SAX(Students Against Xilinx) Accumulators and Registers 3.5E

Team 2D: Caleb Donoho, Brian Jennings, Anderson Yang, Kevin Wang, and Kaiyu Xie

Table of Contents

Executive Summary

In late September 2017, Our group began designing a 16-bit processor that could run basic functions. We created this by meeting seven milestones set out to make sure we were on the right path. We met several times throughout the project resulting in about seven hours per week of work per person. These milestones allowed use to meet with our professor each time to ask questions and see if we needed to take a different direction with some of our ideas. Throughout the process we kept up two documents, a design document and a journal. The design document shows an in depth explanation of our processor from simple plans to full data paths. The journal shows a record of where and when we worked on the project. It also shows what we planned on getting done each day and what we actually completed. Both the design document and journal are attached to this report.

Introduction

This report is designed to step through the process we went through to design our processor. Firstly, the initial process of designing our instructions will be discussed, including our design choices regarding registers. After that, we will show how we designed the RTL and datapath. Next, our part implementation process will be discussed followed by a description of our final Xilinx Model. Finally, we will talk about our testing practices and the final results of our processor.

Instruction Set Design

When beginning our processor, we didn't quite understand what an accumulator-based design entailed. Our first set of instructions were more similar to load-store and were too large. Once we learned of our errors in this first design, we redesigned to what is now our current instruction set and design.

We decided first to have four bit opcodes and four bit register codes. This limited us to only 16 registers. On top of that, three of those were reserved for a zero register and two assembler temporaries. This also limited us to 16 instructions. We chose what instructions we needed by looking through the MIPS green sheet and seeing if we needed an instruction similar. This got us to a list that was slightly too long, so we had to figure out what instructions could be performed using others in our set. For example, addi could be achieved by storing the accumulator in another register, loading an immediate into the accumulator using lui and ori, then adding the value from the register storing the original accumulator back into the accumulator. After weeding out unnecessary instructions and modifying instructions to be more useful, we got our list down to exactly 16 instructions.

Our instructions were structured in three ways, two of which were incredibly similar. The first type of instruction consisted of a four bit opcode followed by a four bit register code and ended with an eight bit immediate/unused. The second type of instruction was exactly the same, but it had a four bit immediate/unused instead of the four bit register code. The last type of the instruction had a four bit opcode, a four bit register code, another four bit register code, and then four unused bits.

One unique part of our instructions is the immediate/unused bits in the first two types of instructions. Because we were limited to 16 bit instructions, in order to do proper slt and sub without having two instructions for each, we used the unused bits to be a tag for what order to do ALU operations. Using this otherwise unused bits is what allowed us to have as few instructions as we did. For the instructions where order doesn't matter in the ALU, it can still recognize the order bits but as implied, flipping the inputs on the ALU won't change the outcome on them.

As far as our design choices for registers, we modeled our types of registers with our knowledge of MIPS. We have argument registers, return registers, temporary registers, and safe registers. One unique thing about our choice for registers is that we have two assembler temporary registers. This allowed us to write more complex pseudo-instructions where we needed to store multiple things. We knew we would need this because all the operations go through the accumulator, so we would need to store multiple things during pseudo-instructions.

