Final Report

C-Biscuits, Team 4
Caleb Harper
Lissa Avery
Matt Salisbury
Mike Turpyn

Table of Contents

Executive Summary

For the final project in CS232, we were to design “a miniscule instruction set.” We were then asked to model our design, test it, debug it, assess its performance, and possibly implement it on a Field Programmable Gate Array (FPGA) microchip. This executive summary is intended to give the reader a brief summary of our design flow and our current status.

For the first milestone of our project, we had to design the foundation of our processor. We decided we would have 16 registers. We thought it would be convenient to have a “zero” register. Other registers included two argument registers, three temporary registers, three saved temporary registers, two kernel registers, a stack register, a return address register, a register for the assembler, a register for the results, and a display register. We also decided in milestone I to have four different instruction types: R1, R2, C, and B types.

For the next milestone, we decided on certain components we would need for our processor. These parts included a memory, instruction register, register file, temporary registers, an ALU, a Sign Extend, and a concatenate part. The basic functionalities of each part can be found in our design document. Also in milestone II, we designed our first RTL (which also can be found in our design document).

For milestone III, we had to produce the first model of the datapath. We decided to keep our datapath similar to MIPS because we understood it better than other processor types, and we could use MIPS as a model for future development. Another section of milestone III was determining the function of each control signal. We ended up having 18 different control signals. Also in milestone III, we had to define a test plan for our processor. We felt that our test plan was extensive and covered every functionality required so that our processor could be successful.

In milestone IV, we designed our control component. Our processor’s control was implemented using a finite state machine. The combinational logic unit, called the ALUOp Calculator, uses the Op code and function code, both of which are inputs to the Control, to determine the appropriate ALUOp for the instruction. Also in this section we had to develop a test plan for the control part. Our test included several simulations to ensure correct transitions.

For the last milestone of the project, we needed to implement our components into Xilinx. Each one of us was assigned various parts to complete. We were supposed to design, test, and create a part of each one of our components. We felt that if each individual part was tested that it would ensure that our final product would be successful. We used ModelSim to test our individually components, partly because Caleb understood how to use it, and also because upperclassmen who completed the project in the past recommended testing everything using ModelSim.

In the end, the processor was capable of executing several instructions—including branching, storing and loading words to and from memory, and R-type instructions—but not the entire GCD program. Several issues with developed with Xilinx and the integration of the components during the last few hours of project time that hindered our success. This includes a problem when the entire processor was moved from one team member's computer to another's that caused an error ingenerating timing reports and gate counts.

Introduction

The purpose of the project described in this document is to design a small instruction set and model it using Xilinx software. The instruction set was designed to operate on a 16-bit microprocessor, also designed in this project.

The team was given a problem—to find a number relatively prime to a given number using Euclid's algorithm—for which it designed an instruction set to solve. The instruction set needed to be sufficiently robust to be able to handle nested procedures, parameters, and general computations not necessarily needed for the execution of Euclid's algorithm.

In addition to these basic requirements, the processor outlined here supports recursive procedures through stack memory usage and interrupts from five input devices.

Beginning with some knowledge of the 32-bit MIPS instruction set and how that might be implemented, the team devised a 16-bit instruction set meeting the above requirements. The design includes the actual instructions and their translation to machine code, the register transfer language instructions to execute the programmer-level instructions, and a corresponding datapath.

Following this design, the datapath was constructed and tested in Xilinx. CoreGen use was permitted where applicable, and StateCAD/Verilog use was permitted for the control units.

Testing followed implementation. All components were tested using test benches in ModelSim, but the control finite state machine unit underwent preliminary exhaustive testing in StateCAD's StateBench.

Summary of Design

Our initial design phase consisted of choosing what types of instructions we planned to include in our architecture. Using Euclid's algorithm as a source, we translated a version of the algorithm from Java to assembly. This process allowed us to determine what instructions were absolute necessities, and which instructions were optional. Pages 49 and beyond of the accompanying design journal (Appendix B) has detailed descriptions of each instruction type as well as each individual instruction. In order to leave as many options open in the event a future revision was required, our group created a single branch instruction. This *branch if not equal* instruction assumes the role of its counterpart; however, it requires manipulation on the part of the programmer. Additional drawbacks include the limited ability to branch at most 31 instructions in either direction. This has obvious limitations in the size and scope of any program written in our assembly language. Nevertheless, this instruction fulfills the requirements for the specified program.

Most notable deviations from the standard MIPS instruction set are the following instructions: *jump register and link, load address, move to co-processor,* and *move-from-coprocessor*. In an effort to make our instruction set as compact and efficient as possible, we combined the three MIPS jump instructions into one single instruction. This instruction jumps to the value of the given register, and automatically stores the value of the PC in the return register. While this shortens our instruction list, it makes programming slightly more difficult. The programmer must be aware that they will change the return address every time that they perform a jump.

The load address instruction is actually a pseudo-instruction that places the memory location of the given label into the given register. We initially wanted to avoid the inclusion of pseudo-instructions in our architecture to keep things as simple as possible, however we eventually decided the benefits to the programmer outweighed the slight complication of our instruction set. The final design of the load address instruction condensed a *load upper*, *load lower*, and *or* instruction all into one instruction.

Another design choice was the decision to treat labels as immediate values for branching purposes. While this simplified our instruction set, it limits the distance that can be branched to the length of an immediate value. However, since this value was already limited by other constraints, we did not actually need to make any additional tradeoffs to make this change. It was simply an easier method to implement labels within our architecture.

In order to more easily handle exceptions, we included room for manipulation of the exception data in our main instruction set. We needed some means of moving the data to and from the cause register, and solved this problem by creating a *move to co-processor, move-from-coprocessor.* Although the addition of these instructions limited the number of other instructions that we could include, we found enough room to include everything.

Once we had determined which instructions would be included in our architecture, we needed to finalize the format for each type of instruction. Loosely basing our design off of MIPS, we began each instruction with the operation code, and ended them with the function code. This was done to make the processor easier to wire when it came time to actually build the design.

The second major issue that we addressed in our initial design phase was the number of registers to include. We wanted to include a power of two, to limit the number of wasted bits in our design. We compiled a list of all of the registers that would be a necessity and then rounded up to the nearest power of two. A full list of the registers and their purposes can be found on pages 8 and 9 of the design journal.

There were three design changes at this point in the project that our group would consider were we able to do the project again. The first of these changes would be to increase the amount of temporary registers available. This would allow for greater versatility when programming. The drawback, however, is that there was little to no room in our current design to fit these. If they were added, then another bit would need to be included to account for the extra register, but this would greatly affect our instruction design. At that point, it might be easier to expand the processor to 32 bits instead. Another option for our group was to intentionally increase the number of register up to 64 from 16. This idea would have the same drawbacks as the previous idea; however, we now have additional registers to work with. A possible solution would be to specify particular registers to particular instructions. Another design change we might want to make is the addition of more pseudo-instructions, such as a load immediate instruction. This would make things easier for the programmer, while not affect the performance of the processor, as the compiler would take care of the translation to machine code. Once we finalized our instruction set, register conventions, and op-code designations, we were ready to move on to the next phase of our project.

The second phase of our project consisted of an RTL description of each instruction type and a list of the processor components we wanted to include, and specifically what each one would accomplish. Page 30 (Appendix A) shows the RTL description of our group's architecture. The benefits of our datapath design are the relatively few clock cycles that our instructions require. The tradeoff here is that our datapath and control became more complicated, however from a performance standpoint; a lower CPI is preferable to a simplified datapath.

Our component list looked very similar to the list of components for the MIPS instruction set. A list of our components and their inputs and outputs as well as a brief description of each can be found on page 27. However, there were some noticeable design difference between our list and the MIPS list. The first of these is the number of control bits in Memory. MIPS uses two 1-bit control bits, MemRead and MemWrite. The inclusion of MemRead is only for the Memory data register. Our more simplified design did not require such a register, and so we had no need for its accompanying control bit. Other design choices include a separate set of co-processor registers to deal with the mfc0 and mtc0 instructions. This register block is designed to hold the EPC and Cause registers, as well as Display and other co-processes. While this will add additional components to our datapath, we felt the accessibility of these registers and resulting ease of exception handling was a good design tradeoff.

Another design decision was the inclusion of the Status register as a separate component. A control bit is sent directly to the Status register, and it is wired directly back to control. This register not only prevents one exception from interrupting another, leading to an infinite loop, but also does so as soon as it changes, disregarding any triggering from the processor clock.

To facilitate the implementation of our branch instruction, the ALU was given an output bit called NotZero. This value was a 1 if the two inputs were not equal and 0 if they were equal. We believe that this output would simplify the branch instruction without adding any drawbacks.

The next phase of our design process required our group to create both the block diagram of our datapath. When dealing with the two exception instructions, however, we chose to simplify control by adding another clock cycle to the instructions. In this case, however, since exceptions will not account for the majority of the instructions, we felt that this was an acceptable tradeoff to ensure that our group could finish and test our processor before the deadline.

While creating the datapath, we found that we needed to make some adjustments in the way that our design was implemented. Almost all of these changes were done by using multiplexers. While this added additional components and control bits to our design, it made further modifications much easier. Also, if we decided to undo one of the changes that we had made, it would be a simple process of removing the multiplexer.

If the number of registers in our project design was increased, then it would have possible to eliminate the co-processor register block from our datapath by combining it with our normal register block. While this design would have all of the drawbacks mentioned in the previous changes, it would have an additional advantage of a simpler datapath. Not only could we follow the same path for the exception instructions, but the control bits would be eliminated, making the creation of the processor much easier. Another change that our group would make if we were able to do the project again would be to write the user input to control instead of the display register.

Conclusion

As mentioned in the Executive Summary, the processor is not working completely according to specifications, making performance information not the most accurate. Although not all data can be directly obtained, some can be calculated or derived from what we *can* get.

Our instruction set required 56 instructions to calculate the greatest common denominator using Euclid's algorithm. This yields a CPI of 4.219.

The processor's clock cycle frequency is 7.193 MHz, which leads to a cycle time of 139 ns.

For the calculation of a number relatively prime to 0x13B0, the execution of 470 instructions are required to obtain a result of 0x000B, which gives a cycle count of 1982.93 and an execution time of 275.67 ns.

Problems with Xilinx file migrations prevented the obtainment of a complete gate count of the processor.

Appendix A – Design Documentation

Instructions and usage

Introduction

The following document details the instructions which will be performed by our processor. Due to the limitations of a 16-bit machine, several of our instructions have been condensed to the point that an ill-prepared user might get into trouble. Please pay special attention to all notes marked with an asterisk '*'. These provide essential warnings for our instruction set.

Example

'command' 'argument'

Op	$rt	0	Func
2	4	8	2

This is just an example to help demonstrate the format of the instructions. The operation code ("Op code") and function bits contain the appropriate information to select the particular instruction desired. Some commands have no function code. The other bits contain either the address of a register or an immediate value, depending on the particular command. The numbers underneath the table detail the number of bits that each part of the instructions occupies.

R1- and R2-Type Instructions

Addition

add $rd, $rs, $rt

Op	$rs	$rt	$rd	Func
2	4	4	4	2

The *add* command sums the values of the registers in the registers *$rt* and *$rs*, then stores the value in the register *$rd*.

Subtraction

sub $rd, $rs, $rt

Op	$rs	$rt	$rd	Func
2	4	4	4	2

Similar to the add command, the sub command takes the value in register $rs, subtracts from it the value in register $rt, and stores the value in register $rd.

Store Word

sw $rd, $rs, $rt

Op	$rs	$rt	$rd	Func
2	4	4	4	2

The format of sw is slightly different than the standard MIPS instruction. The $rd register contains the value which is to be stored into memory, the $rs register contains the address of the destination in memory, and the $rd register contains the offset of the memory address.

Set Less Than

slt $rd, $rs, $rt

Op	$rs	$rt	$rd	Func
2	4	4	4	2

The slt instruction changes the value of register $rd based on the value of $rs relative to register $rt. If the value in register $rs is less than the value in register $rt, then the value of $rd is set to 1. Otherwise, the vale of $rd is set to 0.

Shift Left Logical

sub $rd, $rs, $rt

Op	$rs	$rt	$rd	Func
2	4	4	4	2

In this instruction set, the shift left logical command is formatted as displayed above. The value in register $rs is shifted to the left by the amount in the register $rd, the resultant calculation is stored into register $rd.

And

and $rd, $rs, $rt

Op	$rs	$rt	$rd	Func
2	4	4	4	2

This instruction performs a bitwise logical nand on two registers, $rs and $rt and stores the resulting value into register $rd.

Or

or $rd, $rs, $rt

Op	$rs	$rt	$rd	Func
2	4	4	4	2

This instruction performs a bitwise logical or on two registers, $rs and $rt and stores the resulting value into register $rd.

Load Word

lw $rd, $rs, $rt

Op	$rs	$rt	$rd	Func
2	4	4	4	2

The format of lw is slightly different than the standard MIPS instruction. The $rd register contains the value which is to be loaded from memory, the $rs register contains the address of the memory destination, and the $rd register contains the offset of the memory address.

B-Type Instructions

Branch If Not Equal

bne $rs, $rt, label

Op	$rs	$rt	Offset
2	4	4	6

Since all branch type commands can be calculated using either branch if not equal or branch if equal, and given the limited space for instructions, only one of the branch instructions was included in the instruction set. The command compares the values in registers $rs and $rt, and if the two are not equal it tries to branch to the address of label.

* The maximum distance bne can go is $2^5 – 1 = 31$ instructions forward or backward.

C-Type Instructions

Load Upper Immediate

lui $rt, imm

Op	$rs	Immediate	Func
2	4	8	2

The load upper immediate manipulates the eight most significant bits in a given register. It replaces the eight upper bits in the register $rs with the value specified in the immediate field, and the lower eight bits are subsequently replaced with zeros.

Load Lower Immediate

lli $rt, imm

Op	$rs	Immediate	Func
2	4	8	2

The load lower immediate manipulates the eight least significant bits in a given register. It replaces the lower bits in the register $rs with the value specified in the immediate field, and the upper eight bits are subsequently replaced with zeros.

