Webbased instruction level simulation of a parameterisable dynamic processor

Bachelorarbeit
von

Maximilian Senftleben

29. März 2011

Referent:	Prof. Dr. Klaus Schneider
Betreuer:	Jens Brandt

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema "Webbased instruction level simulation of a parameterisable dynamic processor" selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 29. März 2011

(Unterschrift)

____

Maximilian Senftleben

Abstract

This thesis describes the design and implementation of a simulator for a parameterisable dynamic scheduled processor and its underlying principles and required techniques. The principles of dynamic scheduling and speculative execution are explained. Some characteristics of the resulting simulator are defined and important hazards, which come along with the implementation, and their solutions are mentioned. The implemented simulator is described with its core components and most important functions.

Kurzüberblick

Diese Arbeit beschreibt das Entwerfen und die Umsetzung eines Simulators für einen parametrisierbaren, dynamischen Prozessor sowie die zugrundeliegenden Prinzipien und benötigten Verfahren. Die Grundlagen von dynamischer Prozessorablaufplanung und spekulativer Befehlsausführung werden beschrieben. Diverse Eigenschaften des resultierenden Simulators werden definiert und wichtige, die Implementierung begleitenden, Problematiken genannt und behandelt. Der implementierte Simulator wird mit seinen Kernbestandteilen und den wichtigsten Funktionen beschrieben.

Contents

List of Figures

List of Tables

“There are three arts which are concerned with all things: one which uses, another which makes, and a third which imitates them.”

Plato (427 BC - 347 BC)

Chapter 1

Introduction

1.1 Subject

One of the most important approaches to improve processor performance is pipelining. It divides the execution of an instruction into several steps and overlaps their execution to increase instruction throughput; e.g. the classic 5 stage RISC pipeline with the stages IF, ID, EX, MEM, WB[1] as used in the DLX and early MIPS processors. [1, App. A]

Superscalar execution is a more advanced improvement based on pipelining. The approach aims at the execution of multiple instructions from one instruction stream in parallel on different function units.

To minimize structural conflicts[2] and to avoid problems resulting from parallelization of an instruction stream there exist two different superscalar processor types: static and dynamic scheduled. [1, p. 114] The order of the instruction stream of a static scheduled processors is determined by a compiler based on detail knowledge about the target architecture. In contrast, a dynamic scheduled processor, reorders the instruction stream in execution time by itself to utilize the maximum number of available function units on the one hand while considering data dependencies[3] between instructions on the other hand.

[1]IF: instruction fetch, ID: instruction decode, EX: execute, MEM: memory update, WB: writeback
[2]hardware limitation, e.g. not enough integer function units
[3]e.g. RAW: read after write: an instruction should read a value written by a preceding instruction

1.2 Motivation

"In the author's opinion, entirely too much programmers' time has been spent in writing such simulators and entirely too much computer time has been wasted in using them." - Donald E. Knuth [2, p. 202]

Starting with this quote, I want to point out the mass of exisiting simulators and say a few words why I build another one.

As described before, dynamic scheduling is a powerful performance enhancement technology which therefore should be illustrated in a way easy to understand.

In my opinion, a simulator providing an example superscalar dynamic scheduled processor and offering the possibility to understand the operating principles by exploring them via trial-and-error and exercising based on given examples would be an valuable addition to computer science education.

In this thesis, I therefore want to address that issue by the approach to design and implement a simulation of a simplified processor providing the desired superscalar and dynamic scheduled architecture.

The result should offer a high degree of customizability to address a wide range of possible usage scenarios and offering a huge number of different exercises. To address a broad public it should be ready to use without much preparatory work, should not require additional software and preferably should be platform independent. The user interface should not require much explanation and allow an easy use of the simulator.

1.3 Structure and outline

At first the motivation to design a new simulator is stated in the first chapter along with a short introduction to the subject.

The next chapter 2 consists of reviews of common simulators. Here their usefulness to serve the described purpose is analysed. A conclusion explains reasons to design a new simulator instead of using one of the reviewed ones.

Chapter 3 establishes a knowledge base with some fundamental techniques required in this thesis.

The following chapter 4 states and discusses some design decisions made to simplify and adjust the simulator to the aimed usage scenarios. Furthermore, hazards are mentioned which have to be considered during the implementation.

The implementation itself is described in chapter Chapter 5. The most important structures and functions are described in detail and the occuring hazards and their solutions are explained.

Chapter 6 gives a short overview on the graphical user interface and describes how to use the interface and which restrictions have to be considered.

Finally in the last chapter 7 the resulting simulator is successfully validated. Along a short comment and suggestions for further work I conclude the successful thesis.

In this document I used some parts of the latex template of [3].

Chapter 2

Related Work

Because of the advantages a simulator can offer, many have been developed so far and others will be designed in the future. These simulators serve different purposes as well as doing so in varying ways with other degrees of complexity.

