Comprehensive Thesis: Synthesizing a CPU

Donald Stayner

with
Peter Millet & Brian Collett

Hamilton College
Department of Physics

December 15, 2014

Table of Contents

Table of Figures

Acknowledgements

I would like to thank my advisor, Peter Millet, for helping me throughout the entire project and always being willing to meet on short notice. Professor Millet's insights and advice were crucial to this project's completion, and I very honestly would not have been able to do it without him. I would also like to thank my technical advisor, Brian Collett, for his willingness to take on yet another responsibility this term, and also for tolerating my slow engulfment of the entire E&C lab (and for stealing his chair).

Further acknowledgement goes to Mr. David Picard, the engineering lab instructor at Dartmouth's Thayer School of Engineering. His many hours of instruction this past spring were crucial in developing a working knowledge of the tools I used for implementing this CPU onto hardware.

Finally, I would like to acknowledge Alan Clements, author of *Principles of Computer Hardware*, who surprisingly responded to my email on the best techniques for making circuit diagrams.

Abstract

This project documents the design and implementation of a simple 16-bit CPU. This CPU is designed and implemented with a priority for clarity and understanding over efficiency and performance.

The design features a Harvard architecture comprising a simple datapath with eight registers and dual-port RAM, a simple ALU (Arithmetic Logic Unit), ROM for storing programs, and a control unit that connects programs to the datapath. The resulting CPU, implemented on a programmable processor, can successfully execute code written in its machine language.

Project Background

The goal of this project is to document the process of designing a simple, pedagogically-oriented CPU and subsequent implementation using VHDL, an industry-standard tool for engineering integrated circuits.

What is a CPU?

The CPU, or central processing unit, is the part of a computer responsible for performing the basic arithmetic, logical and input/output operations of the system. With a CPU, one can perform different tasks by specifying different instructions, rather than physically rewiring the computer. Thus, rather than programming an integrated circuit specifically for the task at hand, you can describe the task with instructions and have them executed by the CPU.

CPUs are designed with a number of basic operations they are able to perform, such as arithmetic or conditional program execution (if x do y). Instructions for the CPU are in the form of machine code, which is unique to the processor. Each instruction is a pattern of bits that is executed directly by the hardware. Taken together, the processor's set of supported instructions is known as its Instruction Set. The Instruction Set Architecture (ISA) defines how these instructions are represented in machine language that can be read by the CPU. Using a processor's ISA, a programmer could write instructions in machine and load them directly into memory for execution. However, coding in machine language is tedious and prone to error, as every instruction is simply a line of binary 0's and 1's. Thus, coding at the machine language level is done in assembly, where each instruction (and associated operands) is represented as a mnemonic. (We will define our instruction set as assembly commands, and the ISA as how these assembly commands are represented in machine language.) Over time, "high-level" languages have developed where any one instruction is associated with several (or even hundreds) of equivalent machine language instructions.

Regardless of the level of abstraction, though, any program written for a computer will end up in machine code once it is compiled to be executed.

CPU History

Alan Turing, the Father of Modern Computing

The principle of the modern computer was first described by Alan Turing in his 1936 paper, *On Computable Numbers*.[1] Turing introduced the concept of the Turing machine, a hypothetical device able to compute anything for which there is an effective algorithm. Turing's machine is a kind of state machine, represented by a tape of infinite length divided into squares of arbitrary symbols. The machine inspects squares of the tape one at a time, as directed by a table of transition rules that serve as the program for the machine. Each transition rule is of the form, "if the machine is in state [x], and the cell being scanned has symbol [y], then take action [z] and move to state [w]. As actions, a Turing machine can either write to the current square, or move to the left or right on the tape. Based on values of the tape and a defined table of transition rules, the Turing machine will execute until the answer is all that remains on the tape.[2]

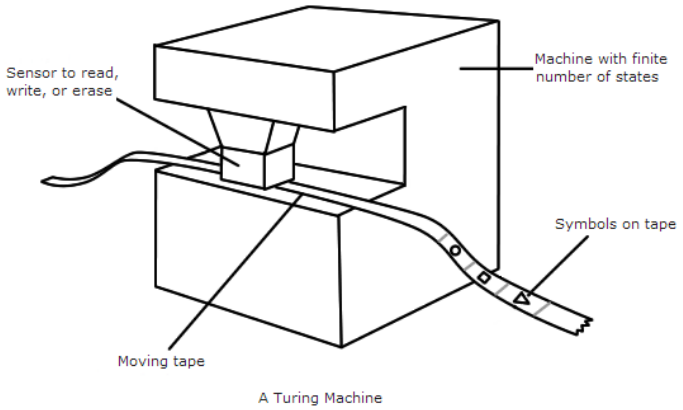

Figure 1: Illustration of Turing Machine[3]

In his same paper, Turing went on to define the Universal Turing Machine, which is capable of not just computing anything that is computable, but also performing the tasks of any other Turing machine – including itself. The UTM, then, is a sort of Turing machine interpreter, written in the language of Turing machines. This is achieved by storing the transition rules of a Turing machine on the tape of the UTM itself with binary symbols (0s and 1s). Thus, rather than using transition rules to solve a specific computation, the transition rules of a UTM are universal rules governing how it executes a given Turing Machine stored on tape. [4]

The Universal Turing Machine is noteworthy because it served as a blueprint for the CPU, which executes a program (or Turing Machine) using its transition rules, which govern how the program's instructions are executed by the processor. Indeed, Von Neumann – whose eponymous CPU structure is the basis of modern computer design – acknowledged that the central concept of the modern computer was due to Turing's paper.[5]

Turing's concept of the Universal Turing Machine has come to define the theoretical model of computation that characterizes the central processing unit; except for the limitations imposed by their finite memory stores, modern computers are said to be Turing complete – meaning they have algorithm execution capability equivalent to a UTM.

Early CPU Designs

The first truly Turing-complete computer was the US-built Electronic Numerical Integrator and Computer (EINAC) in February 1946. The EINAC was the first electronic programmable computer built in the US, and could compute any problem that would fit into its memory. [6] However, the EINAC was not a stored-program computer, for programs stored in its memory were defined by the state of its patch cables and switches.[7]

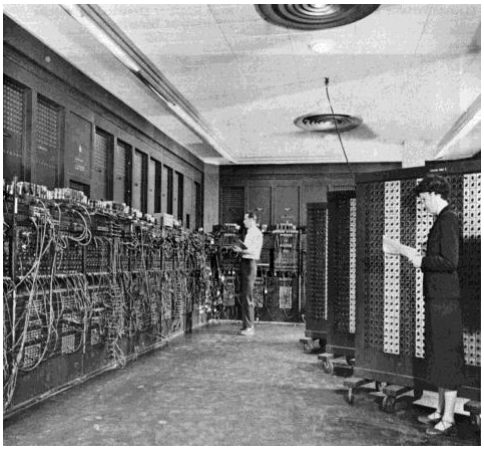

Figure 2: ENIAC, the first electronic general-purpose computer

The Manchester Small-Scale Experimental Machine (or Manchester "baby") was the world's first stored-program computer, meaning that in addition to being Turing-complete, programs were stored in electronic memory via a novel use of secondary emission of a cathode ray tube to temporarily store binary data.[8] The Manchester "baby" ran its first program in June of 1948, and was primarily intended as a demonstration of feasibility. Once feasibility was demonstrated, development began for a more usable successor. This resulted in the Ferranti Mark 1, the world's first commercially available general-purpose computer (delivered in February 1951). [9]

These early computers were massively expensive; the UNIVAC I, the first mass produced computer, sold for $9 million each in today's dollars. Over time, they became smaller and less expensive as new technologies, such as magnetic storage and transistors (rather than vacuum tubes) were developed.

The Integrated Circuit

The integrated circuit is a set of electronic circuits on one small plate of semiconductor material. Before the integrated circuit, computers were built of discrete components such as

resistors and transistors. The first practical integrated circuit was invented by Jack Kilby at Texas Instruments in 1958. The invention of the integrated circuit led to the development of the microprocessor: a CPU wholly contained on an integrated circuit. In 1965, Gordon Moore made his famous prediction that the density of transistors on an integrated circuit would roughly double every 12 months (later revised to every 24 months in 1975). Moore's law is has proven to be accurate to this day; however this is partially because the semiconductor industry adopted it as a guide for long-term planning.

The development of microprocessors in the 1960s led to the development of microcomputers – computers cheap enough to be owned by individuals – in the 1970s. Since then, CPUs have continued to become more powerful, through a combination of Moore's law, and techniques such as pipelining to optimize efficiency. While today's CPUs are much more powerful than those of the 1950s, their basic design and function retains the architecture articulated by Turing an implemented by Von Neumann.

How are modern CPUs designed?

