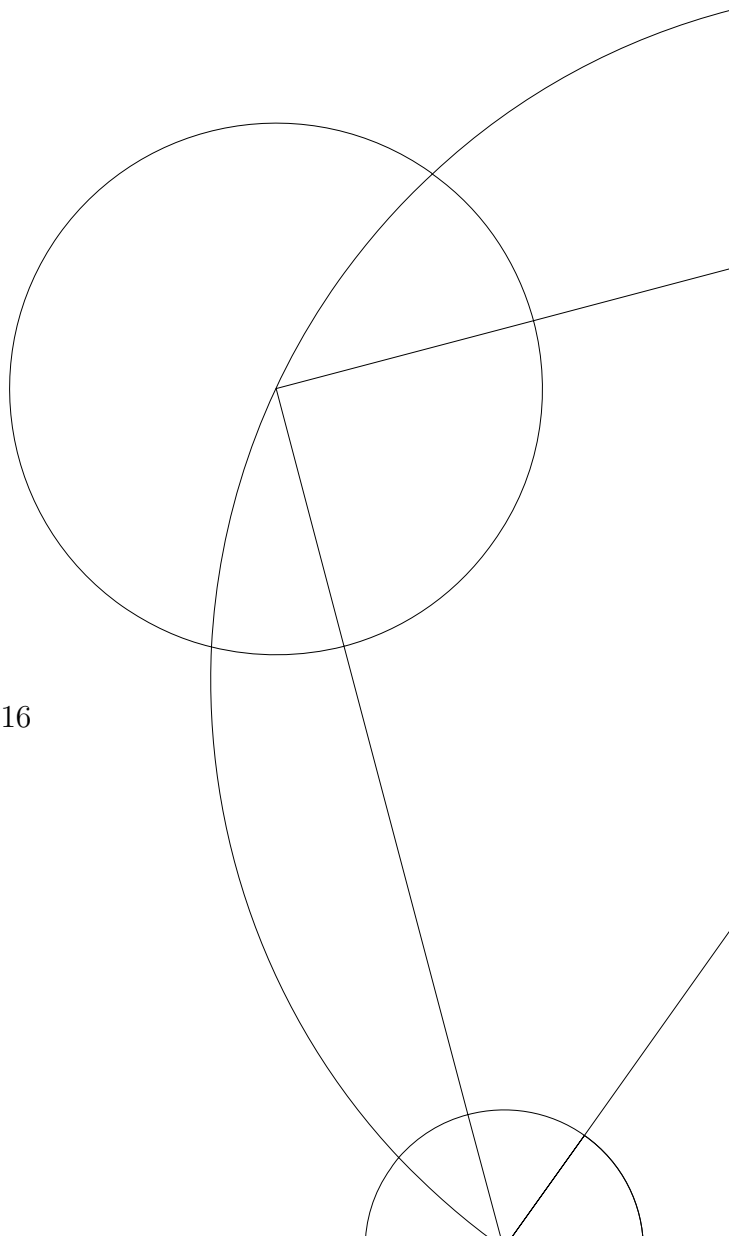
UNIVERSITY OF COPENHAGEN

Simulation of a MIPS machine

Jan Mezník
PZJ895



April 10, 2016

Abstract

TODO

Contents

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. [3]

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. [9] 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. [8]

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 [7]:

- *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:

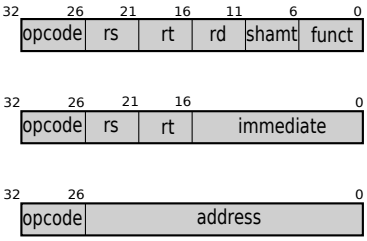


Figure 1: The 3 MIPS instruction formats.

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. As it only has 26 bits to address an 32 byte memory location, it shifts the whole value twice to the left, as to align the value in words. The upper 4 bits are retrieved from PC. In practise, this is enough to jump to any address in the program.

Arithmetic Logic Unit

Without any extension co-processing unit, the Arithmentic Logic Unit (ALU) in MIPS32 only supports operations on integers.
The ALU supports basic mathematical operations such as adding (`add`), subtracting (`sub`), as well as logical shifting to both left and right (`sll, srl`), which also can be used to division or multiplication by even numbers.

