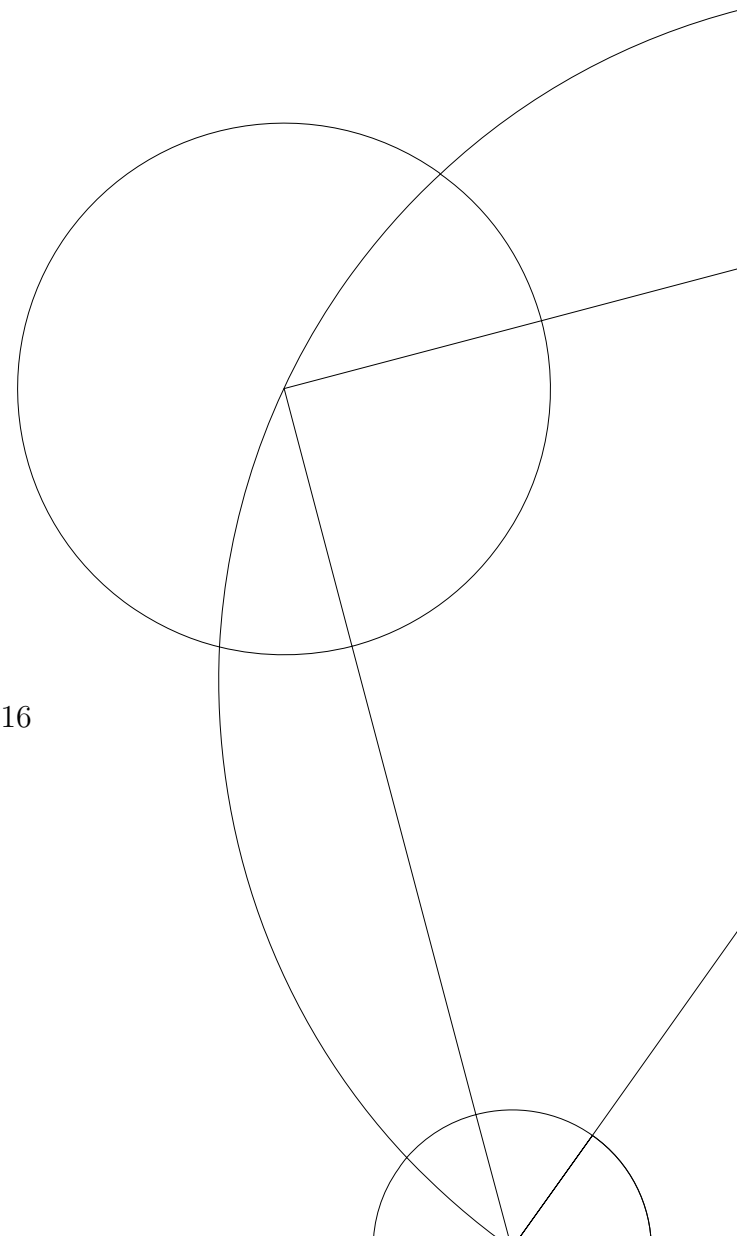
UNIVERSITY OF COPENHAGEN

Simulation of a MIPS machine

Jan Mezník
PZJ895



April 3, 2016

Abstract

TODO

Contents

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 A simulator 1

2 CPU Architectures **1**
- 2.1 Instruction Set Architectures 1
- 2.2 MIPS Architecture 2

3 MIPS Core Processing Unit **2**
- 3.1 Registers 3
  - 3.1.1 General Purpose Registers 3
  - 3.1.2 Special Registers . . . 3
  - 3.1.3 Co-processor 0 registers 3
- 3.2 Instructions 3
- 3.3 Arithmetic 4
- 3.4 Load and Store 4
- 3.5 Jumping and Branching . . . 4
- 3.6 Interrupts 4

4 Pipeline **4**

5 TLB **4**

6 MMU **4**

7 User and Kernel mode **4**

8 SMP **4**

9 Tests **4**

10 Performance **4**

11 Conclusion **4**

Appendices **5**

Introduction

This report describes the development of a MIPS simulator, intended to support the operating system KUDOS. The simulator will be written in C, and will support the most important processor required to run KUDOS, such as the translation lookaside buffer (TLB), memory management unit (MMU), user and kernel CPU modes, multiple cores (SMP), and I/O device emulation.

Motivation

KUDOS is a small operating system skeleton intended to be used by students attending operating system project courses at university of Copenhagen. It is used to explore operating system concepts by extending and improving on existing system. Initially, KUDOS targetsthe MIPS32 architecture, which leverages on the advantages of a reduced instruction set computing - RISC. To ease the development and debugging of KUDOS, it is desireable to run the OS in a simulated machine. This enables the students and other developers to better inspect the state of the machine while executing, as well as making up for the difference in the hardware of the host machine.

