My Final College Paper

A Thesis
Presented to
The Division of Mathematical and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Sebastian Andrada Ottonello

May 2024

Approved for the Division
(Computer Science)

Charles McGuffey

Acknowledgements

I want to thank a few people.

Preface

This is an example of a thesis setup to use the reed thesis document class.

List of Abbreviations

You can always change the way your abbreviations are formatted. Play around with it yourself, use tables, or come to CUS if you'd like to change the way it looks. You can also completely remove this chapter if you have no need for a list of abbreviations. Here is an example of what this could look like:

ALU	Arithmetic Logic Unit
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
ISA	Instruction Set Architecture
RISC	Reduced Instruction Set Computer
RISC-V	Reduced Instruction Set Computer Five

Table of Contents

List of Tables

List of Figures

Abstract

The preface pretty much says it all.

Dedication

You can have a dedication here if you wish.

Introduction

Computer Systems is the study of the physical, metal-and-silicon devices that make up a functioning computer, and how they operate at their lowest level; these devices are the final backbone of just about any work done on a modern computer, and the field is therefore well worth exploring.

Of the countless pieces of hardware to study and delve into within this field, the CPU (Central Processing Unit) can be argued to be the most essential one. It is the heart of any computer, the piece that actually carries out the programs and computations that it was built for. It comes as no surprise, then, that most college-level Computer Systems courses begin by taking a close look at the inner workings of a simple CPU, so that they can then begin to understand how the other components of a computer support its functionality, and how it can produce computation out of just an assembly program, wires, and logic gates. Learning of this is all well and good, but getting hands-on experience with the topic is harder than one might expect: The powerful CPUs in the personal computers of students run on are all closed-source designs that obfuscate how they carry out their instructions; and even if they were transparent, the sheer breadth and depth of modifications they employ to get the performance they do would make their design completely indecipherable to a novice.

Therefore, the aim of this project is to develop a pedagogical simulator program, developed in the Rust programming language and capable of running on modern computers, that simulates a simple RISC-V CPU at the hardware level, running assembly code and then transparently displaying how the code and computed information traverse the inside of the processor. While simple, the simulated CPU implements a 5-stage pipeline design similar to the ones that are mandatory for modern CPUs. The project also seeks to include some user-facing features, like a graphical interface and the ability to step through and rewind through the execution of a program, to make the simulator more intuitive to use.

The results are promising, but with room for improvement. Certain useful features have been deemed to be beyond the scope of the current project, but the finished simulator fully supports the 32-bit RISC-V Base Integer specifications, and features a GUI, support for user-inputted programs, and rewinding capabilities for its ease of use as well.

Chapter 1

Background

1.1 An abstract view of how a CPU functions

In order to build and understand the design of a CPU, we must first understand what it, in general terms, does. We've already established that a CPU carries out a program consisting of instructions from its ISA. However, what exactly do those instructions demand the CPU do, and what does the CPU have access to to carry it out?

We will go over the exact design of the model of CPU shown here later, with all of its intricacies and small parts. But, in broad terms, the CPU has access to: an **Instruction Memory**, where the program to be executed is stored instruction-by-instruction. A **Decoder** that reads this raw instruction, made of bits, and gleans the important details from it, like what instruction it is, what the inputs and outputs are, or other instruction-specific data. A **Register Memory**, which holds a small amount of numbered registers that store information; think of them as set mini-variables built directly into the CPU that it has quick and easy access to. A **Data Memory**, external and much larger than the previous two, in which to store larger values; and finally an **ALU**, Arithmetic Logic Unit, which is simply capable of performing arithmetic operations on the values it is given.

This explanation leaves out many crucial components and smaller parts that make the whole thing possible, but they're enough to understand the CPU's general plan, detailed here below. Note that this plan, and the designs described later, do not accurately describe all CPUs; rather, it is a simple design that, despite that, is fully equipped to execute base RISC-V programs. (REDO AS FOOTNOTE?)

1. Read the first instruction from the Instruction Memory.**(IF: Instruction Fetch)**

2. Decode the fetched instruction into useful information: The operation to be performed, the registers to be used, etc. **(ID: Instruction Decode)**

3. Read the values of whichever registers the instruction calls for from the Register Memory. **(ID: Instruction Decode again.)**

4. Perform whichever arithmetic operation the instruction calls for, using values from the registers or from the instruction itself as input. **(EX: Execute)**

5. If the instruction calls for a value to be read from or stored to the Data Memory, do so. **(MEM: Memory)**

6. Write back the end-result value of the instruction into the destination register it specified. **(WB: WriteBack)**

These six repeating steps are enough to accomplish everything that base version of RISC-V calls for, shockingly! It is enough for the CPU to successfully run low-level programs. How these simple instructions build up to the complex operating systems and graphical applications that most are familiar with today is beyond the scope of this project, but the important thing is that it is enough.

Do note that the dividing of the process of executing one instruction into six steps is entirely arbitrary; in fact, you may have noticed that the steps have been given titles in bold, but steps 2 and 3 are both lumped into the same "ID" name; These names are called "stages", and my 6 steps and the 5 stages are both arbitrary. You could subdivide this exact same process into less or more parts depending on how deep you look. However, this arbitrary model of "stages" is going to be incredibly helpful to implementing the pipeline design/feature of this simulator, which will be described later.

There are many simplifications and concerns ignored in this explanation, but one is particularly important to the CPU's design, relating to steps 1 and 2. In the modern day, we want multiple CPUs of different designs to all be capable of running the same machine code. How does the CPU know which set of bits correspond to which instruction? How does it knows which operations its designers should ensure it can compute?

1.2 What is an ISAs?

These questions all have the same answer; and so does an entirely different question, one you might've asked yourself back in the introduction: What does it even mean that a CPU is *RISC-V*?

The answer is, ISAs. ISAs are how CPUs know how to read machine code not specifically tailored to them; and RISC-V is one of many ISAs, belonging to the RISC family of design. So, what are they?

Starting from the top, ISA stands for Instruction Set Architecture; you may have also heard it reffered to as just a CPU's "architecture", or of popular ISAs like x86 and ARM. We will go into the details later, but for now an ISA is simply a list of machine-code instructions that a CPU can carry out: These instructions are things like basic arithmetic operations (addition, subtraction, etc), calls to load or store data from the memory, or even jump instructions to alter the flow of the program; it specifies how to decode any machine-code instruction, stored as a set of bits, exactly. Ultimately, a program is just a list of these instructions, one after the other, that the CPU executes in order.

The ISA also contains certain hardware specifications that the CPU should be assumed to have: Things like the amount of registers available and how large a value they can store, or how the program counter ought to be measured, or even certain demands to make life easier for programmers. For example, the MIPS ISA demands that the 0th register, "Register r0", be hardcoded to the value zero at all times, so that programmers have easy access to it. While centered around the instructions, these additional demands are a glimpse into how important and widespread the ISA is in a CPU's design. (INSERT RISC-V SPECIFIC STUFF HERE)

A CPU is considered to use a certain ISA if it supports and has implemented all of its instructions. ISAs were created for the goal of hardware compatibility: As long as two different CPUs made by competing manufacturers both use the same ISA, any low-level program will run on both processors because they both guarantee that they'll support the same instruction set, no matter how different their underlying design is.

Ultimately, an ISA is a standard for how a CPU should take any given line of code from a low-level program - encoded as an integer in binary, this close to the hardware - and translate that into an actionable operation that the CPU should execute when it reaches that part of the program.

1.3 Why RISC-V?

Though the ISA doesn't ascribe an exact design for how its demands should be met(that's called the *implementation*, and is the domain of the CPU's manufacturer), it so heavily shapes the CPU's capabilities, and therefore indirectly its design, that choosing which ISA to implement is the most important decision one can make when designing a CPU, whether it's real or virtual. As alluded to in the introduction, there are multiple ISAs to choose from, and many more popular ones than RISC-V. So, what makes competing ISAs so different from one another, and why choose RISC-V for this project instead of any of the more popular options?