All bitwise logical operators `and, or, and nor` are implemented Using these, which additional missing logical operations can be created, such as `nand` and `not`.
Since both the operand and destination registers are 32 bit wide, an overflow in the result might occur - that is, the result is larger than what a 32 bit register can hold. In that situation, an exception is raised in the processor, and code to recover from this error is run [7]. This will be explained in later sections.

Load and Store

MIPS is a "load/store" architecture, where memory is only accessed by specific load and store instructions [2]. This design is a very common for RISC architectures, as it greatly simplifies the pipeline stages and clock timings. In contrast, CISC architectures have many instructions that can do operations on both memory and registers at the same time. For example, on the x86_64 architecture, the `MOVSW` instruction reads from a memory location pointed to by register SI, stores it in memory location DI, and at last, increments (or decrements[3]) both registers [4]. This adds additional stall and hazard logic to the processor, and makes it is hard for the CPU to determine how many clock-ticks the instruction will take.

MIPS32 uses `lw` for loading a word from the main memory into the register, and a `sw`, which stores the value from register into the specified memory location. In reality, MIPS32 also has `LH`, `LB` and their store counterparts `SH SB`, which operate on half-word and byte sized loads and stores. However, for performance reasons, the main memory always reads a word (4 bytes), and so, the desired size is computed in the CPU.

Jumping and Branching

To be turing complete, the processor needs to be able to do conditional jumps to other memory locations. This is done with

[3] This is determined by the direction flag, which determines whether the CPU reads memory from top to bottom or in reverse.

the jumping instructions: jump (`j`), jump-register (`jr`) and jump-and-link (`jal`). The conditional jumps are: branch-equal (`beq`) and branch-not-equal (`bne`).
On the bare-metal level of the processor, these instructions simply modify the value of the Program Counter register, which is otherwise inaccessible from assembly.

Interrupts

Interrupts is a special way to control what the CPU. It actively "interrupts" the CPU from its current job, and makes it execute a special function, specified by an interrupt number. There usually 3 types of interrupts [6]:

- Exceptions
  Exceptions occur in software, usually when an error has occurred that needs attention from the kernel. This is usually caused by reading from illegal memory addresses or when arithmetic overflow occurs.

- Hardware Interrupt
  Hardware interrupts are initiated from hardware devices, such as a mouse or a keyboard. When a user presses a key or moves the mouse, the hardware devices send a signal to the CPU that something has happened that needs attention from the kernel.

- System Call (syscall)
  Syscalls are usually used by programs, when they need attention from the kernel. An operating system and the underlying kernel will usually expose an interface with a whole set of functions, that the program can access by syscalls. This can be everything from reporting termination of a program to writing data to the disk.

The action that the CPU has to perform is determined by an interrupt vector table. For each interrupt vector, there is specific code to be executed. Because the interrupt vector table is limited in size, operating systems, such as Linux, use a single interrupt vector number 0x80. Additional arguments for further determination of the service are passed in service number, which is stored in the general purpose registers, and if needed, on the stack.
System call handling is made somewhat easier in MIPS. Whereas in x86_64, you have to set the appropriate system-calls arguments and then do an interrupt on the correct vector, MIPS has a dedicated system-call instruction `syscall`. The operating system can choose however the arguments are passed, but usually, the service number is stored in `$v0`, and the arguments in `$a0-$a3` [7].

Memory

Memory is an essential part of a working computer. The memory stores both the program code (the instructions) as well as the program variables, which cannot fit in all the CPU registers.
The memory is a large chuck of consecutive bytes. While these can store practically anything, operating systems designate memory areas into multiple segments (also called sections), which eases memory management. While many operating systems create many different segments in the memory map of a program, these segments usually fall into 4 categories [1]:

- `.text`
  This section consists of the executable contents of a program, which is the instructions. In modern processors and OS, this section cannot be written to.

- `.data`
  This segment contains the initialized variables used in the program. This section can be both written and read from, but is not executable.

- `.bss`
  Unitialized memory segment. This allocates memory for variables, which are not necessarily initialized by the OS, and can contain random values.

- `stack`
  The stack usually starts at the top of the memory area. It is a Last-In First-Out (LIFO) structure, meaning that

elements added last are the first ones being removed, hence the name. The top of the stack is usually pointed to by a designated register in the CPU, in MIPS32, that is the `$SP` register. The stack is used for local variables within functions, which usually have a very short lifespan.