2.1 Review of common simulators

Written in 1992 SIMPLESCALAR has become one of the most popular simulators for computer architectures. "For example, in 2000 more than one-third of all papers published in top computer architecture conferences used the SimpleScalar tools to evaluate their designs."[4] It supports the instruction sets Alpha, PISA[1], ARM and x86 with and without microcode. The SIMPLESCALAR tool set contains some sample simulators including a fast functional processor, a dynamic scheduled processor with branch prediction and others. [5]

SESC[2] simulates the MIPS instruction set on different architectures, eg. a dynamic superscalar processor, CMP[3], processor-in-memory and speculative multithreading architectures in contrast to SIMPLESCALAR which is primarily designed to simulate single processors. [6]

A full-system simulation of the SPARCv9 instruction set is offered by GEMS[4] which is based on SIMICS. [7, 8]

[1]PISA: Portable ISA, similar to MIPS
[2]SESC: SuperESCalar Simulator
[3]CMP: Chip level multiprocessing
[4]GEMS: General Execution-driven Multiprocessor Simulator

The M5 Simulator is a highly object orientated simulator and therefore supports the usage of multiprocessors and a broad range of instruction sets: Alpha and SPARC in full-system simulation and MIPS, ARM and x86 currently in syscall emulation only. [9, 10]

PTLsim is a cycle accurate superscalar out of order x86 and x86-64 simulator with full support of x86-64, SSE/SSE2/SSE3, MMX, x87 and therefore "[...] being the only open source cycle accurate x86 microarchitectural simulator [of real and commercial x86 microarchitectures, A/N] available to researchers [...]"[11]. It is used by many universities, researchers and the processor vendors Intel and AMD themselves. [12, 13]

In contrast to previously named simulators, MARS[5] simulates the functional behaviour of assemblercode on a MIPS32 architecture. It is used to teach the assembler language as well as to test MIPS32 assembler programs. [14]

The JavaHASE applets illustrate and simulate different computer architectures step-by-step primarily for teaching purposes. JavaHASE is based on the powerful HASE[6] and the SimJava simulation package with animation facilities. The available applets range from a simple pipelined DLX processor to a dynamic-scheduling processor based on the Tomasulo algorithm. [15, 16, 17, 18]

SPIM[7] runs MIPS32 assembler programs and simulates there functional behaviour while offering the possibility to debug the program and to manipulate register contents. [19, App. A]

2.2 Distinction and benefits

The majority of the reviewed simulators are designed for high speed execution instead of illustrating step-by-step simulations. Each of them offers different features like debuggers, support for multiple instruction sets or implementation of state-of-the-art processor techniques. Anyhow, to me most of them offer far too much additional functionality or just can not suit the educational goal to understand how exactly a dynamic scheduled processor operates and the way it benefits from its design.

SPIM, which supports singlestep execution, continuous monitoring of the current processor state and the possibility to alter the register values is a good approach to the

[5]MARS: MIPS Assembler and Runtime Simulator
[6]HASE: Hierarchical computer Architecture design and Simulation Environment
[7]SPIM: palindrome of 'MIPS'

previous defined goals (section 1.2). However, it does not model a dynamic scheduled processor so it can not fulfill the intended purpose. Also it has to be installed before it can be used and in my opinion the design aims more at experienced users than computer architecture beginners.

JavaHASE on the other hand matches the intended purpose even more, since it does not require any installation, offers an applet for Tomasulo's dynamic scheduling algorithm and comes with an interface very easy to understand while maintaining the possibility to influence the simulation example (e.g. altering memory data, register data and latencies). Still it offers no possibility to adjust the model of the simulation (e.g. numbers of functional units etc.), the applets are deprecated and are no longer maintained and I regard the design as limited and the nomenclature of its components (especially those in the parameters dialog) as not self-explanatory enough.

The resulting simulator designed and implemented during this thesis should feature a step-by-step simulation of a dynamic scheduled processor without requiring the user to install additional software. In order to allow a comparison of the impact of different structural processor characterisitics (e.g. number of functional units, buffer sizes) on program execution times and the instrution flow and to provide many varying examples, the simulation should be customizable to some degree but keep the interface and the setup easy to understand.

Chapter 3

Fundamentals

The design of the resulting simulator of this thesis is based on some common computer architecture concepts described below.

3.1 MIPS

MIPS32[1] ISA[2] is classified as a RISC[3] ISA. Its instructions share an uniform length of 32 bits and they are constructed in accordance with just a few different patterns. MIPS, developed since 1981, was one of the first RISC architectures used. It plays an essential role in embedded systems today and is also used in some recent super computers. [20] Therefore, it is often studied at universities as an example computer architecture. [21]

3.2 Tomasulo Algorithm

The Tomasulo Algorithm describes a scheme to enable out-of-order execution and to reduce the number of read and write hazards. It minimizes RAW[4] hazards by forwarding the operands for instructions as soon as they are available. To minimize WAR[5]

[1]MIPS (originally): Microprocessor without Interlocked Pipeline Stages
[2]ISA: instruction set architecture
[3]RISC: reduced instruction set computer
[4]RAW: read after write
[5]WAR: write after read

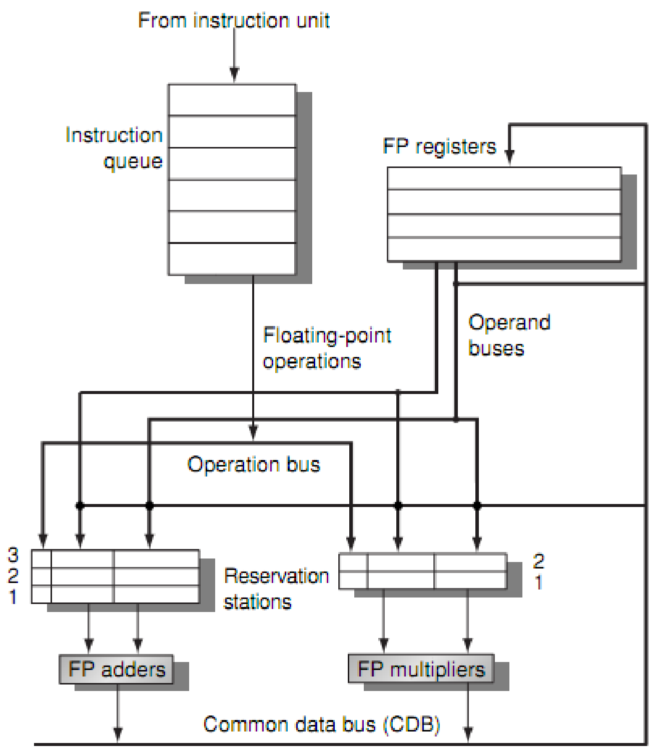


Figure 3.1: The basic structure of a MIPS floating-point unit using Tomasulo's algorithm [1, p. 94 figure 2.9, modified]

and WAW[6] conflicts the approach uses register renaming to preserve temporary register values.