Jump Register and Link

jalr $rt

Op	Immediate	$rt	Immediate	Func
2	4	4	4	2

The jump register and link is the only jump command in the instruction set. It jumps to the value specified in register $rs, and it also automatically loads the return value (current value of the PC register) into the register $ra. A standard jump can be accomplished if the address for a given label is loaded into a register and the jalr command is called referencing that particular register.

Load Address

la $rs, label
pseudoinstruction

The load address commands places the memory location of the given label into register $rs.

Move To Co-Processor

mtc0 $rs, $co

Op	$rs	c0	Extra	Secondary Func	Func
2	4	4	3	0	2

This command moves information from a general purpose register to one of the registers in the co-processor. This command is especially important for interrupt and exception handling. The information specified in the co-processor register $co will be overwritten upon completion of the command.

Move From Co-Processor

mfc0 $rs, $co

Op	$rs	c0	Extra	Secondary Func	Func
2	4	4	3	1	2

This command moves information from one of the co-processor register to one of the general purpose registers. The information in the general purpose register $rs will be overwritten when the data is moved. This command is used primarily for dealing with interrupts and exceptions.

Registers and Conventions

Register name	Number	Usage
$zero	0	Constant 0
$at	1	Reserved for assembler
$v0	2	Results of a function
$a0	3	Argument 1
$a1	4	Argument 2
$t0	5	Temporary (not saved across calls)
$t1	6	Temporary (not saved across calls)
$t2	7	Temporary (not saved across calls)
$s0	8	Saved temporary
$s1	9	Saved temporary
$s2	10	Saved temporary
$k0	11	Kernel use only
$k1	12	Kernel use only
$sp	13	Stack pointer
$ra	14	Return address for function calls
$t3	15	Temporary (not saved across calls)

Coprocessor registers	Number	Usage
$EPC	0	Address of interrupt-causing instruction
$Cause	1	Interrupt type
$display	2	Output to display

In addition, there is a program counter register that is not part of either register set. It holds the address of the instruction following what is currently being executed.

For optimal register usage by large-sized programs, it is best to define a set of conventions defining how and where registers can be read and written. In this processor, there are 16 general purpose registers to be managed in the defined conventions; one is wired to ground (effectively a "0" register) and cannot be written by the programmer.

Reg #	Reg name	Applicable Conventions
0	$0	Hardwired to ground, always zero
1	$at	A temporary register used by the assembler, primarily used for storing jump addresses and executing jumps. This register can be used by the coder, but should be done so sparingly and with extreme caution since the value is liable to change without warning
2	$v0	This register is used to return values from a procedure call. A value that is to be returned from a procedure should be stored here prior to returning. Extra values should be returned on the stack, following stack conventions.
3-4	$a0-$a1	These register are used for passing arguments into a procedure. Values that are to be passed should be loaded into these registers prior to calling the procedure. Extra values could be sent on the stack, as long as the caller follow proper stack convention

5-8	$t0-$t3	These registers are for temporary data. There is no guarantee that they will be saved across calls
9-11	$s0-$s2	These registers are for saved data. Before these registers can be used, the values must be stored by the callee to the stack; the original values should then be restored before returning to the calling function.
12-13	$k0-$k1	These are registers used by the processor for exception calls. They should NOT be used by the coder.
14	$sp	The stack pointer register points to the current position on the stack. Any modifications made to this register should ultimately be ‘undone’ prior to returning (i.e. if 4 is subtracted from $sp, then 4 must be added before returning)
15	$ra	This register will contain the return address anytime a jalr command is used. It should NOT be used by the coder except to obtain the return address for a later purpose

Instruction Formats

R1 Type Instructions

opcode	rs	rt	rd	func code
2-bit	4-bit	4-bit	4-bit	2-bit

opcode = Basic operation of the instruction

func code = This field selects the specific variant of the operation in the opcode field.

rs = The first register source operand

rt = The second register source operand

rd = the register destination operand (gets the result of the operation).

R2 Type Instructions

opcode	rs	rt	rd	func code
2-bit	4-bit	4-bit	4-bit	2-bit

opcode = Basic operation of the instruction

func code = This field selects the specific variant of the operation in the opcode field.

rs = The first register source operand

rt = The second register source operand

rd = the register destination operand (gets the result of the operation).

B Type Instructions

opcode	rs	rt	offset
2-bit	4-bit	4-bit	4-bit

opcode = Basic operation of the instruction

rs = The first register being compared

rt = The second register being compared

offset = a register when data is transferring

C Type Instructions

opcode	rt	undesignated	func code
2-bit	4-bit	8-bit	2-bit

opcode = Basic operation of the instruction

func code = This field selects the specific variant of the operation in the opcode field.

rt = The register containing an address

undesignated = bits not currently being used

Instruction translation to machine language

R1-Type Instructions

Opcode - 10

slt

opcode	rs	rt	rd	func code
10	….	….	….	00

sll

opcode	rs	rt	rd	func code
10	….	….	….	01

or

opcode	rs	rt	rd	func code
10	….	….	….	10

and

opcode	rs	rt	rd	func code
10	….	….	….	11

R2 Type Instructions

Opcode - 11

add

opcode	rs	rt	rd	func code
11	….	….	….	00

sub

opcode	rs	rt	rd	func code
11	….	….	….	01

lw

opcode	rs	rt	rd	func code
11	….	….	….	10

sw

opcode	rs	rt	rd	func code
11	….	….	….	11

C Type Instructions

Opcode - 00

lli

opcode	rs	imm	func code
00	….	….	00

lui

opcode	rs	imm	func code
00	….	….	01

la

pseudoinstruction

jalr

opcode	imm	rt	imm	func code
00	….	….	….	11

mtc0

opcode	rs	c0	extra	secondary func	func code
00	….	….	0	0	10

mfc0

opcode	rs	c0	extra	secondary func	func code
00	….	….	0	1	10

B Type Instructions

Opcode - 01

bne

opcode	rs	rt	offset
01	….	….	……

Example assembly-language programs

Euclid's Algorithm

```
main: lli $s0, 2        # set "m" to 2 initially
      mfc0 $t0, $2      # grab the "n" from the display register

      add $a0, $t0, $0  # set our first parameter to "n"
      add $a1, $s0, $0  # set second parameter to "m"

      la $s1, loop      # create our loop label for usage as a jump
                        # site
      la $s2, GCD # likewise for the GCD function

loop: jalr $s2          # call GCD

      lli $t1, 1        # the constant 1; the limited registers means
                        # this must be set every time around

      add $s0, $s0, $t1 # increment "m"

      mfc0 $t0, $2      # grab the "n" from the display register
                        # the scarce register situation leads to this
                        # repetition

      add $a0, $t0, $0  # set our first parameter to "n"
      add $a1, $s0, $0  # set second parameter to "m"

      bne $v0, $t1, $s1 # loop; if the return variable wasn't a one (1)
                        # jump back and resume looping

      la $t1, exit
      jalr $t1

GCD:  lli $t0, -1       # there are three saved variables to save off
                        # to the stack
      lli $t1, 1        # a good, useful constant to have

      add $sp, $sp, $t0 # increase stack space

      sw $s0, $sp, $0   # save the first one
      add $sp, $sp, $t0
      sw $s1, $sp, $0   # and the second one...
      add $sp, $sp, $t0
      sw $s2, $sp, $0   # and the final one...

      add $s0, $a0, $0  # store parameter "n" for our usage;
                        # also known as "a"
      add $s1, $a1, $0  # store parameter "m" for our usage as well;
                        # also known as "b"

      la $s2, GCD_loop
      add $sp, $sp, $t0
      sw $ra, $sp, $0   # store the return address, since we are doing
                        # a bit of internal jumping
```

```
GCD_loop: la $t2, return
      slt $t0, $0, $s1  # if b !> 0, get out of dodge
      bne $t0, $t1, $t2 # Note this is compared to a *1*, not a zero

      la $t2, switcher

      slt $t0, $s0, $s1 # if b > a, swap 'em.
      bne $t0, $t1, $t2 # Again, this compares the slt with 1

      sub $s0, $s0, $s1 # otherwise, subtract b from a and store in a


switcher: add $t2, $s0, $0 # swapperoo
      add $s0, $s1, $0
      add $s1, $t2, $0
      jalr $s2          # back to the loop


return: add $v0, $s0, $0      # set the return variable to "a"

      lli $t0, 1        # about to unload the stack
      lw $ra, $sp, $0
      add $sp, $sp, $t0
      lw $s2, $sp, $0
      add $sp, $sp, $t0
      lw $s1, $sp, $0
      add $sp, $sp, $t0
      lw $s0, $sp, $0
      add $sp, $sp, $t0
      jalr $ra

exit: # done
```

Implementation of Exam 1 Earnings Calculation

# C-biscuit implementation of Exam 1 assembly code

.data

```
numHours:   .word 9
rate:       .word 30
earnings:   .word 0

.text

main:       lui $at, 0
            lli $t0, numHours
            or $t0, $t0, $at
            lli $t1, rate
            or $t1, $t1, $at

            add $a0, $t0, $0
            add $a1, $t1, $0

            la $s0, CalcEarnings
            jalr $s0

            add $t0, $v0, $0
            la $s1, earnings
            sw $t0, $s1, $0

            lli $v0, 10
            syscall

CalcEarnings:     add $t0, $a0, $0
            add $t1, $a1, $0

            # keep same $a0, $a1

            lui $at, 0
            lli $t2, -2             # preparation for stack usage for
return addy
            or $t2, $t2, $at
            add $sp, $sp, $t2
            sw $ra, $sp, $0
            add $sp, $sp, $t2
            sw $s0, $sp, $0

            la $t2, CalcGross
            jalr $t2

            add $s0, $v0, $0        # snag the gross income
            add $a0, $s0, $0  # back out for consumption

            #la $t2, CalcTax
            #jalr $t2

            #add $t0, $v0, $0 # grab calculated tax
            lli $t0, $0

            sub $t2, $s0, $t0 # net = gross - tax

            add $v0, $t1, $0

            lli $t2, 2
            or $t2, $t2, $0

            lw $ra, $sp, $0
            add $sp, $sp, $0
            lw $s0, $sp, $0
            add $sp, $sp, $t2
            jalr $ra

CalcGross:  lli $t2, -2       # preparation for stack usage for caller-
saved registers
            or $t2, $t2, $0
            add $sp, $sp, $t2
            sw $ra, $sp, $0
            add $sp, $sp, $t2
            sw $s2, $sp, $0
            add $sp, $sp, $t2
            sw $s1, $sp, $0
            add $sp, $sp, $t2
            sw $s0, $sp, $0

            lli $t0, 0
            or $t0, $t0, $0
            lli $s0, 0              # a temp gross
            or $s0, $s0, $0
            la $s1, cg_loop

      cg_loop: lli $t1, 1
            or $t1, $t1, $0
            add $s0, $s0, $a1 # add the rate to the gross again
            add $t0, $t0, $t1

            slt $t2, $t0, $a0 # while our counter is less than the
                              # number of hours, add the rate to
                              # itself.  Horrible multiplication
            bne $t2, $t1, $s1 # hop back to re-loop

            # done with looping
            add $v0, $s0, $0

            lli $t2, 2        # preparation for stack usage for caller
                              # saved registers
            or $t2, $t2, $0
            sw $s0, $sp, $0
            add $sp, $sp, $t2
            sw $s1, $sp, $0
            add $sp, $sp, $t2
            sw $s2, $sp, $0
            add $sp, $sp, $t2
            sw $ra, $sp, $0
            add $sp, $sp, $t2

            jalr $ra
```

Exception-Handling Code

```
            lli $at, -1
            add $sp, $at, $0
            sw $t3, $sp, $0

            mfc0 $t3, $Cause
            bne $t3, $0, BeginEuclid #If we have the second interrupt,
                                     # start the program

#Otherwise, we're dealing with the loading of the number
LoadNum:    lli $k0, 176
            lui $k1, 19
            or $k0, $k0, $k1
            mtc0 $Display, $k1
            bne $k0, $0, Wait

BeginEuclid:      lli $at, 1
            add $sp, $at, $0
            lw $t3, $sp, $0

            mfc0 $k0, $EPC
            jalr $k0 #Unfortunately, this kills whatever was in $ra
                     # previously; a flaw, although in our case, there
                     # shouldn't be anything in $ra when this is
                     # executed.
```

```
Wait:        #We need to exit and wait for the next interrupt.  We don't
             # have an instruction for this--a flaw, alas.
```

Machine Code – Euclid’s Algorithm

```
0x2008
0x1486
0x291c
0x2a40
0x1428
0x1801
0xa95a
0x1454
0x1801
0xad5a

0x02c3
0x1801
0xad5a
0x2c03
0x1801
0xe658
0x1486
0xcd40
0xd240
0x49b9
0x14e0
0x1801
0x995a
0x0183

0x17fc
0x1804
0xfb94
0xe783
0xfb94
0xeb83
0xfb94
0xef83
0xe4c0
0xe900
0x1490
0x1801
0xad5a
0xfb94
0xff83


0x9428
0x5588
0x9668
0x5581
0xe669

0xde40
0xe680
0xe9c0
```

```
0x02c3

0xce40
0x1404
0xff82
0xfb94
0xef82
0xfb94
0xeb82
0xfb94
0xe782
0xfb94
0x03c3
```

Machine Code – Earnings Calculation

```
0x0500
0x0608
0xc503
0xc604
0x282c
0x3806
0xc205
0x0980
0xf905
0x020a
0x4001

0xc305
0xc406
0x07fe
0xcd7d
0xfd0e
0xcd7d
0xfd08
0x277c
0x270c
0xc208
0xc803
0x0500
0xd857
0xc602
0x0702
0xed0e
0xcd0d
0xed08
0xcd7d
0x3e01

0x07fe
0xcd7d
0xfd0e
0xcd7d
0xfd0a
0xcd7d
0xfd09
0xcd7d
```

```
0xfd08
0x0500
0x0800
0x29ac

0x0601
0xc848
0xc565
0x8537
0x5dbc
0xc802
0x0702
0xfd08
0xcd7d
0xfd0a
0xcd7d
0xfd0e
0xcd7d
0x3e00
```