First emerging in 1987, VHDL (Very High Speed Integrated Circuit Hardware Description Language) is a syntax that can be used to describe a digital system. Before hardware description languages, digital systems were developed with schematics. While schematics are useful for simple circuits, they are not well-suited as complexity scales. One way this was addressed was by blackboxing simple integrated circuits and using them as physical building blocks for more complex systems. This development process results in printed circuit boards – which connect integrated circuits as well as other discrete components together to form a larger circuit. Galaga, the famous arcade game, for example, is run using a printed circuit board.

Hardware description languages disrupted this paradigm, however, for they allowed the development of integrated circuits at the register-transfer level (RTL) – a level of abstraction that allows for the development of much more complex integrated circuits. In addition, VHDL allows one to simulate a circuit and track down issues before translating the design into real hardware. VHDL development was initiated by the American Department of Defense, who requested a language for describing hardware to document the behavior of integrated circuits it purchased from suppliers. The language was structured so that the code itself served as chip documentation. VHDL was standardized by the American Institute of Electrical and Electronics Engineers (IEEE) in 1987.

Today, hardware description languages such as VHDL are the industry standard for developing digital systems – which are, for the most part, specialized integrated circuits designed for a specific purpose. Circuits described with VHDL can also be synthesized to Field-Programmable Gate Arrays, or FPGAs. These are essentially re-programmable integrated circuits, and what I use to implement the CPU design laid out in this report. With VHDL, we use the FPGA to implement a basic example of the integrated circuit upon which the entire computing revolution has taken place – that of the Central Processing Unit, the heart of the modern computer.

Communicating with the CPU

Computers execute machine code, a primitive language consisting of simple operations such as addition and subtraction, Boolean operations, and data movement. The statements and constructs of a high-level language are translated into sequences of machine code instructions by a compiler (this process is shown in Figure 3). Machine language instructions are loaded into memory sequentially; the CPU executes instructions one at a time.

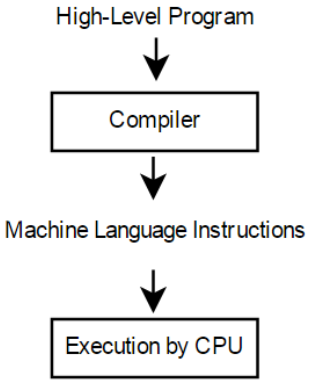

Figure 3: Execution of high-level programs on the CPU

Overview of Assembly Language

Assembly Language represents the human-readable form of machine code. For example, the binary string "1000011001010001" could be the a machine code instruction represented in Assembly as ADD R1, R2, #1 (add 1 to the number in register R2, and store the result in register R1).

It is somewhat simplistic to say that assembly is just a human-readable version of machine code; modern assembly languages implement strategies, such as assigning variables to certain commands, to make it easier for a human to write a program. We will not be addressing this aspect of assembly; we are most concerned with understanding the format of basic assembly instructions. Depending on the assembler, the basic format of assembly instructions can vary as well, for assembly languages are designed for specific processors, or families of processors. This is at odds with higher-level languages, whose sources can be compiled and run on a wide variety of computer systems. A program written in Assembly for one processor will likely not be able to run on another computer system. There are a number of different assembly languages widely used today, each based on a processor family. Some well-known processor families are Motorola 68x00, Intel x86, SUN Sparc, Vax,

and IBM-370.

Today, assembly language is primarily used for two purposes: debugging or reverse-engineering code written in a higher-level language, and for applications that require direct access to computer hardware. For example, if one were to write a printer driver, it would be difficult to do so using Java, which does not permit direct access to hardware. With Assembly, however, hardware access is straightforward and simple.

We will review the syntax and semantics of assembly language later on in the background, after introducing the basic components of the CPU.

Basic CPU Design

Before we examine how the CPU works, we must first consider the relationship between the program, CPU and memory.

Memory and RAM

Memory is typically defined as an indexed array that can be used to store and later access data. Programs are stored line by line in sequential addresses of memory, and the CPU executes machine language instructions by accessing them from the program memory.

RAM is memory organized so that it takes the same amount of time to access any address, regardless of its location. The access time doesn't depend on where the data being accessed is located. This is important for the CPU, because interfacing with memory would become dramatically more complex if it had to calculate how much time to wait based on the location of the given address. RAM is defined in opposition to serial access memory, where the time to access a certain piece of data is dependent on its physical location; examples include CDs, magnetic tape, and hard drives.

Harvard vs Von Neumann Architecture: Two Different Approaches

When the instructions of a program are stored in memory separate from the memory used to store data, the CPU is known as a *Harvard architecture CPU* (which is the approach we are taking). Harvard Architecture CPUs have separate paths for data and instructions, which is arguably easier to understand when presented visually. When the instructions and data are stored in the same memory, it is known as a *Von Neumann architecture CPU*; in this case, data and instructions flow along the same datapaths. The Von Neumann architecture processor is a more direct analog to the Universal Turing Machine we discussed earlier.[1]

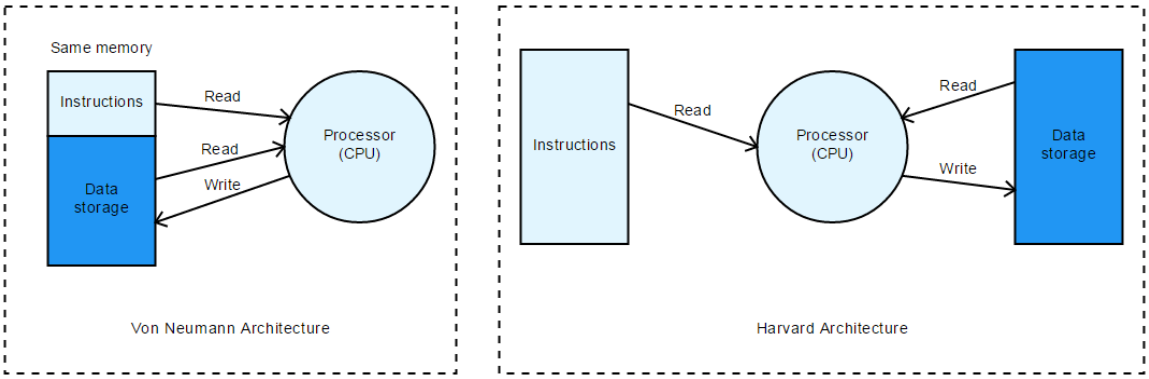

Figure 4: Harvard vs Von Neumann Architectures

Figure 4 illustrates how the two architectures differ in their relationships to memory. On this abstract level, both approaches interface the memory to the processor via three buses, or information paths. The address bus is a one-way path from memory to processor, whereas the data bus is bi-directional and can flow both ways. Also, it is important to note that the memory used for instructions in the Harvard architecture is technically ROM-RAM (read-only random access memory), but will be referred to as ROM or Read-Only Memory.

---

[1] While the Harvard Architecture CPU is also a UTM, the implementation is not as obvious because instructions and data are not stored in the same memory. While the specifics are unimportant, the Harvard Architecture is equivalent to Von Neumann in that both are able to perform the tasks of any other Turing machine.

Components of CPU

The CPU can be generally divided into two parts: the control unit and the datapath.

In the Harvard architecture, the control unit manages the CPU's relationship with instruction memory. It is responsible for decoding machine language instructions, and translating them into the signals that drive execution.

The Control Unit also contains the Program Counter, which keeps track of the address of the current instruction in instruction memory. As we mentioned before, instructions are stored sequentially in memory. After each instruction is executed, the Program Counter will increment by one to access the next instruction, unless the instruction specifically directs the program counter to a different address (more on that later).

The datapath uses signals from the control unit to execute a program's instruction. The datapath also serves as the CPU's interface with data memory – which it can both write to and read from. Finally, the datapath sends *status signals* back to the control unit; these are typically used for helping determine the next instruction that should be executed. A high-level overview of the CPU's two parts is shown in Figure 5 below.

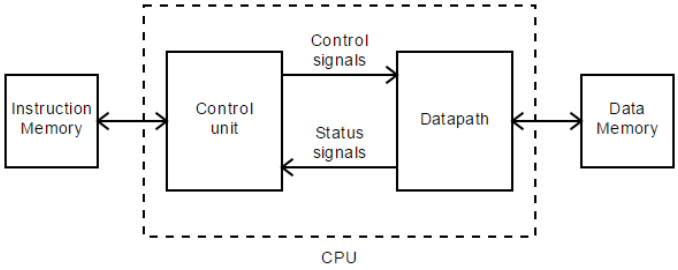

Figure 5: Block Diagram of Processor

In addition to these two components, the CPU has an internal "clock" which generates a sequence of repetitive pulses. The clock is used for parts of the CPU that use sequential

logic, where the logic is driven by both current inputs and past history. Every instruction requires a minimum of one clock cycle to execute. The clock's frequency is one of the most visible "specs" of a CPU – for example, a "2.6 GHz Intel i7 CPU" has an internal clock with a frequency of 2.6 GHz. Modern chips run at frequencies of up to 3.8 GHz – meaning the cycle time is as little as 0.26ns! The clock also synchronizes the internal operations of the CPU with other system components.