RISC-V is a modern ISA, albeit one that has not yet experienced widespread success; while it very rapidly growing in use, its two contemporaries, x86 and ARM, dominate the Personal Computer CPU and Mobile CPU market entirely for the time being. And yet, RISC-V has two major advantages over them that make it the ideal choice for this simulator.

The first is, simply, that RISC-V is fully open source. Unlike its two proprietary cousins, the full listings of all the RISC-V instructions are freely available, with details on what behavior is expected of each instruction and exactly how to glean an instruction's parameter from its 32-bit integer representation. Working with open-source sorftware and specifications is good by its very nature in that its development benefits everyone, not just its creators, but it's also a better choice practically.

The second difference is that RISC-V is a RISC ISA, while x86 is a CISC ISA; it's even in the name. These stands for *Reduced Instruction Set Computer* and *Complex Instruction Set Computer* respectively, and they are opposing schools of design for an ISA. The difference ultimately comes down to how many different instructions the ISA lists out: The idea of a CISC ISA is that by providing a very large variety of different instructions, most individual things you'd want a program to do can be performed with just one instruction each. By contrast, a RISC ISA aims to keep its total number of instructions low: While this would mean that code would need more total instructions to accomplish the same task, the idea is that keeping the demands simpler would allow RISC processors to have faster and more efficient designs that offset this cost [1]. This also coincidentally makes RISC processors a much better example for students to study, while still remaining a modern architecture that's growing in widespread use.

1.4 RISC-V Instructions

It is also enough to get a rough idea of how any given RISC-V instruction is carried out. The full list of instructions can be found on the official *RISC-V Instruction Set Manual* that is freely available online, or on a number of more user-friendly guides like Five EmbedDev's *RISC-V ISA Instruction Quick Reference*. A list of all the instructions in the most basic version of RISC-V, the RV32I Base Integer Instruction Set, can be found (INSERT HERE), but the instructions are broadly divided into the following four categories by the *RISC-V Instruction Set Manual*:

Register-Register Computation: These instructions take two registers as input, perform some sort of operation on their two values (an arithmetic operation like addition, or a logical operation like a bitwise AND or NOT), and store the result in a third register.

Register-Immediate Computation: These are the same as RR (Register-Register) instructions, but take as input one register and one hard-coded value that's written directly into the program, called an Immediate. It's the difference between "z = x + y" and "z = x + 3".

Load/Store: The purpose of these instructions are simply to interact with the Data Memory, either storing a value to a specific address or reading a value from it.

Control Transfer: Instructions that call for the CPU to move on to a place in the code that is NOT the next instruction, potentially. These are what make the If-Then statements and Loops of high-level programming languages possible. These work because there is a simple **Program Counter**, PC, attached to the Instruction Memory that simply keeps track of which instruction should be performed next.

Each of these instructions in these categories, conveniently, move through the 5-stage process in roughly the same way based on their category.

RR Computation: Run through the steps as usual, feeding the value of its two input registers as the input for the ALU. ignore step 4, MEM, since Comptutational Instructions do not interact with the Data Memory.

RI Computation: Same as RR; but only one register is read at step 2. the other input value is the immediate, which the Decoder reads from the instruction itself.

Load: Load Instructions have a Register and an Immediate as input, representing the address in data memory to read from, as well as a destination register. They run through all of the steps; in the EX stage, the register value and immediate are added together to get the final address, which is then fed to the Data Memory in the MEM stage. Finally, the data loaded from Data Memory is stored in the destination register.

[1] Denning (1993)

Store: Store Instructions are similar to Load, but have two registers and an Immediate, with no destination register. The first register + Immediate are the address as usual, but the second register holds the value that is to be stored into memory; and is sent directly to the Data Memory through its own special wire.

Control Transfer: Control Transfer Instructions have an Immediate that describes where the program should jump to, and possibly some registers. They run normally until the EX stage: Here, rather than operate on registers, the ALU is given the Immediate and the current value of the PC, which are added together to determine the Jump location. Some additional computation may be performed in this stage, such as a comparison between registers, and for some instructions the jump destination is written to an output register, but the CPU, at this point, goes back to the IF stage while editing the value of the Program Counter to complete the jump.

Do note that these categories are loose and arbitrary, to an extent. the RISC-V Instruction Manual itself has six "Types" it divides its instructions into, but these have less to do with what the instruction accomplishes and more with the details of how the CPU should read its bits. The categories described above help give a broad overlook of what sort of instructions the RISC-V ISA expects CPU implementations to handle.

1.5 RISC-V Hardware Specifications

Despite what was stated earlier, the does ISA also makes certain demands, or specifications, for the hardware of the CPUs that implement it. Generally, the hardware side of things is the CPU's job, not the ISA; but in order to be compatible with every CPU running under it, an ISA needs to have certain guarantees. What is a RISC-V program tries to write a value to register "r19", but the CPU running it has only 16 registers?

RISC-V has two base variations to pick from, The *RV32I Base Integer Instruction Set* and the *RV64I Base Integer Instruction Set*. It also has many optional extensions, which are part of why RISC-V can have a simple core while remaining viable for commercial applications. However, it is the Base Integer Instruction Sets, reffered to as RV32I and RV64I respectively from now on, that determines hardware specification, so that is what we will look at now.

RV32I calls for a total of 32 registers. Registers r1 through r31 are general-purpose registers, each capable of holding a 32-bit-wide integer value. r0 is a special register hardwired to the constant value 0, which is useful for many situations.

It specifies that each Instruction should be 32-bits in length, though it doesn't specify how many Instructions the Instruction Memory should be able to hold. And, while the ISA doesn't make any demands of the Data Memory, and actually warns that the environment a program is executed in (like, say, an Operating System) gets to decide which parts of the Data Memory it can access, the amount it can access is upper-bound by the addresses that the RV32I Load and Store instructions can incode in their 32 bits.

RV64I is rather similar, with the difference that the 32 registers have a width of

64 bits each, and that Instructions themselves are 64-bits long.

The RV32I is simpler, and easier to understand at a glance, making it a better choice for an educational simulator.

1.6 A more detailed look of how a CPU functions

This information is enough to build a full hardware model of the CPU that the simulator will use; at least if Pipelining is ignored for now. The following is a visual representation of this model:

(INSERT DIAGRAM HERE)

Most of the components shown here have already been explained earlier, albeit with some modifications and additions. It should be noted that the components are arranged from left to right in the order of the stages; with the wires of the WB stage looping back to where the outputs should be stored. The notable changes are as follows:

1. The decoder is actually two components: The Decoder and the Immediates Decoder. The Decoder focuses on getting the indices of the two input registers and the destination register; and then a set of values called the opcode, funct3, and funct7. These are 8-bit, 3-bit, and 7-bit numbers respectively that, together, describe which exact instruction it is. The Immediates code gets the opcode in order to get the immediates out of the instruction; depending on the exact instruction, the immediate is found in different bits; RR instructions have no immediates at all.

2. There is a Branch Comparator component alongside the ALU. This allows Branch instructions to, while the ALU is busy computing the final destination of the jump, calculate whether the jump should happen or not.

3. Lastly, there are multiplexors, drawn as (INSERT INFO HERE), throughout. These simply take multiple inputs and, based on the state of the CPU, decide which one should pass through. For example, the two multiplexors right before the ALU need to decide to let the values from registers r1 and r2 pass on an RR instruction, but let the current PC and the Immediate pass in during a Control Transfer instruction.

This is enough to understand the CPU simulator, for the most part; rather than simply take in a machine-code program and execute it as efficiently as possible, which would be trivial to do in a modern high-level programming language, the simulator represents each of these components individually as structs, so that the flow of data through the CPU circuits can be displayed accurately.