RTL and Datapath Design

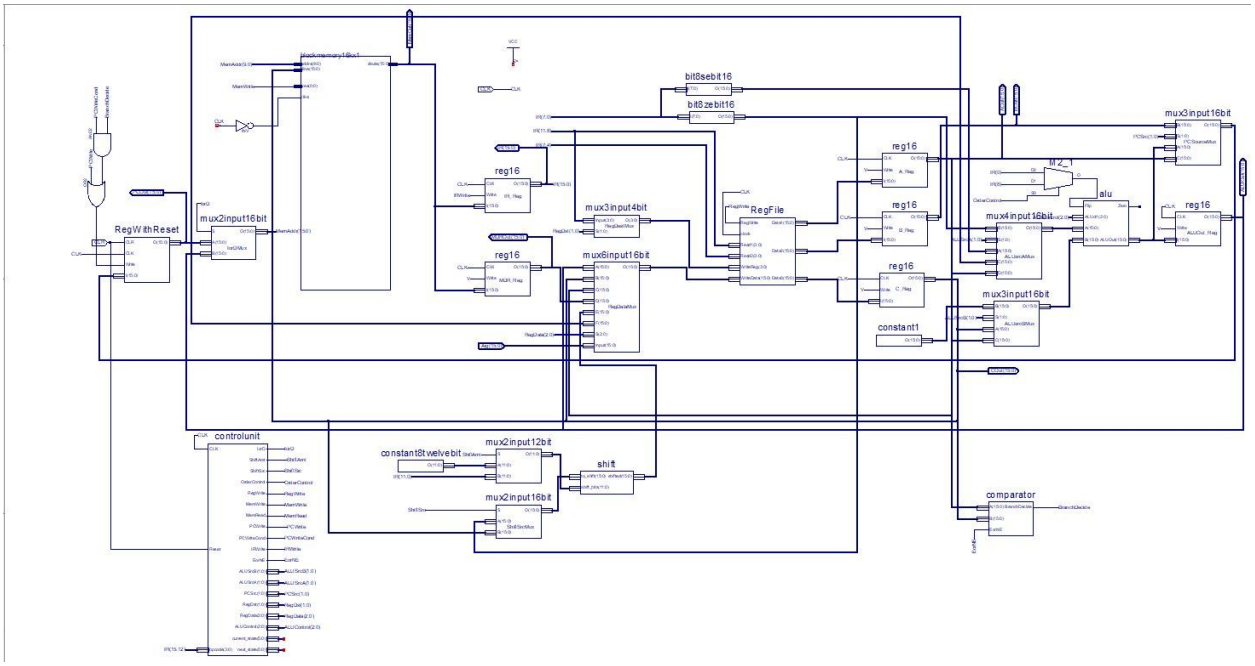
When designing the RTL, we first went through each of our 16 instructions and wrote out RTL for each. After we had completed the RTL for each instruction, we looked through all the instructions to see what was common amongst all of them. These pieces then became the first two stages of the RTL. After we found the similarities among all the instructions, we grouped instructions together by similarities which resulted in fewer categories after the first two cycles, resulting in a less complicated datapath design. There were even a few iterations of the RTL to simplify it further and debug some of the issues with how the cycles were set up. This resulted in a very refined set of RTL.

Once the RTL was designed, we went about designing the processor. To design the processor, we went cycle by cycle in the RTL and added onto the datapath to accomplish what was needed for that cycle. If parts were already in the datapath and were needed for another cycle, we wouldn't make an excess part and instead used the part already on the datapath. This resulted in a pretty confusing datapath still, so we then looked through and simplified it further. In our meetings with our professor, we received more ideas on how to simplify the datapath, primarily by reducing the number of muxes and thus reducing the number of control signals. By the end of the project, the datapath was very manageable.

Implementation

Our implementation of parts was done on an individual basis. We delegated what parts each person would do so that the workload would be split up. Most of the parts were built in Verilog, but some were built in schematic. Most of the muxes were actually built in schematic. Other parts, such as the ALU, Register File, and Control Unit, were easier to build using Verilog. The parts were designed based around the requirements of the design we made beforehand. After implementing the full datapath, some parts needed to be modified slightly as we debugged, but no major changes were made to parts once they were finished. One example is that we added mux inputs to introduce I/O. Because implementation of parts was on an individual basis, testing of parts was up to the individual designing the part. Most parts were pretty heavily tested, which will be expanded upon later in the report.

Xilinx Model



When building our Xilinx Model out of the parts we built, we followed our datapath design for a vast majority. We built the full datapath in schematic because we felt it would be easier to visualize and debug if we used the schematic tool. After we built the datapath we had designed on paper, we had to add I/O then debug.

To add an input, we added another option for the RegData mux and another option for the RegDst mux. This allowed us to put an input value into $a0. In order to do this though, we had to add an additional state to our control unit that was exactly like the Fetch state from before,

but it had RegWrite turned on and chose the input to go into $a0. None of the other states pointed to this new state, so it only ran at the start of a program when Reset was on. As far as outputs, we put outputs out of every register so that their values could be read.