The Arithmetic Logic Unit (ALU)

The ALU is a part of the datapath responsible for the actual processing by the CPU of given inputs. The ALU performs arithmetic and bitwise logical operations on the inputs, which are integer binary numbers. While there are several different ways to represent integers in binary, the standard in processor design is two's complement – which allows us to represent negative numbers at the cost of one bit. So, if our ALU were to accept two 8-bit binary integers, we would only be able to represent $2^{8-1} - 1 = 127$ numbers rather than $2^8 - 1 = 255$.

The arithmetic or logical operation performed on inputs is determined by the control unit. Values are fed to the to the ALU from the second component of the datapath – a number of internal registers used for temporary data storage.

The Registers

As mentioned above, the registers are a part of the datapath used for temporarily storing information. While information can also be stored in memory, the registers are notable because reading and writing from a register takes just one clock cycle. The typical CPU has between just ten and thirty registers – a small number in contrast to the amount of addressable storage in RAM. A typical register has three inputs and one output. The inputs are the input data, the clock signal, and a load enable signal. Every clock cycle, if load enable

= 1, then the register will write the input data to its output. Otherwise, the data output remains constant. Thus, reading a value from a register is just a matter of accessing its output. An *n-bit* register stores data that is n bits long. On a circuit diagram, an input, output or signal longer than one bit is denoted by a slash through its line, and a number representing its length. This notation is shown below in Figure 6 for our n-bit register.

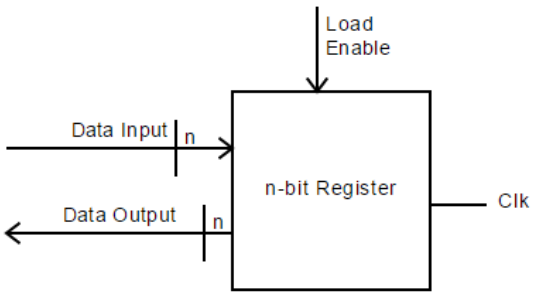

Figure 6: Diagram of basic register

The datapath is constructed such that the ALU uses register outputs for calculations, and subsequently write the result back to another register. Thanks to the speed of the registers, these types of operations (known as *data manipulations*) can typically be executed in just one clock cycle. The registers are also used when writing to or reading from the data memory.

Assembly Language Syntax, Semantics and RTL

Now that we have gone over the basic components of the CPU, we are in a position to understand the instructions that are typically written at the assembly language level. Since instruction structure differs from machine to machine, we will go over the instruction format I lay out the processor in the next chapter on the development process. In addition, for each instruction we will also go over its equivalent in *Register Transfer Language* (RTL) – an algebraic notation that is essentially an architecture-neutral way of representing assembler instructions. In addition, it is the notation used to describe digital circuits at the

Register Transfer Level, which is the level of abstraction used when programming in VHDL. (So, learning RTL will also be helpful for understanding the CPU's source code.) There are three basic instruction types in assembly: data manipulation, data movement and control flow.

Data Manipulation Instructions

Data manipulation instructions tell the CPU to take some values stored in the registers, perform a given operation on them using the ALU, and write the result back to another register. Data manipulation instructions are of the form

```
operation      destination, source1, source2 ; comment here
```

where *operation* defines the nature of the instruction, *destination* is where the result should be stored, *source1* is the location of the first operand, *source2* is the location of the second operand, and text after the semicolon is commented code for documentation. Common operations of this form include addition, subtraction, and bitwise logic (AND, OR, NOT, etc.). The sources of data manipulation operations can be specified in two ways – as the values of registers, or as "immediate" values passed directly from the instruction. Note the difference in syntax for an example of each below:

```
ADD     R3, R1, R2   ; R3 <- R1 + R2
ADD     R3, R1, #1   ; R3 <- R1 + '1'
```

The second command denotes '1' in decimal with the # sign. Hexadecimal (base 16) is denoted with $, and binary with %. (So #25 is equivalent to $19 and %11001.) The equivalent instructions in RTL are noted as comments. RTL uses the left arrow to indicate the transfer of data.

Data Movement Instructions

Data movement instructions are used to transfer data from registers to memory and vice versa. Examples of both are listed below, along with their RTL equivalents as comments.

```
LOAD    R0, (R1)        ; R0 <- M[R1]
STORE   (R0), R1        ; M[R0] <- R1
```

The LOAD command accesses the data at the memory address value stored in R1, and writes the data to R0 (register 0). The equivalent RTL uses the left arrow to denote transfer of data, and M[R1] denotes that the value is being pulled from memory at the address specified in R1. If we were to directly specify the address, the RTL would be $R0 \leftarrow M[address]$. The STORE command loads data from R1 into memory at the address specified by R0. The RTL again shows data transfer with the left arrow, though the data is traveling in the opposite direction to a LOAD command.

Control Flow Instructions

Control flow instructions (or branch instructions) are used to control the flow of a program. As mentioned in the overview of the Control Unit, there is a Program Counter that usually increments after each instruction is executed to access the next one. Control flow instructions specifically instruct the Program Counter to continue execution at the specified target address. The simplest of these is the unconditional jump, JMP target, which always forces a jump to the instruction at the target address. More complex are conditional branches usually relative to the current one (i.e. back 4 instructions). These instructions are either conditional or unconditional. The unconditional is known as a BREAK, and simply specifies the address to jump to. The conditional first carries out a test, and decides whether to branch based on the results of that test. An example of this is “break if zero”, or BRZ. The BRZ command will specify both a register and a target address. If the specified register has a

value equal to zero, then the Program Counter will branch to the target address; and if not, it will simply increment by one to the next instruction. The conditional branch is notable because it is what allows for any form of conditional behavior!

Development Process: Designing the CPU

Before I set about researching how to develop a CPU, my first step was to determine whether it was possible. This question was easily solved; searching online yielded no less than 15 different implementations of CPUs onto FPGAs. Confident that the project was possible, I defined some basic parameters for the CPU. It was to be fully 16-bit, meaning that addresses, data and instructions would all be 16 bits long. In addition, it would use the Harvard architecture, because it is easier to understand in the form of a visual diagram. Design decisions were to be driven by a desire to make the CPU as simple and understandable as possible.

I then decided to use one of the online VHDL implementations I had found as a guide for understanding the CPU. More specifically, the blog of Francesco Balducci, an Italian embedded software developer. On his blog I found a detailed overview and implementation of "XENTRAL", a 32-bit *Harvard Architecture* CPU using VHDL[10]. Reading through Mr. Balducci's blog, I began sketching out a simplified version of his implementation as a rough guide. However, I became very bogged down in his write-up, which was based on confusing schematics that left me stumped. My next approach was to comb through his actual VHDL code, and see if I could piece together his approach by tracking the various signals in the design. Unfortunately this too was unsuccessful.

Having reached a dead end, I met with Professor Millet and at his suggestion, purchased two Schaums guides on computer architecture and assembly. In reading through these, I realized that my approach to designing a CPU had been completely backwards. Before diving deep into the building stage, and learning by example, I needed to learn assembly. Up to this point, I had assumed that assembly language was a nebulous concept that I could focus on after developing the CPU. Moreover, I assumed that by building a simple CPU, I

would be able to take any given assembly code and run it on the CPU. The crucial turning point was understanding that I needed to learn assembly, identify a few basic instructions to implement, and design an Instruction Set Architecture (ISA) that would translate these instructions to binary machine code – because the ISA, once defined, is what informs the design of the rest of the processor.

To learn assembly, I purchased a textbook on Computer Architecture called *Assembly Language for x86 Processors* and started reading.[11] I identified that there are three main categories of assembly functions – data manipulation, data storage, and control flow. As a starting point, then, I decided to design a CPU that supported the basics of each category:

- Data manipulation: addition, subtraction, loading to registers, and bitwise logic functions – NOT, AND & OR
- Data movement: Writing to, and reading from RAM
- Control flow: unconditional jump to new instruction, and conditional jump

These basic functions are enough to guarantee the Turing completeness of the processor, which means that it is equivalent to a Universal Turing Machine. In other words, the processor is able to compute anything that is computable, or for which there exists an algorithm that results in a solution in a finite number of steps.