Chapter 2

CPU and Simulator Design

Before looking at the build of the actual CPU simulator program, it is worth going over one, in particular, of the features that this simulator seeks to implement.

2.1 Pipelining

This feature, Pipelining, must be described first: It is not a feature of the simulator as much as it is a feature of the CPU itself that this program simulates, and the design of the CPU itself must be majorly altered in order for it to be successfully implemented. However, this feature is quite powerful, and used near-universally in RISC-V CPU designs (CITATION NEEDED), and so the simulator is made with it in mind.

The previously-described CPU design has one simple issue: It is painfully slow. This design is capable of executing one instruction per cycle: one instruction per full use of each of the CPU's parts. This may seem good, but at any given moment, most of the CPU is actually going unused: The decoder is waiting on the instruction memory to fetch the instruction it needs to decode, which is waited upon by the register memory that needs the register indices to fetch their data, which is waited upon by the ALU that wants its inputs, so on. When, say, the ALU is firing and computing, the data memory, decoder, instruction memory, and so on are all entirely dormant. They could, theoretically, be doing something useful at the same time; but because our design handles one instruction at a time, the components before the ALU have already done their part and the components after the ALU are waiting on it to finish and present its output. Theoretically, one could be multiplying efficiency of this CPU many times.

(DIAGRAM HERE OF THE STEPS, SHOW WASTED SPACE)

The solution is simple in theory, though difficult in execution: have the CPU be performing multiple instructions at once. Five of them, to be exact. This is where the five stages come into play: each stage does its part, performing its needed function for the current instruction; but when that instruction is passed off to the next stage, rather than waiting around for the full round trip, it's already receiving the information for the next instruction right away. If the WB stage is executing instruction number 9, the MEM stage is executing instruction 8, and the EX is executing instruction 7, so on.

(FOOTNOTE??)

(DIAGRAM HERE OF THE PIPELINED STEPS)

This is a good time to remind that the choice to subdivide this process into five stages, specifically, is entirely arbitrary. It's easy to see how two of these stages could be combined to have less-than-5 pipeline stages, or how the Instruction Decoding and Register Fetching could be separared into their own stages, but you can go much further than that: Most modern CPUs have somewhere around 5 to 14 distinct pipeline stages, and the outdated *Pentium 4 Prescott* CPU had a whopping 31 stages! (CITATION NEEDED). Theoretically, you'd want to split your CPU into as many pipelined stages as you possibly can; the more instructions you're running at once, the more efficiently the CPU is being utilized. There are tradeoffs and reasons for why this is not the case, which are better to describe later in this section.

Still, this is much, much faster; in optimal conditions, the CPU can execute nearly 5 cycles in the time it would've taken it to do 1 cycle before! Sure, each of those cycles only performs one-fifth of five instructions, but the end result over a long program would still be a five-fold increase in speed!

Of course, conditions are not always optimal. A certain addition will be needed to this diagram for pipelining to be possible at all; A few smaller ones will be needed to handle new issues that arise out of pipelining specifically; and some of those new issues can only be mitigated, not avoided fully. Let's go over these.

The first modification, which makes pipelining possible on a basic level, is to handle coordination. Theoretically, if every subcomponent of a CPU was perfectly coordinated and each of the five stages each always took the exact same amount of time to complete, one could simply set the CPU going, with a slight alteration to the PC so that it increments its program count 5 times per cycle instead of once. However, this is not the case: In real hardware, one cannot count on parts being perfectly coordinated; and some stages will take more or less time than others. The memory stage, notably, can require time to perform that is magnitudes longer than the other stages when it is required to go off the CPU and into a separate RAM stick, as is often used in more complex modern computers. (NEED FIXING/CITATION) Data memory structures even tend to have specialized on-CPU caches to minimize the need for this; it was originally intended for the CPU simulator to represent these fully as well, but this was proven to be beyond the scope of the project. Instead, the Data Memory is abstracted as a "black-box" subcomponent, which stores an array of data accessible by address through unknown means. Indeed, this entire timing aspect will be largely abstracted away in the simulator, operating entirely on the time unit of a "step" in which each of the instructions currently in the CPU advance by one stage; however, the alteration used to solve this timing issue in real hardware is so essential to the design, and additionally used in some of the solutions for smaller problems that cannot be abstracted away, that they will be included in the simulator regardless.

2.1.1 The Backbone of Pipelining: Latches

So, what is this modification? The main thing to be prevented here is that no stage of the CPU should recieve the inputs for the next instruction before it is done performing its computations for its current structure; to keep all five stages in sync. The ideal way to do this is to ”hold” the outputs of each stage once they are computed, until the slowest stage completes its computations, and then release them all at once to begin the next step. This ”holding” of outputs is done through the addition of a set of ”Latches” to the CPU:

(INSERT CPU + LATCHES DIAGRAM)

There’s one latch inbetween each CPU stage, with all of the outputs passing through said latch. They are named after the two stages they separate, with the leftmost being the ”IF-ID Latch” for example. No latch is needed after WB, since that stage simply consists of delivering the instruction result to where it needs to go. The latches are their own kind of register; in fact, ”Pipeline Register” is another name for them. The latch holds one value for every wire that would pass through it, and constantly outputs said value through to the output end. Then, at the start of every cycle If the ALU’s output wire read ”13” before the EX-MEM latch was closed, and ”4” after, the latch will continue to output ”13” to the wire leading from the latch to the Data Memory until the latch is opened again. This achieves the ’holding’ effect that was desired, and makes pipelining possible at all! In order to pipeline possible... (REFRESH ON HARDWARE LEVEL OF LATCH OPERATION)

This is one of the reasons why having more pipeline stages incurrs diminishing returns, along other tradeoffs: Simply put, Latches take a small amount of time to open, during which no computation can ocurr. The more pipeline stages your CPU has, the more times this needs to happen during one cycle, and the longer the cycle gets; therefore, adding more pipeline stages inherently has dimishing returns for the CPU’s effiency. There are entire studies about measuring the efficiency of CPUs, and how this is impacted by design decisions such as its latch placement.

However, even with pipeline and latches in place, even with the efficiency tradeoffs in mind, there are still other issues that will arise from this feature. Ultimately, whoever wrote the code that this CPU is running did so under the fundamental assumption that the CPU executes one instruction of code after the other, sequentially, waiting to finish one instruction before beginning the next. This is not the case in our faster, pipelined CPU; and while that’s fine in most cases, it isn’t always. There are certain scenarios that can occur in a machine code program where, due to this false assumption, our pipelined CPU as-it-is will produce an output that is wildly different from what the program correctly should. These scenarios are called ”Hazards” In order for the pipelined CPU model to be valid, it must have some way to avoid or correct any hazards that come up! Hazards can be broadly divided into these groups:

- Data Hazards: Hazards that involve the reading and writing of data to registers.
- Control Hazards: Hazards that involve jump instructions.

(... WERE THERE MORE??)

Let’s look at these hazards in more details, and at how the pipeline CPU design can be altereted to handle them.

2.1.2 Data Hazards

Data hazards are best introduced with a simple example, shown in assembly code. Assembly Code that is only one step removed from Machine Code: It's the same line-by-line instructions that deal directly with the CPU's registers and the Data Memory, for the most part, but presented in a way that is somewhat human readable instead of as rows of bits.

For this example, all you need to know is the RISC-V instruction ADDI. The instruction "addi b, a, i", where a and b are registers and i is an immediate value, adds the contents of the a register to the immediate i and stores the result in register b. It is not unlike the line of code "b = a + 10" (FORMATTING) in a python program.

Imagine the following program snippet, where $\$r0$, $\$r1$, and $\$r2$ are registers and all registers start with the value 0.

```
addi $r1, $r0, 3
addi $r2, $r1, 4
```

In a non-pipelined CPU, this program runs fine, as shown below. The steps relevant to this example are described by cycle; the IF and MEM stages do not matter to the example and will thus be skipped over.