The algorithm describes the usage of different units connected via the so called common data bus (CDB). For each functional unit or groups of equal function units a so called reservation station stores the next instructions to execute, their operands value and the state of availability of these values. Each instruction passes three steps. In the first step called 'issue' the instruction is fetched from the instruction queue and inserted into a matching reservation station if it is empty. While insertion the algorithm checks if operands are available and if not keeps track of the functional unit which will produce the required operand(s). The next step called 'execute' describes the updating of the reservation station entries and the assignment of an instruction towards a functional unit. The last step is called 'write result'. In this step, available results are fetched from the functional units and written to the registers, reservation stations and memory via the CDB. [1, p. 92 et seq.]

The out-of-order execution with the Tomasulo algorithm provides a better performance

[6]WAW: write after write

because independent instructions can execute instead of waiting for previous stalled instructions and the cycle frequency can be increased due the seperation of different execution units.

An example implementation of the Tomasulo algorithm is illustrated in figure 3.1. The figure shows a MIPS unit for floating point operations with disjunct reservation stations for addition and multiplication.

3.3 Hardware-Based Speculation

A bottleneck of Tomasulos Algorithm are branch instructions because they require the instruction fetching to pause until they leave their execution unit and the next instruction can be fetched according to their result. To eliminate this bottleneck, modern processors use hardware-based speculation. They predict the outcome of the branch evaluation and act accordingly. If their speculation was wrong all speculative fetched and executed instructions must be removed, the processor must be reset and the instruction fetch must restart fetching at the correct branch target. To achieve this a processor remembers the prediction and compares it to the branch result in the Writeback stage. If prediction and result do not match the PC[7] is set to the branch target and the processor is reset to its initial state. This procedure in combination with the Tomasulo algorithm guarantees that no speculative wrong-fetched instruction result is written to the register or memory.

One approach to achieve better predictions is to maintain a so called ‘branch history table’. It keeps track of previous predictions and their results and is updated if the outcome is different from the prediction. Because of that the processor may make better predictions for loops in most programs.

[7]PC: Program Counter - address of executed instruction

Chapter 4

Design Decisions

4.1 Assumptions

As most computers are implemented that way, for the purpose of this thesis I assume that MIPS uses two's complement for signed values and the General Purpose Register 0 is invariably set to zero. Therefore, I will implement it that way too.

To simplify the implementation, I assume the flushing of the processor while it is reset due to a branch misprediction is completed in the same cycle as it is initialized and therefore does not generate additional performance costs.

4.2 Restrictions

4.2.1 Instruction Set

The simulator is going to use a reduced and simplified MIPS instruction set named Little MIPS which only implements a subset of the MIPS instructions. Little MIPS like MIPS uses a 6bit opcode but no additional function field (or similar) and therefore can implement 64 operations. As the simulator will have a variable register size, the length of the register address will vary too and as a consequence the instruction encoding depends on the register size. A instruction word has the size of 6bit plus 3 times the length of a register address. Because of that, immediate operands have the same length as a register address and the target of a jump instruction has the size of 3 times the register length. For example a simulated processor using 64 registers

Encod.	Int	Opcode	signed	MIPS opc-fct	Format	Wr	Rd	Desc
000000	0	TERM	-	-	-	-	-	Terminate Processor
000010	2	J	-	id	off3	-	-	Jump
000100	4	BEQ	y	id	rs,rt,off	-	rs,rt	Branch iff equal
000101	5	BNE	y	id	rs,rt,off	-	rs,rt	Branch iff not equal
000110	6	BLEZ	y	id	rs,off	-	rs	Branch iff <= 0
000111	7	BGTZ	y	id	rs,off	-	rs	Branch iff > 0
001000	8	ADDI	y	id	rt,rs,im	rt	rs	Add Immediate
001001	9	ADDIU	y	id	rt,rs,im	rt	rs	like ADDI
010000	16	ADD	y	000000-100000	rd,rs,rt	rd	rs,rt	Addition
010001	17	ADDU	y	000000-100001	rd,rs,rt	rd	rs,rt	like ADD
010010	18	SUB	y	000000-100010	rd,rs,rt	rd	rs,rt	Subtraction
010011	19	SUBU	y	000000-100011	rd,rs,rt	rd	rs,rt	like SUB
010100	20	MULT	y	000000-011000	rs,rt	hi,lo	rs,rt	Multiplication
010101	21	MULTU	n	000000-011001	rs,rt	hi,lo	rs,rt	Multiplication unsigned
010110	22	DIV	y	000000-011010	rs,rt	hi,lo	rs,rt	Division
010111	23	DIVU	n	000000-011011	rs,rt	hi,lo	rs,rt	Division unsigned
011000	24	MUL	y	011100-000010	rd,rs,rt	rd	rs,rt	Full multiplication
011010	26	SLT	y	000000-101010	rd,rs,rt	rd	rs,rt	Set on Less Than
011011	27	SLTU	n	000000-101011	rd,rs,rt	rd	rs,rt	Set on Less Than unsigned
011100	28	AND	-	000000-100100	rd,rs,rt	rd	rs,rt	Bitwise AND
011101	29	OR	-	000000-100101	rd,rs,rt	rd	rs,rt	Bitwise OR
011110	30	XOR	-	000000-100110	rd,rs,rt	rd	rs,rt	Bitwise XOR
011111	31	NOR	-	000000-100111	rd,rs,rt	rd	rs,rt	Bitwise NOR
100011	35	LW	-	id	rt,off(rs)	rt	rs,mem	Load Word
101011	43	SW	-	id	rt,off(rs)	mem	rs,rt	Save Word
110000	48	MFHI	-	000000-010000	rd	rd	hi	Move From High
110010	50	MFLO	-	000000-010010	rd	rd	lo	Move From Low