Defining the Instruction Set

Eventually, I came across a series of lectures from CS 231 at the University of Illinois Urbana-Champaign that were extremely helpful in developing the instruction set.[12] I decided to use their instruction set with a few changes, also borrowing elements from a presentation at a German hackers conference.[13]

Loading Registers: Immediate Value

A LDI instruction loads a register with an immediate value specified in the assembly instruction. LDI instructions are of the format

```
LDI      R1, #1       ; R1 <- '1'
```

Add and Subtract

Addition and subtraction (ADD and SUB respectively) can use either register operands or constant operands; like with LDI, constant operands in decimal are denoted with a hash mark. (As mentioned in previous chapter, the $ denotes hexadecimal and the % denotes binary.)

```
ADD      R3, R1, R0   ; R3 <- R1 + R2
SUB      R3, R1, #1   ; R3 <- R1 + '1'
```

Bitwise Logic Instructions

AND, OR and NOT can also use either register operands or constant operands. These should be specified in binary using the %.

```
AND      R3, R1, R2      ; R3 <- R1 AND R2
OR       R3, R1, %1001   ; R3 <- R1 OR 1001₂
NOT      R2, R1          ; R2 <- NOT(R1)
```

Loading Registers: From RAM

A LOAD instruction copies data from a RAM address to one of the registers. The RAM address must come from one of the registers. LOAD instructions are of the format

```
LOAD     R0, (R1)        ; R0 <- M[R1]
```

Storing From Registers to RAM

A STORE instruction copies data from a register to an address in RAM. The RAM address must come from one of the registers.

```
STORE   (R0), R1        ; M[R0] <- R1
```

Control Flow Instructions: Jump and Branch

We covered the use of jumps and conditional branches in the previous chapter (see Control Flow Instructions, pg. 19). We will use one unconditional jump – JMP – and the conditional "branch if zero" (BRZ). A JMP instruction always changes the value of the Program Counter, whereas BRZ will only jump to the target address if the specified register's value is equal to zero. Both instructions will use Program Counter-relative addressing – meaning that the target address is relative to the current one. So a value of #-4 would jump back 4 instructions, and a value of #2 would jump 2 ahead. The syntax of both instructions is illustrated below.

```
JMP     #2          ; Jump ahead 2 instructions
BRZ     R0, #2      ; Jump ahead 2 instructions if R0 == 0
```

These ten instructions, taken together, make up our entire instruction set architecture. The next logical step after defining the ISA is to define the rules of the assembler – that is, how each instruction should be converted to machine language. However, the structure of the machine language is driven by what is required from the Control Unit. First I had to make a Datapath, and figure out how it would execute each of the instructions defined in the ISA. The UIUC lecture slides helped me take this ISA and define a datapath that would execute the required instructions

Building the Datapath

The datapath of a processor has three main components: the Arithmetic Logic Unit, a set of registers, and a block of RAM memory. Of the instructions in the ISA, the datapath is responsible for executing data manipulation and data movement.

Defining Behavior of the Arithmetic Logic Unit

The ALU is the part of the processor that performs computations. Our ALU is fed two 16-bit data inputs, A and B, as well as a selector signal OP, which picks the function to be used on the inputs. ALU functions vary from processor to processor; ours are based on the data manipulation instructions in the ISA, as well as "passthroughs" for the two inputs. I have chosen basic arithmetic, bitwise logic and pass-through functions that are illustrated in Figure 7 below.

Operation	OP Code	Description
Bitwise NOT	0000	Out = A NOT B
Bitwise AND	0001	Out = A AND B
Bitwise OR	0010	Out = A OR B
Addition	0011	Out = A + B
Subtraction	0100	Out = A - B
Passthru A	0101	Out = A
Passthru B	0110	Out = B

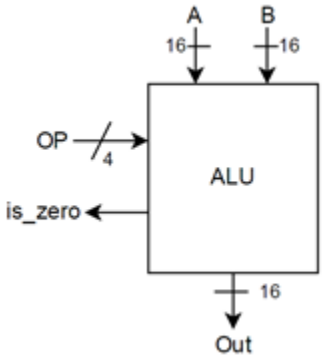

Figure 7: Behavior and Operations for ALU

In the ALU, the two 16-bit inputs are fed to digital circuits for each operation, and we create a signal "OP" that selects the appropriate operation according to the table above. OP is the first signal that we will need from the Control Unit. In addition, I included an is_zero output that can be used by the Program Counter for the BRZ instruction.

Defining Behavior of RAM

My RAM design is a modified version of XENTRAL's RAM.[14] Behaviorally, our RAM must be able to do two things: for a given address, either write or read data. Thus, the RAM must have an address input, a data input, a selector input, and a data output. The selector input tells the RAM whether it is writing to or reading from the memory. It is important to note that the RAM is a clocked module, for this will have an effect on the number of clock cycles required to complete memory read/write instructions. The size of the RAM is unimportant; we will address that during implementation.

The Registers

Understanding how the registers relate to the datapath and the rest of the processor was one of the largest pain points of the project. When attempting to understand XENTRAL's design, it continually referenced specialized registers such as the Memory Address Register, Memory Block Register and the Stack Pointer. While all of these are extremely important in modern-day processors, I felt that it would be better to do without them if possible. So, the processor instead has eight general registers, numbered R0 through R7. Each register is clocked, and has the structure illustrated in Figure 8. As a clocked cycle, data input is written to the data output each clock cycle if load enable is equal to 1.

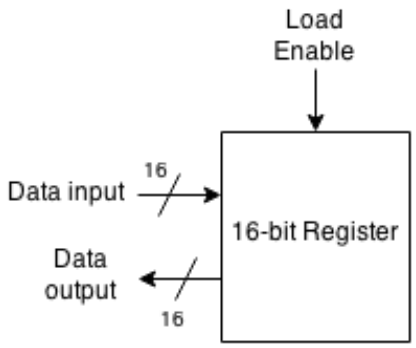

Figure 8: 16-bit register. The input data is can be stored to the register by setting Load Enable to 1

To determine how the registers fit in with the ALU and RAM, we can run through the different assembly instructions and what they require from the datapath.

3-Register Instructions: ADD, SUB, AND, OR

3-register instructions specify an ALU operation, a destination register, and two source registers:

```
ADD   R3, R1, R0   ; R3 <- R1 AND R2
```

Thus, we need a way to select R1 and R2 to send to the ALU, and write the output to R3. We can do this by indexing the registers in a sort of "Register File" with two outputs to the ALU and one input from the ALU. There must be signals from the control unit to determine which registers are read from and written to.

Constant Operand Instruction: LDI, ADD, SUB

Instructions with a constant operand operate like 3-register instructions above, however they will have a constant that must be accounted for:

```
LDI      R1, #1       ; R1 <- '1'
```

Thus we need a way to insert constants specified by the control unit into the ALU. We can do this using a 2-to-1 MUX on one of the ALU's inputs that selects between the Register File and the constant. We will do this on ALU input "bus B".

Writing to RAM

The STORE command requires the contents of one register be sent to the address input of the RAM, and the contents of another sent to the data input:

```
STORE   (R0), R1      ; M[R0] <- R1
```

Thus there must be a connection between the two outputs of the Register File and the two inputs of the RAM.

Reading from RAM

The STORE command sends the output of one register to the address input of the RAM, and writes the resulting RAM output to a second register.

```
LOAD  R0, (R1)       ; R0 <- M[R1]
```

Thus, we must be able to select whether the data input to the Register File is coming from the ALU versus the RAM. We can do this with a 2-to-1 mux selector.