(INSERT DATAHAZARD-NOPIPELINE FIGURE)

Cycle 2: Fetch the value of input register $\$r0$, 0, from Register Memory.

Cycle 3: ALU executes $0 + 3$, its output is 3.

Cycle 5: The output, 3, is written back to the $\$r1$ register in the Register Memory.

Cycle 7: Fetch the value of input register $\$r1$, 3, from Register Memory.

Cycle 8: ALU executes $3 + 4$, its output is 7.

Cycle 10: The output, 7, is written back to the $\$r2$ register.

The end result is a state where $\$r1$ holds the value of 3 and $\$r2$ holds the value of 4. This is the correct output, and what any such person would expect this code snippet to perform with the described starting conditions. However, look at what happens if we run the same code snippet through our pipelined design:

(INSERT DATAHAZARD-PIPELINE FIGURE)

Cycle 2:

Instr 1 ID: Fetch value of input register $\$r0$, 0, from R Mem.

Cycle 3:

Instr 1 EX: ALU excecutes $0 + 3$, output is 3

Instr 2 ID: Fetch value of input register $\$r1$, 0, from R Mem.

Cycle 4:

Instr 2 EX: ALU executes $0 + 4$, output is 4

Cycle 5:

Instr 1 WB: The output, 3, is written back to the $\$r1$ register.

Cycle 6

Instr 2 WB: The output, 4, is written back to the $\$r2$ register.

This time, the end value of $\$r2$ is erroneous, 4 instead of 7! The pipelined CPU gives an incorrect output, and the reason boils down to this: **the output of Instruction 1 is needed for the correct input of Instruction 2, since $\$r1$ is both the former's destination register and the latter's input register**. But, Instruction 1's WB stage happens **after** Instruction 2's ID stage. In most cases, this would be fine; but not when the second instruction is trying to read from the same register that the first instruction modifies. It will happen even if these two instructions aren't directly sequential; only after the two are far enough temporally that the former's WB stage ocurrs at the same time as the latter's ID stage does it stop being an issue.

At the very least, having a program return a consistently incorrect output is unnaceptable, and we need to fix that. But, how?

2.1.3 Stalling

The first solution that might come to mind is: "Why don't we pause the CPU when a Data Hazard is incoming, so that it has time to catch up?". And, this is shockingly a decent suggestion! The idea is simple: As it is in the example, the update to $\$r1$, ocurring at the start of Instruction 1's WB stage, was needed halfway through Instruction 2's ID stage (Register acessing happens during the latter half of the ID stage, after the Decoder has gone). The red arrow in the diagram represents this desired update that needs to happen for correctness.

(DATAHAZARD-NOSTALL DIAGRAM)

So, we could just delay the execution of Instruction 2 by three cycles...

(DATAHAZARD-STALL DIAGRAM)

And now the program will execute correctly! This process, called "Stalling", can be applied dynamically, too; if a Data Hazard happens by just one stage of difference, the offending instruction only needs to be stalled one cycle. This method inherently costs us some of the efficiency that pipelining offers either way, though.

But, how is this Stalling implemented? As it is now, our pipelined CPU functions, but it will simply barrel forward without control: At the start of every CPU cycle, all the latches fire and transfer their information forward to the next stage. Luckily, we already have the perfect tool to control this flow, and it's those latches themselves!

Let's add a new functionality to our latches. As they are now, the latches are in "Transfer" mode:

Transfer: At the start of cycle, take the incoming currents of the previous stage and update the outgoing currents to match.

Now, let's add two other possible modes:

Stall: At the start of cycle, change nothing; keep the previous outgoing currents as they were.

Bubble: Set the outgoing currents to all-zeroes, regardless of what the ingoing currents were.

Lastly, we add a "Stall Control" subcomponent to the CPU. All it does is determine whether and how it needs to Stall for the given cycle, and change the latches between these three states to perform the stall. Note: There is an additional fifth "Latch" in this setup, where the Stall Control hooks up to somewhere in the Program

Counter circuitry. This simply allows the Stall Control to pause the Program Count's natural per-cycle increment; so long as the instructions are stalling, the program count needs to stall as well or the CPU will start skipping instructions entirely.

But, why are all these modes needed at all? Well, we cannot simply stall out the entire CPU, that is the same as if it wasn't running at all. What we need to do is allow for some instructions, those before the hazard-bearing instruction, to finish running while the rest of the program stops and waits for the desired number of stall cycles.

The Stall Control has all latches set to "Transfer" during normal operation. However, once an instruction with a Data Hazard reaches the ID stage, it alters the latch modes as follows.

(INSERT STALL CONTROL FIGURE HERE)

At this point, the EX, MEM, and WB stages hold the instructions that you do not wish to stall; those are Transfered so they can finish. The IF and ID stages are Stalled (using the PC and IF-ID Latch respectively), holding their current instructions. Lastly, the EX stage gets "Bubbled"; set to an output matching a "NOP" instruction that simply does absolutely nothing. This is quite important: if the EX stage was simply stalled as well, you'd actually start getting duplicates of the instruction immediately predecing the hazard-holding instruction inserted into the pipeline. Thanks to the bubble, harmless 'nop' instructions that alter nothing get sent out instead to fill that gap.

Once the instruction that holds the needed alteration reaches the WB stage, the Stall Control simply flips all latches to Transfer mode, and operation continues as normal. This does successfully resolve the data hazard, and fixes these correctness issues! However, it is still painfully slow, and gives up much of the advantage of pipelining anytime that a register is written to and then read from in quick succession; as it turns, this is rather common. We've already accepted that conditions will not often be optimal for pipelining, but this is a serious blow. Can we do better than that?

2.1.4 Forwarding

The answer is that yes, we can! Let's go back to our data hazard example, when it is un-stalled:

(DATAHAZARD-NOSTALL DIAGRAM)

Obviously, the issue is clear here: The data needed in the ID isn't ready until Instruction 1's WB stage, and is needed in Instruction 2's ID stage. Except, this isn't really true. Technically, that data has already been computed by the end of Instruction 1's EX stage, found in the output of the ALU. And, it is Instruction 2's EX stage that needs the data for the addition calculation, not the ID; as long as the ALU gets the correct integer input of 4 instead of 0, it doesn't care where it came from. So, technically, the desired data to resolve this hazard *is* available by the time it's needed; it just doesn't have the pathway to get there in time. So... what if we added that pathway ourselves?

That is exactly what forwarding is! Extra data pathways are added into the ALU input; controlled by a multiplexor that reads register indices to decide if forwarding is needed. The pathways come from the MEM-stage output if the offending instructions are adjacent, and from the WB-stage output if they are one instruction apart; any more than that, and there is no Data Hazard at all. These are called "EX-EX Forwarding" and "MEM-EX Forwarding" respectively. There is a third type of Data Hazard, which occurs when a value is loaded from the Data Memory to a register and then stored to somewhere else in the Data Memory immediately after; the store will still have the stale value from that register, from before it was read to. An extra pathway and multiplexor, from the WB stage to the Data Memory, handles that; this is called "MEM-MEM Forwarding" . With that, all of the Data Hazards are resolved, this time without performance loss!

(INSERT FORWARDING TYPES DIAGRAM)

Doesn't this make stalling entirely obsolete? Well, not exactly; there's still one more unhandled data hazard case. What if happens if a load instruction is immediately followed by an arithmetic instruction, using the register that was loaded to? (Or by a store instruction, using the register to calculate an adress rather than as the to-store data). As seen below, the data is retrieved at the end of the Load's MEM stage, yet is needed at the beginning of the EX stage so it can be used in the arithmetic operation. In this case, the data truly doesn't come soon enough!

The ideal solution is a hybrid of the two techniques: If you stall the second instruction by just one cycle, the data can now arrive just in time. It can even use the same "MEM-EX Forwarding" from earlier without any new pathways needed!

2.1.5 Control Hazards

There is a second reason why stalling, or at least the Stalling Control we implemented, is still useful: Control Hazards.

