GROUP

1-4

CSSE 232 FINAL PROJECT REPORT

Adam Westhusing
Jeremy Fox
Jason Hochstedler
Gabriel Golcher
Geoffrey Ulman

General Purpose Instruction Set Processor

DOCUMENT CONTAINING

Design and Process Documentation

11/9/03
Fall Term 2003-2004 School Year
Rose-Hulman Institute of Technology
Dr. Laurence Merkle, Instructor
G:\csse\csse232\0304a\turnin\team1-4\index.htm

Table of Contents

Section 1

Project Introduction and Executive Summary

Information Icons	3
Interrupt Handling	4
Java Assembler	4
Process and Documentation	4
Testing and Implementation	5
Field Programmable Gate Array	6
Documentation For Future Users	6
What's Next?	6

Section 2

Instruction Set Design

Beginning Stages: Instruction Types	7
Inventing a New Language	8
A Type Instructions	8
I Type Instructions	10
J Type Instructions	10
Creating a Standard	11
Argument and Return Registers	11
Zero Register	11
Button State Register	11
Machine Code Formatting	12
Putting it All Together	12
Java Assembler	13
Assembler Read-Me	13

Section 3

Design Implementation

Register Transfer Language Design	16
RTL For 2nd Clock Cycle	17
Data-path Design	17
FPGA Processor Re-Design	18
Individual Component Design	18

Section 4

Xilinx Model

Initial Process	22
Xilinx Testing Process	23

Section 5

Testing Methodology and Final Results

Error/Change Tracking Log	24
Systematic Testing Process	25
Making a Second Pass	28

Section 6

FPGA Implementation

FPGA Testing	29
Initial Testing and Setup	30
Online Documentation	30
Step-By-Step Guide	31
Implementing the Processor	36
Lessons Learned	37

Section 7

Conclusion

General Conclusions	38
Total Component Count	39
Processor Timing Information	42

Section 8

Individual Reflections

Geoffrey Ulman's ReflectionsError! Bookmark not defined.

Jeremy Fox's ReflectionsError! Bookmark not defined.

Gabriel Golcher's ReflectionsError! Bookmark not defined.

Adam Westhusing's ReflectionsError! Bookmark not defined.

Jason Hochstedler's Reflections Error! Bookmark not defined.

Section 1

Project Introduction and Executive Summary

This section gives an executive summary of the brainstorming, design, implementation, and testing steps which the group went through in order to develop a fully functional processor..

In its' final form, our processor is a 16-bit general purpose processor which uses a load-store architecture and contains 16 general purpose resisters. It has 14 instructions available to the programmer as well as one instruction used during hardware interrupts. Since 4 bit op-codes are used, this implementation leaves one instruction free to allow for easy expansion in further versions of the processor.

ICON KEY

See Appendix For Details

See Reference Files Attached

Interesting Extra Info.

The processor can handle inputs from up to 15 different one bit I/0 ports (in this context these correspond to 15 buttons the processor could be hooked up to) as well as one 16 bit input port (in this context the 4 bit input from the FPGA dip-switches is used).

Truly taking the "simplicity favors regularity" design technique to heart, our processor is designed to accomplish its stated tasks and functions with as little extraneous hardware as necessary. For example, our processor forgoes a shifting unit entirely, instead relying on repeated use of the addition function of the ALU to shift operands. Other design trade-offs which resulted from this overall design philosophy are discussed in later sections.

Information Icons

The icons listed in the Icon Key above are provided to categorize boxes containing extra information relevant to some aspect of the processor. The folder icon indicates that more information can be found in the appendix of this document. The disk icon indicates that there are additional electronic resources available (included in the e-

mail this document was sent in, or on our team folder). Finally, the book icon covers all other relevant side-information.

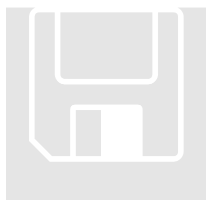
Two special purpose registers are provided in addition to the general propose ones. One special purpose register always holds the value of the dip-switches and can be used to input numerical data into the processor. The other is directly linked to the 16 bit output port which is intended to be connected to a seven segment display.

Interrupt Handling

The processor uses a unique interrupt handling scheme in which an "interrupt op-code" is specified. Whenever the control receives this op-code, it outputs the sequence of control signals necessary to handle the interrupt.

Whenever one of the I/0 ports becomes high for a clock cycle a high signal is stored in a 1-bit register which holds that signal until reset by the control. This signal cases the next instruction to be read from a static value containing the interrupt handling op-code instead of from memory. This op-code causes the processor to do a number of things: store the value of the program counter, jump to the instruction handling routine, disable interrupts, and reset the latch. A high value is also stored in the appropriate position in the button state register to indicate which specific button was pressed.

Java Assembler



Our team also chose to create a Java assembler to aid in our testing and verification process. The assembler can read an input file written in our assembly language and outputs a machine language file in either hexadecimal or binary with the specified number of words per line. It also handles the "load" pseudo instruction which allows an entire 16 bit immediate to be loaded into a register. More detailed instructions along with the program itself are available in our team folder and in the appendix of this report.

Process and Documentation

Much of our team's energy during the project went towards creating and maintaining good documentation. Not only did we keep a very thorough and well formatted Design Process Journal as required, we designated one team member each meeting to take "scratch" notes on everything that was said. These allowed us to create a very detailed and accurate Design Process Journal. We also designated one main person to keep the team folder and website up to date—this avoided confusion about which versions of our documents were most recent.

All our design documents as well as our Memos and Design Process Journal are included in the Appendix.

Having these documents turned out to be incredibly useful throughout the process. For example, during our team meetings with Dr. Merkle we brought the Design Process Journal with us and used it as a guide to remind us of questions and difficult design decisions that came up during our meetings.

At the end of each team meeting, we also recorded what each team member's assignments were and when the next group meeting would be. This ensured that work was constantly being done—thus there was no confused down time between meetings where no one really knew what they should be doing to move the project forward.

Testing and Implementation

This was by far the most exciting and rewarding part of the process. There is no feeling quite like taking a program written in an assembly language you designed, translating it to a machine language you designed using an assembler you wrote, and having the zeros and ones which are outputted actually make the processor you designed to what you specified in the assembly language. It definitely made all the hard work worthwhile.

However, in order to get to the point where this was possible, we had to bridge the gap between good documentation and actual implementation. Our process followed fairly straightforward steps:

1. We implemented and tested each component which we needed separately.

2. Once all the components were working, we connected them based on our data path design except we left the memory component out of the data path. Thus, instead of dealing right away with the added complication of instructions coming from memory, we inputted them manually from the test bench.

3. After testing a few instructions this way and correcting the main errors in the data path, we hooked up the memory component and wrote one "mini program" to test each of the 14 different instructions sequentially.

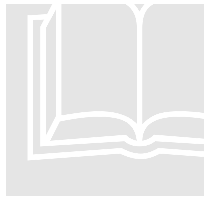
A Error Change Tracking Log was kept throughout this entire process. This was not required, but we found it extremely helpful to document every single fix that was made so all team members knew exactly what had been done. This document also contained a table listing the current testing status of each of our instructions. It also helped us keep track of outstanding issues which we needed to address and any changes that we made to the documentation that had to be cleared through Dr. Merkle. We feel that

this document was a very useful extra addition to the testing process and allowed our testing to proceed as quickly and as painlessly as it did.

Field Programmable Gate Array

Our team finished the implementation and testing of our processor approximately one week ahead of schedule and thus had time to attempt to implement it on the FPGA. Although we were ultimately unsuccessful in this endeavor, we still feel that it was worthwhile in many ways. We learned a lot about how to gather data from technical documents and also provided thorough documentation of our process which could be used by future groups to circumvent many of the problems which we had.

In retrospect, one of the things which would have helped us immensely would have been using an iterative approach to implementing the design on the FPGA (similar to the process which we used to implement the processor in Xilinx). This would have allowed us to catch errors as they came up instead of having to implement the entire processor and debug all at once.



Documentation For Future Users

Section 6 of this final report contains a collection of the information which we compiled that could be used by future students. Among these resources are a list of pin associations between the Spartan2E board and the peripheral board, a step-by-step guide for implementing simple IO, and a list of other problems we encountered.

What's Next?

The remainder of this document describes both the process and all aspects of our final product in greater detail. The purpose of this document is to provide the user with both a complete understanding of the workings of our processor as well as an understanding of the process which we employed to complete the project.

Section

2

Instruction Set Design

This section describes the design decisions that led to our final assembly language and machine language specifications. It also provides details about the unique points of the final specifications..

Beginning Stages: Instruction Types

The first step in designing an instruction set architecture was to determine the nature of the architecture of the processor. Because the memory of the processor was determined by the professor to be 16-bit, it was logical to use a 16-bit processor. A 16-bit architecture indicates that the processor will use 16-bit instructions, 16-bit registers, and 16-bit memory addresses.

In keeping with the 16-bit theme, it was determined that 16 general purpose registers would be appropriate. This allows for a 4-bit register address, which is conveniently a denominator of 16. This would allow each 16-bit instruction to address 4 registers (if no op-code exists), or, more realistically, 3 registers (if an op-code exists). Having an instruction capable of handling the address of three registers makes it convenient to use instructions much like that of the R-type instructions in the MIPS architecture. For example, the MIPS add instruction uses three register addresses: add $rd, $rs, $rt.

Because the majority of instruction address 3 registers, this leaves 4 bits unused. It was determined that 4 bits would be an appropriate length for an op-code, as it would allow up to 16 different instructions for the processor, and does not waste any bits. Thus, the first instruction type was created: Arithmetic (A-Type) Instructions. As the name implies, the A-Type instruction format is handy for arithmetic operations that require two arguments and a destination location. These instructions are listed in the Assembly Language Specifications document.