Table 4.1: The encoding of Little MIPS instructions, their relation to MIPS and their IO behaviour (In MIPS ADDU (ADDIU,SUBU) differs from ADD (ADDI,SUB) only in exception handling: ADD (ADDI,SUB) raises overflow exceptions, ADDU (ADDIU,SUBU) does not.)

needs 6bit for each register address. In this case a signed immediate operand would have a value out of {-32,...,31} and a jump target would have a maximum value of 524287. The MIPS functions only identified via opcode have the same encoding in Little MIPS, other functions are assigned to unused opcodes. [22]

The syntax of Little MIPS instructions is the following:

R-Type: opc(6), rs(r), rt(r), rd(r)
I-type: opc(6), rs(r), rt(r), immediate(r)
J-type: opc(6), target(3r)
r : length of registeraddress

Table 4.1 shows the encoding of the Little MIPS instructions and their according MIPS encoding. The table also lists which inputs an instruction requires and which outputs it modifies. [22]

4.2.2 Memory

To prevent possible dependencies between write instructions and the instruction fetch stage, the processor is implemented as a Harvard architecture to completely seperate instruction memory and data memory. If the simulator is going to allow the simulation of multi processors no additional problems beside datamemory cache coherence need to be considered.

I assume that a memory access unit can not be interrupted and implement it that way. This means that an initiated load will return the data whether they are still needed or not.

4.3 Other

As the simulator design should not cover too much details I decided not to implement processor exceptions (e.g. integer addition overflow) as they are not that important to understand the operating principles of dynamic scheduling.

Other decisions are motivated by the available data structures and types of the programming language. Arrays with (default) int(32) indices limit the potential size of register and memory as they are implemented as arrays. The usage of int(32) to hold an encoded instruction limits the register address length to a maximum of 8 bits (6 +

3 * 8 = 30). As the datatype long(64) is used to hold the memory data and register values their maximum length is 64 bit which limits the adressable memory as well.

The branch prediction uses a branch history table. It uses the modulo operation to assign memory entries to branch history entries. The branch history table features a variable history depth, which determines when the prediction should be updated, and variable table size.

4.4 Hazards

As no parallel programming language is used the data dependencies between the processor stages have to be concidered during implementation. As each stage produces data required one clock cycle later by the next stage they are all linear independend. Due to other dependencies, e.g. jump (in ID) and branch (in WB) instructions manipulating the PC used in the IF stage, the dependencies form a cycle. Figure 4.1 shows some of the dependencies which have to be considered.

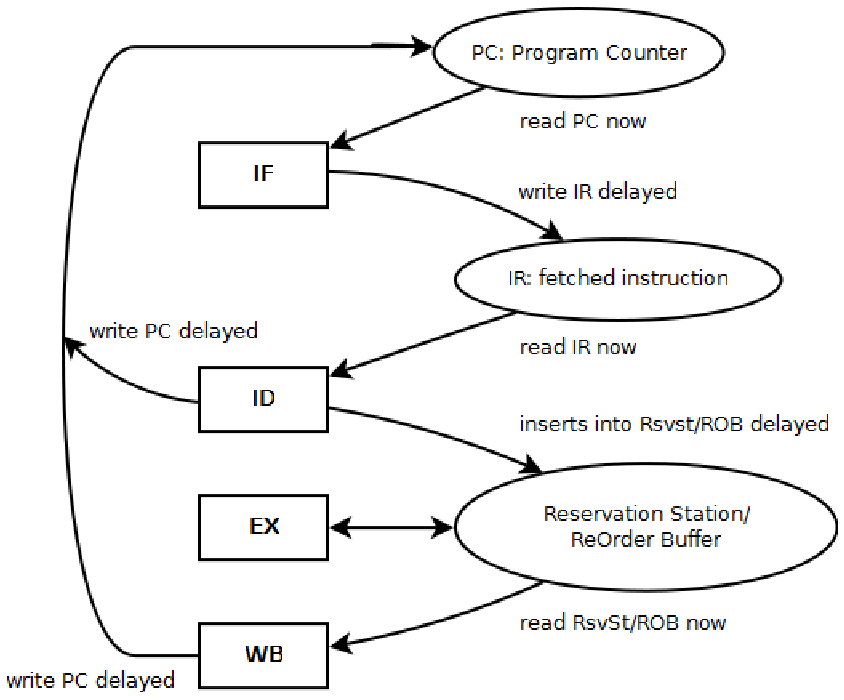


Figure 4.1: Data dependency graph for the process stages, revealing a cyclic dependency

Because load instructions do not calculate their result during execute but their memory address, a memory controller must fetch the result value before the reorder buffer entry can be considered as ready.