So, when do control hazards occur? Well, anytime any jump instruction is called. The problem is simple: anytime that a jump instruction occurs, the new address to jump to is already needed by start of the next instruction's IF stage: but it is calcuated in the jump's EX stage, where the ALU adds the jump value to the existing Progam Count.

This hazard is going to lose us cycles no matter what; but to mitigate that, the first step is to add another forwarding pathway from the ALU output directly to the PC in the IF Stage. This almost resolves the issue, with the jump happening at a 2-cycle delay, though there's one more problem: Inbetween the jump instruction entering the IF stage and the jump address being calculated in its EX stage, two instructions directly after the Jump get loaded in; instructions that shouldn't run at all, since the program supposedly jumps to another location. After those two, though, the output of the EX stage gets forwarded back to the PC in the IF stage, so the solution is simple: On the cycle that the jump instruction is in the EX stage, we tell the Stall Control to set the latches of the IF and ID stages (the stages holding the two erroneus instructions), to go on "Bubble Mode". This harmlessly flushes out those two instructions before they have a chance to perform any alterations, and the instruction following them will be from the correct jumped-to address! The end result is akin to stalling the instruction following a jump for 2 cycles, in order to avoid the Control Hazard.

In fact, you could achieve the same result by stalling the instruction 2 cycles, using the method described in the earlier section on stalling. This would work: there, the two erroneous instructions never enter the pipeline at all. It would be simple if all hazards were handled through the same method, so why not just do that?

It has to do with *Branching* control instructions, or Branches. A Branch is similar to a jump, but it also makes some sort of logical comparison between two inputted register value. If the comparison resolves to true, the Branch jumps to the destination; if it's false, the jump is not taken and the program continues to the next sequential instruction. Branches are the main option assembly programs have for introducing, well, branching pathways to the control flow of their code, and what compilers most heavily use when compiling if-statements and loops from high-level programming languages.

Well, as it turns out, handling jumps with the flushing method is actually a small optimization that potentially saves time for *branching* control instructions. Branching control instructions are almost the same as jump instructions: there's just an extra subcomponent in the EX stage that compares its values to check if the Branch is to be taken, which decides if the Program Count is changed or left alone. However since we cannot know whether the branch will be taken or not *until* the EX stage, this causes a difference between the flushing and stalling methods: In the stalling method, the instruction after the Branch has to get stalled regardless of whether the Branch gets taken or not; it has to start stalling right away. In the flushing method, the CPU flushes and jumps as usual if the Branch gets taken; however, if it does not get taken, the Stall Control simply... doesn't do anything. Those two "erroneous instructions" are actually now the correct path forward for the program to take, and it just lets them happen. So, flushing the Branches instead of stalling them makes no diffence if the branch is taken, but saves cycles if it isn't! This method is also called the "Fetch and Cancel When Taken" method of handling control hazards, since the "cancel" or flush only happens if the Branch is taken.

2.1.6 Final CPU Design

(INSERT FIANLIZED CPU DESIGN HERE)

The final design of the CPU we will simulate is therefore as shown above. Instruction, Decoder, Registers, Arithmetic, Data Memory, and Writeback; built with the intent that the simulator will display its own internal state to a similar level of granularity as the diagram above.

The flow of instructions through the wires is separated into 5 arbitrary stages in order to **Pipeline** them for major performance boosts, with the separation enforced by a set of Latches.

Then, in order to deal with the correctness hazards that this pipelining incurrs, additional **Forwarding** pathways and a **Stall control** in the latches are implemented to sidesteps those hazards, while also minimizing the performance loss incurred.

2.2 Simulator Design

The actual simulator is a program written in the Rust programming language, and aims to faithfully replicate the operation of the CPU at the same level of abstraction used in the diagrams above! as a result, it can be compiled and run on most modern personal computers. Its final iteration fully supports the Risc-V 32-bit Base Integer Instruction Set, with the exception of system-call instructions that are meaningless without the context of an OS outside the program, and the failsafe for a particular Data Hazard that turned out to be rather difficult to impelement.

The goal of the simulator itself is simple: to take in a machine-code RISC-V program as input, set up a virtual construct in Rust that replicates the low-level, semi-abstracted model of a CPU that has been discussed in the background chapter, and then run said program through it in a way where the simulated CPU's internal state is always transparent. Other goals, such as a User Interface, are additional features and not part of the core design. This design was eventually accomplished, though the project had to go through a major restart partway.

First Draft

The first draft was quite neat in theory, though rather idealistic. The core idea is to rely on the way that the diagrams above dive the CPU into separate ”subcomponents”, such as the Program Counter or the ALU, which only pass along information through predefined wires; this is something the final design will keep.

The core idea of the first draft is that each of these subcomponents should be represented by its own unique ”struct” in Rust; despite the name, these are much closer to the Classes and Objects of Object-Oriented Programming Languages like Python and C++, made to hold a specific set of variables as well as functions tailored to those variables. Every ”subcomponent” on the graph would be represented with its own unique struct: It would have variables for both every input and output connection it had with other subcomponents; and if it was a stateful subcomponent that stores data, like the Program Counter or any of the three Memories (Instruction, Register, Data), it would also have variables for recording said state. References to the structs that they need to fetch input data from, as well as functions to fetch said data and to perform the subcomponent's intended purpose complete the struct. Then, all of this would be hooked up to some sort of master ”setup” and ”step” functions; the former would initialize all these structs and insert the program into the Instruction Memory, while the latter would activate these structs one by one to perform a single full CPU cycle, over and over.

The advantages of such a design are apparent: The different structs are entirely separate from one another, making them easy to test on an individual level and to see the state of any one given subcomponent at a time. It also meant that the layout of the code resembled the design of the actual CPU somewhat intuitivelty.

However, this design also has two glaring flaws: The first is more manageable, and it is simply that the design's method of passing data between structs, that being of giving each struct a persistent reference to the structures that send information to it, is entirely incompatible with Rust's memory-safety failsafes. Well, the idea of doing that, in itself, is not, but the problem comes from the fact that, due to the CPU's cyclical nature, these links of one struct borrowing a refference to the previous struct eventually form a loop. The details are not relevant to this project, but using such persistent references in a loop with no clear top-level "owner" goes against the language's philosophy to the point of not being allowed at all. This issue could've been resolved on its own if it was the only problem with the design; for example, by simply having the step function handle all the wiring itself and using the subcomponent structs for their internal computations only.

That said, the second issue is that the design was unbearably repetitive, to the point of being confusing to look at. The distinct input-output variables on each struct meant that every piece of information in the CPU essentially took up two variables instead of one, and the process of computing a piece of data and sending it to the next subcomponent in the stage became a chore with many unecessary middle-man steps inbetween.

Final design

The second and final iteration of the simulator took a slightly different approach, one that relies more closely on the nature of this CPU as being ultimately made of logic gates and wires, and employs style of design often used to simulate simpler logic gate machines.

The idea now is to divide the CPU design into two categories, and therefore two structs: Its State and its Logic. The State consists of all the naturally stateful elements of the CPU as described before: The Program Counter, the three Memories, and also all of the Latches, which essentially store the supposed current state of the wires they are supposed to clamp down on. The Logic stores everything else that has no state; or, rather, it stores all of the *wires*, since that is all else that matters. A subcomponent like the ALU has no internal state; it reads data of the two operands and the opcode, and calculates its output solely off of that; even to be accurately rerpesented, this representation can be done simply as a piece of code that performs the ALU's function. Even the wires are only stored in persistent variables for the sake of visually displaying the CPU's state later. So, the entirety of the CPU is separated into a State struct and a Logic struct, and each struct has an "Update" function written to it that takes the other struct as a mutable reference and updates the state of all its own variables at once: For the State Struct, this means handling memory changes like updating the PC, enacting the write of data into the Writeback register, and Storing data into the Data Memory when Store instructions pass by. For the Logic struct, it means reading the state of latches and registers and determining what signals should be traveling through the wire as a result of their changes.