Putting the Datapath Together

Taking into account the different use cases of the datapath above, we can construct a resulting data structure that can successfully execute all the instructions in our architecture. We will use what is called the three-bus structure, where every execution cycle bus A and B are driven from the registers and constants, and bus C is routed back to the registers. We also need to get the following signals from the Control Unit:

- Selectors for the Register File input and two outputs
- Write enable for the Register File for commands that require writing to a register
- Signal to alert when constant operand instruction
- Signal carrying the constant operand
- RAM mode signal to select between writing to and reading from RAM
- Signal OP to select proper ALU function
- Signal to select between ALU and RAM for Register File input

These signals, as well as the resulting Datapath structure, are noted below:

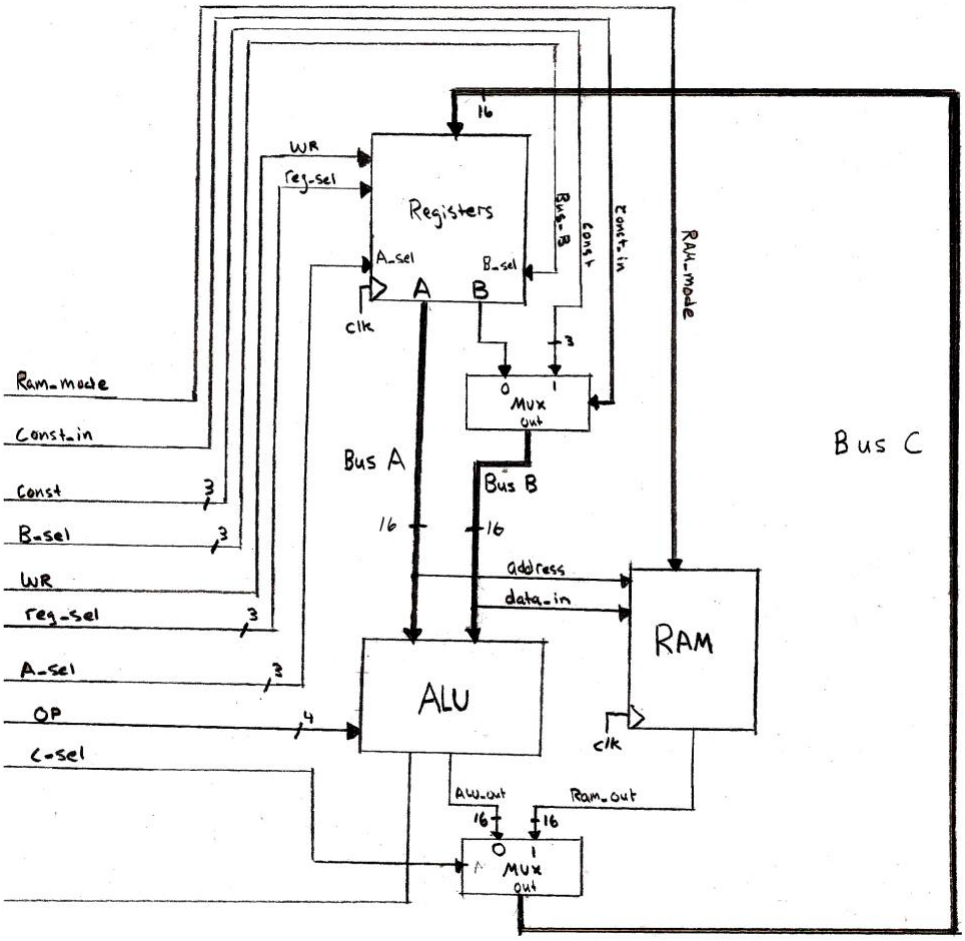

Figure 9: Circuit Diagram of Datapath with Required Signals

The required signals from the control unit for each type of instruction are summarized below in Table 1. The values of certain signals do not matter for some instruction types; these situations are noted with x's.

Instruction Type	Example	A_sel	B_sel	C_sel	WR	Reg_sel	const	const_in	ram_mode
Register ALU	ADD R3, R1, R2	001	010	0	1	011	xxx	0	0
Constant Operand ALU	LDI R4, #5	xxx	xxx	0	1	100	101	1	0
Memory Load	LOAD R1, (R5)	101	xxx	1	1	001	xxx	0	0
Memory Read	STORE (R5), R1	101	001	x	0	xxx	xxx	0	1

Table 1: Instruction Types for Execution Unit

With the datapath defined, our next step is to examine the Program Counter, which is also driven by the Control Unit.

Defining the Program Counter

The Program Counter is responsible for the last category of instructions in our ISA: control flow, with unconditional jump JMP and conditional break BRZ.

For data manipulation, which takes one clock cycle, the program counter will increment after each clock cycle. For data movement, which takes two clock cycles because the RAM is a clocked process, the program counter will increment after two clock cycles. For unconditional jumps, the Program Counter will jump to the specified relative address; this instruction only requires one clock cycle. For BRZ, the Program Counter will wait one clock cycle for the datapath to check the register, and on the next clock cycle use the signal is_zero from the datapath to determine if it should break to the given address.

To execute these actions, the Program Counter must be given the relative address value for all control flow instructions, an is_zero value specifically for BRZ, and a way to know that a jump or break command has been given. We will call these three signals jmp_value, is_zero, and instr_type respectively. The Program Counter with these required signals is represented below.

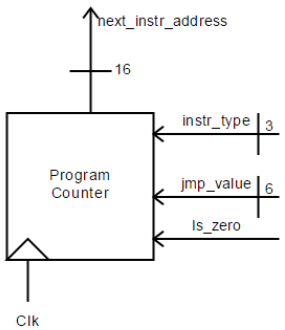

Figure 10: Signal Diagram of Program Counter

Full CPU Circuit Diagram

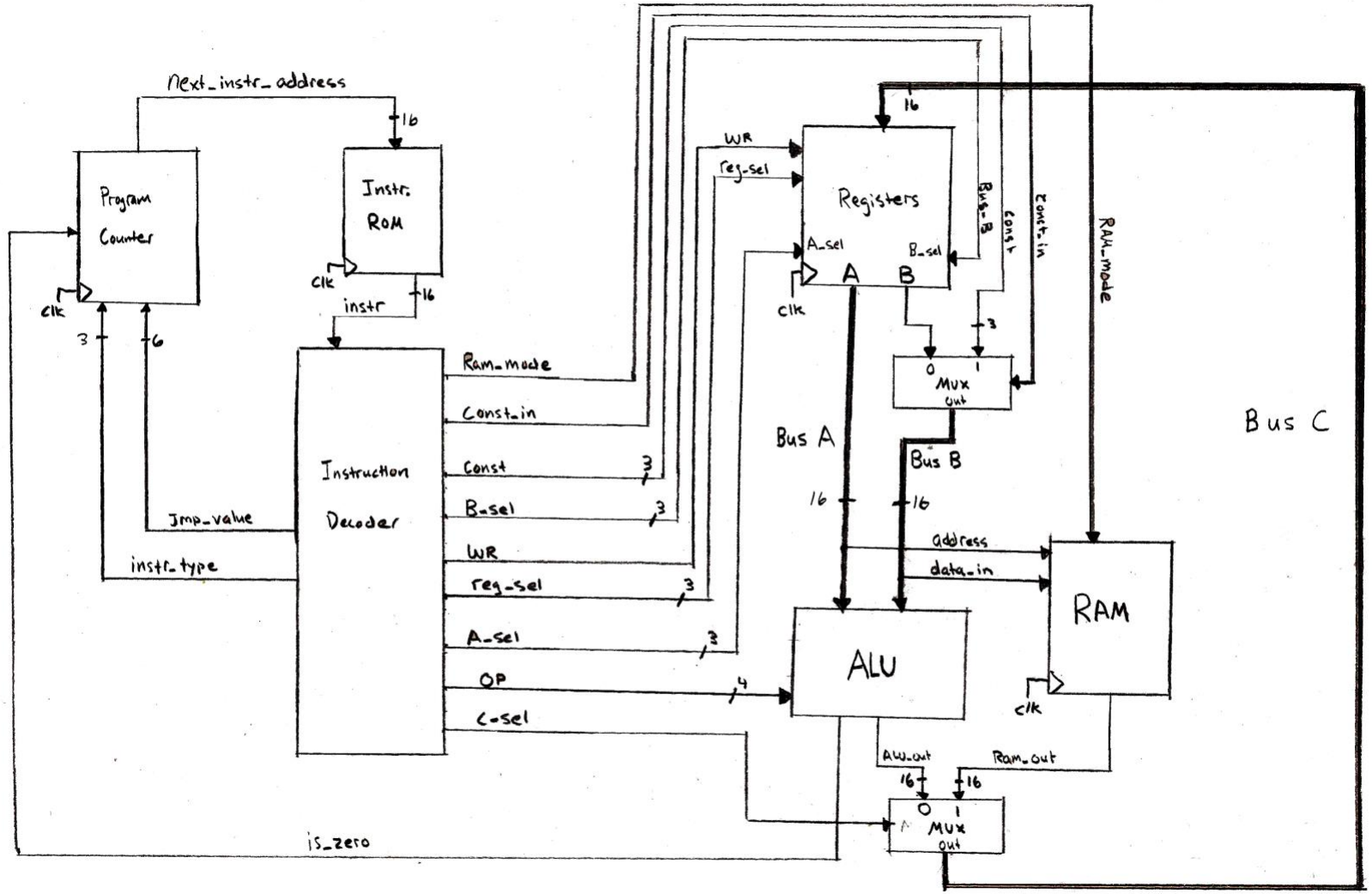

Figure 11: Full Circuit Diagram for CPU

Defining Our Instruction Set Architecture

Now that we know what signals are needed by the processor for each type of assembler instruction, we can go about coding them into machine language. As this is a 16-bit CPU, each of our instructions is coded into a 16-bit vector. Thus, we need to define rules that convert each line of assembly into machine language. We begin by defining the rules for each type of instruction.