- `heap`
  The heap starts immediatelly after the initialized segments (`.text` and `.data`), and is used for variables with a longer lifespan. This memory area is maintained by the program, and can store practically any variables and other data-structures. This area can also be extended by the operating system, which makes it ideal area for storing large amounts of data.

While the memory layout of a process is completelly implementation specific on the underlying operating-system, on most UNIX systems, the memory area is very similar to what can be seen on figure 2.

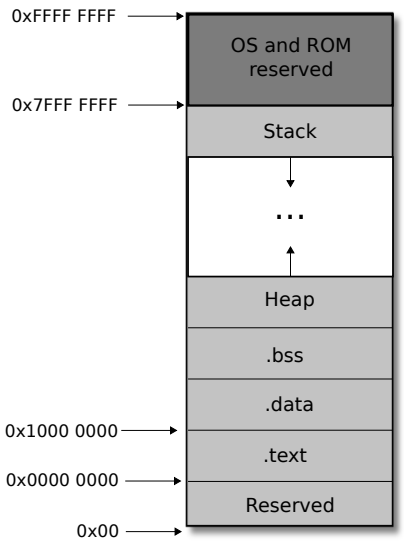


Figure 2: Process memory map.

Although the address space is 32 bit, the top half is not available for the user, and it is usually used by the operating system [10].

Pipeline

CPU speeds are usually measured by timing the execution time of programs. Since a computer program is just a collection instructions, the speed of the CPU is determined by how fast it can process each instruction. Every CPU has a clock, which ticks at a given rate. For every tick, a new instruction is executed. This clock ensures that all instructions "flow" through the processor without problems, and that the electrical components, such as the ALU or the control-unit, can manage to carry out their tasks in that time. Naturally, electrical engineers have pushed the limits of the circuits to manage the highest clock rate. The clock rate of the very first processors was measured in hertz and kiloHertz (kHz), but most modern desktop CPUs reach in multiple GigaHertz (GHz) [12]. However, even with those speeds, the demand for faster processing units is ever-growing, and other techniques to speed-up the execution are used.
One of those techniques is pipelining, which separates the circuit into multiple stages, much like the assembly lines in factories. In such factories, workers have their own station at the assembly line, do a specific task repeatedly, and forward it down the line. This greatly increases the throughput of a factory and decreases the labor need. In MIPS, this idea is implemented by separating the processor into 5 stages [7]:

- Instruction Fetch (IF)
  Fetches the next instruction.
- Instruction Decode (ID)
  Reads the instruction, sets the appropriate control flags, reads the relevant registers and sends the data to the next stage.
- Execute
  Executes the instruction. This is typically done by the ALU with the appropriate operation supplied.
- Memory Access
  All operations on memory happen here. This stage either loads a memory

address or stores a value at an appropriate address.

- Write Back
  Writes the results to the CPU registers.

Each of these stages will naturally use less time than all of them combined, and since the clock is shared in all stages, it is set to the slowest stage in the pipeline.
Not only do we have faster tick rate on our clock, but we are also able to perform multiple operations concurrently. Figures 3 and 4 show the timing of each instruction, and how pipelining might improve the whole process.

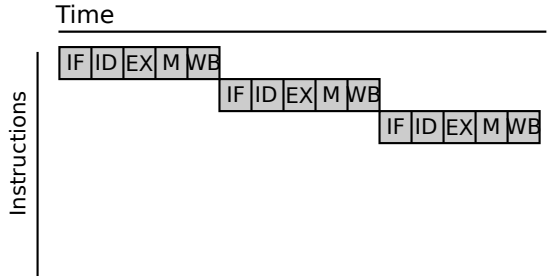


Figure 3: Single cycle implementation.

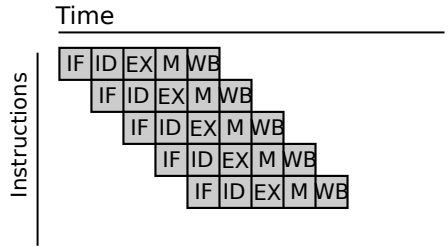


Figure 4: Pipelined approach.

Design of the MIPS32 Pipeline

The advantages of a pipelined design does not come without a price. Although the single-cycle implementation of the processor is very similar to the pipelined approach, it has its fair sets of challenges. The main problem with executing instructions concurrently is that the instructions will often rely on the result of the previous instructions. These situations are referred to as hazards.