Now that a template for A-Type instructions has been framed, it is necessary for the user (programmer) to input data directly into the processor through the program. For this, an immediate is necessary.

It was determined that the best way to input an immediate into a processor's register would be to have instructions that consisted of fields for an op-code, a destination register address, and an immediate. It was already established that an appropriate length for an op-code is 4 bits and likewise for the register address. This leaves 8 bits for an immediate, which is convenient because it is exactly half the length of the register. In this manner, an entire 16-bit number could be loaded using two instructions and without "wasting" any bits. Thus, the Immediate (I-Type) instruction is born. More information will be provided later in this document describing the instructions involved in this process.

Finally, a program will need to be able to jump to other locations in memory. Since the processor uses 16 bits to address memory and the instruction is only 16 bits long, something must be done to accommodate the ability for the instruction to jump to any memory location. It was decided that since each register file is 16 bits long, it is very easy to place an entire memory address in each register file. Therefore, two I-Type instructions could be used to load a memory address into a register, and a single jump instruction could be used to access the memory address stored in a register and jump to that location. This jump instruction only needs one argument to specify the register that contains the memory location, which is 4 bits. Also, an op-code is necessary; another 4 bits. Thus, only 8 bits are needed for a jump instruction. The remaining 8 bits in the instruction are not used, and are by convention set to 0. This constitutes the need for one final instruction type: the Jump (J-Type) Instruction.

It was decided that the op-codes for all instruction types be logical, and not random. The distribution of commands is as follows: 10 A-Type instructions, 2 I-Type instructions, 2 J-Type instructions, and a special "exception" instruction (described later). It works out that with a 4-bit op-code, there are exactly 12 combinations where the two most significant bits can be logically "and-ed" together to get a value of 0. The 10 J-Type instructions (as well as the exception instruction) are each uniquely assigned to one of these 12 combinations.

Since there are 4 op-code combinations left (with which a logical "and" of the fist two bits produces a value of 1), they are distributed to the remaining 4 instructions. I-Type instructions are assigned such that the third-most significant bit is 0. The J-Type instructions are set such that the third-most significant bit is 1.

Now that all instruction types have been established, it is necessary to determine what instructions should be implemented.

Inventing a New Language

A Type Instructions

By far the most common instructions for a processor are A-Type instructions, since they are the instructions that actually manipulate data and perform calculations. Basic mathematical knowledge tells us that probably the most important mathematical

operations are addition, subtraction, multiplication, and division. Almost any mathematical procedure can be broken down to a series of these instructions. However, upon a closer look, multiplication, division, and even subtraction can all be done with addition. Therefore, the most important mathematical instruction is addition, which is the first instruction on the processor's architecture.

Even though an addition operation can perform subtraction (by adding a negative), restricting the processor only to addition requires an operation to find the inverse of a number. Therefore, nothing is gained by omitting subtraction, and including the instruction allows for more intuitive programming. It was determined that the processor should indeed support subtraction.

Some instructions (other than arithmetic instructions) may need to use a similar format as the A-Type. Such an example would be to store a word to memory and retrieve a word from memory. Since being able to read/write to memory is essential for a processor's ability to hold large programs and manipulate large amounts of data, it was determined that the processor should implement both a load and a store instruction. Since these are the only two instructions that access memory, this processor's architecture has been implemented as a "load-store" architecture.

Some instructions that don't seem intuitively arithmetic require the use of some sort of arithmetic logic unit in which two arguments are given and some operation is performed on them. These include a bitwise 'and', branch instructions, and a set less than instruction.

The bitwise 'and' instruction was deemed necessary in order to do certain operations with the later described "button state register". It can be used to reset certain bits and keep the others in tact.

Branch instructions are necessary for a program's ability to loop and to jump to a certain part of a program based on a certain condition. Based on this, it was determined necessary to implement two branch instructions: "Branch if Equal To" and "Branch if Not Equal To". Both of these instructions take two arguments and jump to the memory location specified in a register if the condition is true. If the condition is false, the execution continues to branch the next line of code.

In conjunction with the instructions, it is necessary for a programmer to know if one number is greater than or less than another number (often times to determine if a branch is necessary). This calls for a "Set Less Than" instruction that sets a specified register to "0x0001" if the first argument is less than the second argument. Otherwise, the specified register is set to "0x0000". It was determined that a "Set Greater Than" instruction is not necessary, since the same functionality can be implemented by reversing the order of the arguments in "Set Less Than".

The last two A-Type instructions have necessity further described later. These instructions are "Retrieve Value" and "Output Value". They are necessary to move

data back-and-forth between the special purpose registers and the general purpose registers.

Retrieve Value takes a 16-bit word out of a special purpose register and places it into a specified general purpose register. Output Value takes a value in a general purpose register and places it into a special purpose register. This means both instructions have an op-code and two arguments, leave 4 unused bits. Even though their structure is similar to a J-Type instruction in that they have unused bits, "Retrieve Value" and "Output Value" are considered A-Type because they never cause an execution jump.

I Type Instructions

As stated above, I-Type instructions have one basic purpose: to place data from a program into the processor. However, since an immediate field is only 8 bits long, and a register is 16 bits, two instructions are necessary in order to load an entire 16-bit integer (signed). One instruction must load the first 8 (most significant) bits to a register location, and the other must load the last 8 (least significant) bits to a register. For that, the instructions: "Load Upper Immediate" and "Load Lower Immediate" were selected.

Load Upper Immediate takes the 8 bits of the immediate field and places them into the most significant 8 bits of a specified register. Load Lower Immediate takes the 8 bits of the immediate field and places them into the least significant 8 bits of a specified register. Note that neither instruction changes any bits to a register other than those being loaded.

J Type Instructions

There are two purposes for jumping in a program. One is to unconditionally jump to a certain memory location, and the other is to call a procedure. Because we wanted to create a general purpose processor, we decided to implement both commands.

Jump to Register unconditionally jumps to whatever memory address is specified in $rs. Jump and Link unconditionally jumps to whatever memory address is specified in $rs as well, however it stores the next instruction's memory address into register $ar. This allows for a called procedure to return execution to where it was left off in the calling procedure.

One final instruction, the "exception" instruction, is necessary for our processor to be able to handle exceptions. This instruction is NOT available to a programmer and is only used internally by the processor. This instruction is fed to the processor when an exception has occurred. The exception instruction tells the processor to jump to an exception address in memory (hard-wired to the processor). It also tells the processor to save the memory address of the next instruction to the Interrupt Return Register ($ir). While the processor is jumping to the exception memory location, it sets the Interrupt Enable Bit to 1 (this process is described later). This is so the exception handling routine can jump back to the main program.

Creating a Standard

With all instructions in place, it was time to decide a standard in usage of registers. Much like MIPS, we decided it was best to compromise between a "callee" based convention, or a "caller" based convention, meaning there will be certain saved registers that a procedure should not overwrite, and certain temporary registers which procedures are allowed to overwrite. By convention, there are four of each of these types. Saved Registers: $s0 - $s3. Temporary Registers: $t0 - $t3.

Argument and Return Registers

To allow for procedures to be called, two Argument Registers ($a0 and $a1), two Return Registers ($v0, $v1), and a Return Address Register ($ar) are implemented. The Argument Registers are used as arguments for a procedure, the Return Registers are used to return values from a procedure, and the Return Address Register is used so that a procedure can return to where it was called.

Zero Register

For convenience sake, a Zero Register ($zero) was implemented to assist with mathematical instructions/calculations. The Zero Register ALWAYS contains the value 0x0000. This value can never be changed.

Button State Register

The Button State Register ($bs) is used so that a program can easily see what buttons have been pressed on an external board containing pushbuttons. Once a button has been pressed, the button's corresponding value in $bs is changed to a 1, and remains that way until reset by the program.

The most significant bit of $bs has special purpose. This bit is called the "Interrupt Enable Bit". If this bit is 0, the processor is accepting interrupts. If it is 1, the processor ignores all interrupts. When an interrupt does occur, the processor sets the bit to 1 (disabling interrupts) until it returns from the interrupt handling routine. Placing this functionality in a general purpose register allows a program to enable/disable interrupts.

A bank of special purpose registers (only accessible by "Output Value" and "Retrieve Value") are also included on the processor. These registers include the Value Register ($vr) and the Display Register ($dr). Even though there are only two registers in this bank, they are still referred to with a 4-bit address (such that they follow a similar format to the general purpose registers).

The Value Register is a register that is "hard-wired" to dip-switches on an external board. The Value Register always holds a value reflecting the state of these switches.

The Display Register is a register that is also connected to four seven-segment displays. It holds the value (in binary) that should be output to these displays.

Machine Code Formatting

Each instruction in the architecture should follow a logical flow, such that programming is as intuitive as possible. Therefore, a standard should be made in the syntax of statements.

General Instruction Format

InstructionName $destination, $argument1, $argument2

- All instructions require InstructionName.
- Only the A-Type instructions in this architecture have all 3 fields described above, however all instructions use the general format.
- If an instruction requires only one argument, only $argument1 is used.
- If an instruction requires only a destination, only $destination is used.
- All I-Type instructions combine $argument1 and $argument2 into a single argument field (to allow 8 bits
- Sometimes the $destination is used as a source (such as with Load Word).

The machine code itself follows the same format, saving confusion. The op-code is ALWAYS the first 4 bits. The destination is always the second 4 bits, the first argument is the third 4 bits, and the second argument is the last 4 bits. This allows for a total of 16 bits per instruction.