As described in section 4.2.2 memory access units can not be interrupted. As a result, an instruction fetch memory access can not be interrupted even if the processor is flushed and the PC is reset. It has to be dealt with the possibility that unwanted data returns from a memory controller.

Chapter 5

Implementation

5.1 Programming language

The simulator is implemented as a webservice and therefore ASP.NET[1] with the programming language C# is used. The interface is visualized via HTML[2] and controlled by JavaScript.

Java was rejected because it would require the installation of a Java Runtime Environment on the one hand and its performance disadvantage towards C#.

Another candidate PHP is not used due to its weak type system.

5.2 Objects

To offer some degree of customizability the simulator uses classes and objects to represent certain units of the processor.

[1]ASP: active server pages (Microsoft)
[2]HTML: Hypertext Markup Language

5.2.1 Processor

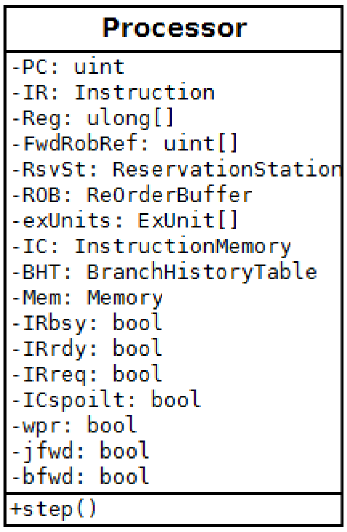


Figure 5.1: Processor class diagram

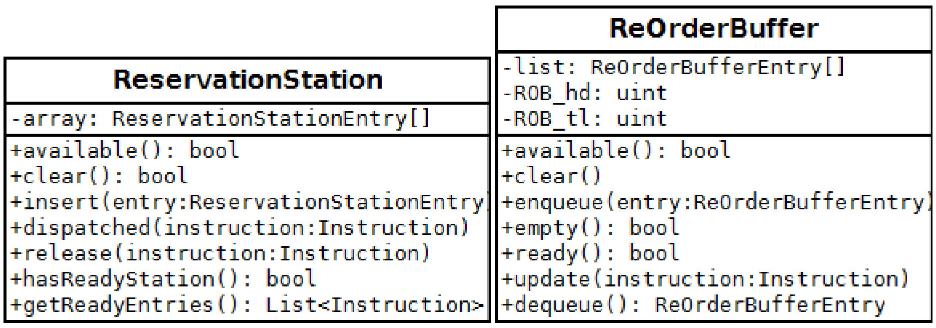
The processor class represents a processor and provides the primary object used for the simulation. As figure 5.1 shows, it allocates variables for the processor states, interrupt handling (due to jump or branch instructions updating the PC), registers and the different architectural units. The public method step() represents the actual simulation of a processor cylce.

As mentioned in the previous chapter the registers are implemented as 64bit (unsigned) integer array. The FwdRobRef array is used for register renaming. FwdRobRef[i] determines if Register Reg[i] is up to date or which ROB[3] entry will produce the required value for it. The per processor instruction memory is seperated from the main memory passed as reference to the processor.

5.2.2 Reservation Station

The reservation station consists of an internal array of reservation station entries and some functions for interaction listed in the class diagram in 5.2a; It offers functions to check if a slot is available for insertion and the insertion itself, others to check for the availability of executable instructions, to dispatch them to execution units and to release them if they leave the execution units and another one to clear the reservation station in case of a processor flush/reset.

[3]ROB: ReOrder Buffer


ReservationStation
-array: ReservationStationEntry[]
+available(): bool
+clear(): bool
+insert(entry:ReservationStationEntry
+dispatched(instruction:Instruction)
+release(instruction:Instruction)
+hasReadyStation(): bool
+getReadyEntries(): List<Instruction>
ReOrderBuffer
-list: ReOrderBufferEntry[]
-ROB_hd: uint
-ROB_tl: uint
+available(): bool
+clear()
+enqueue(entry:ReOrderBufferEntry)
+empty(): bool
+ready(): bool
+update(instruction:Instruction)
+dequeue(): ReOrderBufferEntry

(a) ReservationStation (b) ReOrderBuffer

Figure 5.2: Class diagrams

5.2.3 ROB

In contrast to the reservation station the ROB is implemented as a list in order to support a FIFO[4] data flow. Because the reorder buffer has a fixed size during simulation an array with head and tail indices is used instead of an dynamic list type. The functionality provided by the ROB is shown in figure 5.2b. Just like the reservation station it provides a function to check for an available slot, another to clear the reorder buffer in case of a processor flush/reset and others to insert and remove entries from the reorder buffer although this only works in a FIFO manner. Additionally it offers functions to check for emptiness of the reorder buffer or to check for the availability of the first instruction for the WB stage.

5.2.4 Memory

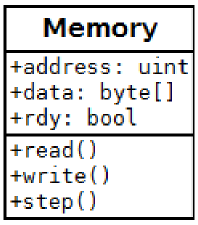


Figure 5.3: Memory class diagram

As instruction memory and main memory are implemented in a similiar way only the latter one will be introduced. They implement an interface which allows the interchangeability with cached memory architectures which could be implemented in

[4]FIFO: First in - First out

further work. The interface, as shown in figure 5.3, offers variables to set the target address, check for and return the result and functions to initiate a load or store query. Further it provides a function called every processor cylcle which simulates a cycle of the memory and at first considers memory latencies but could implement multilevel cache hierachies with different latencies and their hit/miss behaviour in a later release as well.

5.3 Cycle Simulation

The different stages described by the Tomasulo Algorithm are realized as functions.