First, since there are six different instruction types, we will reserve the first three bits to specify what type of instruction the decoder receives. We will define each type with Table 2:

Table 2: Instruction types and their corresponding Opcode designations

Instruction Type	
Description	**Opcode bits 15-13**
Register Format ALU	000
Const Operand ALU	001
Memory Write	010
Memory Read	011
Conditional branch (BRZ)	100
Unconditional Jump	101

Now we will go over the specific syntax of each instruction type.

Case 1: Register Format Instructions

Register format instructions contain four pieces of information: the required ALU function, the output register, and two input registers. To represent the ALU function, we can use the same 4-bit OP code defined in the ALU on page 14. We also need to reserve the first three bits to specify the instruction type. This leaves bits 8 through 0 for representing the three registers. Thus, each register gets three bits, meaning we can have a maximum of 8 or 2^3 registers in our design.

We will code register format instructions as follows:

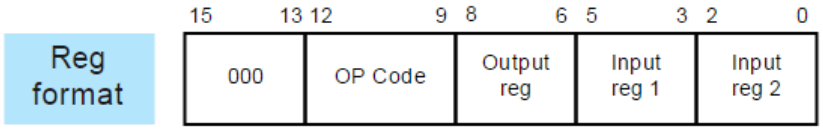

Figure 12: Instruction syntax for Register ALU operations

Case 2: Instruction with Constant Operand

Instructions with a constant operand should have the same format as register instructions; however the last three bits should be replaced by the constant operand instead of the second input register.

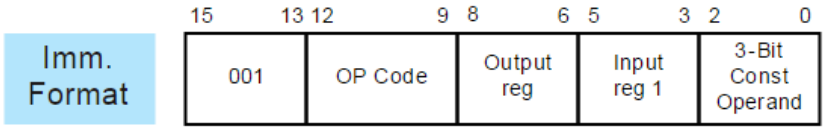

Figure 13: Instruction syntax for Immediate ALU operations

There is also a specific syntax for the LDI command, which always uses the OP code corresponding to "B pass-through" so that registers can be loaded with the given constant operand:

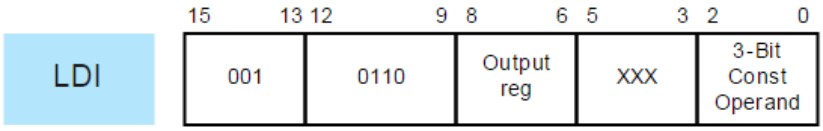

Figure 14: Instruction syntax for LDI (load immediate)

Case 3: Memory read and write

Memory read and write operations specify just two registers – one containing the RAM address, and the other that will either be read from or written to.

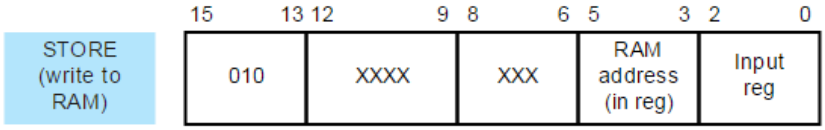

Figure 15: Instruction syntax for STORE (write to RAM)

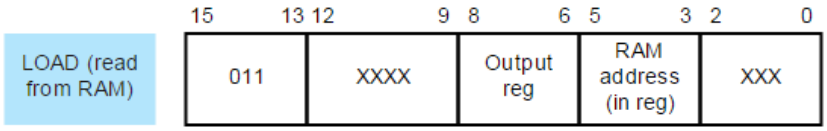

Figure 16: Instruction syntax for LOAD (read from RAM)

Case 4: Branch or Jump

Branches and jumps should have a different structure because they contain a relative address comprised of six bits. Since we have been coding the source registers as the last six bits of the instruction, we will do the same here.

BRZ uses the specific ALU OP code for "A pass-through" so that the input register can be examined to determine whether it is equal to zero.

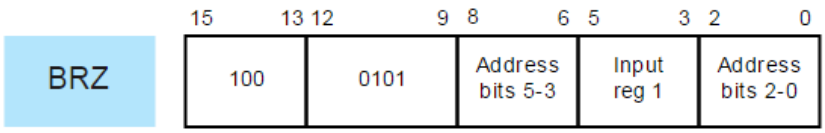

Figure 17: Instruction syntax for BRZ

The jump instruction, JMP, solely requires an address.

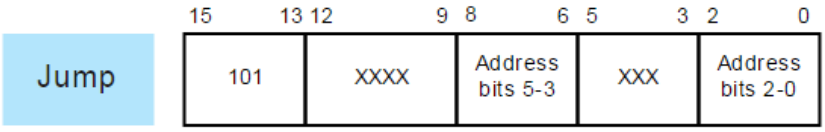

Figure 18: Instruction syntax for JMP

Putting it all together: The Instruction Decoder

With our assembly ISA decoded to machine language, we can now go about linking the different parts of each machine language instruction to the appropriate signals to drive execution in the datapath and program counter – thus completing the link between instruction and execution.

The first three bits of each instruction denotes the instruction type. The Instruction Decoder routes signals in six different ways depending on the instruction type. For datapath instructions, actions are represented in the table below.

Table 3: Instruction Routing Signals for Datapath

Instruction Type		Resulting signals to route							
Description	**Opcode bits 15-13**	**Reg_sel**	**A_sel**	**B_sel**	**C_sel**	**WR**	**const**	**const_in**	**ram_mode**
Register ALU	000	dest reg	src reg 1	src reg 2	0	1	xxx	0	0
Const operand ALU	001	dest reg	src reg 1	xxx	0	1	const operand	1	0
Memory Write	010	xxx	RAM address	input reg	1	1	xxx	0	0
Memory Read	011	dest reg	RAM address	xxx	x	0	xxx	0	1

Likewise, control flow instructions for the program counter are below as well:

Table 4: Instruction Routing Signals for Program Counter

Instruction Type		Resulting signals to route	
Description	**Opcode bits 15-13**	**jmp_value**	**instr_type**
BRZ (branch if zero)	100	address bits 5-3 & 2-0	100
Unconditional Jump	101	address bits 5-3 & 2-0	101

These two tables of instruction types and resulting signal routing make up the logic of the Instruction Decoder. Implementation into gates can be complex; however the ISA is actually designed to simplify this process. This is due to the presence of x's, or "don't cares" for many of the signals.

Thus, if we were to wire Reg_sel, A_sel and B_sel directly to bits 8-0 of the instruction, they would route the appropriate register values for register ALU functions, *and* the appropriate register values for RAM read and write operations. This is because we formatted the memory read and write instructions such that the address is specified with bus A, and the data to write with bus B. And for constant operand ALU instructions, register values for the destination and src1 are routed appropriately, whereas the value of B_sel is a don't care because of the subsequent mux where we inject the constant operand value. For BRZ, wiring bits 5-3 of the instruction to the source register feeds that register's value to the ALU along input A, where the ALU is programmed with OP code 0101 to determine if A is equal to zero.

Using these techniques, the logic of the Instruction Decoder is actually quite simple; however, we need not go into these details for programming the functionality using VHDL. Rather, we can just describe the desired logic at the level of Tables Table 3 and Table 4, and let the synthesizer do the simplification and optimization for us.

Development Process: Implementing the CPU with VHDL

The process of implementing the CPU as developed in the previous section was a complex task. Rather than walking the reader through the entire implementation, I will instead focus on some key aspects of the language and how certain parts of the circuit – such as muxes – are represented.

To start, we will examine the basic structure of VHDL modules.

Basic VHDL Code Structure

A module of VHDL that describes a circuit's structure can be broken down into two parts: the entity declaration and the circuit's architecture.

The entity declaration gives the circuit a name, and defines its inputs and outputs. For example, consider the example below:

```
entity is_equal is

        Port (    input1          : in std_logic;
                  input2          : in std_logic;
                  output1         : out std_logic);
end is_equal;
architecture Behavioral of is_equal is
-- Define components (if needed)
-- Define internal signals (if needed)
begin
--This circuit outputs '1' if the inputs are equal, and '0' if not
        process(input1, input2) begin
                if input1 = input2 then
                        output1 <= '1';
                else
                        output1 <= '0';
        end process;
end Behavioral;
```

In the above circuit, we can see the circuit is defined as a black box with two inputs and one output in the entity declaration. This is equivalent to the diagram below.