Putting it All Together

As described above, the processor's Instruction Set Architecture should be logical, methodical, and intuitive. By creating 3 different instruction types, we have divided instructions into specific categories based on their similarities, but we have kept only a minimum number of categories in order to save confusion.

When deciding on what instructions to put into the architecture, special consideration was given to importance of the instruction, its necessity, and how much convenience is

enhanced with the addition of the instruction. The final instruction set evolved slightly from the original implemented set in order to provide functionality that was deemed necessary later in development.

The syntax and binary implementation was designed to be as intuitive as possible by utilizing a set of rules with which to go by. There are few enough rules that they are not difficult to remember, simple to follow, and apply to every instruction in the Instruction Set Architecture. This instruction set architecture truly aspires to follow the 1st rule of thumb from the Patterson Hennessy text book, "simplicity favors regularity."

Java Assembler

The command-line Java Assembler program which our group created based on our Assembly and Machine Language Specifications proved to be an amazing help throughout our implementation as well as our Xilinx testing and FPGA testing stages. It allowed us to test our instruction set on our processor model by writing over 20 different mini-programs which each highlighted a basic function of the processor.

The Assembler not only provides simple direct conversion of register and instructions to their binary values, but also implements pseudo instructions, can output in either hex or binary and with a variable number of words per line, automatically calculates jump target addresses, can output general information about the program, and provides basic debugging features.

Assembler Read-Me

The Assembler program is well documented. A full description of its functionality is included in the Assembly Language Specification document included in the Appendix. A read-me.txt file is also included with the electronic version of the source code and program file. Also note, the source code is also well commented.

The only added restrictions which the Assembler requires are a series of "tags" (which are text strings like assembly instructions except they begin with a '.' in front of the instruction). For example, each assembly language program to be translated by the assembler must begin with ".address 0x0000" where 0x0000 is some 16-bit hex number which specifies the location of the first line of the program in memory (and allows the program to calculate jump addresses). Similarly, each assembly language program must end with a ".end" tag.

Example Program with Assembler Tags

```
.address 0x0000          # sets the program starting address
.label loop              # declares "loop" as a jump target

# some assembly language instructions
add $s0, $zero, $zero
lli $s0, 0x01

loop:                    # labels the next line as jump target "loop"
add $t0,$s0,$t0

# a pseudo-instruction which loads the address of loop
load $s1 loop
jr $s1

.end                     # this tag indicates the end of the program
```

As stated before, the Assembler is run from a command line interface as shown below and can thus take a number of run-time parameters to modify how it formats the machine language in the outputted file.

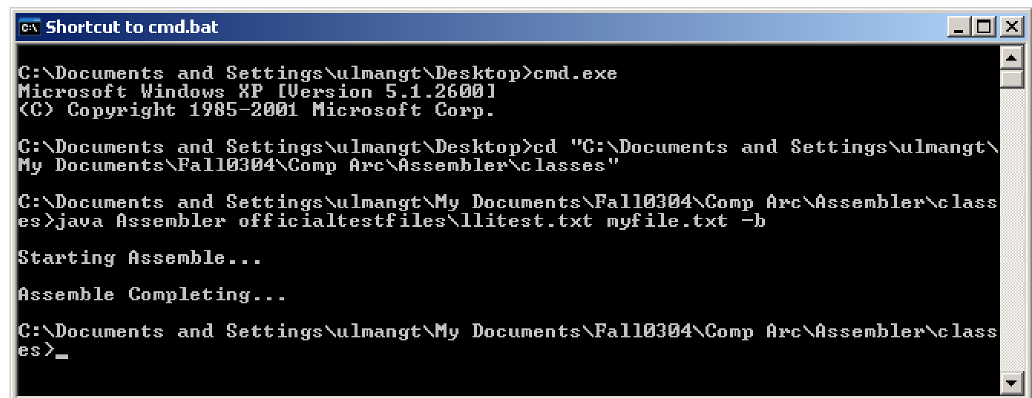


FIGURE 1 The command line interface for the Assembler. This shows the program assembling the assembly language file "llitest.txt" in the "officialtestfiles" directory and outputting the results in binary (the result of the "-b" tag) to the file "myfile.txt." The process is extremely quick and requires only minimal special formatting of the assembly language file. This made the Assembler a great asset when creating multiple test programs to be run on the processor.

The result of all these features and considerations was a versatile tool which was easily able to adapt to changes in the design criteria of the project. For example, we originally assembled all of our example programs into binary with one instruction (two words) per line. However, when Dr. Merkle released the first version of the memory component, we realized that the machine language files would have to be in

hexadecimal with one word per line. However, this merely meant re-running the Assembler with different command line arguments.

Thus, overall we feel that this was a useful tool for our group and was well worth the time invested in its development.

Section

3

Design Implementation

This section details the steps we took to design our components and our data path in order to implement our assembly language and register transfer language specifications in hardware..

Register Transfer Language Design

Once our assembly language and machine language were completed, our task was to move on to the design of the hardware part of our processor. We began with the RTLs for each of the instructions in our processor. The full text of the RTLs can be viewed in the appendix. In any case, there are some notable characteristics in our RTLs which gave uniqueness to our data-path. First off, unlike MIPS, which is a 32-bit processor, our processor, being 16-bit does not need to increment the Program Counter by 4 every instruction. Instead, we increment it by 2, which is rather unsurprising.

More interesting is that the $vr register in the assembly commands rv and ov is mentioned for the first time as an independent special register that takes the value of the dip switches directly. Here is the RTL for ov as an example of our general RTL and to illustrate the appearance of the Special Register.

Output Value (ov) Register Transfer Language

1.	PC = PC + 2	# add 2 to the program counter
	IR = Mem[PC]	# store the value in the PC to the IR
2.	A = Reg[IR[8-11]]	# reads values into the registers
	B = Reg [IR[12-15]]	
	C = Reg[IR[4-7]]	
	If(IR[0-3] == 5) then	# jumps based on the op-code
3.	Special[IR[4-7]] = A	# store A into the special register file

Unlike the regular register, this one is modified by the user when a dip switch is changed. This occurs in command rv and then the machine works on its own from

there. Once our value is computed, the machine outputs the result to the same register in command ov and displays it in the seven-segment LED, so this register is basically the "middleman" between the user the display.

Another notable idea in our processor is the fact that the exception RTL works just like a regular operation. Our first two lines are exactly like the other RTLs down to the conditional that checks for the op-code of the operation to see which one to execute. Since we have 15 out of 16 possible operations, our exception handling takes the form of our "16th" operation. Here we have step 2 of our interrupt RTL:

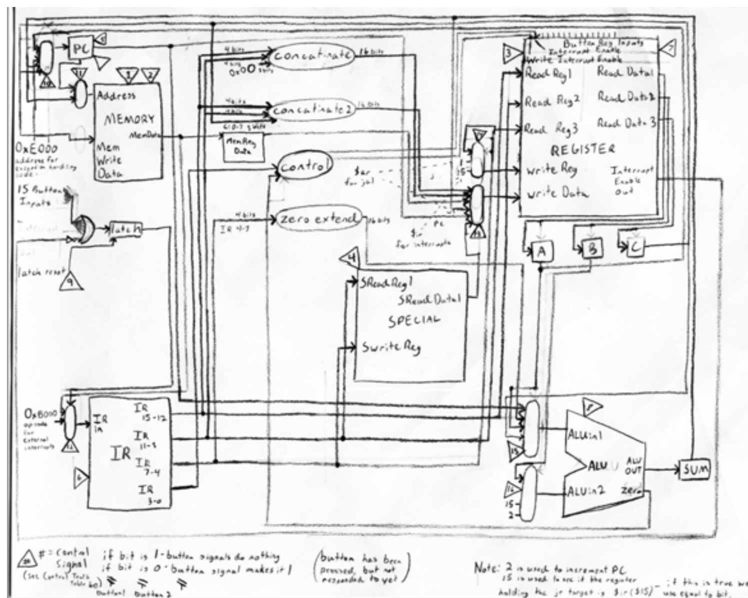
RTL For 2nd Clock Cycle

```
A = Reg[IR[8-11]], B = Reg[IR[12-15]], C =
Reg[IR[4-7]]

If (IR[0-3] == ##) then (jump to ##)
```

This was very lucky of us, because if we had needed one more operation, we would've had no way of checking for interrupts, which would have messed our careful balance, because we would've needed to devote one more bit to the op-code or change the way we did things. The rest of the RTL specifications are available in the appendix. We basically specified the steps taken in our decisions made previously.