Because each stage produces data required by the following stage in the next clock cycle, they would overwrite the data required by the following stages in the actual clock cylce. To solve these dependencies, each simulated clock cycle the stages are executed sequentially reversed: first WB, then EX followed by ID and at last IF. As mentioned in 4.4 more dependencies have to be considered.

5.3.1 IF

If the processor is not in a terminated state IF fetchs a new instruction from the instruction memory if ID requests a new instruction (IRreq == true) and IF is not busy with already fetching an instruction. In case of the instruction memory not being ready yet IRbsy indicates that no new instruction request needs to be issued the next clock cycle. If the processor has been reset then ICspoilt indicates that the result of an ongoing instruction memory request must be discarded. IRrdy and IRreq represent signals, send and read in the same cycle and reset to 0 in the next one, of a physical processor implementation. IF sends IRrdy iff IR holds a valid requested instruction provided by the instruction memory. Because IF is simulated at last and sets IRrdy if IR holds a valid instruction, it is equivalent to IF sending IRrdy the next clock cycle when IR holds a valid instruction or just has changed to the new instruction. Other dependencies like the wrong branch prediction interrupt or jump and branch forwarding which affects IF in the same clock cycle they occur are resolved by IF beeing the last executed stage as well.

5.3.2 ID

ID checks whether an instruction is ready (IRrdy), decodes it and inserts it into the reservation station and reorder buffer. In case of a jump instruction jfwd is set and PC updated to the jump target. If the instruction is a branch the branch is predicted according to the matching branch history table entry. While insertion into the reservation station ID checks the forward reference table if the operands are available or which reservation stations provides them. Write instructions update the forward reference table to their reservation station id as their result is the operand for further instructions reading the same register. If there are free slots in the reservation station and reorder buffer ID sets IRreq to request a new instruction fetch.

5.3.3 EX

Stage EX is split into two subfunctions: Dispatch and Fetch.

Dispatch checks for instructions ready for execution (in FIFO order) and dispatchs them to available fitting execution units if possible. Each execution unit has its own list of supported instructions and therefore many different settings can be simultated.

Fetch observes the execution units for finished instructions, writes their results to the reorder buffer and updates instructions waiting for these operands in the reservation station. However load and store instructions must be handled differently as they only calculated the target address and still need to be passed to the memory controller. To address this issue reorder buffer entries with load/store instructions are not considered ready (for writeback) until they additionally were processed by the memory controller.

5.3.4 WB

The last stage WB checks whether the reorder buffer has an instruction ready for writeback and performs the according tasks. An instruction is ready for writeback if the execution unit has calculated the result, it is the topmost entry of the reorder buffer and if it is a load or store instruction and its memory request is finished. If the affected instruction is a branch instruction and the execution revealed the branch was incorrectly taken then the processor is flushed, that means all parts of the processor except the PC are reset to their initial state. The PC is set to the corrected branch target and the processor starts fetching instructions again.

Chapter 6

User Interaction

6.1 GUI

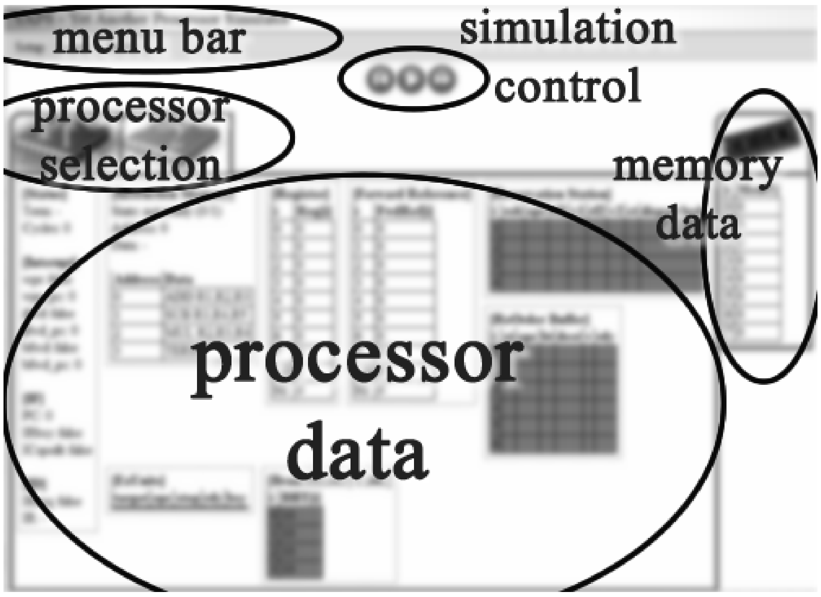


Figure 6.1: Graphical user interface overview

As shown in figure 6.1 and figure 6.2 the GUI[1] is seperated into a titlebar, a menubar, the simulation control, processor selection, processor data and memory data. To maintain some degree of clarity only one processor (of a multiprocessor simulation) is displayed at the same time. All elements of the GUI are clearly seperated and easy to

[1]GUI: Graphical User Interface

identify. Graphics and colors are used to offer a more comfortable and easy to understandable interface than a text-only approach would offer.