List of components and their interfaces

Program Counter (PC)
Input: 16-bit address selected from multiplexor.
Output: 16-bit address
Controls: PCWrite: 1-bit bus

Memory
Inputs: 16-bit address from ALUOut or PC; 16-bits of data from register B
Output: 16-bit contents of a memory location
Controls: MemWrite: 1-bit

Instruction Register (IR)
Input: 16-bit instruction from memory
Output: 16-bit instruction
Control: InstructionWrite: 1-bit

Register File (Reg)
Inputs: two 4-bit Read Registers; one 4-bit Write Register; one 16-bit Write Data
Outputs: 16-bit Read Data 1 and 2.
Control: 1-bit RegWrite

Co-processor Register File (C0Reg)
Inputs: one 4-bit Read Register; one 4-bit Write Register; one 16-bit Write Data
Outputs: 16-bit Read Data 1.
Control: 1-bit RegWrite

Register A (A)
Inputs: 16-bits of data from the Register File (Read Data 1)
Output: unaltered 16-bits of data
Control: none

Register B (B)
Inputs: 16-bits of data from the Register File (Read Data 2)
Output: unaltered 16-bits of data
Control: none

Register C (C)
Inputs: 16-bits of data from the Co-processor Register File (Read Data)
Output: unaltered 16-bits of data
Control: none

Status Register (Status)
Inputs: none
Output: none
Control: 1-bit Status

Arithmetic Logic Unit (ALU)
Inputs: two 16-bit operands
Output: 1-bit NotZero; 16-bit ALUResult
Control: 4-bit operation

ALUOut Register (ALUOut)
Input: 16-bit ALUResult
Output: 16-bit ALUResult, unaltered
Control: none

Sign Extend by 10 bits
Inputs: 6-bit branch offset
Ouput: 16-bit sign-extended word address
Control: none

Concatenate right by 8 (Conc8 Hi)
Inputs: 8-bit immediate
Output: 16-bit number with immediate in upper bits
Control: none

Concatenate left by 8 (Conc8 Low)
Input: 8-bit immediate
Output: 16-bit number with immediate in lower bits
Control: none

Descriptions of components

The program counter (PC) contains the 16-bit address of the instruction to be executed after the current instruction. Its value comes from the increment of the address (which is done via the addition of two to the current PC value), a jump address, or a branch address. Its only output is its own 16-bit value. It accepts one bit from the control specifying whether to its contents should be rewritten with waiting values.

The memory holds user-accessible data, accessible by 16-bit address. Given a 16-bit address from ALUOut or the PC, a 16-bit instruction or piece of datum, whichever lies at the given address, can be output. In this case, the one-bit MemRead control input must be enabled. For data to be written to memory, the one-bit MemWrite control input must be enabled. Likewise, the memory component will accept 16-bits of data from register B, to be placed in the address given by ALUOut.

The instruction register (IR) is a temporary register used for holding an entire instruction before it is decoded. It takes its 16-bit value from memory when the 1-bit InstructionWrite control is enabled, and outputs that same value for instruction decoding.

The register file (Reg) holds the contents of the processor's 16 general-use registers, accessible by four-bit number. Given two four-bit register numbers, the 16-bit contents of those two registers are output to Read Data 1 and Read Data 2. When the RegWrite control bit is asserted, the contents of the 16-bit Write Data input are written to the register indicated by the four-bit Write Register input.

Registers A and B are temporary registers used for holding the contents of a register (each) before they are processed further. They always write their contents at the appropriate time, so there are no control signals. The contents are obtained from the Register File's Read Data 1 and Read Data 2 for Registers A and B, respectively.

The co-processor's register file (C0Reg) holds three 16-bit registers, accessible by four-bit number. Given a four-bit register number, the 16-bit contents of that register are output to Read Data. When the C0RegWrite control signal is assert, the data at the 16-bit Write Date input are written to the register indicated by the four-bit Write Register input.

Register C is a temporary register used for holding the value from the co-processor's register file's Read Data output.

The Status register stops the processor from accepting more than one exception at a time.

The Arithmetic Logic Unit (ALU) performs the primary mathematical operations of the processor. Given two operands, it will perform an operation on them (as determined by the ALUOp control signal) and output the result. In addition, the ALU subtracts the two operands and asserts a NotZero output if the two numbers are unequal.

The ALUOut register (ALUOut) is another temporary register that is perpetually in write-mode. It obtains and holds the results of an ALU operation for a clock cycle.

There are also several concatenate left and right components and a sign extension component. These are simply hard-wired components used to properly place or multiply numbers and have no control signals.

RTL Description of Instructions

We broke our instructions down into 5 different sets of RTL blocks depending on the instruction. All of our R-type instructions will be executed with practically the same RTL, the LW (Load Word) and SW (Store Word) will require slight modifications.

In the C-type, the LLI (Load Lower Immediate) and LUI (Load Upper Immediate) will both have similar RTL blocks, but the JALR (Jump And Link Register) will be different, and will have its own set. Finally, the BNE (Branch if Not Equal) instruction will have an RTL chunk all to itself.

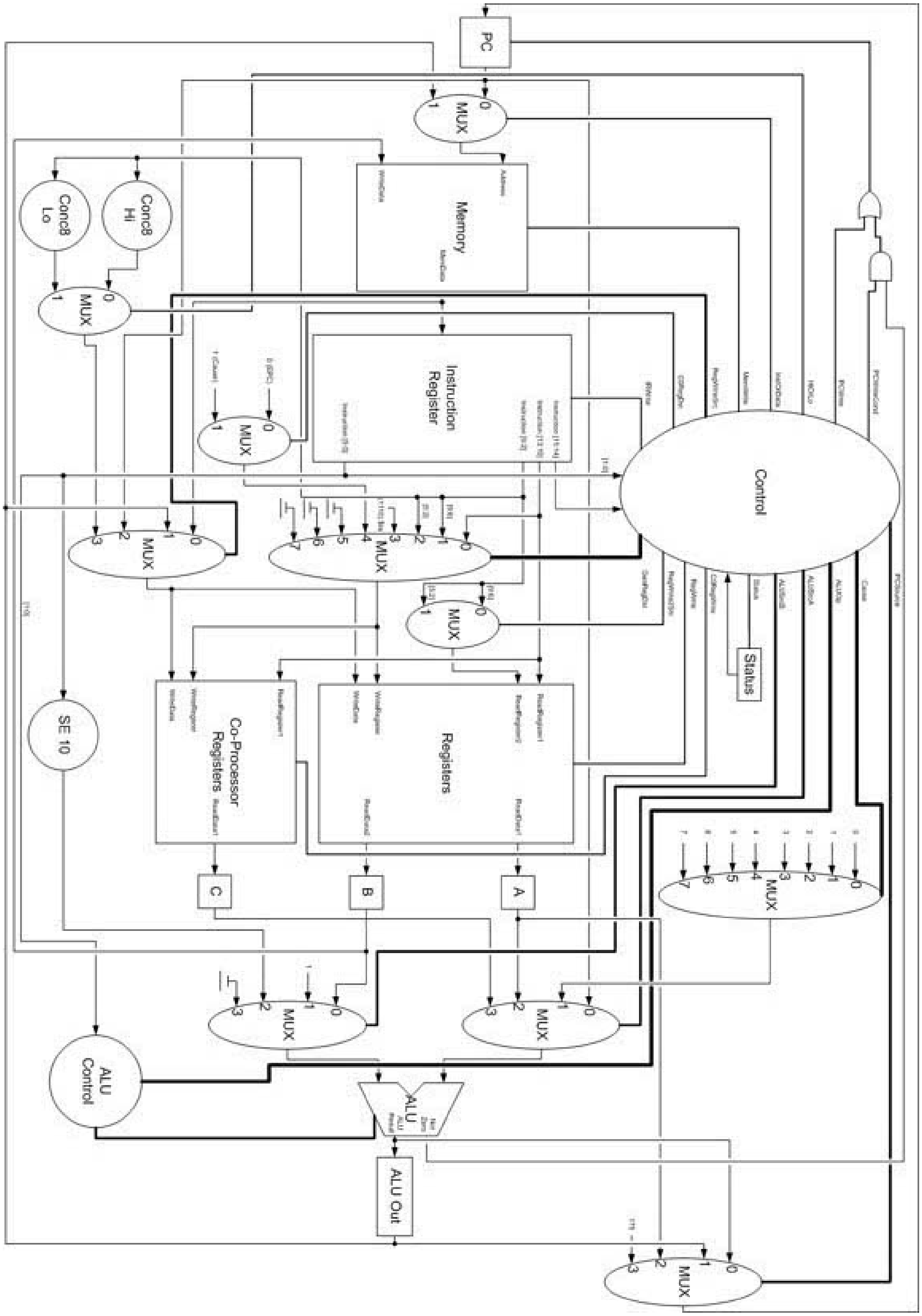
We differ slightly from MIPS in that only the first step is common amongst all of our different instructions. Differences occur in the second step, which allows us to do our jump command in just two steps instead of three. It does, however, complicate the control logic, but, hopefully, our small number of instructions will help keep it from being overwhelming.

Step Description	Standard R-type	Load Word/ Store Word	Branch if Not Equal	Load Lower Immediate/ Load Upper Immediate	Jump And Link Register
Get Instruction & increment PC	Instruction Register <= Memory[PC] PC<= PC+2				
	A<= Reg([IR[13:10]) B<= Reg(IR[9:6]) ALUOut<=PC+(SE[5:0] <<1)			Upper: reg([IR[13:10])<= IR[9:2] conc8* Lower: reg([IR[13:10])<= conc8(IR[9:2])	$ra<= PC PC<= reg(IR[13:10])
	ALUOut <= A op B	ALUOut <= A + B B<= reg[IR(5:2)]	If (A != B) PC<= ALUOut		
	Reg([IR(5:2)]<=ALUOut	Load: reg([IR(5:2)] <= Mem(ALUOut) Store: Mem(ALUOut <= B			

Step Description	mtco	mfco
Get Instruction & increment PC	Instruction Register <= Memory[PC] PC<= PC+2	
	A<= Reg(IR[13:10])	C<= CoProReg(IR[9:6])
Additional step (simplifies control)	ALUout<=A	ALUout<=C
	CoProReg(IR[9:6]) <= ALUout	Reg(IR[13:10]) <= ALUout

*Concatenate 8-bits (00000000)

Block Design



Control Signals

This processor implementation has a multitude of one-bit control signals, along with several two-bit control signals, and one three-bit signal.

One-bit control signals

The *RegWrite* control signal is a single-bit control signal that, when asserted, writes the values of the Write Data input to the general-purpose register selected by the Write Register number. When this is deasserted, it has no effect.

The *C0RegWrite* control signal is a single-bit control signal that, when asserted, writes the values of the Write Data input to the co-processor register selected by the Write Register number. When this is deasserted, it has no effect.

The *InstOrData* control signal is a one-bit signal that, when asserted, uses the contents of ALUOut to specify the address of the memory unit to be accessed. When it is deasserted, the value of the PC is used.

When asserted, the *MemWrite* one-bit control signal permits the contents of the memory unit at the location given by the address input to be overwritten by the data at the Write Data input. When not asserted, this control signal has no effect.

IRWrite is another one-bit control signal. When asserted, the output from the memory unit is placed in the Instruction Register. When it is not asserted, there is no effect.

Similar to the *IRWrite* control signal, the *PCWrite* control signal permits the PC to be written when asserted. When deasserted, there is no effect.

When asserted, the *PCWriteCond* permits the writing of the PC if the Not Zero output from the ALU is also active.

HiOrLo determines whether a load upper immediate or load lower immediate instruction will take place. When it has a value of one, a concatenation of eight bits to the lower end has taken place, indicating a load upper immediate is taking place. Otherwise, a load lower is taking place.

The *ReadReg2Src* control signal selects which bits of the instruction will dictate the register chosen for reading. When zero, the instruction bits 9 thru 6 are used. Otherwise, the bits 5 thru 2 are used, which is typical of a load word or store word command.

The *C0RegDest* control signal selects either the EPC (0) or the Cause (1) as the register to be written to by the Control unit.

Status changes the status register to indicate whether or not the program can currently process an interrupt. When asserted, the processor can*not* process an interrupt due to either being in the middle of an instruction or due to the fact that it is already handling an interrupt. When not asserted, any interrupt-handling code will be executed if necessary.

Two-bit control signals

The *ALUSrcA* determines the source for the first operand to the ALU. When the control signal is 00, the contents of the PC are selected. When the value is 01, the data from the A register is selected. When the control signal is 10, the value selected as the cause of an exception is chosen as the first input for the ALU. Finally, when the control signal is 11, register C, the co-processor register output, is selected.

Like *ALUSrcA*, *ALUSrcB* determines the source for the second operand to the ALU. When the control signal is 00, the B register is selected. When the control signal is 01, then one (1) is selected. When the control signal is 10, then the second input to the ALU is the sign-extended word-aligned branch offset from the Instruction Register. When the control signal is 11, the value passed along is zero.

The *PCSource* determines the source for the next update of the PC. If the signal is set to 00, the ALUResult becomes the next PC value. When the control signal is 01, ALUOut (the branch target address) is the new value of the PC. A control signal of 10 sets the PC to the jump destination address. The value of the PC is set to the exception-handling address of *0x6666* when the *PCSource* control signal is set to 11.

The *RegWriteSrc* selects what data to write to the general-purpose registers or co-processor register, whether it be from memory (00), ALUOut (01), the PC (10), or a concatenated immediate (11).

Three-bit control signals

The *ALUOp* determines the operation to be performed by the ALU. When the control signal is 000, the ALU performs addition on its two operands. When the control signal is 001, a set-less-than operation is performed--subtraction is performed and the NotZero output is set appropriately. When the control signal is 010, the ALU performs a bitwise and on the operands. A control signal value of 011 performs a bitwise or. If the control signal is 100, the first operand is passed through without modification. A control signal of 101 indicates a shift left logical will be performed.