Data-path Design



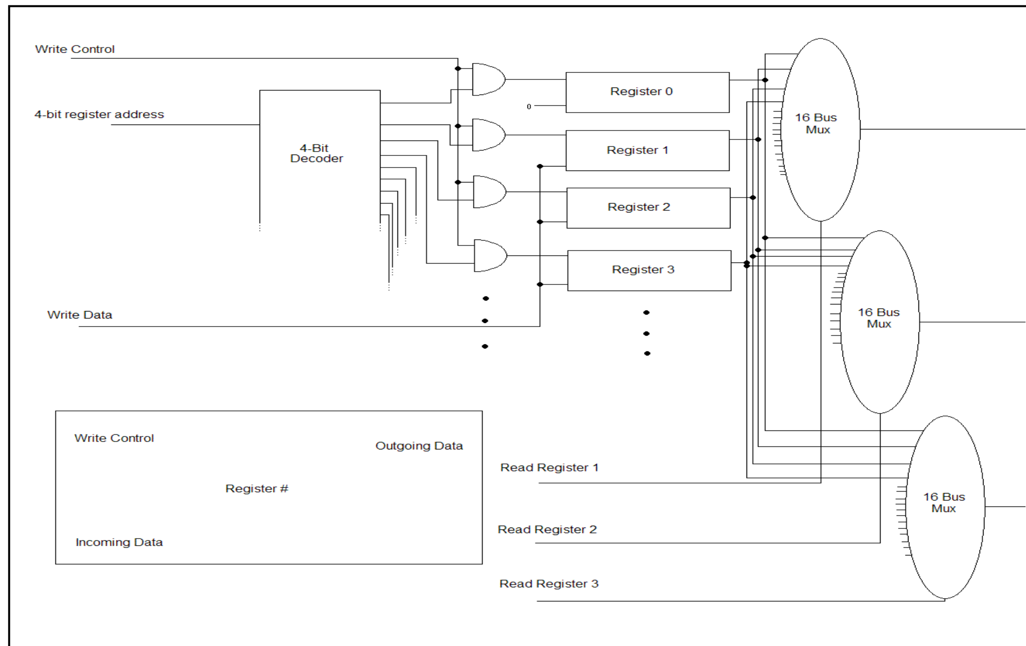
Data-Path Diagram

Where we start to find innovation again is in our data-path. Its design is hardly revolutionary, with most components being pretty straight-forward. Of course, it's their use in this case, which interests us. Here is a picture of the design, which can be found expanded in the Appendix:

As we can see, we sport many of the common operations, like incrementing the PC with the ALU or sending the different bits of the IR into the Register component. Of course, some of the uses of these components are quite interesting.

For example, we can see that there is a latch in the middle left side. What the latch does is that it controls the multiplexer going in to the IR. This latch has two possible inputs, either the press of a button by the user or an exception being handled. If either of these two is activated, the latch activates and disables the interaction of the memory with the IR and instead feeds the IR the op-code for external interrupts so it can tell the control to begin handling them.

The Special Register is pretty much an autonomous component. It can keep any value that is presently set with the dip switches without having any effect on the program. It's main task is to output the value into the seven-segment LED so the user can see what value is being output. Now, only when the processor is looping, waiting for an input, and the first button is pressed does the value in the special register get loaded into the register.



Register File Diagram

The Register itself is hooked up to the buttons and the Register keeps track of which buttons are pressed specifically. In our case, we support up to 15 different buttons, each hooked up to 15 button registers (with a 16th button register being the interrupt-specifying register), but we currently only use two of buttons. The first button loads whichever value is stored in the Special Register into the Register and the second button initiates the calculations and creates the interrupt which prohibits any other value from being input. The previously-mentioned interrupt button register lets the program know when an interrupt has occurred and ignore button presses. These button registers are advantageous because they can also function like regular registers to be used by the processor at will.

FPGA Processor Re-Design

This register file design was not used in the version of the processor that was implemented on the FPGA board because it is actually a very inefficient design. For the FPGA, this register file was re-written in Verilog code which allowed Xilinx maximum freedom to optimize it..

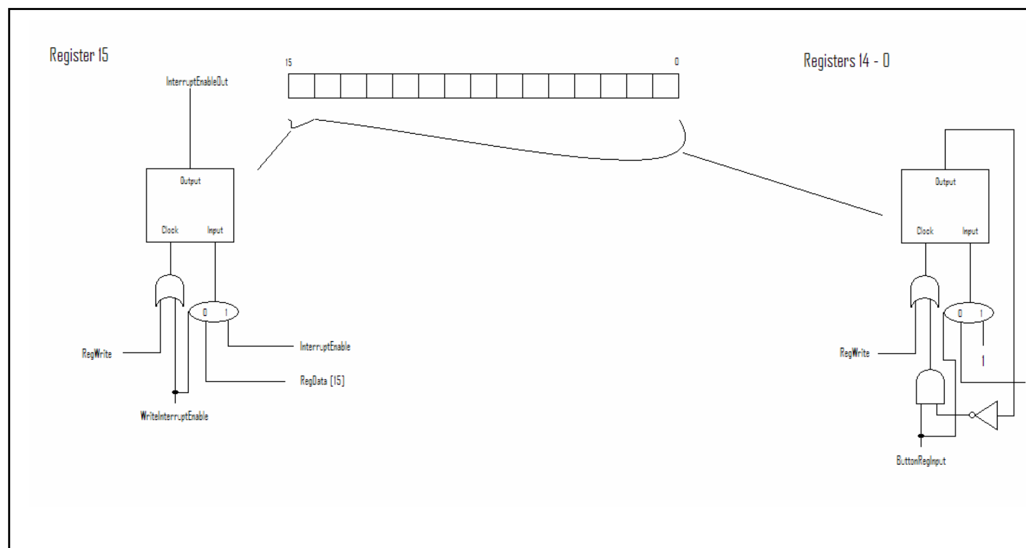
Individual Component Design

Now we will delve deeper into the hardware-side of things and look at the internal make-up of the specific components. Many of these specific components are pretty generic. As an example, the memory was even provided by the instructor. Others, like the PC or special-purpose registers A, B, and C are only registers composed of D flip-flops, hardly interesting. For completeness sake, I will enumerate all of our different

components and then simply elaborate on the most interesting ones. Our processor consists of 7 multiplexers; 8 special-purpose registers (with varying bit capacities) including the PC (which has a value that is incremented periodically), the latch, and the IR (whose function is to split up the bit inputs); the memory component; 3 concatenate components which just rearrange bits and add zeros; the Register; the Special Register; the Control Module; the ALU; and assorted gates.

Of these, the ones worth mentioning are the Register, the Special Register, the Control Module, and the ALU. We will begin with the Register. It basically consists of three parts, the register logic itself, interrupt enable bit logic, and the button bit segment logic.

The register logic is pretty common. The registers themselves are just regular 16-bit registers As well,. there is a component which specifies to the multiplexers which register's value to use. In the write control, the bits received are just interpreted by a 4-bit decoder that specifies the register being written to.



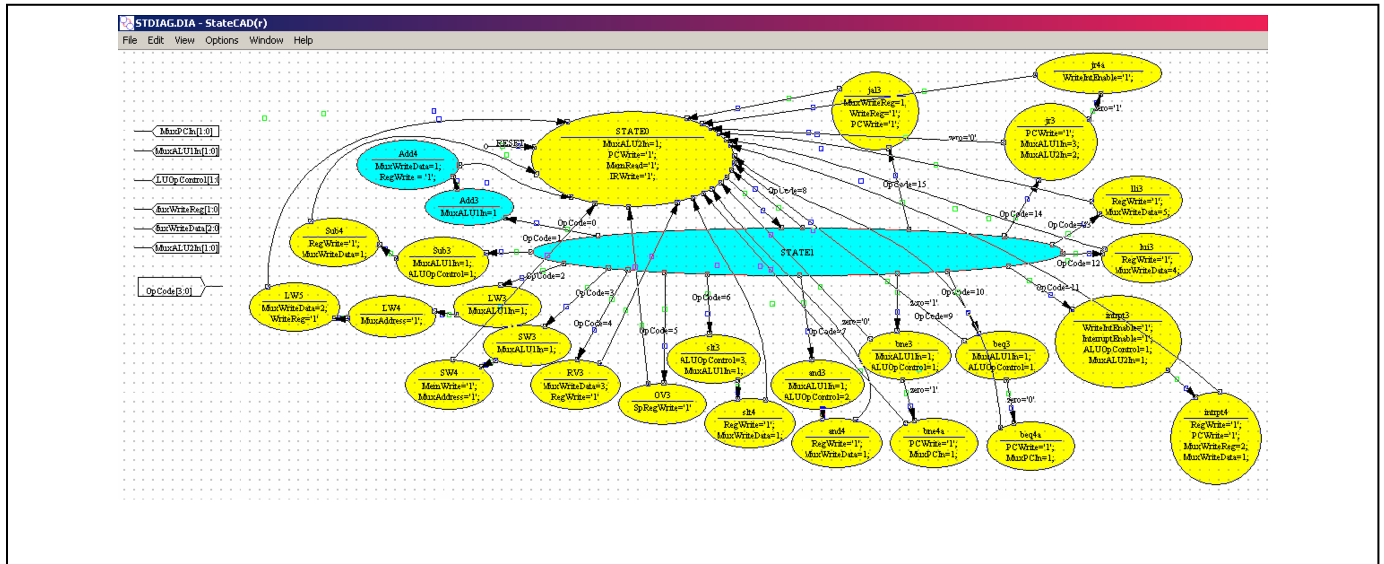
Button State Register Design

On the right, the interrupt enable bit consists of the actual D flip-flop and then a miniature mux and some gates. Basically, what happens is, if the control tells it that the register wants to write to it or that an interrupt occurred, then it will either receive the value of the register or set itself to 1 for the interrupt. It will output this result to tell the multiplexer in front of the IR if an interrupt has occurred or not.

On the left a regular button bit appears. The clock enable for the bit writes to it only if it is set to 0 and a button is pressed (because a button press cannot and in any case should not set it to 0) or when the Register itself deems to give it a value (in this case it can be either 1 or 0).

The next component is the Special Register. It is composed of two 16-bit registers: the value and the display registers. The value register is hardwired to the dip switches and always holds their value. The display register gets the value from the value register and displays it in the seven-segment LED. These are both connected to a multiplexer whose purpose is to feed the value in the value register into the Register when the first button is pressed and to send it to the display register when a digit is chosen with said button.