A simulator

Simulation is the act of imitating the operation of an existing system. In our case, we will be imitating, or rather, simulating a MIPS32 machine running KUDOS.
Unlike emulating a system, the simulator will execute every instruction comparably to as how a hardware machine might do. This involves modelling the internal state of the machine, which accuratelly reflects the target it is simulating. This allows the developer to not only see how a program behaves, but also observe properties of the machine which are also present in the original target.

CPU Architectures

At the heart of every computer lies the Central Processing Unit (CPU), which is an electronic circuit that carries out the basic arithmethic- and logic calculation as well as process and redirect input and output to other devices in the computer, using the shared busses.

Most modern CPUs are contained on a very small, yet packed intergrated circuit chip, which can also house memory caches, multiple cores, and other processing units.

The functionality of all processors is fundamentally the same. The processor executes some primitive operation by fetching an instruction in the form of binary signals, act upon the instruction and store the result in either one of the its registers, or in the main memory.

A single instruction does very little, but a collection of instructions make up a program. In the very early computing days, computers were programmed in an assembly language, which is simply human-readable instruction code. As the computers grew more powerful, more complex and much faster, larger programs could be executed. Because it is hard and time-consuming to write programs using only the assembly language, compilers are used to remove this complexity. A compiler takes a high-level language, such as C, C++ or Java, and creates the corresponding assembly language program for the specific architecture, containing the instructions. This assembly language file is in turn assembled or translated to binary, that the particular CPU can understand.

Besides hiding the complexity of the underlying architecture away from the programmer, it can usually also compile programs to multiple architectures as well as optimising the code to run faster.

Instruction Set Architectures

The instructions supported by a particular processor is determined by the Instruction Set Architecture (ISA), which is the specification of how the CPU works. An ISA determines the instructions supported,

the registers available, memory architecture, addressing modes as well as handling of interrupts.

There exists many different types of ISAs, with both their advantages and disadvantages. For example, some architectures have a very few instructions and registers, which is very practical for small embedded devices, whereas large servers might make use of a large array of registers for complex computations.

Besides the current use of an architecture, designers must also take into account its future uses and applications. As the world of computation is ever growing and evolving at exponential rates, the architectures must be up to the challenge of future computing. Introducing a completely new architecture to the market is very troublesome, and causes a list of problems. One of the main issues is that old software written for older architectures will no longer work, and it requires to be either recompiled, rewritten, or even emulated. One such example is the Intel Itanium (IA-64) architecture, which had a very bad marked reception due to its lack of backwards compatibility with the x86 architecture. The emulation of the architecture on IA-64 yielded suboptimal performance and ultimately lost to the AMD x86_64, which in turn was compatible. [1]

Indeed, there are a lot of factors to take into account when designing a new architecture, and every decision has big implications on the future of the whole ISA.

MIPS Architecture

The MIPS architecture (acronym for Microprocessor without Interlocked Pipeline Stages) was first created in the early 1980s. [4] MIPS is a reduced instruction set architecture (RISC), developed by MIPS technologies, to bring new levels of performance and efficiency into the world of processing units. As an RISC architecture, MIPS aims to implement only the most essential instructions, so that they in return can get highly optimised. This is the based on the RISC philosophy, that by implementing only the most common instructions, the architects can simplify the design and speed up the crucial parts of the instructions. This enables the processor to execute programs faster, but also removes a lot of complexity of implementing large programs.

In contrast to RISC, complex instruction set architecture (CISC) aims to reduce the number of instructions needed to execute a program by implementing instructions packed with functionality. This means that a single instruction in CISC can execute several operations at once, such as loading from memory, arithmetic and storing. While complex programs indeed execute faster on a CISC architecture, the burden of implementing efficient and maintainable code and compilers can outweigh its advantages. [3]

Besides the inspiration from RISC, MIPS has added its own design principles, which are honored and used to question every change, implementation, or design. These are [2]:

- *Design Principle 1:* Simplicity favors regularity.
- *Design Principle 2:* Smaller is faster.
- *Design Principle 3:* Good design demands good compromises.
- *Design Principle 4:* Make common case fast.

These decisions withstood the trial by fire and proved, that honoring these principles yields good design, easing implementation as well as simplifying hardware.