As far as debugging, a lot of it had to do with the control unit because individual parts had been tested extensively as will be talked about in the next section. When we had first designed our control state diagram on paper, we had forgotten some key signals in certain states. For example, in the Fetch state, we had forgotten to include the ALU Operation code, so the PC never incremented by one because there was no control on the ALU. After fixing these bugs, it almost worked. The last change that needed to be made to the design was that the clock to the memory needed to be inverted. We found this through debugging and finding that all accesses to memory were loading a cycle late. Upon discussing this with our professor, he suggested that the memory needed to be on a different clock. Inverting the clock fixed the memory delay issues and resulted in a working processor.

Testing Method

Since we are using a top-down design, we test the part we wrote before we go to next step. For the assembly part, we wrote some short codes to achieve some commonly used in usual programs. This makes sure that we can actually write code to deal with these situations. Then we wrote the assembly code for relprime. We then tried to turn it into machine code to make sure we can turn it into codes that can actually use as input. We also found problems like do zero extend or sign extend and how to deal with the jumps to labels.

Then we wrote the RTL tests. This is also a manual test. We split into two groups: one wrote all the RTL and the other wrote the test with the instructions given by the other group. Therefore, we can write the test without any assumptions on it. Most of the rtl works well as we wrote it.

Then we began to build our components. This includes three steps, individual test, integration test and full datapath test. For the individual test, we did thorough tests for almost all of them. Since it is the most basic step, we have to make sure it works well. We wrote the tests in Verilog, so we can finally really run the test cases to see if it works. These tests are important to us, because we really found several parts that we need to fix during the test. Then we combine some of the components together to do the integration tests. We did many tests on this to make sure every instruction that we will need to work can work well. This gives us a solid foundation for the whole datapath. We found some problems during passing data through components, which is related to the clock. We also did some fixes to our components.

The final step is the whole datapath test. The test for control unit it did together with it, because we need the whole datapath to show if the control units. We first wrote a little chunk of code to see how it works as a whole datapath. We made a chart to write the things we expected and

compare them in the waveform. We found some problems in the control unit, assembler, assembly code and clock. We fix all these things and it works well on our final test which is the relprime.

Final Results

Our processor has a satisfying result. When we try to calculate the smallest relative prime for 5040, we used 274 byte of memory, and ran through 316509 instructions. Even though the number of instructions is large, it is acceptable since we need to store value into accumulator register before we perform any ALU operation on it. Therefore, our processor tend to have more instructions than a simple load-store processor.

It took us 949662 clock cycles to run all these instructions, which means 3.0004265 clock cycle for an instruction on average. This is consistent with our RTL since most of our instructions takes 3 cycles to complete. According to Xilinx Synthesis report, our cycle time is 14.794ns and it took 15.194744ms to execute the relative prime in total. We can improve this performance data by building our ALU and all other components written in Verilog from bottom up. Therefore, we can use less gates (currently we used around 1800 gates) and improve our overall performance.

Conclusion

From this final project, we proved the feasibility of accumulator based and load-and-store based processor. The accumulator based processor is a register based processor that has a general purpose accumulator. It has advantages of simple instructions, simple designs and simple operations. Because most instructions defaultly uses the accumulator, each instruction has particular uses for the accumulator. It makes the codes clear and easy to understand. The single accumulator decrease the difficulty of design, since we always do operations about the accumulator.

On the other hand, the simple operations also mean that we need more lines of code to do functions than other architecture. Because we have only a single accumulator that used for calculation, it is sometimes complex to do operations that requires parallel computing.

Load-and-store helps the problem that the accumulator cannot do operations simultaneously. It allows the processor to store the current value into an address, or load a value from an address. Although loading and storing will spend many cycles and increase the necessary time, load-and-store based architecture enables the design to run codes more flexibly.

Accumulator based architecture and load-and-store based architecture are both great architectures for a processor, but their shortcomings are obvious. The mixture of them will find a compromise of their shortcoming, and still keep its characteristics.