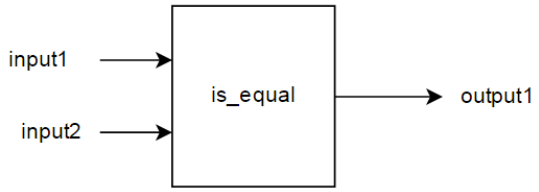

Figure 19: Diagram of example "is_equal" VHDL code

The next part of the code is the architecture. The architecture is divided into two sections, before and after the "begin." The circuit's signals and components are defined above, and below is the circuit's logic. In our simple example, we do not have any components or signals. The circuit's logic is organized into processes. Each process has a sensitivity list, which spells out the inputs that drive the process's logic. In this example, there are two binary inputs, and one binary output. The logic is formatted such that the output is dependent entirely on the input: if the two inputs are equal, the output is a '1'. If not, the output is a '0'. This is an example of combinational logic – where the output is solely dependent on the given inputs.

Next we will inspect the VHDL for an example of combinational logic. We will consider the simple 16-bit register, eight of which are used in the datapath.

```
entity simple_register is
    Port ( clk      : in  STD_LOGIC;
           data_in  : in  STD_LOGIC_VECTOR (15 downto 0);
           load_en  : in  STD_LOGIC;
           data_out : out  STD_LOGIC_VECTOR (15 downto 0));
end simple_register;

architecture Behavioral of simple_register is

begin

process(clk) begin

        if rising_edge(clk) then
                if load_en = '1' then
                        data_out <= data_in;
                end if;
        end if;
```

```
end process;

end Behavioral;
```

Note in this simple register that the logic is clocked – the register will only load a value if both load_en = '1' *and* there is a rising edge of the clock.[2] In addition, the std_logic_vector is a vector, or array, of multiple std_logic elements. So the type declaration "std_logic_vector(15 downto 0)" defines a 16-bit vector of std_logic elements, where the bit positions within the vector are numbered from 15 to 0. The downto keyword defines the ordering of the elements within the vector, indicating that element 15 is in the leftmost position. The statement "data_out <= "1000000000000000"", then, would assign the value 1 to position 15 of the vector data_out and the value 0 to the remaining 15 positions.

Now that we have gone over a few simple examples of digital components, we will examine how to wire components together to create a digital system. In VHDL, a digital system is defined using a top-level entity that can contain other entities as components. Below is the syntax of a top-level entity implementing a NAND gate.

```
entity NAND_gate is
        Port (    in1  : in std_logic;
                  in2  : in std_logic;
                  out1 : out std_logic);
end NAND_gate;

architecture Behaviorial of NAND_gate is

        component AND_gate is
                Port( and_in1 : in std_logic;
                          and_in2    : in std_logic;
                          and_output : out std_logic);
        end component;

        component NOT_gate is
                Port(    not_in   : in std_logic;
                          not_out : out std_logic);
```

[2] The rising edge of the clock is used for all sequential logic; using the rising edge as a condition simply means that every clock cycle, all sequential logic is updated simultaneously.

```
        end component;
signal sig1 : std_logic; -- internal wire between AND and NOT gate
begin
AND_1: AND_gate
        port map (              and_in1     => in1,
                                and_in2     => in2,
                                and_output  => sig1);
NOT_1: NOT_gate
        port map (              not_in      => sig1,
                                not_out     => out1);
end Behavioral;
```

In this example, note how the architecture starts by declaring what components the design will be using. These components must be defined elsewhere in the description's file or another file. Each component declaration must define the inputs and outputs of each component, and those inputs and outputs must match the component's entity declaration exactly. Now consider the diagram in Figure 20 for the NAND gate we just described.

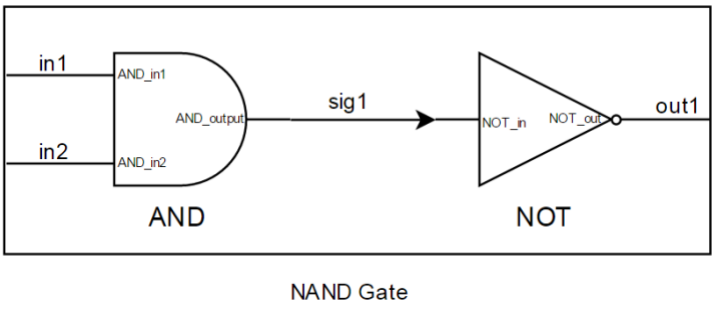

Figure 20: Diagram of example NAND Gate VHDL implementation

Note how the inputs and and outputs of each component are linked. Looking specifically at the AND, its two inputs connect to in1 and in2, while its output connects to the internal *signal* sig1 that links it to the subsequent NOT gate.

The Source Code of the 16-bit Processor

The source code of the 16-bit processor is implemented such that that the ALU is purely combinational, non-clocked logic. While the Instruction Decoder *can* be coded with combinational logic, my implementation is clocked because it allows for the code to be

much clearer. Finally, the instruction ROM, data RAM, Program Counter and registers are all implemented using sequential logic. The Instruction Decoder and Program Counter are contained within a "control_unit" module, while the registers, ALU, RAM and muxes defined in Figure 9 are contained within a datapath module. These two modules are then wired together in a top-level entity to complete the design of the CPU. Signal names in the source code correspond to those shown in Figure 11, the illustrated overall circuit diagram. In addition, the source code is commented to aid in understanding.

We will now review a few key strategies I employ in VHDL, and how they translate to the behavior defined in the previous section.

The Multiplexer, or mux

The multiplexer is a logical component that has a number of inputs and one output. A control signal is used to select which input is sent to the output. For a mux with 2^n inputs, we will have an n-bit selector signal. In the datapath, we use 2-to-1 muxes to select between the ALU and RAM for Bus C, and also to insert a constant operand. These muxes have 1-bit select signals. We use 8-to-1 muxes for selecting the one input and two outputs of the register file; these control signals are Reg_sel, A_sel and B_sel respectively. Muxes are implemented into VHDL in the following format:

```
with control_signal select mux_output <=
        input_1 when "00",
        input_2 when "01",
        input_3 when "10",
        input_4 when "11";
```

This notation is frequently used in the source code, especially in the datapath.

The "Register File" – an array of registers

The Register File is implemented into VHDL by instantiating an array of eight registers, and wiring each of their outputs to an array of 16-bit std_logic_vectors. Bus C is wired directly to the inputs of each register, and the load_enable signal is extracted from an 8-bit reg_en array. The value of Reg_sel is sent to an 8-bit mux that converts the binary value to an 8-bit one-high signal of the form "00000001" for R0, "00000010" for R1 and so on. If WR (write to register) goes high, then this 8-bit signal is split up and sent to each register. R7 gets the left-most digit; R6 gets the one to the right, and so on down to R0. In the one-high format, the register specified by Reg_sel will be written to, while the other registers retain their values. This process is illustrated in Figure 21 below.

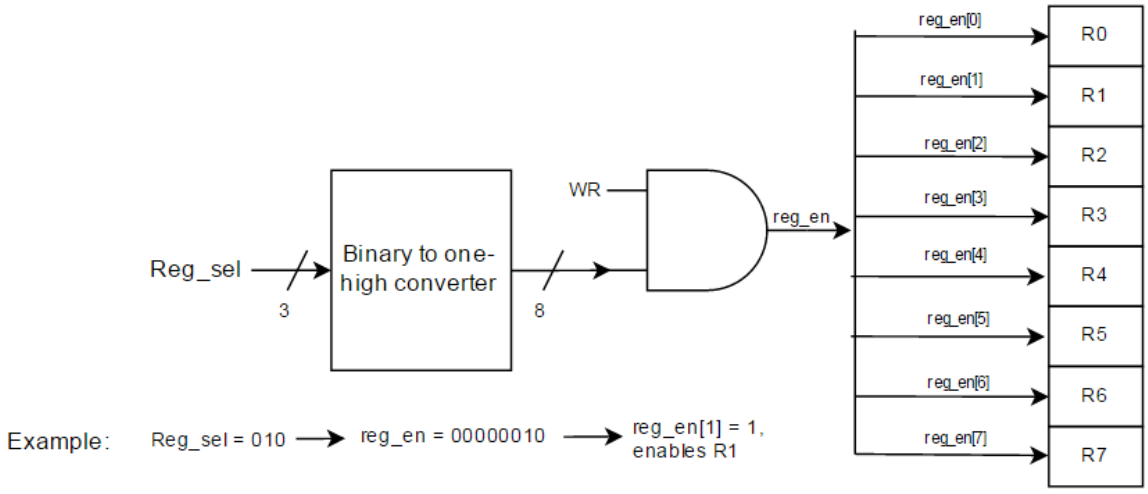

Figure 21: Illustration of writing to Register File

The Case Statement

The case statement is a means of describing sequential logic. The case statement takes an input, and describes behavior to be executed for every possible value of the input. The processor uses two case statements: one for the Instruction Decoder, and one for the