The *RegDest* control signal determines the correct number of the register to be written to by the instruction, depending on the type and nature of the instruction. When the control signal is set to 000, an R1 or R2 type instruction is being executed, and the destination register is set to *$rd*. When *RegDest* is set to 001, a move-to-co-processor instruction is being executed, and the write destination is *$rs in the co-processor register*. When the control signal is set to 010, a C type instruction is being executed, and the destination register is set to *$rs*. A control signal value of 011 indicates a jump instruction is being executed, and the destination register is correspondingly set to *$ra*. A control signal of 100 allows the co-processor register selected by the *C0RegDest* control signal to be used.

The *Cause* control signal selects one of up to five exceptions. There are external exceptions from up to five devices (numbered 000 thru 100).

Control Units

The processor's Control is implemented using a finite state machine and a portion of combinational logic.

The combinational logic unit, called the ALUOp Calculator here, uses the Op code and function code, both of which are inputs to the Control, to determine the appropriate ALUOp for the instruction.

Below is a truth table that completes this mandate. It should be noted that when the Op code is 11 and the function code is either 10 or 11, a load word or store word instruction is being processed and the state of the ALU does not matter for this branch of the state sequence.

Opcode[0]	Opcode[1]	Funct[0]	Funct[1]	ALUOp[0]	ALUOp[1]	ALUOp[2]
1	0	0	0	0	0	1
1	0	0	1	1	0	1
1	0	1	0	0	1	1
1	0	1	1	0	1	0
1	1	0	0	0	0	0
1	1	0	1	1	1	0
1	1	1	0	x	x	x
1	1	1	1	x	x	x

This resolves into the following equations:

ALUOp[0] = (Funct[0])' * Funct[1]
ALUOp[1] = Opcode[1] * (Funct[0])' + Funct[0]
ALUOp[2] = (Opcode[1])' * (Funct[0])' + Funct[0] * (Funct[1])'

Another block of combinational logic handles the interrupts. There are five devices that give interrupts in our design, although only two are implemented in software. The five different button presses are registered with individual J/K flip-flops that have enables and reset options. They are wired such that once one button press has been acknowledged, all the flip-flops are disabled until they are cleared by the Control. This ensures that only one exception occurs at a time, although it does mean that only the first button pressed during the execution of an instruction is noted.

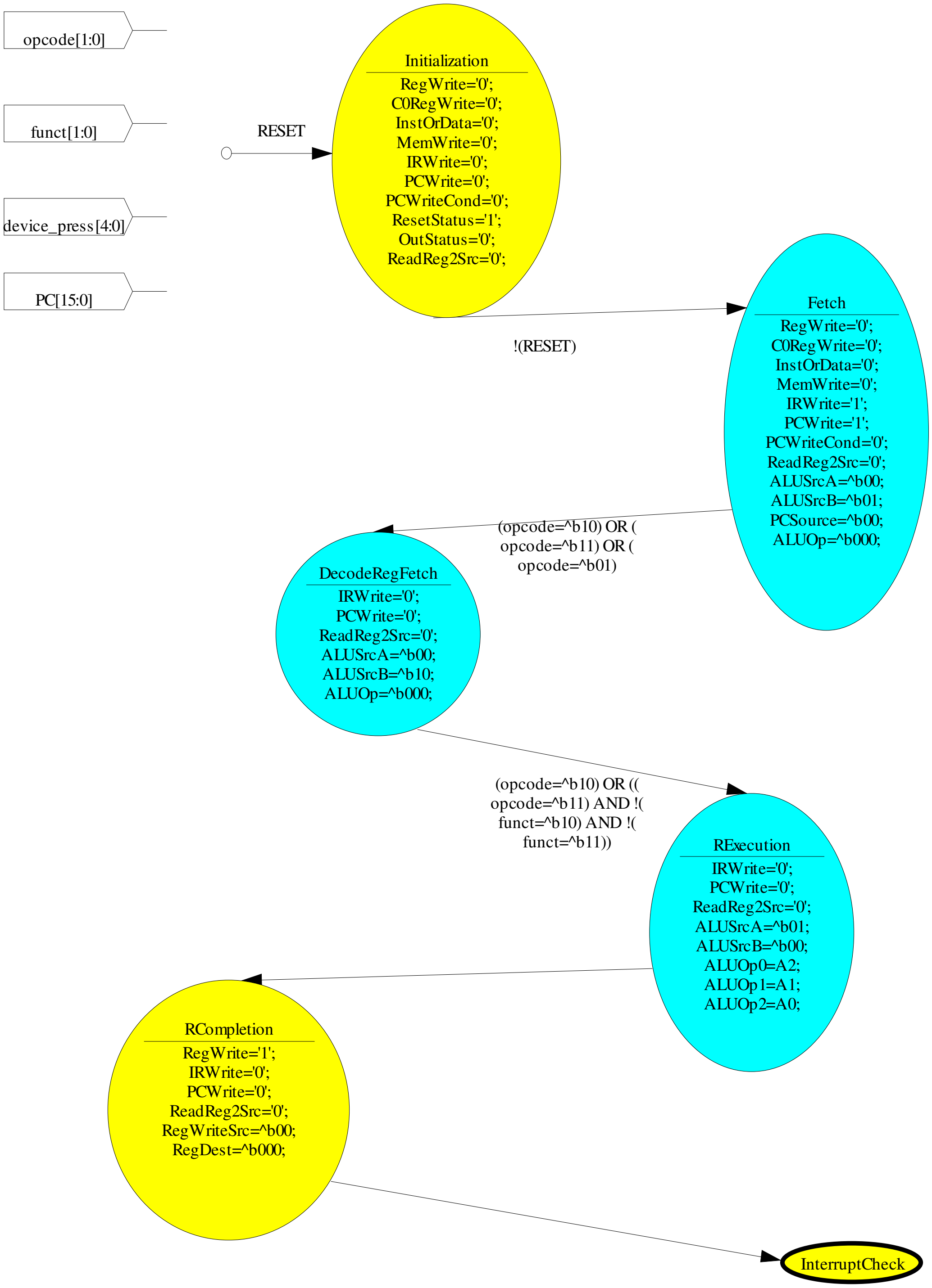
The five devices, however, must translate into a three-bit Cause that is passed along the datapath for exception-handling. Below is the truth table for this conversion.

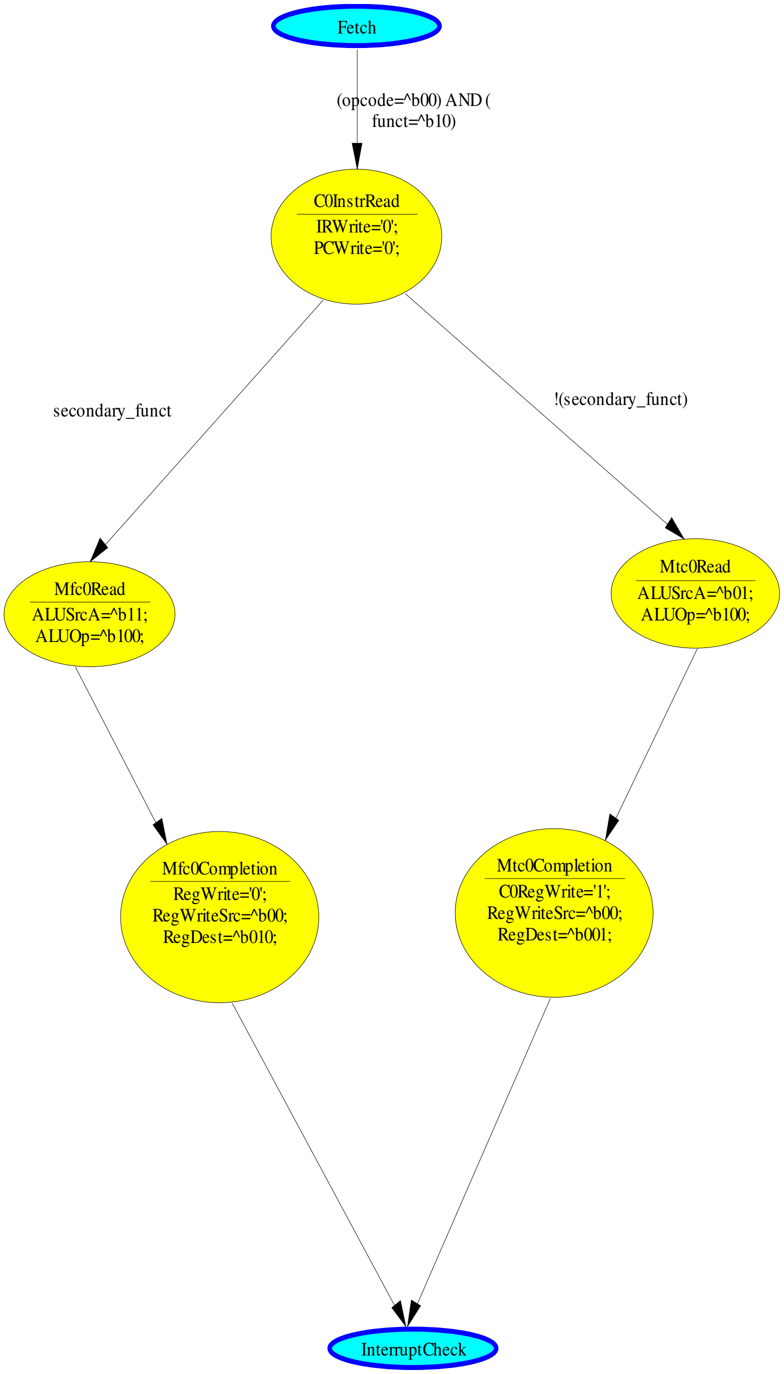
Device0	Device1	Device2	Device3	Device4	Cause[0]	Cause[1]	Cause[2]
1	0	0	0	0	0	0	1
0	1	0	0	0	0	1	0
0	0	1	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	0	0	1	1	0	1

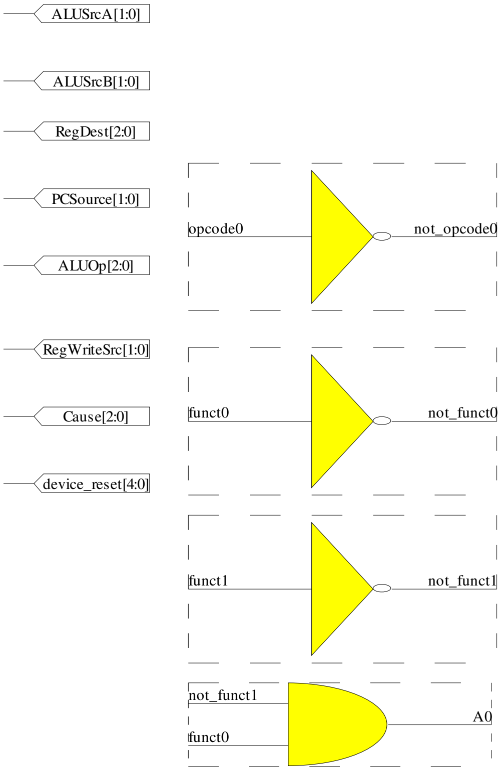
The primary control unit is the finite state machine described in the state transition diagram in **Appendix D**.

The system should initialize (or reset) into the state given by the control signals in the diagram below. There are several signals whose initialization values are not set, as there is nothing appropriate to set them to. For example, there is no immediately correct setting for the HiOrLo control signal, which is set to zero when the upper eight bits of an input are set to zero, indicating a load lower immediate is being performed. These "don't care" control signals are not included in this initialization state or any subsequent states. The diagram will show only those control signals that change from the previous state.

In addition, it should be noted that "assert" indicates that the control signal is active low.

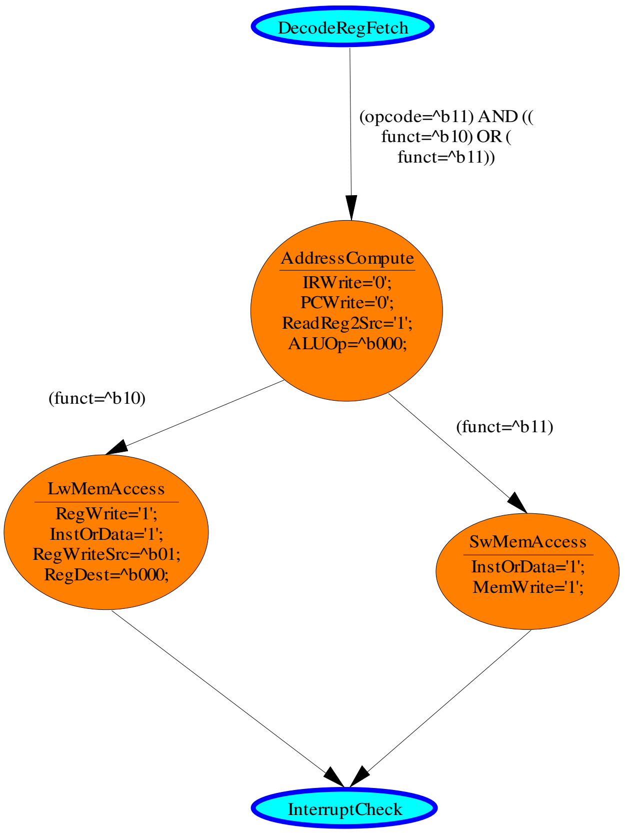

opcode[1:0]
funct[1:0]
device_press[4:0]
PC[15:0]
RESET
Initialization
RegWrite='0';
C0RegWrite='0';
InstOrData='0';
MemWrite='0';
IRWrite='0';
PCWrite='0';
PCWriteCond='0';
ResetStatus='1';
OutStatus='0';
ReadReg2Src='0';
!(RESET)
Fetch
RegWrite='0';
C0RegWrite='0';
InstOrData='0';
MemWrite='0';
IRWrite='1';
PCWrite='1';
PCWriteCond='0';
ReadReg2Src='0';
ALUSrcA=^b00;
ALUSrcB=^b01;
PCSource=^b00;
ALUOp=^b000;
(opcode=^b10) OR (
opcode=^b11) OR (
opcode=^b01)
DecodeRegFetch
IRWrite='0';
PCWrite='0';
ReadReg2Src='0';
ALUSrcA=^b00;
ALUSrcB=^b10;
ALUOp=^b000;
(opcode=^b10) OR ((
opcode=^b11) AND !(
funct=^b10) AND !(
funct=^b11))
RExecution
IRWrite='0';
PCWrite='0';
ReadReg2Src='0';
ALUSrcA=^b01;
ALUSrcB=^b00;
ALUOp0=A2;
ALUOp1=A1;
ALUOp2=A0;
RCompletion
RegWrite='1';
IRWrite='0';
PCWrite='0';
ReadReg2Src='0';
RegWriteSrc=^b00;
RegDest=^b000;
InterruptCheck