Data Hazard

Data hazards mainly occur when an instruction cannot continue, because it must wait for the result from an earlier instruction. Suppose a program wants to calculate the sum of 4 integers:

$$A = A + B + C + D$$

In MIPS32 assembly, this would be written as:

```
# t0 = A, t1 = B, t2 = C, t3 = D
add $s0, $t0, $t1   # s0 = A + B
add $s1, $t2, $t3   # s1 = C + D
add $v0, $s0, $s1   # $v0 = result
```

Figure 5: Code exposing data hazard situation.

Here, the first two instruction will have no trouble executing, as they do not share any source or destination registers. The third instruction however, will not be able to fetch the updated values. When it is in the ID stage, where it decodes the register `s0` and `s1` values, the previous instructions are still in the pipeline, in the EX and MEM stage! These instructions have not written back their results in the appropriate registers, and so, instruction 3 cannot fetch the correct value of `s0` and `s1`, unless it waits 3 clock cycles. This situation can be visualized in a sequential graph show in figure 5, where it can be seen that the result of the first two instructions is needed in the third.

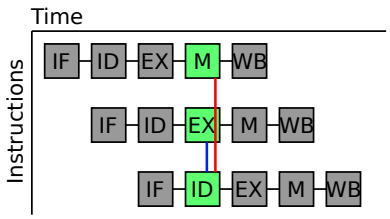


Figure 6: Time-sheet of the program listed in figure 5.

The problem is very clear - the data is needed before it saved at the appropriate place. However, the data is usually already available in the EX stage in the ALU, and

can therefore be used in the same clock-cycle in the ID stage. This is called forwarding or bypassing, and is handled by the Forwarding Unit. This unit is wired to both EX, MEM and WB stages, and has a logic unit that checks for matching registers in the instruction, in which case it forwards the correct unit back in the pipeline.

Control Hazard

Another type of hazard is the Control Hazard, which occurs when the processor must make a decision based on an instruction, while others are executing. Whether a branch is taken or not is determined in the EX stage, as that is the point where forwarding unit can bypass the most recent values, if any. If a branch is taken, the challenge emerges that we already have the next instruction in the ID stage, even though the program has taken another branch. There are many possible ways to solve this issue, the most simple, but inefficient one, is to stall the pipeline by inserting a No-Operation instructions after the branch. This way, the instruction after the branch will do nothing, but in both situations, we loose a clock cycle. Although this does not sound like much, considering that branching instruction make up about 25% of any assembly code [5], this can be a lot. A better, but much more complex technique to solve this issue, is to have a dedicated unit in the processor, which will predict whether a branch will be taken or not.

Implementation

The implementation of the pipelined simulator uses a very object-oriented approach, where each device or system has its own C structure.

Simulator structures

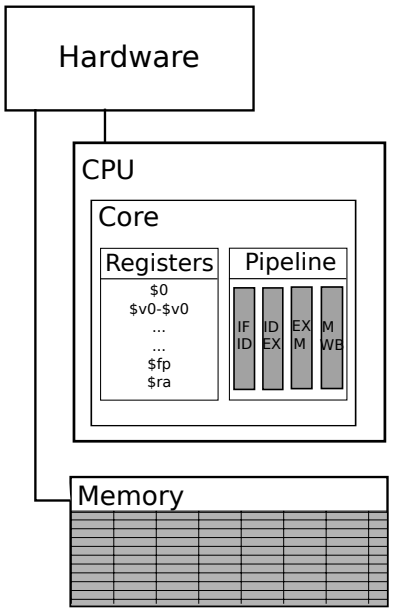


Figure 7: Relations between the simulator structures.

In the simulator, the processor cores, pipeline registers, and memory unit all belong to a "hardware" structure, that binds them together, making up a simulated machine. The relations between the components can be seen in figure 7.
In the simulator, the hardware structure will be used to pass along the simulated hardware to functions, which then can access the underlying devices, modifying them properly.

Pipeline functions

The pipeline is implemented by running each pipeline stage separately. For each clock-tick, the pipeline stage functions are being called in reverse. This can seem very unnatural, but this prevents pipeline registers to be overwritten before they are used.

```
void tick() {
        interpret_wb(...);
        interpret_mem(...);
        interpret_ex(...);
        interpret_id(...);
        interpret_if(...);
}
```

Each of the function receive the relevant pointer to the CPU core structure, which

holds all the relevant information about the state of the core.