Design document

Group 2D

Registers

$0: A register that always contains 0. Reg #0

$acc: The accumulator register. Reg #1

$a0-3: argument temporary registers. Reg #2-5

$v0-1: return temporary registers. Reg #6-7

$t0-1: temporary registers. Reg #8-9

$s0-1: safe registers. Reg #10-11

$ra: A register which contains the return address. Reg #12

$sp: A register which contains the stack pointer. Reg #13

$at0-1: Assebler temporaries. Reg #14-15

Instructions

Types

Instruction Fields:

Type I:

op	reg	imm/unused
4 bits	4 bits	8 bits

op: Basic operation of the instruction.

des: The register/accumulator destinator operand

imm: immediate value that represents a constant value or an address in memory.

Type II:

op	imm/unused	imm
4 bits	4 bits	8 bits

op: Basic operation of the instruction

imm: immediate value.

Type III: For beq and bne

op	reg	des	unused
4 bits	4 bits	4 bits	4 bits

op: Basic operation of the instruction

List of instructions

add: Type I instruction with opcode 0 0

Adds value at register into accumulator value and stores at accumulator

sub: Type I instruction with opcode 1 1

Subtracts value at register with accumulator value and stores at accumulator(decide order using unused bits. If no imm in instruction, assembler assumes 0. 1 means acc = acc - A, 0 means acc = A - acc)

lmem: Type I instruction with opcode 10 2

Takes the value at a register, adds the immediate to create a memory address, then gets the value from memory and puts it in the accumulator

smem: Type I instruction with opcode 11 3

Takes the value at a register, adds the immediate to create a memory address, then sets the memory at the location to the value in the accumulator

beq: Type III instruction with opcode 100 4

Conditionally branch if reg == acc, then go to the mem location contained in dest.

bne: Type III instruction with opcode 101 5

Conditionally branch if reg != acc, then go to the mem location contained in dest.

sll: Type II instruction with opcode 110 6

Left shift the value in accumulator by constant value

slt: Type I instruction with opcode 111 7

Decide order using unused bits. If no imm in instruction, assembler assumes 0. 0 means acc<reg, 1 means reg<acc. If inequality is true, acc gets 1. else acc gets 0.

or: Type I instruction with opcode 1000 8

Bitwise OR operation for a reg and acc

and: Type I instruction with opcode 1001 9

Bitwise AND operation for a reg and acc

lui: Type II instruction with opcode 1010 A

Load the upper half of an immediate into accumulator

jr: Type I instruction with opcode 1011 B

Jump to an address within a register

jal: Type I instruction with opcode1100 C

Jump to an address within a register and save ra in the register

ori: Type II instruction with opcode 1101 D

Bitwise or zero extended immediate into the accumulator

lacc: Type I instruction with opcode 1110 E

Load value in specified register into the acc

sacc: Type I instruction with opcode 1111 F

Store value in acc to given reg

Pattern

Procedure calls

During procedure calls, we will still back up all the registers we need including $ra into stack, and restore them when finished. We use $a0-$a3 to store arguments, and $v0 and $v1 to store return values. If we need more arguments or return values, we will store them in the stack. Backing up registers looks like the following: If acc is needed, back it up first. Afterwards, move values from registers into the acc, then onto the stack.

Translate to machine code

For logical instructions, it will zero extend immediates. For arithmetic instructions, it will be sign extend.

For op codes and registers, it will zero extend to 4 bits.

For beq, bne, jr, and jal we use direct addressing, meaning it will go to the address in the register specified.