(DIAGRAM OF CPU, COLOR-CODED FOR STATE AND LOGIC MAYBE?)

This setup is so useful because it makes the actual update of a CPU cycle incredibly simple: Just run the update function for the State struct, followed by the update function for the Logic Struct. This is not entirely true, as the implementation of stalling will complicate things a little, But for the most part, this makes up the base of the simulated CPU and the process for executing one of its cycles:

```
struct State {
        pc : u32,
        instr_mem : Vec<u32>
        // ... more State variables
}

struct Logic {
        added_pc : u32,
        instruction_out : u32,
        //... more Logic variables
}

impl State {
        fn update(&mut self, &mut logic: Logic){
                self.pc = logic.added_pc:
                //... more State updates
        }
}

impl Logic {
        fn update(&mut self, &mut state: State){
                self.added_pc = state.pc + 4;
                self.instruction_out = state.instr_mem[(state.pc / 4)]
                //... more Logic updates
        }
}

fn step(state: &mut State, logic: &mut Logic) {
        state.update(logic);
        logic.update(state);
}
```

This code is entirely functional as a rough base for the simulator's design; all the ,main body of the program needs to do outside of this function is to initalize the two structs, get the user-fed instruction into the instruction memory, and then run the $update(state, logic)$ function repeatedly until the program has run its course!

Other features can and will be implemented as well, of course; but, the design shown above is a sweeping generalization. It's worth taking a closer look at what the final, detailed program looks like, now that its basic structure is understood.

Structs and Sub-Structs

The final design could indeed have just unceremoniously dumped all of its State and Logic variables into their respective structs; but this would've made a rather huge mess, resulted in individual wires (especially different wires that carry the same piece of data at different stages) being hard to keep track of, and would've been rather confusing and inelegant in general; this is one of the issues the first draft design sought to resolve. Therefore, the final design for the structs instead looks like this:

```
#[derive(Clone)]
pub struct Registers {
        pub ifid: IFIDLatch,
        pub idex: IDEXLatch,
        pub exmem: EXMEMLatch,
        pub memwb: MEMWBLatch,

        pub pc: u32,

        pub instr_mem: Vec<u32>,
        pub reg_mem: Vec<u32>,
        pub data_mem: HashMap<u32, u32>,
}

//this structs holds all the wiring of each stage

#[derive(Clone, Default)]
pub struct Logic {
        pub fetch: IFLogic,
        pub decode: IDLogic,
        pub execute: EXLogic,
        pub memory: MEMLogic,
        pub writeback: WBLogic,

        pub pc_stall: bool,
}
```

Rather than having a soup of variables, all of the states tied to a specific latch and all the wires tied to a specific pipeline stage are sequestered off in their own sub-struct; when used like this, these sub-structs are just fancy ways of holding variables and don't get in the way like they did in the previous design. It also allows for said sub-structs to be stored in their own file, leading to a much cleaner folder structure for the source code. In a way, these sub-structs are effectively just organizational folders for the variables within the program.

```
use crate::components::*;

//The IF-ID Latch
#[derive(Clone, Copy, Default)]
pub struct IFIDLatch {
    pub base_pc: u32,
    pub added_pc: u32,
```

```
    pub instruction: u32,

    pub id_stall: u8, //stall, bubble, or neither?
}

impl IFIDLatch {
    pub fn bubble(&mut self) {
        self.base_pc = 0;
        self.added_pc = 0;
        self.instruction = 0;
    }
}

//holds all of the wiring of the ID stage
#[derive(Clone, Default)]
pub struct IDLogic {
    pub decode_r1: u8,
    pub decode_r2: u8,
    pub decode_opcode: u8,
    pub decode_rd: u8,
    pub decode_funct3: u8,
    pub decode_funct7: u8,

    pub regmem_r1: u32,
    pub regmem_r2: u32,

    pub immediates: u32,
}
```

(FOOTNOTE EXPLAINING THIS IS DECODE.RS) (MAYBE DIAGRAM OF DECODE STAGE TO COMPARE?)

This is an example of one such file; in genral, everything is stored as simply as possible, with the data types one would expect: Most wires carry a 32-bit unsigned integer; since this is a 32-bit system, that is the format that the CPU preferst to handle its data in. while wires for the smalled opcode, funct-3, and funct-7 all use unsigned 8-bit integers. Technically, the latter should use u7, u3, and u7 respectively, but establishing custom-length integer data types just for them was deemed to not be a very high priority.

The instruction memory and register memory are simply represented as u32 Vectors, with the former having a length equal to the number of instructions in the program and the other having 32 integers exactly; this makes accessing registers incredibly intuitive, as the nth register can be found in $state.reg_mem[n]$. The data memory is orders of magnitude larger than the first two though, and is going to consist mostly of empty addresses, so using a vector would have been incredibly wasteful; instead, it uses a hashmap of 32-bit integers with numbered keys, matching the size of a "word" that the *Store Word* and *Load Word* instructions use. This means that the smaller load-store instructions need to slice off part of the accessed word, but this was fairly manageable.

It is worth noting that there are some wires which aren't often shown in CPU design diagrams, but are nonetheless present in a real CPU, and therefore in our simulation too. For example, the Forwarding Multiplexors are only shown as having the base, EX-EX, and MEM-EX register values going into them, but in reality they also read a few other values, such as the opcode and register indices of the current and next two instructions, to decide which forwarding, if any, should ocurr.

Ultimately, this makes the storing and accessing of information about the CPU pretty intuitive, which will come in very handy for the User Interface.

Update Functions

Unfortunately, the two update functions aren't quite as simple. In a sense, they are: Each one goes has the struct going through all of its variables and updating them according to the internal logic of the CPU design. However, this updating needs to be done in just the right order in order to make things work properly. This is one case where building a virtual simulator is actually harder than making a real CPU: In a real CPU, all of this happens instantaenously and at the exact same time, literally within that given CPU clock cycle, and so it does not need to worry about this. But the simulated CPU runs one-step-at-a-time, and doing certain updates before others can cause massive errors. For example, the latches in the State struct's update function need to be updated in order from right to left, starting with the MEM-WB latch and ending with the IF-ID latch. Otherwise, if the IF-ID latch updates before the ID-EX latch, the information it was supposed to send directly over, like the base PC and added PC, that information vanishes for good. The problem is especially tricky with the Logic update function where, thanks to forwarding and writebacks, certain computations of a pipeline stage rely on information from a stage further to the right. Thankfully, part of the job of the latches here is to serve as a stopgap to separate the stages, so the issue is merely annoying rather than impossible.

A lot of the desired ordering was figured out, in part, by writing a large amount of test cases and catching any correctness errors, though that will be discussed in detail later.

Its ordering aside, the actual code contained within is mostly understandable, just there to perform the functions that those sub-components would in a real CPU. The decoder code takes an instruction and separates it into its relevant pieces using bit-masks and bit-shifts; the register memory writes and reads data off the 32 general-purpose registers, so on. The most crucial one is the ALU, since that's utlimately where the actual computation of most instructions takes place, and even that one is farily simple: It takes in the two operands, looks at the opcode to figure out what sort of arithmetic operation it's supposed to perform and spits out the output to $logic.execute.alu_out$. To whomever is interested in more details, the update functions and my comments on them can be found in the *components.rs* file, right under the structs.

Stalling

There is one piece of the core simulator that doesn't fit neatly into the description above.

2.3 CPU-Simulator Features

2.4 Bibliographies

Of course you will need to cite things, and you will probably accumulate an armful of sources. This is why BibTeX was created. For more information about BibTeX and bibliographies, see our CUS site (`web.reed.edu/cis/help/latex/index.html`)[1]. There are three pages on this topic: *bibtex* (which talks about using BibTeX, at `/latex/bibtex.html`), *bibtexstyles* (about how to find and use the bibliography style that best suits your needs, at `/latex/bibtexstyles.html`) and *bibman* (which covers how to make and maintain a bibliography by hand, without BibTeX, at at `/latex/bibman.html`). The last page will not be useful unless you have only a few sources. There used to be APA stuff here, but we don't need it since I've fixed this with my apa-good natbib style file.