MIPS Core Processing Unit

MIPS CPUs are pipelined, meaning that it implements a pipeline which enables it to execute different stages of multiple instructions at once. This gives the processor a higher throughput that would otherwise be possible at a given clock-rate. The processor has 31 general purpose (GP) registers, with additional registers per co-processing unit. Even the first models of the MIPS CPUs, such as the MIPS

R2000, had memory caches and a translation lookaside-buffer, which improves the speed of the processor by reducing the number of main memory lookups.

Registers

MIPS contains multiple types of registers. The most common and most used registers are the general-purpose registers (GP), which can be used for practically anything by the programmer. Special registers are registers implemented for cases where GP registers were either too small or otherwise unsuitable for the purpose.
For additional functionality, the MIPS co-processor 0 also has its own set of registers that, along with an operating system, bring many features to the system.

General Purpose Registers

In MIPS, there are 32 general-purpose registers, all 32 bit wide. Although they can all theoretically be used however the programmer or assembler wants[1], there are some conventions for the use of the registers.

Mnemonic[2]	#	Use
`$zero`	0	Constant Value 0
`$at`	1	Reserved Temporary
`$v0-$v1`	2-3	Function Results
`$a0-$a3`	4-7	Function Arguments
`$t0-$t7`	8-15	Temporaries
`$s0-$s7`	16-23	Saved Temporaries
`$t8-$t9`	24-25	Temporaries
`$k0-$k1`	26-27	Reserved for OS
`$gp`	28	Global Pointer
`$sp`	29	Stack Pointer
`$fp`	30	Frame Pointer
`$ra`	31	Return Address

Special Registers

The special registers in MIPS cannot directly be accessed from the program. Rather, they are modified by different instructions.

[1]Except the 0'th ($0) register, which can only hold the value 0.

[2]Textual mnemonic used in the assembly language

Name	#	Use
`HI`	-	Hi-word of 64bit value
`LO`	-	Lo-word of 64bit value
`$PC`	-	Program Counter

`HI` and `LO` registers are used to contain the result of a multiplication or division, which, using 2 32bit registers, can end with a 64bit result.
The PC register is pretty self-explanatory, as it simply points the current location in the program (or "counts" the instructions). On other architectures, this register is better known as the Instruction Pointer (IP).

Co-processor 0 registers

Registers in co-processor 0 are mainly used by the system, to provide additional features. The co-processor can have 32 registers, but only few of them are used consistently. Many of the empty registers are also defined by the manufacturer of the processor.

Name	#	Use
`index`	-	TLB entry index
`random`	-	TLB random access register
`entrylo`	-	Low order current TLB entry
`context`	-	Page-Table lookup addr.
`vaddr`	-	Virtual address of exceptions
`entryhi`	-	High order current TLB entry
`status`	-	Processor status
`cause`	-	Exception cause
`epc`	-	PC when exception occured

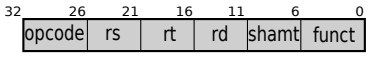
The `status` register is a bit-field of flags used to signal the current state of the processor. It is similar to the `EFLAGS` register on x86 architectures.

The rest of the registers will be discussed in depth in the SMP chapter.

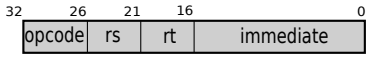
Instructions

Each instruction in MIPS is 32-bit long, aligned to word. This simplifies the instruction fetching, decoding, as well as disassembly of the program, for both the processor as well as the programmer.

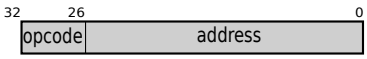
In MIPS, the instructions have 3 basic formats:



(a) R-Format



(b) I-Format



(c) J-Format

It is clear that, as they have common fields, mainly the opcode field, they are easily distinguishable.
The R-format instructions are mainly used when all the data being processed is located in the registers. That includes adding between registers, binary operations on values in registers as well as jumping to an address located in a register.

The I-format instructions can operate on both data from registers and immediate values encoded directly in the instruction (thus the 16-bit immediate field). I-format instruction share a lot of common operations with the R-format, where one of the operands is the immediate.

J-format instructions are used solely for jumping instructions, thus the large address field.

Arithmetic

Load and Store

Jumping and Branching

Interrupts

functionality. The floating-point operation

Pipeline

TLB

MMU

User and Kernel mode

SMP

Tests

Performance

Conclusion

References

[1] Johan De Gelas. Itanium - is there light at the end of the tunnel?, 2005. [Online; accessed 28-March-2016].

[2] David Patterson. *Computer organization and design : the hardware/software interface*. Morgan Kaufmann, Oxford Waltham, MA, USA, 2014.

[3] David A. Patterson and David R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8(6):25–33, October 1980.

[4] Imagination Technologies Group Plc. Mips microprocessors overview, 2014. [Online; accessed 29-March-2016].

Appendices