In what level of detail should I describe the pipeline functions?

Forwarding Unit

The pipeline stages work without regard for data hazards, which is where data is required for an instruction before it is stored by the previous instruction. Therefore, we add a forwarding unit function `forwarding_unit()`, which is run by the end of each clock cycle. Being the last function, it has all the recent information in all the pipeline stages, which enables it to decide which values need to be forwarded.
The forwarding unit only forwards values to the ALU input mutual exclusion units MUX A and MUX B. It does that by first checking if the previous instructions write to a register, that is, modify a register value. If it indeed is the case, the destination registers are compared, and if the match, the value from the previous instruction is bypassed. This is done for both the A and B input to the ALU in both Memory- and Writeback-stages. The code for forwarding values from stage MEM to EX can be seen in figure 8. Forwarding from the WB stage is almost identical.

```
/* Forward to A MUX */
if(EX_MEM.c_reg_write == 1
   && EX_MEM.reg_dst != 0
   && EX_MEM.reg_dst == ID_EX.rs)
       ID_EX.rs_value = EX_MEM.
           alu_res;

/* Forward to B MUX */
if(EX_MEM.c_reg_write == 1
   && EX_MEM.reg_dst != 0
   && EX_MEM.reg_dst == ID_EX.rt)
      ID_EX.rt_value = EX_MEM.
          alu_res;
```

Figure 8: ALU result being forwarded from the MEM stage to EX.

Hazard detection unit

Control hazards arise when the processor is branching to another execution path based on the result of the previous instructions. As there is no "correct" solution to the branching hazard, the MIPS CPU manufacturers often implement their own branch predictions based on the intended use of the processor. For simplicity in our simulator, we will always predict that the branch is *not* taken. In that case, we always run the next instruction after the branch. This works well, because this is a widely accepted approach in many RISC architectures [11]. Knowing that the instruction following a branch is always executed, many standard GNU compilers and assemblers restructure the code so that branch-delay slot is not wasted.
For example, where assignment of a value to a register and a branch would consume 3 instructions, utilization of the branch-delay slot can bring it back to 2. The difference can be seen on figure 9. Note that the order of the instruction does not matter in that case.

```
addi $v0, $0, 10
beq $0, $0, exit
nop
```

```
beq $0, $0, exit
addi $v0, $0, 10
# ...
```

Figure 9: Wasted branch delay slot (left) and utilized branch delay slot (right).

Exceptions and Interrupts

3.6

TLB

MMU

User and Kernel mode

SMP

Tests

Performance

Conclusion

References

[1] Gustavo Duarte. Anatomy of a program in memory. http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/, 2009. [Online; accessed 9-April-2016, Archived by WebCite ®(at http://www.webcitation.org/6gf8eojPM].

[2] Michael Flynn. *Computer architecture : pipelined and parallel processor design*. Jones and Bartlett, Boston, MA, 1995.

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

[4] intel. *Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual*. December 2015. Combined Volumes:1, 2A, 2B, 2C, 3A, 3B, 3C and 3D.

[5] Dr. Lawlor. Instruction encoding & frequency. https://www.cs.uaf.edu/2012/spring/cs641/lecture/01_24_encoding.html, 2012. [Online; accessed 8-April-2016, Archived by WebCite ®http://www.webcitation.org/6gdfU63WO].

[6] OSDev.org. Interrupts. http://wiki.osdev.org/Interrupts, 2016. [Online; accessed 07-April-2016].

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

[8] 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.

[9] Imagination Technologies Group Plc. Mips microprocessors overview. https://imgtec.com/?do-download=4408, 2014. [Online; accessed 29-March-2016].

[10] Central Connecticut State University. Memory layout. http://chortle.ccsu.edu/assemblytutorial/Chapter-10/ass10_3.html, 2015. [Online; accessed 10-April-2016, Archived by WebCite ®at http://www.webcitation.org/6gfKPa3Iu].

[11] Wikipedia. Branch delay slots. https://en.wikipedia.org/wiki/Delay_slot#Branch_delay_slots, 2016. [Online; accessed 9-April-2016, Archived by WebCite ®at http://www.webcitation.org/6geBm0bpx].

[12] Wikipedia. Clock rate. https://en.wikipedia.org/wiki/Clock_rate, 2016. [Online; accessed 7-April-2016].

Appendices