2.4.1 Tips for Bibliographies

1. Like with thesis formatting, the sooner you start compiling your bibliography for something as large as thesis, the better. Typing in source after source is mind-numbing enough; do you really want to do it for hours on end in late April? Think of it as procrastination.

2. The cite key (a citation's label) needs to be unique from the other entries.

3. When you have more than one author or editor, you need to separate each author's name by the word "and" e.g.
   `Author = {Noble, Sam and Youngberg, Jessica},`.

4. Bibliographies made using BibTeX (whether manually or using a manager) accept LaTeX markup, so you can italicize and add symbols as necessary.

5. To force capitalization in an article title or where all lowercase is generally used, bracket the capital letter in curly braces.

6. You can add a Reed Thesis citation[2] option. The best way to do this is to use the phdthesis type of citation, and use the optional "type" field to enter "Reed thesis" or "Undergraduate thesis". Here's a test of Chicago, showing the second cite in a row[3] being different. Also the second time not in a row[4] should be

[1] Reed College (2007)
[2] Noble (2002)
[3] Noble (2002)
[4] Reed College (2007)

different. Of course in other styles they'll all look the same.

2.5 Anything else?

If you'd like to see examples of other things in this template, please contact CUS (email cus@reed.edu) with your suggestions. We love to see people using LaTeX for their theses, and are happy to help.

Chapter 3

Mathematics and Science

3.1 Math

TEX is the best way to typeset mathematics. Donald Knuth designed TEX when he got frustrated at how long it was taking the typesetters to finish his book, which contained a lot of mathematics.

If you are doing a thesis that will involve lots of math, you will want to read the following section which has been commented out. If you're not going to use math, skip over this next big red section. (It's red in the .tex file but does not show up in the .pdf.)

3.2 Chemistry 101: Symbols

Chemical formulas will look best if they are not italicized. Get around math mode's automatic italicizing by using the argument `$\mathrm{formula here}$`, with your formula inside the curly brackets.

So, $\mathrm{Fe_2^{2+}Cr_2O_4}$ is written `$\mathrm{Fe_2^{2+}Cr_2O_4}$`
Exponent or Superscript: $\mathrm{O^-}$
Subscript: $\mathrm{CH_4}$

To stack numbers or letters as in $\mathrm{Fe_2^{2+}}$, the subscript is defined first, and then the superscript is defined.
Angstrom: Å
Bullet: $\mathrm{CuCl \bullet 7H_2O}$
Double Dagger: ‡
Delta: Δ
Reaction Arrows: $\longrightarrow$ or $\xrightarrow{solution}$
Resonance Arrows: $\leftrightarrow$
Reversible Reaction Arrows: $\rightleftharpoons$ or $\xrightleftharpoons{solution}$ (the latter requires the chemarr package)

3.2.1 Typesetting reactions

You may wish to put your reaction in a figure environment, which means that LaTeX will place the reaction where it fits and you can have a figure legend if desired:

$$\mathrm{C_6H_{12}O_6 + 6O_2 \longrightarrow 6CO_2 + 6H_2O}$$

Figure 3.1: Combustion of glucose

3.2.2 Other examples of reactions

$\mathrm{NH_4Cl_{(s)} \rightleftharpoons NH_{3(g)} + HCl_{(g)}}$
$\mathrm{MeCH_2Br + Mg \xrightarrow[below]{above} MeCH_2 \bullet Mg \bullet Br}$

3.3 Physics

Many of the symbols you will need can be found on the math page (`http://web.reed.edu/cis/help/latex/math.html`) and the Comprehensive LaTeX Symbol Guide (enclosed in this template download). You may wish to create custom commands for commonly used symbols, phrases or equations, as described in Chapter **??**.

3.4 Biology

You will probably find the resources at `http://www.lecb.ncifcrf.gov/~toms/latex.html` helpful, particularly the links to bsts for various journals. You may also be interested in TeXShade for nucleotide typesetting (`http://homepages.uni-tuebingen.de/beitz/txe.html`). Be sure to read the proceeding chapter on graphics and tables, and remember that the thesis template has versions of Ecology and Science bsts which support webpage citation formats.

Chapter 4

Tables and Graphics

4.1 Tables

The following section contains examples of tables, most of which have been commented out for brevity. (They will show up in the .tex document in red, but not at all in the .pdf). For more help in constructing a table (or anything else in this document), please see the LaTeX pages on the CUS site.

Table 4.1: Correlation of Inheritance Factors between Parents and Child

Factors	Correlation between Parents & Child	Inherited
Education	-0.49	Yes
Socio-Economic Status	0.28	Slight
Income	0.08	No
Family Size	0.19	Slight
Occupational Prestige	0.21	Slight

If you want to make a table that is longer than a page, you will want to use the longtable environment. Uncomment the table below to see an example, or see our online documentation.

Table 4.2: Chromium Hexacarbonyl Data Collected in 1998–1999

Chromium Hexacarbonyl			
State	Laser wavelength	Buffer gas	Ratio of $\frac{\text{Intensity at vapor pressure}}{\text{Intensity at 240 Torr}}$
$z^7P_4^\circ$	266 nm	Argon	1.5
$z^7P_2^\circ$	355 nm	Argon	0.57
$y^7P_3^\circ$	266 nm	Argon	1
$y^7P_3^\circ$	355 nm	Argon	0.14
$y^7P_2^\circ$	355 nm	Argon	0.14
$z^5P_3^\circ$	266 nm	Argon	1.2
$z^5P_3^\circ$	355 nm	Argon	0.04
$z^5P_3^\circ$	355 nm	Helium	0.02
$z^5P_2^\circ$	355 nm	Argon	0.07
$z^5P_1^\circ$	355 nm	Argon	0.05
$y^5P_3^\circ$	355 nm	Argon	0.05, 0.4
$y^5P_3^\circ$	355 nm	Helium	0.25
$z^5F_4^\circ$	266 nm	Argon	1.4
$z^5F_4^\circ$	355 nm	Argon	0.29
$z^5F_4^\circ$	355 nm	Helium	1.02
$z^5D_4^\circ$	355 nm	Argon	0.3
$z^5D_4^\circ$	355 nm	Helium	0.65
$y^5H_7^\circ$	266 nm	Argon	0.17
$y^5H_7^\circ$	355 nm	Argon	0.13
$y^5H_7^\circ$	355 nm	Helium	0.11
a^5D_3	266 nm	Argon	0.71
a^5D_2	266 nm	Argon	0.77
a^5D_2	355 nm	Argon	0.63
a^3D_3	355 nm	Argon	0.05
a^5S_2	266 nm	Argon	2
a^5S_2	355 nm	Argon	1.5
a^5G_6	355 nm	Argon	0.91
a^3G_4	355 nm	Argon	0.08
e^7D_5	355 nm	Helium	3.5
e^7D_3	355 nm	Helium	3
f^7D_5	355 nm	Helium	0.25
f^7D_5	355 nm	Argon	0.25
f^7D_4	355 nm	Argon	0.2
f^7D_4	355 nm	Helium	0.3
Propyl-ACT			