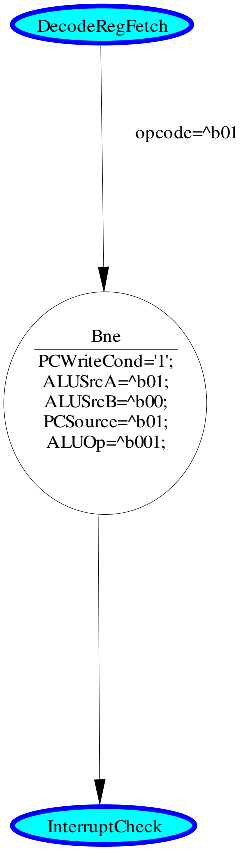

Fetch
(opcode=^b00) AND (
funct=^b10)
C0InstrRead
IRWrite='0';
PCWrite='0';
secondary_funct
!(secondary_funct)
Mfc0Read
ALUSrcA=^b11;
ALUOp=^b100;
Mtc0Read
ALUSrcA=^b01;
ALUOp=^b100;
Mfc0Completion
RegWrite='0';
RegWriteSrc=^b00;
RegDest=^b010;
Mtc0Completion
C0RegWrite='1';
RegWriteSrc=^b00;
RegDest=^b001;
InterruptCheck

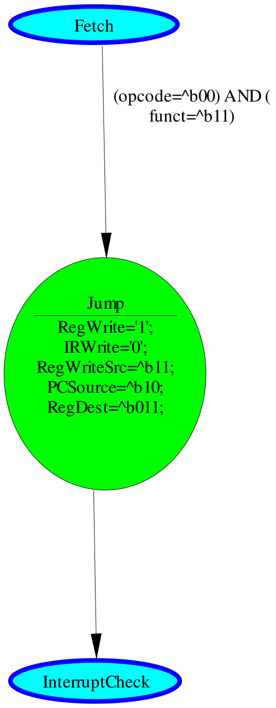


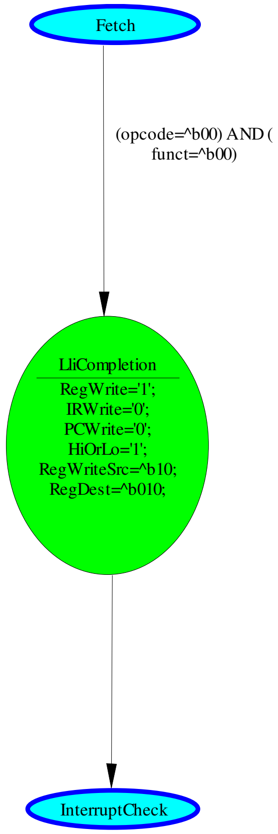
A1 = funct1 OR (opcode0 AND not_funct1 AND funct0)

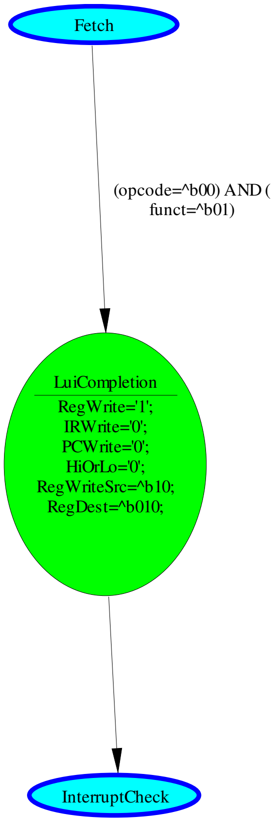
A2 = (not_opcode0 AND not_funct1) OR (funct1 AND not_funct0)

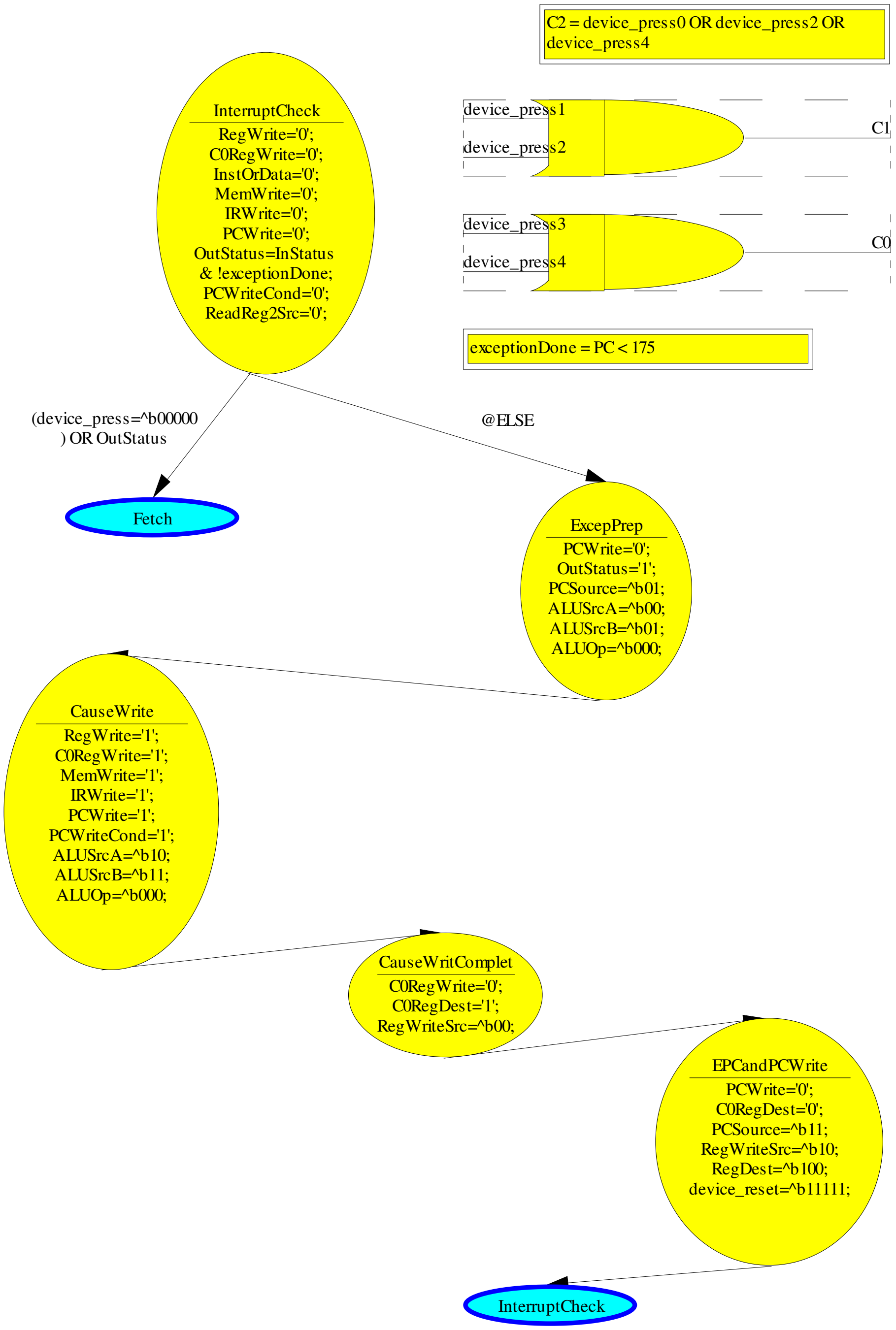

DecodeRegFetch
(opcode=^b11) AND ((
funct=^b10) OR (
funct=^b11))
AddressCompute
IRWrite='0';
PCWrite='0';
ReadReg2Src='1';
ALUOp=^b000;
(funct=^b10)
(funct=^b11)
LwMemAccess
RegWrite='1';
InstOrData='1';
RegWriteSrc=^b01;
RegDest=^b000;
SwMemAccess
InstOrData='1';
MemWrite='1';
InterruptCheck


DecodeRegFetch
opcode=^b01
Bne
PCWriteCond='1';
ALUSrcA=^b01;
ALUSrcB=^b00;
PCSource=^b01;
ALUOp=^b001;
InterruptCheck


Fetch
(opcode=^b00) AND (
funct=^b11)
Jump
RegWrite='1';
IRWrite='0';
RegWriteSrc=^b11;
PCSource=^b10;
RegDest=^b011;
InterruptCheck


Fetch
(opcode=^b00) AND (
funct=^b00)
LliCompletion
RegWrite='1';
IRWrite='0';
PCWrite='0';
HiOrLo='1';
RegWriteSrc=^b10;
RegDest=^b010;
InterruptCheck


Fetch
(opcode=^b00) AND (
funct=^b01)
LuiCompletion
RegWrite='1';
IRWrite='0';
PCWrite='0';
HiOrLo='0';
RegWriteSrc=^b10;
RegDest=^b010;
InterruptCheck


C2 = device_press0 OR device_press2 OR device_press4
device_press1
device_press2
C1
device_press3
device_press4
C0
exceptionDone = PC < 175
InterruptCheck
RegWrite='0';
C0RegWrite='0';
InstOrData='0';
MemWrite='0';
IRWrite='0';
PCWrite='0';
OutStatus=InStatus & !exceptionDone;
PCWriteCond='0';
ReadReg2Src='0';
(device_press=^b00000) OR OutStatus
@ELSE
Fetch
ExcepPrep
PCWrite='0';
OutStatus='1';
PCSource=^b01;
ALUSrcA=^b00;
ALUSrcB=^b01;
ALUOp=^b000;
CauseWrite
RegWrite='1';
C0RegWrite='1';
MemWrite='1';
IRWrite='1';
PCWrite='1';
PCWriteCond='1';
ALUSrcA=^b10;
ALUSrcB=^b11;
ALUOp=^b000;
CauseWritComplet
C0RegWrite='0';
C0RegDest='1';
RegWriteSrc=^b00;
EPCandPCWrite
PCWrite='0';
C0RegDest='0';
PCSource=^b11;
RegWriteSrc=^b10;
RegDest=^b100;
device_reset=^b11111;
InterruptCheck

Control Tests

In order to ensure correct functionality of the control, several simulations need to be run to ensure correct transitions. For example, the Op code and function code for an addition statement can be input using Xilinx's State Bench application; the transitions of the state should follow the diagram above.

Omitted is the initialization state, which the processor enters upon loading.

The progression of states for a particular Op code/function code combination can be seen in the table below.

Instruction	Op code	Funct code (secondary)	State progression				
lli	00	00	Fetch	LliCompletion	InterruptCheck	--	--
lui	00	01	Fetch	LuiCompletion	InterruptCheck	--	--
jalr	00	11	Fetch	Jump	InterruptCheck	--	--
mtc0	00	10 (0)	Fetch	C0InstRead	Mtc0Read	Mtc0Completion	InterruptCheck
mfc0	00	10 (1)	Fetch	C0InstRead	Mfc0Read	Mfc0Completion	InterruptCheck
bne	01	--	Fetch	DecodeRegFetch	Bne	InterruptCheck	--
slt	10	00	Fetch	DecodeRegFetch	RExecution	RCompletion	InterruptCheck
sll	10	01	Fetch	DecodeRegFetch	RExecution	RCompletion	InterruptCheck
or	10	10	Fetch	DecodeRegFetch	RExecution	RCompletion	InterruptCheck
and	10	11	Fetch	DecodeRegFetch	RExecution	RCompletion	InterruptCheck
add	11	00	Fetch	DecodeRegFetch	RExecution	RCompletion	InterruptCheck
sub	11	01	Fetch	DecodeRegFetch	RExecution	RCompletion	InterruptCheck
lw	11	10	Fetch	DecodeRegFetch	Address Compute	LwMemAccess	InterruptCheck
sw	11	11	Fetch	DecodeRegFetch	Address Compute	SwMemAccess	InterruptCheck

Following the test of the state transitions, it is necessary to ensure that all outputs are valid. For example, using the truth table in the previous section for the combinational logic, it should be possible to execute each R-type instruction and verify that the correct ALUOp code is output.

For each state, the below tables can be used to verify the output of the Control. The initialization step includes the "don't care" signals for the sake of creating a single comprehensive list of control signals. Subsequent states only show those signals that change value. Because the processor's components are active low, not asserted means a value of one. Selectors for multiplexers have actual values.