Program Counter. The Instruction Decoder uses a case statement based on the first three values of each instruction. The signals that it routes are different for the different instruction types; consider the case statement used for the Instruction Decoder below:

```
InstructionDecoder: process(clk, instr, instr_type) begin
        case instr_type is
                when "000" => -- register-format ALU operation
                        constant_in  <= '0';
                        C_sel        <= '0';
                        WR           <= '1';
                        RAM_mode     <= '0';
                        OP           <= instr(12 downto 9);
                when "001" => -- Immediate ALU operation
                        constant_in  <= '1';
                        C_sel        <= '0';
                        WR           <= '1';
                        RAM_mode     <= '0';
                        OP           <= instr(12 downto 9);
                when "010" => -- memory write from registers
                        constant_in  <= '0';
                        C_sel        <= '0';
                        WR           <= '0';
                        RAM_mode     <= '1';
                        OP           <= "1111";
                when "011" => -- memory read to registers
                        constant_in  <= '0';
                        C_sel        <= '1';
                        WR           <= '1';
                        RAM_mode     <= '0';
                        OP           <= "1111";
                when "100" => -- conditional branch
                        constant_in  <= '0';
                        C_sel        <= '0';
                        WR           <= '0';
                        RAM_mode     <= '0';
                        OP           <= "0101"; -- register to test will
be selected with A_sel; pass thru to chk for 0
                when "101" => -- jump
                        constant_in  <= '0';
                        C_sel        <= '0';
                        WR           <= '0';
                        RAM_mode     <= '0';
                        OP           <= "1111";
                when others =>
                        constant_in  <= '0';
                        C_sel        <= '0';
                        WR           <= '0';
                        RAM_mode     <= '0';
                        OP           <= "1111";

        end case instr_type;
end process InstructionDecoder;
```

If you compare the assignments for each state to Table 3 and Table, you will see that they are identical other than setting “don’t care” values in the tables to zero.

Programming the RAM and ROM

There are several different ways to program RAM onto an FPGA. The Nexys3 board has two types of memory – block RAM and distributed RAM. Block RAM consists of dedicated RAM areas in the FPGA. Distributed RAM refers to VHDL logic that is configured as RAM. Rather than directly instantiating RAM directly, I followed the approach of XENTRAL and used a behavioral description of the RAM blocks – leaving it up to the FPGA synthesis tools to choose how to implement it. The heart of the module is a clocked process that, on the rising edge, either writes to or reads from the memory for a given input address.

The instruction ROM was written as a simple array of instructions at consecutive 16-bit addresses, starting at address zero. However, this means that in order to load a program into the instruction memory, one must do so by specifying the instructions in the VHDL module rather than while the processor is running on the board. Thus, once the processor works in simulation, the next step is to convert the ROM to dual-port RAM and write an instruction loading program for the processor. Essentially, instructions would be transmitted via serial cable and stored in the RAM sequentially. With the push of a button, the processor would be allowed to execute, and another button would halt the processor. I am currently in the process of coding this capability. However, it is not yet ready for demonstration.

Experimental Results

To demonstrate the functionality of the CPU, I will use a VHDL "testbench," which allow us to step through the logic and verify that the CPU is properly executing a set of instructions.

We will validate the CPU by observing it step through the following assembly instructions:

```
LDI     R1, #3       ; R1 <- '3'
LDI     R0, #1       ; R0 <- '1'
ADD     R4, R1, R0   ; R4 <- R1 + R0
STORE   (R2), R4     ; M[R2] <- R4
LOAD    R3, (R2)     ; R3 <- M[R2]
BRZ     R5, #2       ; IF R5 = 0, INSTR[PC] JUMPS 2
ADD     R6, R3, #5   ; R6 <- R3 + '5'
JMP     #-5          ; Jump back 5 instructions
```

Keeping in mind that all registers are initialized at value zero, we should observe the following sequential behavior:

Instruction Address	Assembly Instruction	Observed Behavior
0	`LDI R1, #3`	Write '3' to register 1
1	`LDI R0, #1`	Write '1' to register 0
2	`ADD R4, R1, R0`	Write R0+R1=4 to register 4
3	`STORE (R2), R4`	Since R2=0, write '4' (value of R4) to memory address 0
4	`LOAD R3, (R2)`	Again since R2=0, write '4' (value of mem address 0) to register 3
5	`BRZ R5, #2`	Since R5=0, this conditional branch should *skip* the next instruction
6	`ADD R6, R3, #5`	This instruction should be skipped
7	`JMP #-5`	The program should jump back 5 instructions to address 2.

Discussion

The idea for this project came on September 4th, when I was determined to complete a Senior Thesis on something related to my courses last year in electrical engineering at Dartmouth. I noted that Professor Collett had done research into developing electronics using Verilog, and mentioned to him my use of VHDL last spring. Together, we came up with the idea of developing a simple, pedagogically-focused CPU using VHDL and implementing it onto an FPGA. We further defined the project such that it would accept program instructions in machine language, and execute them by interfacing with the RAM. While the development process for the CPU was long and had many wrong turns, the overall goal has remained the same since day one: produce something that is easy to understand.

By building a CPU, and especially attempting to explain it in this report, I have gained new familiarity with computer architecture, and especially in how it differs from a specialized, integrated circuit.

There are several next steps that this project could take. I am extremely close to connecting the processor to a serial interface so it can be loaded with programs written in machine language by executing a python script. Once this is working, I want to wire the registers to a VGA output so that one can tangibly observe the execution of instructions. By building out the I/O of the processor, I would make the process of programming it much simpler.

After building out the I/O, it would be extremely interesting in writing a python script that acts as an assembler for the instruction set architecture. Chaining a python script together with serial output to the FPGA would enable programming the ROM on the fly, allowing for further educational value. In addition, those familiar with python would be able to see how the assembly is converted to machine language – something covered in the report, but not implemented using VHDL.

Expanding the instruction set is another possible next step; as of now it is extremely simple and while Turing complete, not something that could run any type of operating system. The funny thing about expanding the instruction set, though, is that almost every change requires a complimentary tweak in the design of the processor. You can't really change one without the other.

This project has been a fantastic learning process, and I hope to continue building out the capabilities of this CPU going forward. Finally, I hope that the architecture laid out in this report proves helpful for the reader looking to understand the basics of a CPU!

---

[1] Turing, Alan Mathison. "On computable numbers, with an application to the Entscheidungsproblem." *J. of Math* 58 (1936): 345-363.
[2] Bezakova, Ivona. "Introduction to Turing Machines." *CS380 Fall 2009*. Web. 10 Dec. 2014. <http://www.cs.rit.edu/~ib/Classes/CS380_Fall09/Slides/082-TuringMachines.pdf>.
[3] http://www.ipod.org.uk/reality/reality_turing.gif
[4] Barker-Plummer, David, "Turing Machines", *The Stanford Encyclopedia of Philosophy* (Summer 2013 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/sum2013/entries/turing-machine/>.
[5] "von Neumann ... firmly emphasized to me, and to others I am sure, that the fundamental conception is owing to Turing—insofar as not anticipated by Babbage, Lovelace and others." Letter by Stanley Frankel to Brian Randell, 1972, quoted in Jack Copeland (2004) The Essential Turing, p22.
[6] Shurkin, Joel (1996). Engines of the mind : the evolution of the computer from mainframes to microprocessors. New York: Norton. ISBN 0-393-31471-5.
[7] Clements, Alan. *The Principles of Computer Hardware*. Oxford, OX: Oxford UP, 1985. 11-12. Print.
[8] Enticknap, Nicholas (Summer 1998), "Computing's Golden Jubilee", Resurrection (The Computer Conservation Society) (20), ISSN 0958-7403, retrieved 19 April 2008
[9] Napper, R. B. E., Introduction to the Mark 1, The University of Manchester, retrieved 4 November 2008
[10] Balducci, Francesco. "Zero Characters Left." *: XENTRAL: A Simple FPGA CPU*. N.p., n.d. Web. 15 Dec. 2014. <http://zerocharactersleft.blogspot.nl/2012/07/xentral-simple-fpga-cpu.html>.
[11] Irvine, Kip R. *Assembly Language for X86 Processors*. Upper Saddle River, NJ: Pearson Education, 2015. Print.
[12] Garzaran, Maria. "CS 231 - Spring 2008." *CS 231 - Spring 2008*. University of Illinois at Urbana-Champaign, 2008. Web. 15 Dec. 2014. <http://www.cs.uiuc.edu/class/sp08/cs231/lectures/>.
[13] Meyer, Dominik. "Introduction to Processor Design." *C3TV* -. C3TV, n.d. Web. 5 Nov. 2014. <http://media.ccc.de/browse/congress/2013/30C3_-_5443_-_en_-_saal_g_-_201312281830_-_introduction_to_processor_design_-_byterazor.html#video&t=1753>.
[14] https://github.com/drxzcl/xentral