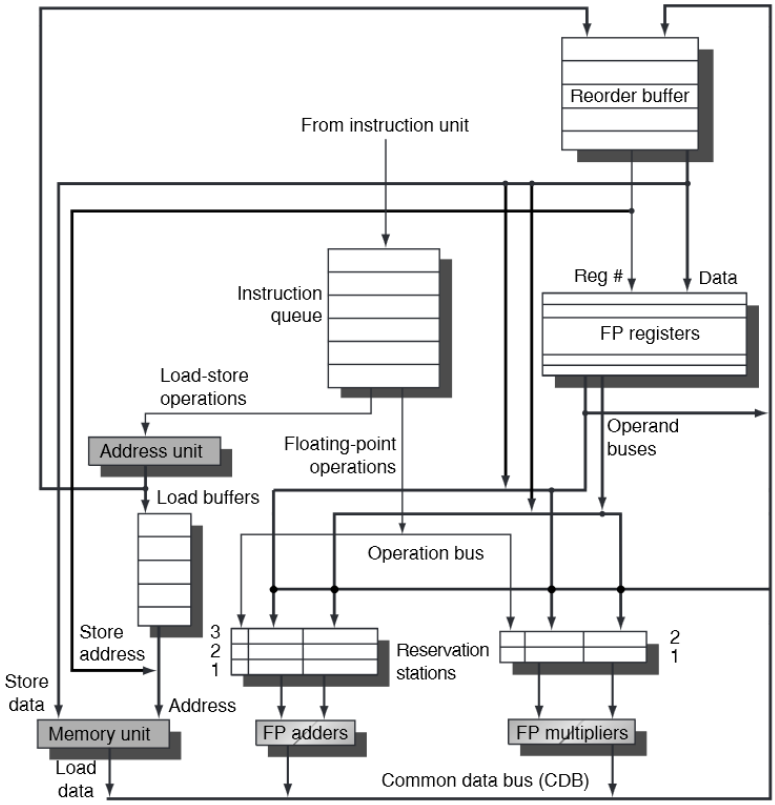
3. TOMASULO

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 commited in-order.



Tomasulo with ROB

4. Implementation Details

We will be reusing most of the Tejas components including Emulator, TransferEngine, Translator, MemorySystem & Branch Predictor. We will be modifying 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 (ReservationStation.java)

It contains an array of ReservationStationEntry.
The ReservationStationEntry.java contains the following fields:

- Op: Operation Type
- Busy: Whether the Reservation Station entry is free or not.
- Qi: Entry Number of Instruction in ROB.
- Qj/Qk: For both source operands, If they are available or not. Contains 0 when value is available in Vj/Vk. Otherwise contains ROB entry Number of Instruction whose destination Register is our source operand.
- Vj/Vk: Value of source operands if available.

The FunctionalUnit.java is updated with a variable of type ReservationStation since each FunctionalUnit needs a reservation station.

4.1.2. Register File (RegisterFile.java)

The main components of RegisterFile.java includes:

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

4.1.3. ROB (ReOrderBuffer.java)

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

- Inst: operation type
- Busy: Status being busy will mean that the entry contains an instruction which is not yet completed.
- Ready: Tells whether the result is computed or not
- Val: result value
- Dst: holds the register number of the destination register.

4.1.4. Common Data Bus (CommonDataBus.java)

Common Data Bus will hold the following fields:

- Data to be transferred
- ROB tag

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

4.1.5. Load/Store Buffer

We will even be needing a load & store buffer for the memory operations. The pending load & store instructions (memory address & the value) are stored in these buffers.

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 (Tomasulo_issue.java extends SimulationElement)
2) Execute (Tomasulo_execute.java extends SimulationElement)
3) WriteResult (Tomasulo_writeResult.java extends SimulationElement)
4) Commit (Tomasulo_commit.java extends SimulationElement)

1) Issue: In this stage, we need to get instructions from the Instruction Queue. Before doing this, we will check 2 conditions:
a) Whether the required 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

2) Execute: In this stage, we actually execute the instruction, once the operands are available in the Reservation station and FU is free.

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.

4) Commit: Instructions are stored in ROB when complete execution but they are committed in order. In this stage, the results are written back to the Register file.

5. TESTING:

We will be using pipelineTests.java for implementing the test cases. Along with this, some junit test cases will be written for the unit testing.

6. REFERENCES:

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