Init

Control Signal	Value
RegWrite	Not asserted
C0RegWrite	Not asserted
InstOrData	0
MemWrite	Not asserted
IRWrite	Not asserted
PCWrite	Not asserted
PCWriteCond	Not asserted
RegWriteSrc (2-bit)	Don't care
Status	Not asserted
HiOrLo	Don't care
ReadReg2Src	0
ALUOp (3-bit)	Don't care
ALUSrcA (2-bit)	Don't care
ALUSrcB (2-bit)	Don't care
PCSource (2-bit)	Don't care

RegDest (3-bit)	Don't care
Cause (3-bit)	Don't care

Fetch Instruction

Control Signal	Value
IRWrite	Asserted
PCWrite	Asserted
Status	Asserted
ReadReg2Src	0
ALUOp (3-bit)	000 (Add)
ALUSrcA (2-bit)	00 (PC)
ALUSrcB (2-bit)	01 ("2")
PCSource (2-bit)	00 (ALUResult)

DecodeRegFetch

Control Signal	Value
IRWrite	Not asserted
PCWrite	Not asserted
ReadReg2Src	0
ALUOp (3-bit)	000 (Add)
ALUSrcA (2-bit)	00 (PC)
ALUSrcB (2-bit)	10 (SE'd and shifted)

RExecution

Control Signal	Value
IRWrite	Asserted
PCWrite	Asserted
ReadReg2Src	0
ALUOp (3-bit)	CalculatedALUOp
ALUSrcA (2-bit)	10 (A)
ALUSrcB (2-bit)	00 (B)

RCompletion

Control Signal	Value
RegWrite	Asserted
IRWrite	Not asserted
PCWrite	Not asserted
RegWriteSrc (2-bit)	01
Status	Not asserted
ReadReg2Src	0
RegDest (3-bit)	010

AddressCompute

Control Signal	Value
IRWrite	Not asserted
PCWrite	Not asserted
ReadReg2Src	1
ALUOp (3-bit)	000 (Add)
ALUSrcA (2-bit)	10 (A)
ALUSrcB (2-bit)	00 (B)

LwMemAccess

Control Signal	Value
RegWrite	Assert
InstOrData	1
RegWriteSrc (2-bit)	01
Status	Not asserted
RegDest (3-bit)	000 (r-type)

SwMemAccess

Control Signal	Value
InstOrData	1

MemWrite	Assert

Bne

Control Signal	Value
PCWriteCond	Assert
Status	Not asserted
ALUOp (3-bit)	110 (subtr)
ALUSrcA (2-bit)	10 (A)
ALUSrcB (2-bit)	00 (B)
PCSource (2-bit)	01

LliCompletion

Control Signal	Value
RegWrite	Assert
IRWrite	Not asserted
PCWrite	Not asserted
RegWriteSrc (2-bit)	11
Status	Not asserted
HiOrLo	1
RegDest (3-bit)	000 (C-type)

LuiCompletion

Control Signal	Value
RegWrite	Assert
IRWrite	Not asserted
PCWrite	Not asserted
RegWriteSrc (2-bit)	11
Status	Not asserted
HiOrLo	0
RegDest (3-bit)	000 (C-type)

Jump

Control Signal	Value
RegWrite	Assert
IRWrite	Not asserted
RegWriteSrc (2-bit)	10 (PC)
Status	Not asserted
PCSource (2-bit)	10 (jump)
RegDest (2-bit)	011 ($ra)
ReadReg2Src	0 (bits[9:6])

C0InstrRead

Control Signal	Value
IRWrite	Not asserted
PCWrite	Not asserted

Mtc0Read

Control Signal	Value
ALUOp (3-bit)	100 (pass-through first)
ALUSrcA (2-bit)	01 (A)

Mtc0Completion

Control Signal	Value
C0RegWrite	Assert
RegWriteSrc (2-bit)	00 (ALUout)
Status	Not asserted
RegDest (2-bit)	001 (mtc0)

Mfc0Read

Control Signal	Value
ALUOp (3-bit)	100 (pass-through first)
ALUSrcA (2-bit)	11 (C)

Mfc0Completion

Control Signal	**Value**
RegWrite	Assert
RegWriteSrc (2-bit)	00 (ALUout)
Status	Not
RegDest (3-bit)	010 (c-type)

Component Specification

Control unit creation

The primary control units were crafted within Xilinx' StateCAD application. Using the state transitions described in tabular format in the Control Tests section, StateCAD was used to construct the finite state machine (FSM), taking care to ensure readability by heavy use of aliases for states. These act as pointers or placeholders for states that have been defined elsewhere, so that not all fifteen options for transitioning from a state must be placed in a single location. This enables each path of the FSM to be placed on a separate page, reducing crowding significantly.

The combinatorial logic in the control unit, described above in the form of the ALUOp and Cause computations, was also implemented in StateCAD using the Logic function to input the above transition formulas. Care had to be taken to match the correct lest-significant bits so that the outputs were not inverted.

ALU design and creation

At the heart of any good processor is a good ALU. We decided to approach the ALU construction by starting with a very small module, a one bit adder, and building up from there. The adder contains nand gates for calculating a carry out and an exclusive or gate for calculating the sum bit. With only three input bits (*input 1*, *input 2*, *cin*) we tested every possible input and checked against desired results.

Now that we had a reliable one bit adder the next move was to make a one bit ALU. The functionality we wanted our 1 bit ALU to perform is exhibited in the following table:

Function	Signal Bits
A plus B	000
Set If A is less than B (SLT)	001
A and B	010
A or B	011
Output the input A	100
Shift left	101
Subtract B from A	110
Output 0	111

An eight bit multiplexer is utilized to provide for the different outputs. Some of these functions are not utilized in our current processor design, but we tried to choose properties which would allow us some flexibility in our design and might be useful for testing the ALU in conjunction with other components. Trial cases consisted of an input combination for each function of the ALU. If the ALU passed all the trial cases, we felt confident that it would pass the other cases as well. Note that the most significant bit (MSB) of the ALU had to be designed differently than the other fifteen. Not only does it have a special output pin for set less than (SLT), but it also contains the logic necessary for calculating overflow. Also, in all cases except the least significant bit (LSB), the input 001 (SLT) into the multiplexer is wired to ground.

In class, we discussed inverting the numbers on the single bit ALU level. However, for our ALU we decided to institute four bit carry look ahead. This broke the sixteen bit ALU down into four chunks. Each chunk contained the carry look ahead logic, four one bit ALU's, and a four bit inverter (for subtraction). Once again, the top chunk of ALU's was slightly different because of overflow calculations and the SLT command. The bottom most chunk was also modified because it is necessary to have a carry in when subtracting using two's compliment. Testing intensified at this level; it consisted of using three or four sets of numbers for each function the ALU is intended to perform. Initially, the subtraction gave erroneous results. After a little examining, we discovered that our fast carry logic was receiving inputs prior to their inversion. This threw the whole subtraction off-kilter, but it was an easy mistake to fix.

The final step in the ALU construction consisted of compiling the four chunks into one sixteen bit monster. We also needed to design and fabricate a ‘shift left box’ that sat beside our ALU and a sixteen bit multiplexer to combine the two. The schematic was fairly easy to assemble, and primarily consisted of connecting parts we’d already designed. Testing, however, was exhaustive. This is the final stage of the ALU, but obviously the test cases are far too large to test every combination. Instead, choose to test do five or six sets of numbers for each function the ALU is supposed to perform. We also tested as many ‘sticky’ situations as we could: adding negative one and one requires numerous carries and a carry out but no overflow, set less than on two numbers of equal value should be false, performing an ‘or’ operation between a binary number containing all ones and another containing all zeros, etc. Once our design passed this rigorous testing, we felt confident that we have a sturdy, consistent ALU.

Memory creation

The Memory piece of our group’s processor was created using the CoreGen feature of the Xilinx computer program. A Distributed Memory block design was used, with the following specifications. The component was named memory, and it was set to a depth of 2^12, or 4096 lines. The width of the data was set to 16, corresponding to the number of bits in our instruction set. The memory type was selected as a Single Port RAM, and the Multiplexer Construction was left to the default LUT based. All other settings were left to their default values, with the exception of the initial contents. The memory was preloaded with the file *testmemory.coe*, which is shown below:

```
memory_initialization_radix=2;
memory_initialization_vector=

0000000000000000,
0000000000000001,
0000000000000010,
0000000000000011,
0000000000000100,
0000000000000101,
0000000000000110,
0000000000000111,
0000000000001000,
0000000000001001,
0000000000001010,
0000000000001011,
0000000000001100,
0000000000001101,
0000000000001110,
0000000000001111;
```

Once CoreGen had created the main piece of memory, we then connected and named the following inputs. The D(15:0) pin was connected to the WriteData(15:0) input bus. This pin and input correspond to the 16-bit number that will be stored into memory. The A(11:0) pin was connected to the Address(11:0) input bus. This bus and input correspond to the 12-bit number that is the address in memory where the data will either be written or read from. The WE pin is the Write Enable pin, and this is connected to the MemWrite control bit. The final input is the CLK pin, and this pin is connected to the CLK clock input of the whole processor. The only output, SPO(15:0), is connected to the output bus MemData(15:0). This bus then takes the information from the memory and carries it to the Instruction Register, and ultimately to the rest of the processor.

Instruction Register

The Instruction Register (IR) has a 16-bit input and four outputs. The instruction register's input is wired out with the memory output. The first output is two bits and will contain bits [15:14] of the instruction. These bits are the Op code of the actual instruction being processed. The second is four bits and will contain bits [13:10] of the instruction.

The third is eight bits and will contain bits [9:2] of the instruction. The last output is 2 bits and will contain bits [1:0] of the instruction. The IR also has the Control signal IRWrite as an input, which determines if the register will write new contents on the next rising clock edge. The IR was made by using 16 flip flops, and wiring them up according to the specifications above.

Register File

We created this part by using CoreGen. We used the dual-port memory as a basis for the register file. Then we had to add a multiplexer to determine when to write and when to read. The register file has four inputs, all of which are four bits. It has two outputs which are also four bits. The register file can be read by the outputs from the instruction register. The register file can be written by the instruction register and/or the Co-Processor register. The outputs always go to either the "A" temporary register or the "B" temporary register.

Co-Register File

The Co-Register piece of our group's processor was created using the CoreGen feature of the Xilinx computer program. A Distributed Memory block design was used, with the following specifications. The component was named memory, and it was set to a depth of 2^4, or 16 lines. The width of the data was set to 16, corresponding to the number of bits in our instruction set. The memory type was selected as a Single Port RAM, and the Multiplexer Construction was left to the default LUT based. All other settings were left to their default values, with the exception of the initial contents. The memory was preloaded with the file *testmemandreg.coe*, which is shown below:

```
memory_initialization_radix=2;
memory_initialization_vector=

0000000000000000,
0000000000000001,
0000000000000010,
0000000000000011,
0000000000000100,
0000000000000101,
0000000000000110,
0000000000000111,
0000000000001000,
0000000000001001,
0000000000001010,
0000000000001011,
0000000000001100,
0000000000001101,
0000000000001110,
0000000000001111;
```

Once CoreGen had created the main piece of the co-register, we then connected and named the following inputs. The D(15:0) pin was connected to the WriteData(15:0) input bus. This pin and input correspond to the 16-bit number that will be stored into memory. The A(3:0) pin was connected to a bus called the

RegChoice(3:0) input bus. This bus and input correspond to the 4-bit number that is the register in memory where the data will either be written or read from.

The RegChoice(3:0) bus is chosen using four two-input MUX devices to decide whether the WriteRegister(3:0) input bus or the ReadRegister(3:0) input bus will be used. The control bit for the Mux devices is the C0RegWrite input bit. This bit is also the input for the WE (Write Enable) pin. The final input is the CLK pin, and this pin is connected to the CLK clock input of the whole processor. The only output, SPO(15:0), is connected to the output bus MemData(15:0). This bus then takes the information from the memory and carries it to the Instruction Register, and ultimately to the rest of the processor.

Integration Plan

Our first step in testing the integration of our processor was to test the ALU with the temp registers. The ALU and the temp registers were wired up using Xilinx. We wired one temp register to InputA(15:0), another to InputB(15:0) and finally wired another temp register up to the output labeled AluOut(15:0). We tested the schematic using ModelSim. We put values into the temp registers, gave the ALU a function, and made sure the proper value was outputted.

The next step was to include the register file with the ALU and the temp registers above. This time, we put values into the register file and made the proper values appeared in the ALUOut. We then created a part which include the ALU, the register file, and the temp registers A, B, and C.

After we had the parts above wired up, we decided to concentrate on the left side of our datapath. We wanted to deal with the memory block first. We made sure that the memory was "Distributed Memory," therefore the instruction is read asynchronously. We wired up the PC temp register with the memory block. Put in some values, and once again, made sure the correct values were outputted.

The next step was to connect the instruction register to the memory block and the PC. We wired the output of the memory block up with the input of the instruction register. The instruction register then breaks up the instruction into 4 different outputs. We made sure that it broke the instruction up correctly before we moved on to the next piece.
We then made a part called "tempregandpc" which encapsulated the memory block, the pc, and the instruction register.

Once we had established that these two chunks of the datapath were working correctly, we connected them together via our design. This also included the uses of the 'conc I hi' and 'conc 8 lo' components. This primarily consisted of generating a variety of different size 'bus muxes' using the Xilinx Coregen. A simple test bench was created to test all each different instruction type, and results were compared to the desired outputs.

Once each of the individual pieces had been tested separately, then in small groups, we decided it was time to give the full blown processor a whirl. The actual construction of the processor was a breeze. Everyone in the group had been very meticulous in construction of their pieces, so it was easy to wire them all together.

After the construction of the processor, the really intense debugging began. We decided it would be best to test one instruction at a time. The way our processor is designed, however, all the instructions are highly dependant on the loading of immediate values (load upper immediate and load lower immediate). Thusly, we had to first establish these two instructions before moving on to any others. Once these two were set, we proceeded to test the add, load word, store word, jump register and link, and the branch if not equal instructions. Of course there were a few minor nuances, as there is likely going to be in a project of this caliber. In general, we corrected these issues by making small modifications to the control module, since it was most convenient to alter and reload.

We did hit one major snag; the jump instruction would not work the way it was currently implemented. The memory modules in the Xilinx CoreGen were not exactly the same was what we were expecting to be using in the project. We had to make some modifications to CoreGen module, or redo some of our design. We opted to modify the module, but a side effect is that we cannot read from the first register address and write to an address at the same time (exactly what needed to happen in our jump instruction). Consequently we were faced with a dilemma; we could either make more modifications to the CoreGen memory module, or modify the jump instruction. Since we had already tested everything with the current design, we chose to make a modification in the jump instruction (simply moving the location of the register bits). With these alterations, the command worked chummily.

Appendix B – Design Journal

Tuesday, September 21 – Tonight was the first time we met as a group. We decided what time would be a good time to meet with everybody's schedules. We also assigned some various tasks. Matt was in charge of the Design Journal, and Caleb was in charge of changing the Euclid's algorithm to assembly language. Our next meeting with be Friday, September 24 at 7:45.