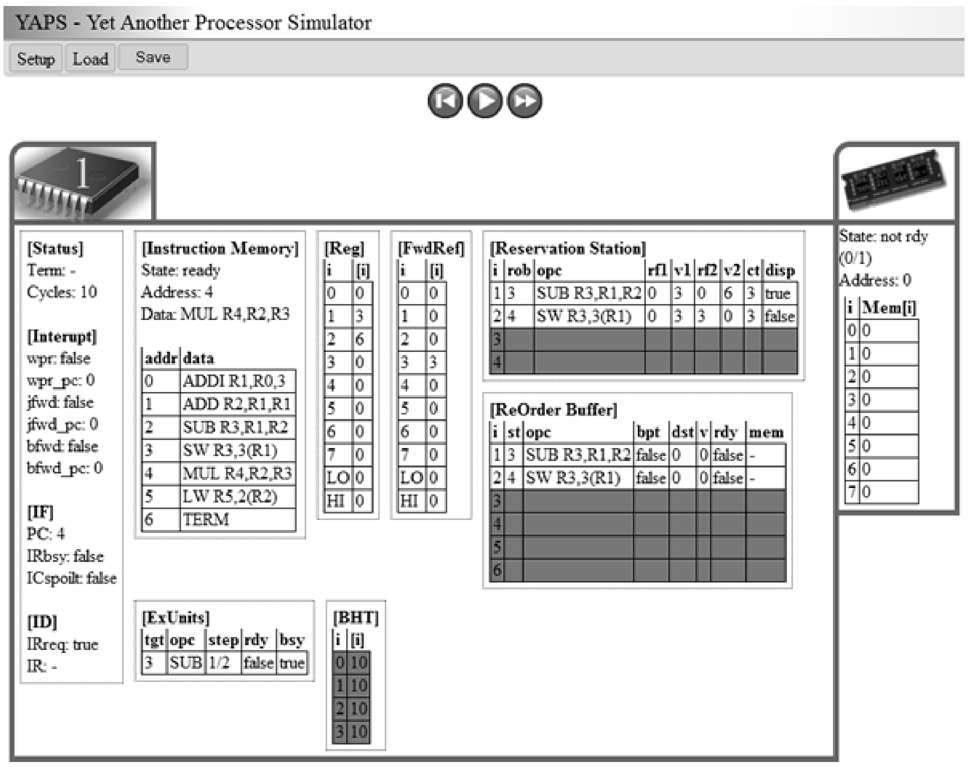
6.2 How To

To use the simulator for the first time a user has to setup a simulation configuration. To do this he starts the setup via the menu entry “Setup”. In the first step he configures the global properties as the numbers of processors to simulate, the word length of data values (memory and register width) and the size of the main memory in number of words. The second step is repeated for each processor to configure. The user has to configure the number of the processor registers, the sizes of the branch history table, the reservation station and the reorder buffer. Also he has to configure the execution units to use by adding as much as he want by selecting the instructions it should process and the latency it has. Finally the user has to insert the instructions to execute in the instruction memory. The size of the instruction memory is dynamically calculated by the number of inserted instructions. To help the user and speed up the configuration process each field is preset with a common default value, a two-cycle omnipotent execution unit is preconfigured and to avoid unintended never-ending programs a ‘TERM’ instruction is autoappended. After the user has setup all processors he finishs the setup and initializes the simulator. Figure 6.2 shows the initiated simulator with an example setup.

To reuse a just setup simulator configuration at a later date the user can save the configuration as a file and reload it when needed.

After initialization the simualator control, the processor selection, processor data and memory data are displayed. The processor selection switches between the different simulated processor and updates the displayed data. The processor data shows the different parts of the simulated processor and important status flags. To maintain a small interface most of the identifiers are abbreviated but reveal their full name when the mouse hovers over them.

The simulation is controlled via three controls which offer the possibilities to reset the simulation to the initial state, to simulate one cylce and to fast-forward by simulating 10 cycles



[Status]
Term: -
Cycles: 10

[Interupt]
wpr: false
wpr_pc: 0
jfwd: false
jfwd_pc: 0
bfwd: false
bfwd_pc: 0

[IF]
PC: 4
IRbsy: false
ICspoilt: false

[ID]
IRreq: true
IR: -

[Instruction Memory]
State: ready
Address: 4
Data: MUL R4,R2,R3

addr	data
0	ADDI R1,R0,3
1	ADD R2,R1,R1
2	SUB R3,R1,R2
3	SW R3,3(R1)
4	MUL R4,R2,R3
5	LW R5,2(R2)
6	TERM

[Reg]

i	[i]
0	0
1	3
2	6
3	0
4	0
5	0
6	0
7	0
LO	0
HI	0

[FwdRef]

i	[i]
0	0
1	0
2	0
3	3
4	0
5	0
6	0
7	0
LO	0
HI	0

[Reservation Station]

i	rob	opc	rf1	v1	rf2	v2	ct	disp
1	3	SUB R3,R1,R2	0	3	0	6	3	true
2	4	SW R3,3(R1)	0	3	3	0	3	false
3								
4								

[ReOrder Buffer]

i	st	opc	bpt	dst	v	rdy	mem
1	3	SUB R3,R1,R2	false	0	0	false	-
2	4	SW R3,3(R1)	false	0	0	false	-
3							
4							
5							
6							

[ExUnits]

tgt	opc	step	rdy	bsy
3	SUB	1/2	false	true

[BHT]

i	[i]
0	10
1	10
2	10
3	10

State: not rdy
(0/1)
Address: 0

i	Mem[i]
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0

Figure 6.2: Graphical user interface with inititated example setup

6.3 Restrictions

As described in chapter 4 the simulator has some restrictions. During the setup incorrect or unsupported values are noticed and the user is prompted to change them. For example the data word length (in bytes) only allows values between 1 and 8.