Example and test cases

Example Assembly Code for Relative Prime:			Hex Code:
relPrime:	#$a0=n		
lui	0		A000
ori	2		D020
sacc	$s0	#s0=m=2	FA00
looop:			
lacc	$s0	#for loop	EA00
sacc	$a1	#a1=m	F200
lacc	$sp	#stack	ED00
lui	-12		AF40
ori	-12		DFF0
add	$sp		0D00
sacc	$sp	#$sp-=12	FD00
	#Code back up $ra, $a0, $s0		
lacc	$ra		EC00
smem	$sp, 0		3D00
lacc	$a0		E200
smem	$sp, 4		3D04
lacc	$s0		EA00
smem	$sp, 8		3D08
lui	gcd		A address of gcd
ori	gcd		D address of gcd
jal	$acc		C100
	#Code to restore $ra, $a0, $s0		
lmem	$sp, 0		2D00
sacc	$ra		FC00
lmem	$sp, 4		2D04
sacc	$a0		F200

```
     lmem  $sp, 8                2D08
     sacc  $s0                   FA00

     lacc  $sp  #stack           ED00
     lui   0                     A000
     ori   12                    D0C0
     add   $sp                   0D00
     sacc  $sp  #$sp+=12         FD00
     lui   output                A address of output
     ori   output                D address of output
     sacc  $t0                   F800
     lui   0                     A000
     ori   1                     D010
     beq   $v0, $t0              4680
     add   $s0                   0A00
     sacc  $s0  #m=m+1           FA00

     lui   looop                 A upper of addr of loop
     ori   looop                 D upper of addr of loop
     jr    $acc                  B100

output:
     lacc  $s0 #$acc = m         EA00
     sacc  $v0 #$v0 = m          F600
     jr    $ra                   BC00

gcd:
     lui   bcase                 A address of bcase
     ori   bcase                 D address of bcase
     sacc  $t0                   F800
     lui   0                     A000
     beq   $a0, $t0              4810
```

```
loop:
     lui   acase                  A address of acase
     ori   acase                  D address of acase
     sacc  $t0                    F800
     lui   0                      A000
     beq   $a1, $t0               4380
     lacc  $a0  #$acc=a           E200
     slt   $a1  #$acc=(b<a?1:0)   7300
     sacc  $t0                    F800
     lui   asub                   A address of asub
     ori   asub                   D address of asub
     sacc  $t1                    F900
     lui   0                      A000
     ori   1                      D010
     beq   $t0, $t1               4890
     lacc  $a1                    E300
     sub   $a0  #a<b                   1200
     sacc  $a1                    F300
     lui   loop                        A upper address of loop
     ori   loop                        D lower address of loop
     jr    $acc                   B100

bcase:
     lacc  $a1                    E300
     sacc  $v0                    F600
     jr    $ra                    BC00

asub:
     lacc  $a0                    E200
     sub   $a1                    1300
     sacc  $a0                    F200
```

```
     lui   loop                          A upper address of loop
     ori   loop                          D lower address of loop
     jr    $acc                    B100

acase:
     lacc  $a0                     E200
     sacc  $v0                     F600
     jr    $ra                     BC00
```

Use psuedo instrutions (we use to run the relprime):

```
relPrime:
li $s0 2
looop:
lacc $s0
sacc $a1
addi $sp $sp 3
lacc $ra
smem $sp 0
lacc $a0
smem $sp -1
lacc $s0
smem $sp -2
jalp gcd
lmem $sp 0
sacc $ra
lmem $sp -1
sacc $a0
lmem $sp -2
sacc $s0
addi $sp $sp -3
li $acc 1
beqp $v0 $acc output
```

```
add $s0
sacc $s0
j looop
output:
lacc $s0
sacc $v0
jr $ra
gcd:
beqp $a0 $0 bcase
loop:
beqp $a1 $0 acase
lacc $a0
slt $a1
sacc $t0
li $acc 1
beqp $t0 $acc asub
lacc $a1
sub $a0 1
sacc $a1
j loop
bcase:
lacc $a1
sacc $v0
jr $ra
asub:
lacc $a0
sub $a1 1
sacc $a0
j loop
acase:
lacc $a0
sacc $v0
```

```
jr $ra
```

Loading an address into accumulator:

```
     lui   addr                    A upper address of addr
     ori   addr                    D lower address of addr
```

Iteration:

```
loop:
     lui   0                  A000
     ori   1                  D010
     add   $a0                0200
     sacc  $a0                F200
     lui   loop                    A upper address of loop
     ori   loop                    D lower address of loop
     jr    $acc               B100
```