Friday, September 24 – We met for a little bit tonight. Caleb presented the assembly language representation of Euclid's algorithm. We also had some preliminary design ideas:

Things that our instructions need to accomplish

1. Reading and writing to the memory is a must
2. Adding two numbers together, or subtracting two numbers
3. moving information from one register to another (maybe by a combination of two or more instructions)
4. jump to a label
5. branch to a label
6. jump and link, a register where we can store the return address
7. jump to a register: necessary for returning from a function call

Registers I don't think we can live without

1. The stack pointer is a must have, it's the best way to get things to and from the memory
2. registers to pass things to memory (can use the stack for this however)
3. registers to save variables (can use the stack for this again)
4. register to store a return address (might be able to use the stack for this too, but I don't know

Monday, September 27 – Tonight we made a lot of progress towards completing the first milestone which is due on Wednesday. We decided on what instructions we will use, as well as what type of instruction we will use. We designated some assignments for each member of the group. We decided to have 4 instruction types. An R1 type, R2 type, B type, and a C type. We decided to have 16 instructions so that we could have a 2 bit opcode, and then a 2 bit function code. The reason we decided to separate the opcode and function code was so that in the B type we would only have the function code, and therefore our offset would be bigger, allowing us to branch further away. We also decided to have 16 registers, that way we can reference any register with 4 bits. A list of the instructions we are using, and well as the format for out instructions are listed in the design document.

Sunday, October 3 – We met today to clear up some deficiencies of our first milestone. We changed the name of the $at register to $jt register. Now, for the jal instruction, we want the programmers to store the labels in $jt register and then use that register for jumps. Another topic we discussed was exception handling. We decided to store the exception handling code into memory.

Sunday, October 10 – We met today to discuss how we will approach milestone 2. Our first step of the day was to break down the datapath into components. We came up with the components of an R-Type to be: the PC, memory, instruction register, register file, output A register, output B register, ALU, and the ALUOut register. There are two exceptions to the R-Type datapath (load word and store word). For the load word and store word, the datapath will be the same as the R-Type except it will also contain the memory data register. For the B-Type, the components are the same except on the shift left instruction, there is a 10-bit sign extension. The datapath is the same except for the ALU there will be a "not-zero" output.

For the load upper immediate, it will be shifted left at 8 bits, and then be put in the register file. For the load lower immediate, it will be shifted right 8-bits, then put in the register file also.

For the Jump Register and Link instruction, we will store the PC+2 in $at, then move $at into $ra after we have jumped, and then when we have to link back, it will jump to $ra.

We also went through the list of input signals, output signals, and control signals for each component, including the number of bits in each signal.

We also divided the milestone into assignments for each person. Here are the assignments:

-Matt and Caleb will do an RTL description of each instruction or set of related instructions.

-Michael will do the description of the tests necessary to verify the correct implementation of your RTL description.

-Caleb will be writing a memo indicating the current status of your design

-Matt will continue updating the webpage

-Everybody will make any changes to the Assembly language and machine language specifications

-Melissa will be putting together an unambiguous English description of each component in terms of its input, output, and control signals

Sunday, October 17 - Mike & Lissa @ 13:00: Decision to move function codes to the end of all instructions to avoid the exception of wiring the branch instruction.

Load address dilemma: if "la" is treated as a pseudoinstruction for a "lui, lli, or" sequence, this would free up an instruction for dealing with the Exception Dilemma.

Discussed exception dilemma: Four registers need to be handled: display (could free up another temporary register if this got moved), cause, status, EPC.

If load address instruction is removed as discussed above, then we have a free C-Type instruction with which to implement the move to/from "coprocessor 0" instructions. As J.P. suggested in our last meeting, we could easily as a second function bit to distinguish between a move to and move from instruction. For instance, op code 00, function code 10 is a Co-Processor move instruction. Secondary function code 0 is a move to instruction, while code 1 is a move from.

Now, to actually solve the Exception Dilemma, one solution would be to add four registers in current register file, then use a control bit to switch between addressing the general registers and the coprocessor's registers. This would save us a component, but would cost us at least two control bits and much added complexity in wiring inputs and outputs on the register file.

A second (and better?) solution would be to simply add a second, smaller register file to contain our four remaining registers. (It should be noted here that the Cause and Status registers can most likely be combined into a single register if the Status register is only to be a single bit--in exception or not. We should have less than 2^16 different types of exceptions, so we can apply a bit of formatting to our sixteen bits and save ourselves some execution time by allowing our processor to get both Status and Cause in one step.)

Keep the Status register as a separate, small register (not in coprocessor file) that is hardwired to the Control signal that flips it on/off. This speeds the setting of Status and simplifies the testing.

This second solution would cause us to add a single, temporary output register ("C"). It would also require the addition of a mux (and Control) to the "Read Register 1" to choose between the general and coprocessor's register files. The RegDst control signal would have to expand to two bits and the corresponding mux would expand to handle three inputs.

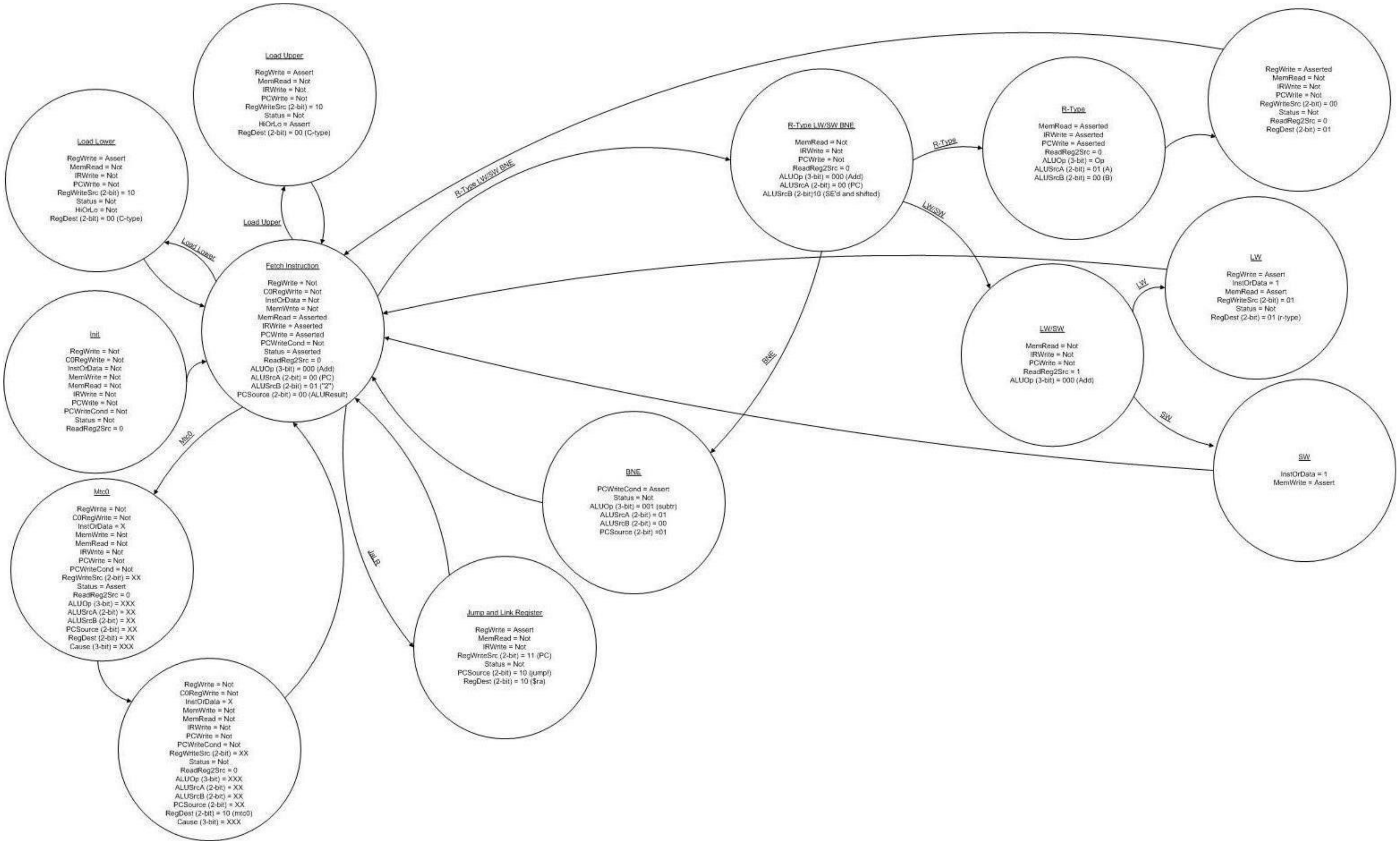
In addition, the pre-ALU muxes would have to expand to include accepting data from the coprocessor's registers. (Upgrading the already-full four-mux to handle eight inputs--better than using three muxes, or worse?) Corresponding control signals would need to expand/be modified.

We chose the mux following the A register to expand to handle four inputs. One of the new inputs will be the register data from "C". The second new input will be a number for the Cause (3-bit?). The Cause will be selected using a Control signal and a mux of options. (So we'll have two muxes, one providing a fourth option for the next.)

For best/simplest execution, let $rs always be the general register, and let $rt always be the coprocessor register, so that the format is consistent with the other C-Types, and to minimize the number of "selections" of where data can come from. This way, only one expanded mux needs to be used to pick the write register (Coproc = 10 or general = 00 or 01, depending on R1/2 type or C-Type), and nothing additional is needed to select the correct bits for a coprocessor write.

Saturday, October 23 – We met today to make sure that milestone 3 was set in stone, and to assign certain parts of milestone 4 to each other. We would like to have something running by next weekend.

Here is a state diagram of our control signals:


Load Upper
RegWrite = Assert
MemRead = Not
IRWrite = Not
PCWrite = Not
RegWriteSrc (2-bit) = 10
Status = Not
HiOrLo = Assert
RegDest (2-bit) = 00 (C-type)
Load Lower
RegWrite = Assert
MemRead = Not
IRWrite = Not
PCWrite = Not
RegWriteSrc (2-bit) = 10
Status = Not
HiOrLo = Not
RegDest (2-bit) = 00 (C-type)
Load Upper
Load Lower
Fetch Instruction
RegWrite = Not
C0RegWrite = Not
InstOrData = Not
MemWrite = Not
MemRead = Asserted
IRWrite = Asserted
PCWrite = Asserted
PCWriteCond = Not
Status = Asserted
ReadReg2Src = 0
ALUOp (3-bit) = 000 (Add)
ALUSrcA (2-bit) = 00 (PC)
ALUSrcB (2-bit) = 01 ("2")
PCSource (2-bit) = 00 (ALUResult)
Init
RegWrite = Not
C0RegWrite = Not
InstOrData = Not
MemWrite = Not
MemRead = Not
IRWrite = Not
PCWrite = Not
PCWriteCond = Not
Status = Not
ReadReg2Src = 0
Mtc0
Mtc0
RegWrite = Not
C0RegWrite = Not
InstOrData = X
MemWrite = Not
MemRead = Not
IRWrite = Not
PCWrite = Not
PCWriteCond = Not
RegWriteSrc (2-bit) = XX
Status = Assert
ReadReg2Src = 0
ALUOp (3-bit) = XXX
ALUSrcA (2-bit) = XX
ALUSrcB (2-bit) = XX
PCSource (2-bit) = XX
RegDest (2-bit) = XX
Cause (3-bit) = XXX
RegWrite = Not
C0RegWrite = Not
InstOrData = X
MemWrite = Not
MemRead = Not
IRWrite = Not
PCWrite = Not
PCWriteCond = Not
RegWriteSrc (2-bit) = XX
Status = Not
ReadReg2Src = 0
ALUOp (3-bit) = XXX
ALUSrcA (2-bit) = XX
ALUSrcB (2-bit) = XX
PCSource (2-bit) = XX
RegDest (2-bit) = 10 (mtc0)
Cause (3-bit) = XXX
JalR
Jump and Link Register
RegWrite = Assert
MemRead = Not
IRWrite = Not
RegWriteSrc (2-bit) = 11 (PC)
Status = Not
PCSource (2-bit) = 10 (jump!)
RegDest (2-bit) = 10 ($ra)
R-Type LW/SW BNE
R-Type LW/SW BNE
MemRead = Not
IRWrite = Not
PCWrite = Not
ReadReg2Src = 0
ALUOp (3-bit) = 000 (Add)
ALUSrcA (2-bit) = 00 (PC)
ALUSrcB (2-bit)10 (SE'd and shifted)
BNE
BNE
PCWriteCond = Assert
Status = Not
ALUOp (3-bit) = 001 (subtr)
ALUSrcA (2-bit) = 01
ALUSrcB (2-bit) = 00
PCSource (2-bit) =01
R-Type
R-Type
MemRead = Asserted
IRWrite = Asserted
PCWrite = Asserted
ReadReg2Src = 0
ALUOp (3-bit) = Op
ALUSrcA (2-bit) = 01 (A)
ALUSrcB (2-bit) = 00 (B)
RegWrite = Asserted
MemRead = Not
IRWrite = Not
PCWrite = Not
RegWriteSrc (2-bit) = 00
Status = Not
ReadReg2Src = 0
RegDest (2-bit) = 01
LW/SW
LW/SW
MemRead = Not
IRWrite = Not
PCWrite = Not
ReadReg2Src = 1
ALUOp (3-bit) = 000 (Add)
LW
LW
RegWrite = Assert
InstOrData = 1
MemRead = Assert
RegWriteSrc (2-bit) = 01
Status = Not
RegDest (2-bit) = 01 (r-type)
SW
SW
InstOrData = 1
MemWrite = Assert

Monday, October 25 – We met today to make sure all our documentation was correct. We met to make sure everybody knew what they were doing on their parts. Out next meeting will be Wednesday night.

Wednesday, October 27 – We met tonight to discuss the milestone 4. We made sure everything was correct in control. We made sure we had all of the components of milestone 4 completed. Lissa did almost all the work on the control. We divided the project in parts. Lissa worked on control. Caleb worked on the ALU. Mike will work on the Memory and the Co-Processor Register. Matt will work on the Instruction Register and the General Purpose Register. Our next meeting will be Saturday, at which point we hope to have all of our components and the documentation completed.