Chapter 7

Result

7.1 Validation

To validate the resulting simulator an empirical approach using a reference simulator is used to prove that the simulator produces the same results as a linear simulator. The reference simulator is a one-cylce simulator, executing one instruction every simulation step. Its implementation is as simple as possible to offer the required reliability to serve as a reference simulator. To validate the simulator randomly generated sequences of instructions and operands are passed to both simulators. Their register and memory values are compared for each instruction after it left both simulators. To avoid infinite test runtime each random program was tested for a maximum runtime of 1000 passed instructions. During validation some errors have been discovered and elminated. In the end the test program discovered no more differences after the execution of around one million random programs with each between 1 and 100 instructions. The simulators generated the same results in a significant number of test cases, hence we can consider the resulting simulator as reliable too.

On the other hand the implementation of the instructions themselves is verified by testing with relevant test cases. Test cases for signed operations cover tests with all combinations of positive and negative operands. All test cases produced the expected result and therefore the implementation is considered reliable.

7.2 Implementation comment

The implementation of the simulator draft revealed some difficulties which were not obvious to me from the basics I have learned in class before. For example results which are already in the reorder buffer must be forwarded to the reservation station as long as they are in the reorder buffer and the load/store instructions required more attention because their reorder behaviour differs from other instructions.

Some details initially planned could not be implemented due to their unexpected complexity or limited available time. It was planned that the resulting simulator should support a cached memory architecture and no seperate memories for per-processor instruction memories and a main memory.

The implementation supports many customizable usage scenarios and due to a object orientated design can be modified or extended without much additional effort. The interface is easy to understand and only shows the essential data, but further work in the interface would be recommended aided by user feedback.

7.3 Suggestions for further work

As mentioned before a cached memory architecture could be developed for the usage with the simulator. This would offer the possibility to demonstrate the effects of multiple processors writing to one memory with and without a cache coherence protocol.

Another optional extension could be the support of exceptions as for example raised by MIPS32 instruction ADD because of an integer overflow.

The support of other processor types, such as a pipelined or a one-cycle implementation, to offer a direct comparison between these processor types, would be another valuable addition.

The GUI could be evaluated with groups of interested students and improved according to the results. Possible improvements may cover i.e. the addition of a tutorial or different detail/context views for different exercises.

7.4 Conclusion

The resulting simulator fits most of the aims specified at the beginning of this document. It shows the working principles of a dynamic superscalar processor and is very customizable. The interface is very simple and easy to understand. The simulator itself requires an ASP.Net Server to be started, but actual users only require a modern webbrowser with enabled JavaScript. According to my opinion the simulator could help to improve the understanding of the operating principles of dynamic scheduling and speculative execution for interested students and therefore should be used in education.

Bibliography

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 2006.

[2] Donald E. Knuth. *The art of computer programming*, volume 1. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 3 edition, 1997. fundamental algorithms.

[3] Tilo Gockel. *Form der wissenschaftlichen Ausarbeitung*. Springer-Verlag, 2008.

[4] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35:59–67, 2002.

[5] SimpleScalar LLC. Simplescalar overview. http://www.simplescalar.com/overview.html. (2011.02.10, http://www.webcitation.org/5wOSsM9im).

[6] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator. http://sesc.sourceforge.net. (2011.02.10, http://www.webcitation.org/5wOT9XvCS).

[7] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33:92–99, November 2005.

[8] Multifacet GEMS Development Team. Multifacet gems. http://www.cs.wisc.edu/gems/. (2011.02.10, http://www.webcitation.org/5wOUSPhDQ).

[9] The m5 simulator system. http://www.m5sim.org. (2011.02.10, http://www.webcitation.org/5wOTbIv68).

[10] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, 2006.

[11] Matt T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS 2007*, 2007.

[12] Matt T. Yourst. Ptlsim x86-64 cycle accurate processor simulation design infrastructure. http://www.ptlsim.org. (2011.02.10, http://www.webcitation.org/5wOi5Nlue).

[13] Hui Zeng, Matt Yourst, Kanad Ghose, and Dmitry Ponomarev. Mptlsim: a cycle-accurate, full-system simulator for x86-64 multicore architectures with coherent caches. *SIGARCH Comput. Archit. News*, 37:2–9, July 2009.

[14] Kenneth Vollmar and Pete Sanderson. Mars: an education-oriented mips assembly language simulator. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, volume 38 of *SIGCSE '06*, pages 239–243, New York, NY, USA, 2006. ACM.

[15] R. N. Ibbett, P. E. Heywood, and F. W. Howell. HASE: A Flexible Toolset for Computer Architects. *The Computer Journal*, 38(10):755–764, 1995.

[16] R. Mcnab and F. W. Howell. Using java for discrete event simulation. In *in Proc. Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW), Univ. of Edinburgh*, pages 219–228, 1996.

[17] University of Edinburgh Institute for Computing Systems Architecture, School of Informatics. Javahase. http://www.icsa.inf.ed.ac.uk/research/groups/hase/javahase/. (2011.02.13, http://www.webcitation.org/5wSzSXSmd).

[18] University of Edinburgh Institute for Computing Systems Architecture, School of Informatics. simjava. http://www.icsa.inf.ed.ac.uk/research/groups/hase/simjava/. (2011.02.13, http://www.webcitation.org/5wSyjk6IH).

[19] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 3rd edition, 2005.

[20] TOP500.Org. The sicortex sc series. http://www.top500.org/2007_overview_recent_supercomputers/sicortex_sc_series. (2011.02.15, http://www.webcitation.org/5wWkGALIu).

[21] MIPS Technologies. *MIPS32 Architecture for Programmers Volume I: Introduction to the MIPS32 Architecture*, 2.60 edition. http://www.mips.com/products/product-materials/processor/mips-architecture/.

[22] MIPS Technologies. *MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set*, 2.62 edition. http://www.mips.com/products/product-materials/processor/mips-architecture/.