The Control Module's design was basically generated automatically by Xlinx, so we will not cover it in this section. In any case, it was created through this state-transition table implementing the control signals that go to the various components:



State Diagram Implementation

The final component we want to contemplate is the ALU. This ALU supports Add, Subtract, Bitwise AND, and SLT operations. The design for the one bit adder and one bit ALU are very similar to those presented in class and in the Xilinx exercises (details are provided in the appendix).

The 16 one bit ALUs were connected as shown on the right to complete our sixteen bit ALU. However, some explanation of the logic surrounding the 16 one bit ALUs may be required.

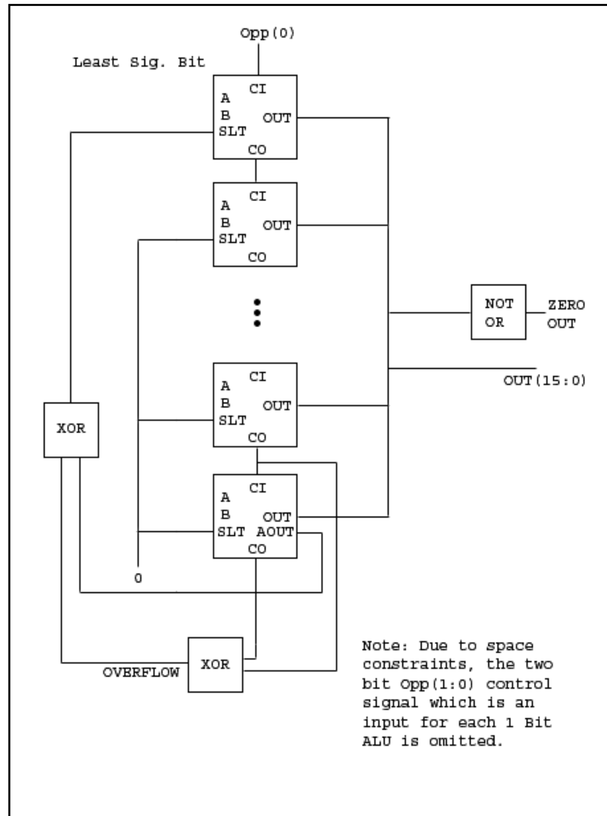
Overflow Detection and Correction

.Out ALU detects overflow by comparing the carry in and carry out bits of the most significant 1 bit ALU. However, this signal is only used in the Set Less Than operation. If overflow in the subtraction that the ALU performs when evaluating a SLT occurs, then the result will be the incorrect output. However, if the SLT bit is combined by an XOR with the overflow bit, then if overflow occurs the SLT signal will be automatically corrected.

Thus, while our processor can output an incorrect value for a large enough addition or subtraction, it will always output the correct value for SLT

As we can see, the ALU design is a ripple-carry design. Here we have an XOR that handles overflow whose inputs are the carry in and carry out of the most significant register. When this overflow signal is asserted, it indicates that overflow has occurred. Finally, the outputs of all bits are hooked up together into a 16 bit bus output. A "zero"

output is also required by the ALU to compute the Branch Not Equal and Branch On Equal instructions. This signal must be asserted if the output of the ALU is 0 and de-asserted all other times—thus the signal is obtained by a NOR of all the 16 output signals of the 1 bit ALUs.



Section

4

Xilinx Model

This section describes the process which we went through while implementing the processor components and data-path in Xilinx.

Initial Process

We began working on our implementation of the Xilinx model during our 9th meeting, which was on the 21st day of October. At first, we were unsure of what some of the components of the model would consist of. The registers could have been made out of several D flip-flops, or possibly some other piece of logic. There were several tactics of how to complete the control and its respective signals, but we were still at a loss of how to implement them in Xilinx. We decided to implement as many as many components of our schematic as possible, then add the more complex pieces, such as the ALU, Control, and Registers at a later time. The construction of the simple pieces of the schematic proved fairly uncomplicated, and went swiftly. While experimenting with labeling the A, B, and C registers, we found that the size of the text had almost no bounds on it's font size. Thus, we added an "Easter egg" to our program, in the form of a titanic letter C with font size 1,000,000 that dwarfed our schematic exponentially (which was later removed).

The design for our ALU was taken from the respective in-class exercise. The only difference is that our ALU does not have the same functionality as that of the one in class. It lacks a few operations, one of which is OR. A 16 bit ALU was then created from our 1 bit ALUs. For our ALU, we decided not to include a separate B-invert signal. This is because the least significant bit of the op code for the ALU could be used as a B-invert. This works because the least significant bit of the AND and ADD op codes is 0, and those operations require B-invert to either be 0, or "don't care". The least significant bit of SUB and SLT op codes is 1. For both of these commands, B-invert needed to be 1.

After some time, the ALU was completed in Xilinx. The ALU was extensively tested with the ADD, SUB, and SLT commands. The parts were shown to work properly.

The next step of our experimentations and implementations led us to conclude that we would have to use 1 bit D flip-flops in order to construct our registers. We had to use these 1 bit registers to create our special button state register, which was one of the 16 general registers. The control would be created with a multiplexer, several counters, and roms. For each separate control signal there would be one counter, and one rom. The counter would cycle thorough as many clock cycles as was needed to complete the instruction. For each clock cycle, the counter would output a unique 4 bit code to the rom, which would output all of the appropriate control signals for that particular cycle. This method would prove to be particularly useful, as the control signals were particularly easy to change if needed.

The registers were created without much difficulty, and were tested by loading particular values into the registers, then outputting the values of the registers in order.

Xilinx Testing Process

After all of the parts of the schematic were created, testing began. Individual components, as well as several components at a time were tested. The inputs were specified in the test bench, and the outputs were evaluated. After the components were tested, they were all integrated, with the absence of only the memory. The IR was temporarily replaced with a 16 bit input for testing purposes. The connections of the entire schematic were debugged using the "Check Schematic" command in Xilinx. It was at this point when we learned that if an input to a component was a bus, then it must be buffered in order to make it an output out of that same component. This required some minor edited, but was rectified quickly. We also had to remove our "Easter egg" at this point, because Xilinx doesn't like text that appears outside of the schematic. After the connection debugging process, the entire schematic was tested by inputting a command on the test bench, via the 16 bit IR we created, and observing all of the signals of the wires, deciphering whether or not we had, in fact, ensured that all of our components were working together properly. Also, instructions were implemented consecutively to ensure that the processor could safely shift from command to command. The java assembler proved useful in this step, as it predicted the desired signals for the processor.

One other problem that we came across was the fact that in our original numbering scheme, the most significant bit of a 16 bit number was 0. This caused much confusion at times, as we sometimes read the wrong bits out of the IR, and other components. We rectified this problem by going through every instance where bits were labeled, and checked to make sure that they matched the Xilinx numbering scheme, where 15 is the most significant bit in a 16 bit number.

Section

5

Testing Methodology and Final Results

This section describes the incremental testing process which we followed to ensure that all aspects of our processor functioned correctly.

Perhaps the most important portion of our process for completion of our project was our testing. In the beginning of our process, each team member created their assigned component and went through a theoretical procedure to determine what inputs and outputs were appropriate at the correct times for a specific component. Then, systematically, components were placed together in the Xilinx Model. Initial testing combined all components except for memory, the PC, and ignored interrupts altogether. At this point for organization (and easier documentation) an Error/Change Tracking Log was created to log the initial state of the processor and the subsequent changes made up to the final model.

Error/Change Tracking Log

One of the features of our testing process that our group is most proud of (and which helped us a great deal during testing) is our Error/Change Tracking Log. This document listed, in bulleted form, every single change which we made to our processor, our documentation, and our test programs during the entire testing process. This meant that any moment we knew exactly what we had done, how we had done it, and what we needed to do next. This document is available in our appendix, and excerpts of it are used throughout this section.

Overall Testing Strategy

1. .Test each component individually in Xilinx.
2. Connect the data-path without the memory component or IR.

3. Manually input an instruction through the test bench waveform (instead of from memory) and debug the data-path until that instruction performs correctly.

4. Repeat this process until relative confidence that the data-path is correct is achieved.

5. Attach the memory component and

6. Test each instruction by writing a small assembly language program using only that instruction and possibly some instructions tested before it.

7. At this stage, interrupt handling was tested and debugged—the last step necessary before we could perform our final acceptance tests.

8. Test the overall processor by doing numerous runs of the final Relative Prime algorithm.

Systematic Testing Process

Xilinx's HDL Bencher and the ModelSim Xe II v5.6e were the backbone for our testing process. Using the HDL Bencher, we could simulate every single signal we wanted, and through the ModelSim our group could insure that the instruction we were testing was outputting correctly for the given input. The correct signals for testing each instruction were derived from earlier documents (Data-path and Control.doc, Register Transfer Language Specifications.doc).

All of our instructions used other instructions to test them, except for our first one, Load Upper Immediate (lui). To test this instruction, we simulated the behavior of memory and the PC, in order to minimize the location of potential problems in the data-path. Once these data-path errors were corrected, we were free to verify the operation of each of our instructions and to fix any timing and/or control signal concerns which arose.

In order to give the reader a full understanding of our testing process, we include below the entire testing cycle for our Branch if Equal (BEQ) instruction.

The first step in testing each instruction was to write the assembly language test program for that instruction (using only the new instruction and any necessary previously tested instructions). The BEQ test program is included below:

Branch On Equal (BEQ) Test Program

```
.address 0x0000
.label loop
.label end

add $t0, $zero, $zero
lli $t0, 0x03
add $s0, $zero, $zero
lli $s0, 0x01

loop:
# decrement $t0 by 1
sub $t0,$t0,$s0

# load the address we jump to on $t0 = 0
load $s1 end

beq $s1, $t0, $zero

# load the address of the loop
load $s1 loop

jr $s1

end:
add $zero, $t0, $s0
add $zero, $s1, $s0
add $t1, $t2, $t3
add $zero, $zero, $zero

.end
```

This program first loads values into registers $t0 and $s0 (which shows how important choosing the order to test the instructions in was—testing BEQ would have been impossible without a way to load instructions into the processor). Then, $t0 is used as a simple counter in a loop which counts down until $to reaches 0 and then branches to the "end" label.