Saturday, October 30 – We met today to finish up our parts and Caleb demonstrated how to test our parts using ModelSim. Out next meeting with be Monday at 9:00.

Monday, November 1 – Tonight we met to see how our components were going. We also had our test procedures finished for all of our components. We talked about some of our flaws as well as some things we have completed and are comfortable with.

Sunday, November 7 – Today we gathered all of our parts together and began trying to integrate them. We worked on integrating for about 4 hours. The meeting was pretty successful, we got a lot of stuff done, and our datapath is looking promising.

Monday, November 8 – We met today for about 2 more hours to do some more integration. Things are still going well. Almost all of our parts are working as planned.

Friday, November 12 – We met for a little bit today and discussed what we needed to do for the final report. We assigned tasks to each person to have done the next day. Our next meeting is Saturday at 3:00.

Saturday, November 13 – We met once again. This time we went over what each one of us needs to do for the presentation on Monday. Caleb is planning on finishing the integration of all our Xilinx pieces tonight. Our next meeting is Sunday at 10:00.

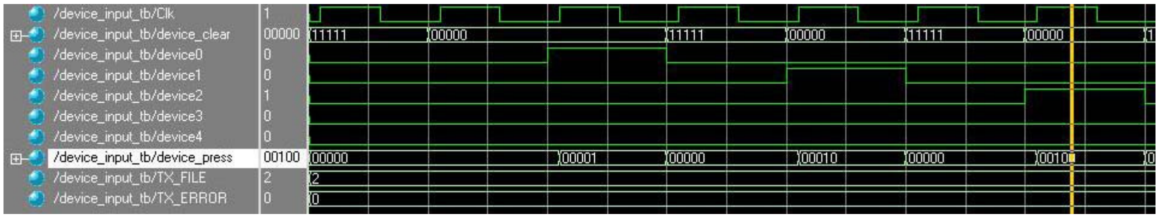
Appendix C – Test Results

ALU

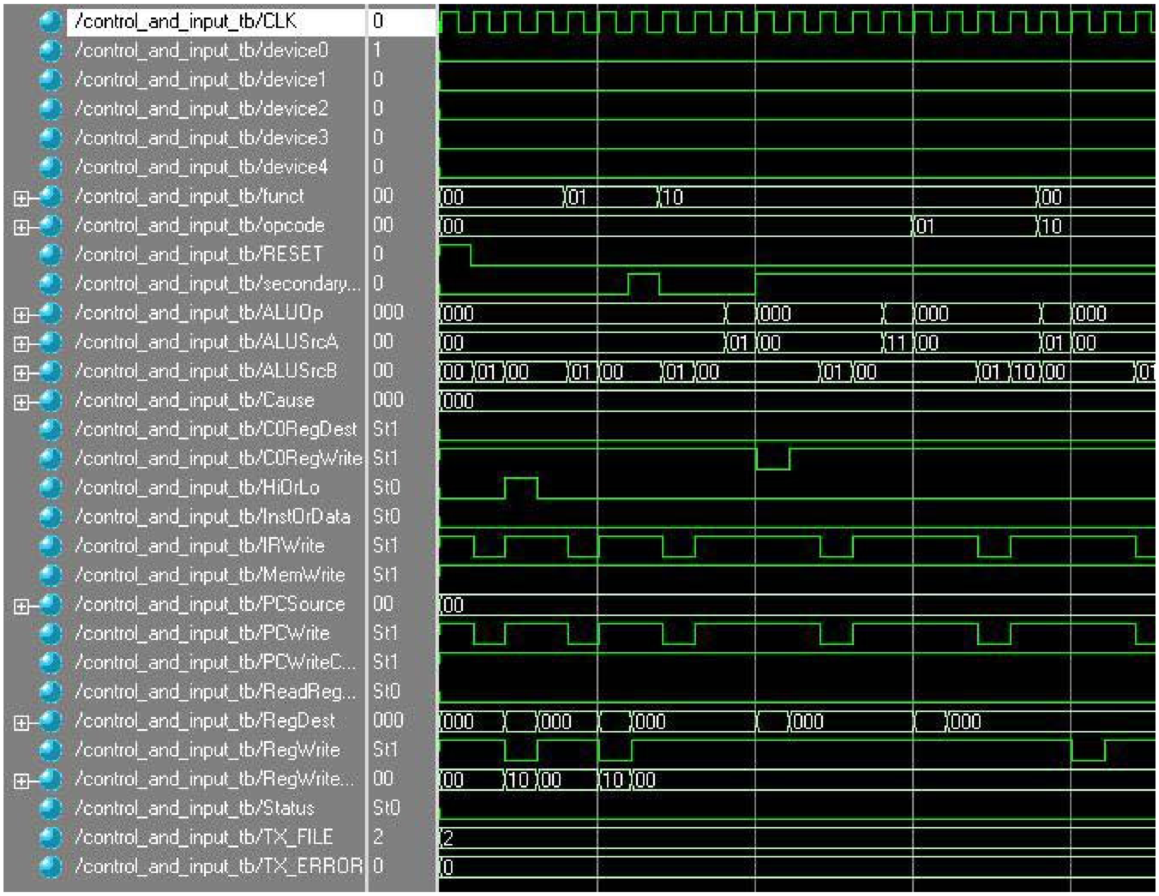
The test cases and testing of the ALU are described in the ALU design and creation section above.

Control and user inputs

The user input device was tested using a test bench constructed in Xilinx and executed in ModelSim. Essentially, the first device was activated, made to hold the value for more than a clock cycle—to ensure that it could—then reset. Then the second device was activated and the five-bit device output was noted. This was continued for the five inputs. A portion of the waveform can be seen directly below.


/device_input_tb/Clk 1
/device_input_tb/device_clear 00000
11111
00000
11111
00000
11111
00000
/device_input_tb/device0 0
/device_input_tb/device1 0
/device_input_tb/device2 1
/device_input_tb/device3 0
/device_input_tb/device4 0
/device_input_tb/device_press 00100
00000
00001
00000
00010
00000
0010
/device_input_tb/TX_FILE 2
2
/device_input_tb/TX_ERROR 0
0

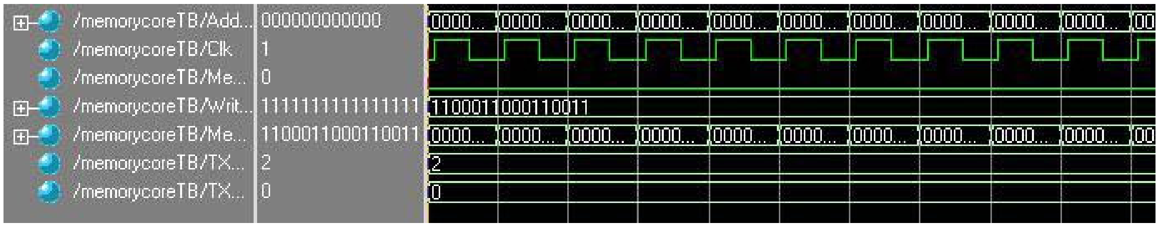
Following the testing of the input device alone, it was connected to the block device of the Control and tested systematically. The Control was transitioned through each state, then run through an interrupt sequence. The first waveform below shows the execution of a couple of instructions.



```
/control_and_input_tb/CLK          0
/control_and_input_tb/device0      1
/control_and_input_tb/device1      0
/control_and_input_tb/device2      0
/control_and_input_tb/device3      0
/control_and_input_tb/device4      0
/control_and_input_tb/funct        00
/control_and_input_tb/opcode       00
/control_and_input_tb/RESET        0
/control_and_input_tb/secondary... 0
/control_and_input_tb/ALUOp        000
/control_and_input_tb/ALUSrcA      00
/control_and_input_tb/ALUSrcB      00
/control_and_input_tb/Cause        000
/control_and_input_tb/C0RegDest    St1
/control_and_input_tb/C0RegWrite   St1
/control_and_input_tb/HiOrLo       St0
/control_and_input_tb/InstOrData   St0
/control_and_input_tb/IRWrite      St1
/control_and_input_tb/MemWrite     St1
/control_and_input_tb/PCSource     00
/control_and_input_tb/PCWrite      St1
/control_and_input_tb/PCWriteC...  St1
/control_and_input_tb/ReadReg...   St0
/control_and_input_tb/RegDest      000
/control_and_input_tb/RegWrite     St1
/control_and_input_tb/RegWrite...  00
/control_and_input_tb/Status       St0
/control_and_input_tb/TX_FILE      2
/control_and_input_tb/TX_ERROR     0
```

Memory

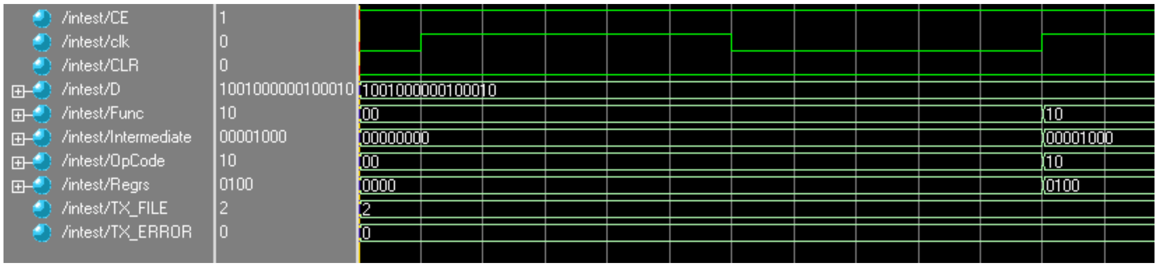
The testing of this piece will use Xilinx to create a test bench file, and then use the ModelSim program to test the file and create a waveform. The waveform will be given predetermined input values, with predetermined output values. These will then be compared to the results of the simulation, and checked for discrepancies. The test will first run through the first several memory addresses with the MemWrite unasserted, to ensure that the file loaded properly. Then, the MemWrite will be asserted and new values written into various locations in memory. Finally, the addresses in memory will be checked again, with MemWrite unasserted, to ensure that the changes to memory remain.



```
/memorycoreTB/Add...  000000000000
/memorycoreTB/Clk     1
/memorycoreTB/Me...   0
/memorycoreTB/Writ... 1111111111111111
/memorycoreTB/Me...   1100011000110011
/memorycoreTB/TX...   2
/memorycoreTB/TX...   0
```

Instruction Register

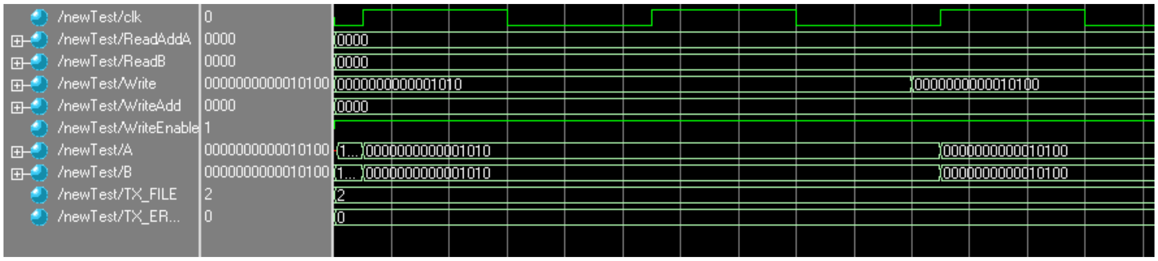
In order to test this register we simulated it in ModelSim. We input a number into the IR and made sure we received the correct number on the output. We constructed test benches to test several cases, and everything seemed to work as expected. Here is an example:



/intest/CE	1
/intest/clk	0
/intest/CLR	0
/intest/D	1001000000100010
/intest/Func	10
/intest/Intermediate	00001000
/intest/OpCode	10
/intest/Regrs	0100
/intest/TX_FILE	2
/intest/TX_ERROR	0

Register File

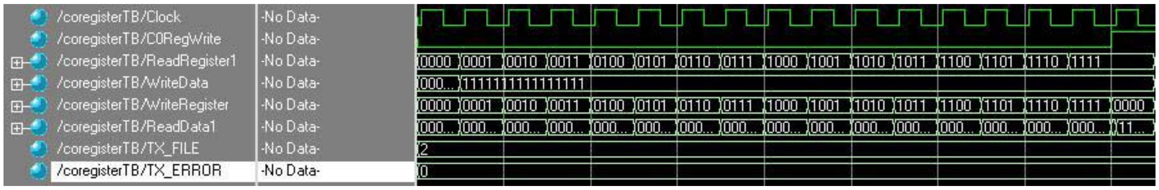
For this part, we used ModelSim to test it. Once again, we put in various numbers, and checked to see if we outputted what we wanted. Again in this case, we were successful. Here is the waveform for the register file:



/newTest/clk	0
/newTest/ReadAddA	0000
/newTest/ReadB	0000
/newTest/Write	0000000000010100
/newTest/WriteAdd	0000
/newTest/WriteEnable	1
/newTest/A	0000000000010100
/newTest/B	0000000000010100
/newTest/TX_FILE	2
/newTest/TX_ER...	0

Co-Register File

The testing of this piece will use Xilinx to create a test bench file, and then use the ModelSim program to test the file and create a waveform. The waveform will be given predetermined input values, with predetermined output values. These will then be compared to the results of the simulation, and checked for discrepancies. The test will first run through the first several memory addresses with the MemWrite unasserted, to ensure that the file loaded properly. Then, the MemWrite will be asserted and new values written into various locations in memory. Finally, the addresses in memory will be checked again, with MemWrite unasserted, to ensure that the changes to memory remain.



/coregisterTB/Clock	-No Data-
/coregisterTB/C0RegWrite	-No Data-
/coregisterTB/ReadRegister1	-No Data-
/coregisterTB/WriteData	-No Data-
/coregisterTB/WriteRegister	-No Data-
/coregisterTB/ReadData1	-No Data-
/coregisterTB/TX_FILE	-No Data-
/coregisterTB/TX_ERROR	-No Data-

Integrated Unit

The integrated unit test involved a simple collection of instructions, this would tell us if things were functioning correctly.