State	Laser wavelength	Buffer gas	Ratio of $\frac{\text{Intensity at vapor pressure}}{\text{Intensity at 240 Torr}}$
$z^7P_4^\circ$	355 nm	Argon	1.5
$z^7P_3^\circ$	355 nm	Argon	1.5
$z^7P_2^\circ$	355 nm	Argon	1.25
$z^7F_5^\circ$	355 nm	Argon	2.85
$y^7P_4^\circ$	355 nm	Argon	0.07
$y^7P_3^\circ$	355 nm	Argon	0.06
$z^5P_3^\circ$	355 nm	Argon	0.12
$z^5P_2^\circ$	355 nm	Argon	0.13
$z^5P_1^\circ$	355 nm	Argon	0.14
Methyl-ACT			
$z^7P_4^\circ$	355 nm	Argon	1.6, 2.5
$z^7P_4^\circ$	355 nm	Helium	3
$z^7P_4^\circ$	266 nm	Argon	1.33
$z^7P_3^\circ$	355 nm	Argon	1.5
$z^7P_2^\circ$	355 nm	Argon	1.25, 1.3
$z^7F_5^\circ$	355 nm	Argon	3
$y^7P_4^\circ$	355 nm	Argon	0.07, 0.08
$y^7P_4^\circ$	355 nm	Helium	0.2
$y^7P_3^\circ$	266 nm	Argon	1.22
$y^7P_3^\circ$	355 nm	Argon	0.08
$y^7P_2^\circ$	355 nm	Argon	0.1
$z^5P_3^\circ$	266 nm	Argon	0.67
$z^5P_3^\circ$	355 nm	Argon	0.08, 0.17
$z^5P_3^\circ$	355 nm	Helium	0.12
$z^5P_2^\circ$	355 nm	Argon	0.13
$z^5P_1^\circ$	355 nm	Argon	0.09
$y^5H_7^\circ$	355 nm	Argon	0.06, 0.05
a^5D_3	266 nm	Argon	2.5
a^5D_2	266 nm	Argon	1.9
a^5D_2	355 nm	Argon	1.17
a^5S_2	266 nm	Argon	2.3
a^5S_2	355 nm	Argon	1.11
a^5G_6	355 nm	Argon	1.6
e^7D_5	355 nm	Argon	1

4.2 Figures

If your thesis has a lot of figures, LaTeX might behave better for you than that other word processor. One thing that may be annoying is the way it handles "floats" like tables and figures. LaTeX will try to find the best place to put your object based on the text around it and until you're really, truly done writing you should just leave it where it lies. There are some optional arguments to the figure and table environments to specify where you want it to appear; see the comments in the first figure.

If you need a graphic or tabular material to be part of the text, you can just put it inline. If you need it to appear in the list of figures or tables, it should be placed in the floating environment.

To get a figure from StatView, JMP, SPSS or other statistics program into a figure, you can print to pdf or save the image as a jpg or png. Precisely how you will do this depends on the program: you may need to copy-paste figures into Photoshop or other graphic program, then save in the appropriate format.

Below we have put a few examples of figures. For more help using graphics and the float environment, see our online documentation.

And this is how you add a figure with a graphic:

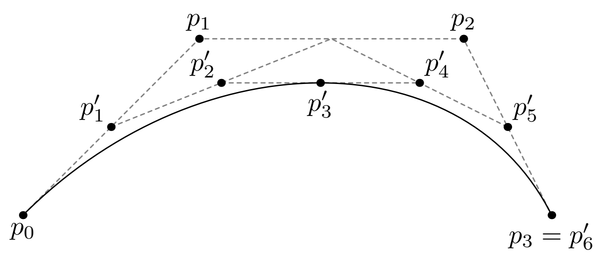

Figure 4.1: A Figure

4.3 More Figure Stuff

You can also scale and rotate figures.

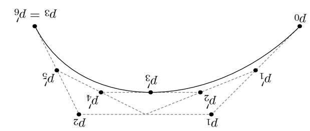

Figure 4.2: A Smaller Figure, Flipped Upside Down

4.4 Even More Figure Stuff

With some clever work you can crop a figure, which is handy if (for instance) your EPS or PDF is a little graphic on a whole sheet of paper. The viewport arguments are the lower-left and upper-right coordinates for the area you want to crop.

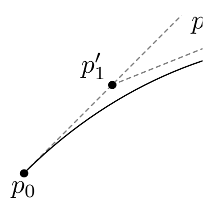

Figure 4.3: A Cropped Figure

4.4.1 Common Modifications

The following figure features the more popular changes thesis students want to their figures. This information is also on the web at `web.reed.edu/cis/help/latex/graphics.html`.

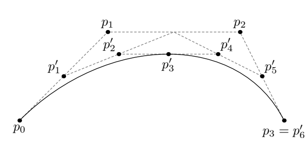

Figure 4.4: Subdivision of arc segments. You can see that $p_3 = p'_6$.

Conclusion

Here's a conclusion, demonstrating the use of all that manual incrementing and table of contents adding that has to happen if you use the starred form of the chapter command. The deal is, the chapter command in LaTeX does a lot of things: it increments the chapter counter, it resets the section counter to zero, it puts the name of the chapter into the table of contents and the running headers, and probably some other stuff.

So, if you remove all that stuff because you don't like it to say "Chapter 4: Conclusion", then you have to manually add all the things LaTeX would normally do for you. Maybe someday we'll write a new chapter macro that doesn't add "Chapter X" to the beginning of every chapter title.

4.1 More info

And here's some other random info: the first paragraph after a chapter title or section head *shouldn't be* indented, because indents are to tell the reader that you're starting a new paragraph. Since that's obvious after a chapter or section title, proper typesetting doesn't add an indent there.

Appendix A

The First Appendix

Appendix B

The Second Appendix, for Fun

References

Angel, E. (2000). *Interactive Computer Graphics : A Top-Down Approach with OpenGL*. Boston, MA: Addison Wesley Longman.

Angel, E. (2001a). *Batch-file Computer Graphics : A Bottom-Up Approach with QuickTime*. Boston, MA: Wesley Addison Longman.

Angel, E. (2001b). *test second book by angel*. Boston, MA: Wesley Addison Longman.

Denning, P. J. (1993). The science of computing: Risc architecture. *American Scientist*, *81*(1), 7–10. http://www.jstor.org/stable/29774812

Deussen, O., & Strothotte, T. (2000). Computer-generated pen-and-ink illustration of trees. *"Proceedings of" SIGGRAPH 2000*, (pp. 13–18).

Fisher, R., Perkins, S., Walker, A., & Wolfart, E. (1997). *Hypermedia Image Processing Reference*. New York, NY: John Wiley & Sons.

Gooch, B., & Gooch, A. (2001a). *Non-Photorealistic Rendering*. Natick, Massachusetts: A K Peters.

Gooch, B., & Gooch, A. (2001b). *Test second book by gooches*. Natick, Massachusetts: A K Peters.

Hertzmann, A., & Zorin, D. (2000). Illustrating smooth surfaces. *Proceedings of SIGGRAPH 2000*, *5*(17), 517–526.

Jain, A. K. (1989). *Fundamentals of Digital Image Processing*. Englewood Cliffs, New Jersey: Prentice-Hall.

Molina, S. T., & Borkovec, T. D. (1994). The Penn State worry questionnaire: Psychometric properties and associated characteristics. In G. C. L. Davey, & F. Tallis (Eds.), *Worrying: Perspectives on theory, assessment and treatment*, (pp. 265–283). New York: Wiley.

Noble, S. G. (2002). *Turning images into simple line-art*. Undergraduate thesis, Reed College.

Reed College (2007). Latex your document. http://web.reed.edu/cis/help/LaTeX/index.html

Russ, J. C. (1995). *The Image Processing Handbook, Second Edition*. Boca Raton, Florida: CRC Press.

Salisbury, M. P., Wong, M. T., Hughes, J. F., & Salesin, D. H. (1997). Orientable textures for image-based pen-and-ink illustration. *"Proceedings of" SIGGRAPH 97*, (pp. 401–406).

Savitch, W. (2001). *JAVA: An Introduction to Computer Science & Programming*. Upper Saddle River, New Jersey: Prentice Hall.

Wong, E. (1999). *Artistic Rendering of Portrait Photographs*. Master's thesis, Cornell University.