This program was then converted into machine language in a format understandable by the provided memory component by using the Java Assembler program. The output file (which was named "main_prog_hex.txt" and placed into the Xilinx project folder as initialization data for the ByteRam.v component) is included below for reference.

Branch On Equal (BEQ) Test Program, Machine Language

0b	1b	c8	00
00	b7	00	87
db	c8	d8	0c
03	00	08	de
07	d8	e8	00
00	16	00	00
d7	a8	00	
01	b0	b7	

The next step was to actually run the test program in ModelSim using a Xilinx test bench. For this instruction the test bench did not need to contain any special input or output. Special test benches were only needed for the interrupt handling tests (where they had to simulate button presses) and for the actual final program test (which included input and output.

The ModelSim waveform output is provided as a sample below (although the sheer number of output signals on the test bench precludes it from being printed here in very large size).

The Error/Change Tracking Log neatly documents the success and confidence of each instruction, along with the consequent changes made to each instruction up till the final model.

Abbreviated Test Status Tracking Table from the Error/Change Log

The following table is an except from the full summary of the testing status of all instructions which was kept in the Error/Change Tracking Log. It lists the dependencies that the current test had to previously completed tests and the confidence of the tester in the correctness of instruction based on the test.

Instruction (in order of test)	Other instructions used in test	Comments
Lui	--	Test Successful, High Confidence
Lli	Lui	Test Successful, High Confidence
Beq	Add, Sub, Lli, Lui, Jr	Test Successful, Medium Confidence *(Update 11/2/03 changed to High Confidence)*

During the testing process for Branch on Equal, it was realized that we had made an error in the Assembly Language Specification for the command—which highlights another important feature of our Error/Change Tracking Log. The document had a "Documentation Change Requests" section where we recorded and kept track of all the changes which has to be approved by Dr. Merkle. This ensured that no unapproved changes snuck in during the testing process which could possibly cause trouble for Dr. Merkle by changing the requirements for the program he wrote for the processor.

Making a Second Pass

After our group was through testing all of our instructions with high or at least medium confidence, a second pass of testing was made to insure that procedures worked properly and also to ensure that our interrupt handing algorithms were working correctly.

One major dilemma our group ran into was fixing our processor so that the first clock cycle (while the global clear signal was being issues to the entire processor) wasn't junk, and the commands worked properly without extending or changing the clock cycle. The initial solution to this was including a "junk" instruction (typically "add $zero $zero

$zero" or 0x0000 in machine language) at the first position in memory. However, after further consideration, we decided that a better solution would be to simply include a reset state in the control unit which would delay the processor one clock cycle when it first started up. The reset would just delay everything one clock cycle without affecting the performance or the execution of the instructions.

Finally, running the gcd and the relativeprime algorithms gave our group a comprehensive test of all of our instructions. Although creating and running large test benches to simulate the processor calculating the lowest prime factor for even a small number took a very long time (both for ModelSim to simulate and for the HDL Bencher to update the ports in the test bench), we were able to run a number of tests on the final processor and were very happy with the final outcome.

As our group approached the end of testing, three outstanding problems still existed, PCWrite was acting up, the memDataRegister seemed to be unnecessary, and the latch was still implemented incorrectly. Subsequent testing fixed all of these problems, and these updates can be seen in the Error/Change Tracking Log.

FPGA Testing

After the process described above was completed, a further round of implementation, testing, and debugging was performed in an attempt (ultimately unsuccessful) to load the processor onto an FPGA. Details on this process are available in section 6.

Section 6

FPGA Implementation

This section discusses and summarizes the work our team did with the Xilinx Spartan2E FPGA and the Digilent Peripheral Board.

Initial Testing and Setup

As we were the first team to attempt synthesizing our board and loading it onto the FPGA, much of our initial effort was devoted to simply understanding the FPGA board, the Xilinx tools associated with loading programs onto it, and the design constraints which all this applied to our Xilinx schematics.

Online Documentation

Documentation files were pulled from a number of various sources in order to understand the FPGA board. The first source of information was the Digilent web page (maker of the Spartan2E FPGA and the IO1 Peripheral board). The pdf documentation for the Spartan2E is available at http://www.digilentinc.com/assets/documents/d2e_rm.pdf and the documentation for the Peripheral board at http://www.digilentinc.com/assets/documents/dio1_rm.pdf. The company also provided a simple I/O demo which demonstrated how to link the single button on the main Spartan2E board to the status LED. This was extremely helpful as it requires that the correct type of I/O buffer is used or the non-clocked button signal will not interact properly with the global clock that controls the chip. This document is available at: http://www.digilentinc.com/assets/referance_designs/d2e_demo.zip.

After about four hours of work on a Sunday night, pouring through this documentation and experimenting with the software, we were able to download a simple programming file to the FPGA which caused the status LED on the Spartan2E board to light when the button on the board was pressed. A basic step-by-step procedure for accomplishing this is provided below as a guide for others:

Step-By-Step Guide

1. Open the Xilinx project Navigator and close any currently open project file. Then select “New Project” from the File menu.

    a. Select **Spartan2E** as the **Device Family**

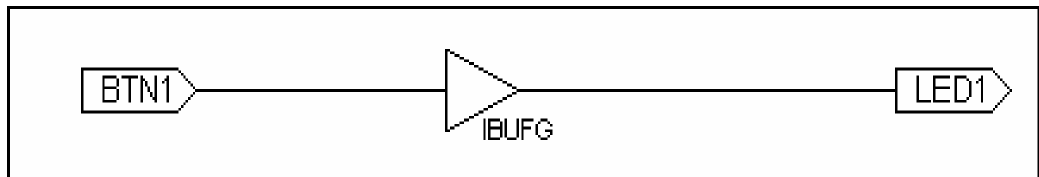
    b. Select **xc2s200e** as the **Device**

    c. Select **pq208** as the **Package**

2. From the Project menu, select New Source. When a dialogue box appears, select Schematic as the type of source and name the schematic. Hit Next, then Finish. The Xilinx Schematic Editor screen should appear.

3. Add a wire to the schematic and name it BTN1. Then, find the component “IBUFG” and place one at the end of the wire. Finally, extend a wire from the opposite end of the IBUFG component and name this wire LED1.

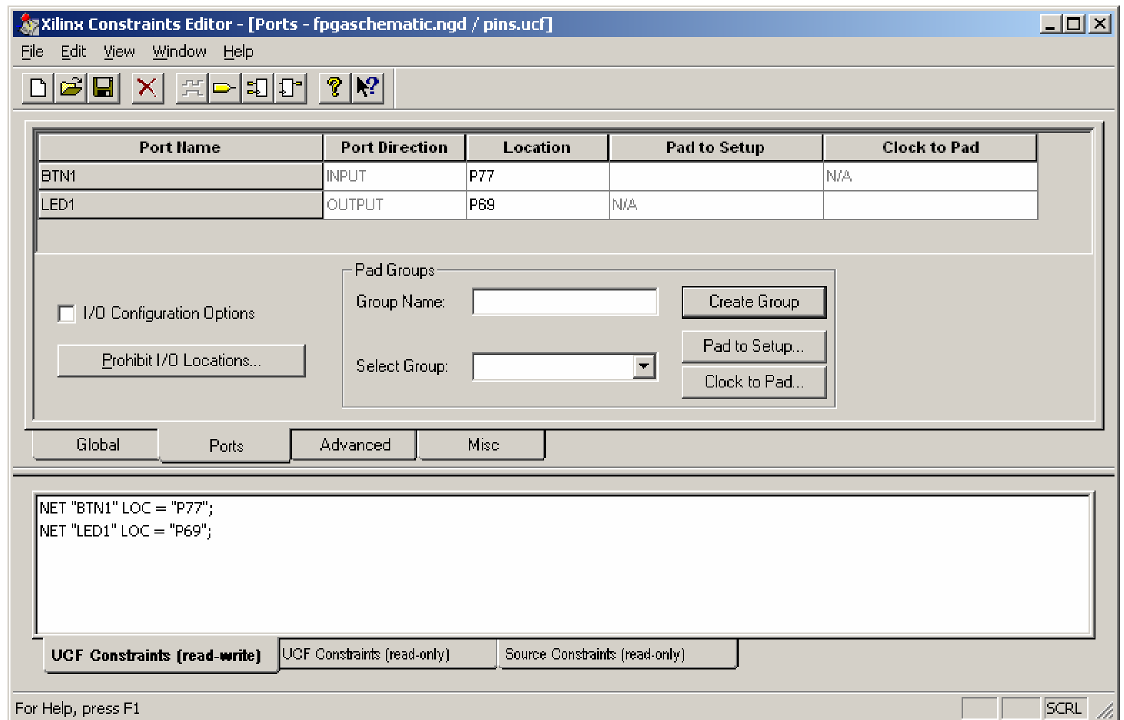
4. Place an input marker on the BTN1 wire and an output marker on the LED1 wire. Your schematic should look like the schematic below. Save the Schematic file and close it.



5. Now select New Source again from the Project menu and this time select Implementation Constraints File as the type of source. Name it, and hit Next. Make sure your schematic file is selected in the next screen, and hit Finish. A “.ucf” file should now appear as a source in your project.

6. Double click on the “.ucf” file to open the Xilinx Constraints Editor. Note: Xilinx will synthesize your project at this time, but there should be no errors and it should not take more than ten seconds or so.

7. Click on the Ports tab near the bottom of the editor and enter values in the Location column so that your screen matches the screen below (BTN1 should have value P77 and LED1 should have value P69):



8. Save the file and close it.

9. Now select the schematic file you created in the Project Navigator but do not open it. Under the Processes for Current Source frame, expand the generate Programming File Generation Report tab and double click on Programming File Generation Report.

10. Xilinx should have placed check marks next to the Synthesize, Implement Design, and Generate Programming File sources. It should also report the following in the Console (note: the exact name of the bit map file depends on what you named your schematic):

```
Running DRC.
DRC detected 0 errors and 0 warnings.
Creating bit map...
Saving bit stream in "fpgaschematic.bit".
Bitstream generation is complete.
Completed process "Programming File Generation Report".
```

11. You will now use the ".bit" file that Xilinx has generated from your schematic to program the FPGA. First, you must hook up the FPGA itself.

12. Plug the power cable into the power input on the board *then* plug in the parallel cable to both the board and the parallel port in the back of your computer. *Note: when the power cable is attached, LED2 on the FPGA board should light up.*

13. Back in Xilinx, select the schematic file again, and under the Processes for Current Source frame, expand the generate Programming File Generation Report tab and double click on Configure Device (iMPACT).

14. iMPACT should open and you will be presented with a dialogue box asking what you want to do first. Select the first option: "Configure Devices" and hit Next.

15. Another box will appear. Again select the first option: "Boundary-Scan Mode" and hit Next.

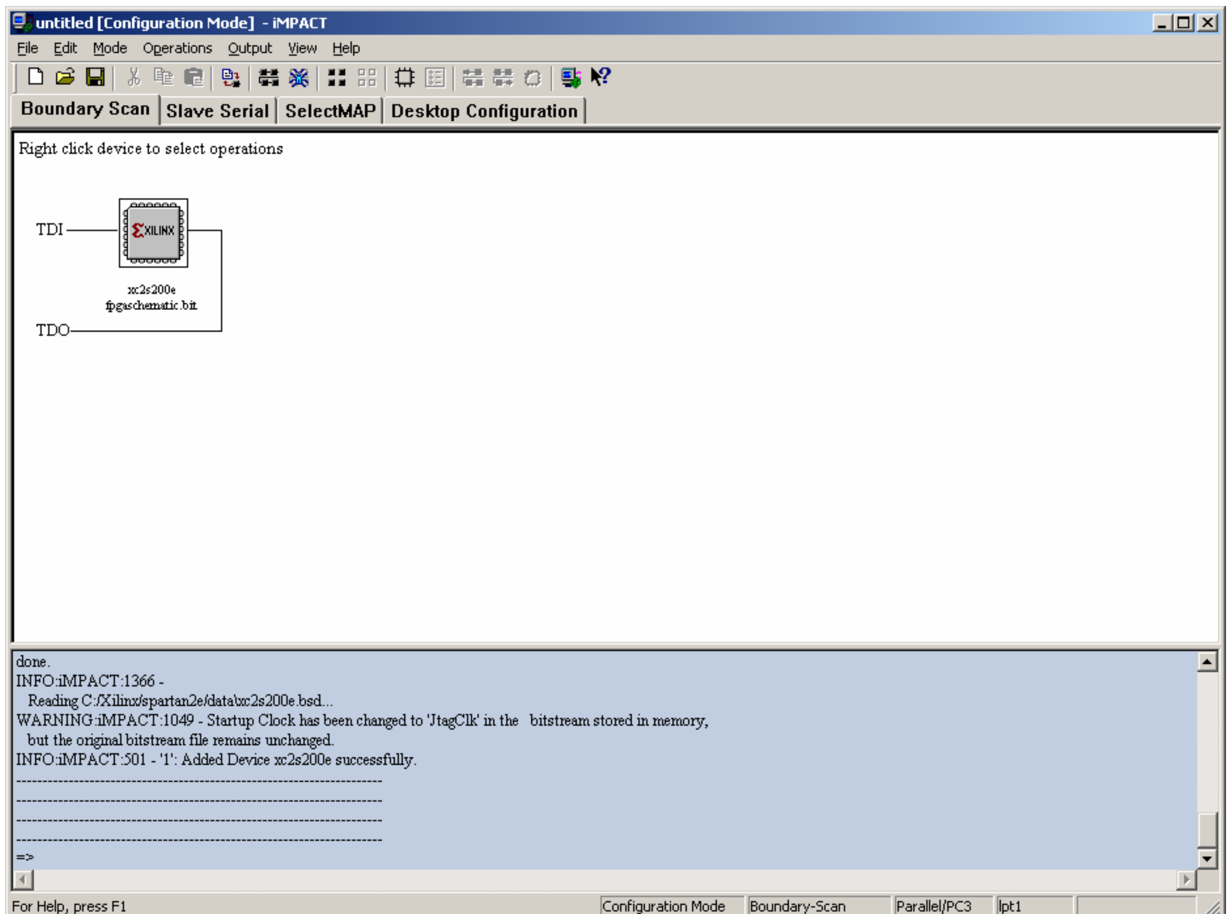
16. Another box will appear. Again select the first option: "Automatically connect to cable and identify Boundary-Scan chain" and hit Finish.

17. A dialogue box titled "Boundary-Scan Chain Contents Summary" will appear. Hit Ok.

18. A box will now appear asking you to Assign New Configuration File. Navigate to the ".bit" file which you created earlier (which should be in your project folder), select it, and hit Open.

19. Another dialogue box will appear. Hit Yes. A warning may occur telling you that the startup clock has been changed. This is normal. Hit Ok.

20. The screen should look like the screenshot taken below.



21. As the final step, right click on the chip you see on the screen and select "Program…" from the pop-up menu that appears. In the Program Options dialogue box that is displayed, make sure that the Verify box is *not* checked and hit OK.

22. You should see a large blue oval with "Programming Succeeded" on the bottom of the screen. You should now be able to push the button labeled BTN1 on the Spartan2E board and the LED next to it, labeled LED1, should light.

The following page contains a table containing pin associations between the Spartan2E FPGA board and the peripheral board. It is explained after the project.

Pin Association Between Peripheral and Main FPGA Board

On DIO1 Peripheral Board		On Spartan2E FPGA Board (A connector)		
Signal	IO Pin	Signal	IO Pin	S-II Pin
GND	39	1	GND	-
VU	40	2	VU	-
VDD33	37	3	VDD33	-
	38	4	A4	P68
	35	5	A5	P64
LD8	36	6	A6	P63
	33	7	A7	P62
LD7	34	8	A8	P61
	31	9	A9	P60
LD6	32	10	A10	P59
	29	11	A11	P58
LD5	30	12	A12	P57
	27	13	A13	P56
LD4	28	14	A14	P55
	25	15	A15	P49
LD3	26	16	A16	P48
	23	17	A17	P47
LD2	24	18	A18	P46
	21	19	A19	P45
LD1	22	20	A20	P44
BTN4	19	21	A21	P43
BTN3	20	22	A22	P42
BTN2	17	23	A23	P41
BTN1	18	24	A24	P40
	15	25	A25	P36
SW8	16	26	A26	P35
	13	27	A27	P34
SW7	14	28	A28	P33
	11	29	A29	P31
SW6	12	30	A30	P30
	9	31	A31	P29
SW5	10	32	A32	P27
	7	33	A33	P24
SW4	8	34	A34	P23
	5	35	A35	P22
SW3	6	36	A36	P21
	3	37	A37	P20
SW2	4	38	A38	P18
	1	39	A39	P17
SW1	2	40	A40	P16

Note: if you want a on output marker on your project to be associated with, the 4th switch on the Peripheral board, the location for that port in the Xilinx Constraints Editor should be set to P23.

Note: The following table does not yet contain a complete listing of the signals on the peripheral board (only the ones currently being used for this project). The abbreviations used are as follows: LD – Light Emitting Diode ; BTN – Button ; SW – Switch.

Once our team got this basic procedure working, our next step was to get I/O between the peripheral board and the Spartan2E FPGA working. In order to do that, we had to compare the technical specifications for each chip and match pin numbers to determine which pin on the Spartan2E each component on the Peripheral board ultimately attached to. The table which we developed (which assumes that the peripheral board is attached to expansion slots A and B on the Spartan2E board) shows these pin associations.

Thus, using this table and the general method described above, one should be able to connect the I/O devices on the Peripheral and Main boards in any combination. Note, however, that all input signals must have a buffer component attached to them. In the step by step example for the Spartan2E board, that buffer was a IBUFG. For the inputs on the Peripheral board, however, an IBUF, not an IBUFG, is required.

Implementing the Processor

Figuring out all the above information and getting simple I/O working was a large task in and of itself and easily represents 20 to 30 hours of work divide among Dr. Merkle Dr. Chidanandan, and our team members. Thus, we were understandably dubious about our chances of getting an infinitely more complicated system up and running in the amount of time (about a week) which we had remaining.

The main implementation problem which we ran across here was that trying to synthesize our design caused a number of Xilinx warnings and when we first loaded the design onto the board, nothing worked. Thus, our first priority was sorting through the various warnings and determining which were causing the processor to not function and which could be ignored.

The first major issue which we came across was that the hex files containing the main memory's initial values were not being loaded onto the memory component during synthesis. Dr. Merkle searched through online documentation files and determined that this was a documented behavior reworked the processor by using an "@ always" clause instead of an "initial" clause to load the values. Thus, in version 1.3 of the memory component, the initial values act as a sort of "BIOS" and are automatically loaded onto the memory component whenever it receives a reset signal.

The other major issue which we came across was the size of our register file. Because we had explicitly designed our register file in a Xilinx schematic using 16 bit registers and large (256 to 16 bit) multiplexers to choose between outputs, our design was too large to be implemented on the FPGA board. Dr. Merkle suggested that we could achieve a much more efficient design by expanding his pre-written register file written in Verilog code. We then created a component from the Verilog and built a on top of this basic register file which included our special button state register and zero register.

However, because we had made very major changes to basic components of our processor (and made other, smaller changes, to almost all components) we had to run a series of ModelSim tests at this point to determine if the processor was still functioning correctly. We ran into a number of problems here, including an infinite loop caused by our "reset" command which caused ModelSim to hang indefinitely.

Lessons Learned

Looking back on the FPGA implementation process, I feel that it was well worth the effort, as we learned a lot—not just about Xilinx and the associated tools—also about how to search through and read technical documentation. We also feel because of our through documentation, those lessons that we learned will be of great help to future classes. Having access to the procedures outlined in this document would have saved us countless hours for sure.

Also, we feel that it would have been much easier to make the final step from Xilinx model of FPGA if we had known about some of the limitations and capabilities of the FPGA before we started the Xilinx model. For example, we would have definitely generated the programming files for each of our components and tested them separately on the FPGA as well as in Xilinx.. As it was, our implementation and testing method for the FPGA implementation was nowhere near as organized or rigorous as our Xilinx testing method. It essentially amounted to a "Big Bang" test because that was all we could do without taking our processor apart and rebuilding it, testing at every stage (a luxury which time did not allow).

Section
7

Conclusion

This section wraps up our final report, describes our final results, and lists some of the most important lessons that we learned throughout the process.

General Conclusions

With a strong start and a strong finish, this project was completed soundly and swiftly. Overall, this project went very well. All of the milestones were met in a timely manner. We held regular team meetings, kept accurate records of our decisions during those meetings, and assigned tasks for group members to complete between meetings to keep the project progressing smoothly. The assembly language we created for our processor succeeded in allowing all the instruction a programmer for our processor would want in under the 16 maximum instructions that our design allowed. Our thorough testing process ensured that testing and implementation went smoothly as well. After verifying that each individual component worked properly, the entire data-path was constructed without the memory component. The first few instructions were tested with this simplified data-path and, once we were sure the data-path was correct, we wrote mini-programs to test the other instructions.

The final step in our implementation was attempting to load our processor onto a Field Programmable Gate Array. For the implementation of the FPGA, we had to first figure out how all of the inputs and led outputs connected to the peripheral board. This step was successfully completed and we feel that we do now have a good understanding of the input-output mechanisms on the board. However, after battling numerous errors from all parts of our processor which were generated during the design synthesis procedure, we were forced to give up due to time considerations.

Overall, both the Xilinx and the FPGA implementations were very strong learning experiences and were both worthwhile, even if we were not able to get the FPGA working.

Total Component Count

The following text box summarizes the Xilinx report on the composition and component count of our processor. **The total number of components was: 2672.**

Component Count for Final Processor

```
Design Statistics
# IOs                    : 301    TOTAL: 301

Macro Statistics :
# Registers              : 53
#      1-bit register    : 53          TOTAL: 53

Cell Usage :
# BELS                   : 1542
#      AND2              : 127
#      AND2B1            : 32
#      AND3              : 32
#      AND3B1            : 32
#      AND5              : 1
#      AND5B1            : 4
#      AND5B2            : 6
#      AND5B3            : 4
#      AND5B4            : 1
#      BUF               : 20
#      GND               : 15
#      INV               : 34
#      LUT1              : 16
#      LUT1_L            : 16
#      LUT2              : 14
#      LUT2_L            : 3
#      LUT3              : 274
#      LUT3_L            : 1
#      LUT4              : 664
#      LUT4_L            : 4
#      MUXCY             : 2
#      MUXCY_L           : 6
#      MUXF5             : 112
#      OR2               : 81
#      OR3               : 16
#      VCC               : 7
#      XOR2              : 2
#      XOR3              : 16          TOTAL: 1542
```

```
# FlipFlops/Latches              : 431
#      FDC                       : 45
#      FDCE                      : 385
#      FDP                       : 1              TOTAL: 431
# Tri-States                     : 28
#      BUFE                      : 28             TOTAL: 28
# Clock Buffers                  : 1
#      BUFGP                     : 1              TOTAL: 1
# IO Buffers                     : 300
#      IBUF                      : 32
#      OBUF                      : 252
#      OBUFT                     : 16             TOTAL: 300
# Logical                        : 8
#      NOR4                      : 8              TOTAL: 8
# Others                         : 8
#      FMAP                      : 8              TOTAL: 8
=====================================================
==========================

Device utilization summary:
---------------------------

Selected Device : 2s200epq208-6

 Number of Slices:              770  out of   2352    32%
 Number of Slice Flip Flops:    431  out of   4704     9%
 Number of 4 input LUTs:        992  out of   4704    21%
 Number of bonded IOBs:         300  out of    146   205% (*)
 Number of TBUFs:                28  out of   2352     1%
 Number of GCLKs:                 1  out of      4    25%
```

Notice the number of bonded IOB's is over the amount provided on the Spartan2E board. Also, the overall component count is quite high. Because of this, we did a large amount of optimization during our Xilinx implementation and feel it is also important to include the component count for the latest version of the processor optimized for the FPGA (note that this version does not yet actually function on the FPGA). **The total component count for the FPGA version was 1317.**

Component Count for Final Processor (FPGA optimized version)

```
Design Statistics
# IOs                            : 17             TOTAL: 17
```

```
Macro Statistics :
# Registers                    : 56
#      1-bit register           : 53
#      16-bit register          : 3
# Multiplexers                  : 3
#      16-bit 16-to-1 multiplexer  : 3                    TOTAL: 56

Cell Usage :
# BELS                         : 992
#      AND2                     : 112
#      AND2B1                      : 32
#      AND3                     : 32
#      AND3B1                      : 32
#      AND4                     : 6
#      AND5                     : 1
#      BUF                    : 23
#      GND                     : 38
#      INV                    : 61
#      LUT1                    : 16
#      LUT1_L                    : 16
#      LUT2                    : 14
#      LUT2_L                    : 3
#      LUT3                    : 130
#      LUT3_L                    : 1
#      LUT4                    : 328
#      LUT4_L                    : 4
#      MUXCY                      : 2
#      MUXCY_L                      : 6
#      MUXF5                     : 16
#      OR2                    : 82
#      OR3                    : 16
#      VCC                    : 3
#      XOR2                    : 2
#      XOR3                    : 16              TOTAL: 992
# FlipFlops/Latches              : 175
#      FD_1                    : 1
#      FDC                    : 44
#      FDCE                    : 129
#      FDP                    : 1                  TOTAL: 175
# Tri-States                    : 44
#      BUFE                    : 28
#      BUFT                    : 16              TOTAL: 44
# IO Buffers                    : 17
#      IBUF                    : 6
```

```
#     IBUFG                  : 2
#     OBUF                   : 9              TOTAL: 17
# Logical                   : 8
#     NOR4                   : 8
# Others                    : 8              TOTAL: 8
#     FMAP                   : 8              TOTAL: 8
======================================================
==========================

Device utilization summary:
---------------------------

Selected Device : 2s200epq208-6

 Number of Slices:                  359  out of   2352    15%
 Number of Slice Flip Flops:         175  out of   4704     3%
 Number of 4 input LUTs:              512  out of   4704    10%
 Number of bonded IOBs:                17  out of    146    11%
 Number of TBUFs:                    44  out of   2352     1%
```

Processor Timing Information

The Xilinx Programing File Generation Report also contained timing information for our processor. Specifically we were interested in the minimum period for each clock cycle and the maximum clock rate which our processor could achieve. Below is a summary of the results which indicates that our **Minimum Clock Period was 68.042ns** and that our **Maximum Clock Rate was 12.697 MHz**.

Timing Summary

```
---------------
   Minimum period: 68.042ns (Maximum Frequency: 14.697MHz)
   Minimum input arrival time before clock:         67.702ns
   Maximum output required time after clock:       63.307ns
   Maximum combinational path delay:    62.967ns
```

Path Analysis –Logic versus Route

The other interesting piece of timing data which Xilinx provided to us was the division of the total clock cycle time taken up between the logic and the data-path itself. The breakdown was:

Logic:19.591ns logic Route: 43.376ns Total 62.967ns

We also ran a final test program which simulated inputting the hexadecimal number 0x13B0 into our processor and calculating the smallest number relatively prime to it. **Our processor got the correct output of 0x000B in 40900720 ns**. Since in the test we were using a 100 ns period clock, this means that it took approximately **409007.2 clock cycles to run the entire program.**

In order to determine the actual number of instructions that were executed throughout the entire run of the program, we has to determine the average number of clock cycles per instruction that the processor did during execution. To do this, we determined that the GCD algorithm and the loop in the Lowest Relative Prime algorithm which called it were run thousands of times more than the other parts of the code, and were thus the only important sections in terms of the average cycles per instruction. We then tallied up the number of instructions of each type in these parts of the program and did a weighted average to get **3.37 as the approximate average number of clock cycles per instruction. Thus, approximately 121367 instructions were executed during the entire run of the program.**

Once we had this information we combined the fact that our ideal clock cycle time would be 62.967ns, with the fact that we had to run 409007.2 clock cycles to execute the entire program, to determine that **our ideal CPU time for that particular input was .0286 seconds.**