Conditional Statement:

```
top:
     lui    0                 A000
     slt    $a0                    7200
     sacc   $t0                    F800
     lui    0                 A000
     lui    case              A address of case
     ori    case              D address of case
     beq    $t0, $acc         4810

case:
     lui    top                    A upper address of top
     ori    top                    D lower address of top
     jr     $acc              B100
```

RTL
See below

Logical/Arithmetic/Slt	sacc/lacc	lmem/smem	sll	beq/bne	lui	jr/jal
IR = Mem[PC] PC = PC + 1						
C = Reg[1] A = Reg[IR[11-8]] B = Reg[IR[7-4]] ori: order = IR[8] otherwise: order = IR[0]						

ori: Use ZE[IR[7-0]] instead of A
 	
if(order==0)
ALUOut=A op C
else
ALUOut=C op A
	sacc:
	Reg[IR[11-8]] = C
	
	lacc:
	Reg[1] = A
		ALUOut=A+ SE(IR[7-0])	Reg[1] = C<<IR[11-0]	beq:
				if(A==C)
				PC=B
				
				bne:
				if(A!=C)
				PC=B
					Reg[1] = ZE(IR[7-0])<<8	jal:
						Reg[12]=PC
						
						jr and jal:
						PC = A

Reg[1] = ALUOut		lmem: MDR = Mem[ALUOut] smem: Mem[ALUOut] = C				
		lmem: Reg[1] = MDR				

Component description

Component	Description	Inputs	Outputs	RTL Interactions
Register file	Used for register operations including the Accumulator. Will take in IR[11-8] and IR[7-4] and output into A and B respectively. The register file will also have a write address input that will mainly be the accumulator register, and a write data which will be the data written into the register. This data will be coming from ALUout and the memory. Lastly, it will output the value in $acc to C. Other register files are A, B, C, ALUOut, IR, MDR, and PC	IR[11-8] - ra1 - 4b IR[7-4] - ra2 - 4b write address - wa - 4b write data - wd - 16b	rd1 - A - 16b rd2 - B - 16b rd3 - C - 16b	IR PC C A B Order
Memory	Will mainly be used to find the instruction, but also will be used to store values. Takes in an input from the PC if the memory read signal is on, and outputs the correct instruction from memory. Also can take in a signal and write it to memory if the Memory write signal is on.	Address in - addin - 16b Data in - din - 16b	Data out - do - 16b	PC IR ALUout MDR
ALU	ALU will take in two inputs and perform an operation depending on the ALU control. This can be +, -, &, |, or <.	input 1 - A - Output of ALU source A mux - 16b	ALUout - 16b	PC A C ZE(IR[11-4])

		input 2 - B - Output of ALU source B mux - 16b ALU control -3b input 3 - order bits that decide which order the inputs should be operated on		
Control	Decides the value of the various muxes, write and read codes to perform a certain operation based on the Instruction Registers output.	instruction from memory - 16b	Various mux signals, write and read signals, and ALU control - bits depend on amount of mux inputs	Controls what RTL runs
Shift left	shifts the value coming in left filling the end with zeroes and then outputting it.	Value coming in from ALU or instruction - 16b Number to shift by - 12b	Value shifted	IR C Register file
Comparator	Compares two signals to see if they are equal	C and A	BranchDecide	A and C

Control Signals

Control Signals	Description
Instruction or data IorD -1b	Controls whether the PC goes into memory to find instruction or data to write to memory
ALU source B ALUSrcB - 2b	Controls which value goes into the second input of the ALU
ALU source A ALUSrcA - 2b	Controls which value goes into the first input of the ALU
PC source PCSrc - 2b	Controls whether PC + 1 goes into the PC or a jump/branch target

Shift Amount ShiftAmt - 1b	Decides the value that goes into the shift input to choose how much to shift by.
Shift Select ShiftSrc - 1b	Dicides the value that will be shifted. Sign extended order or Accumulator.
Register Destination RegDst - 2b	Chooses which register to write to.
Register Data RegData - 3b	Chooses which value gets put in the register.
Order Control OrderControl - 1b	Chooses what bit of the instruction actually constitutes the order bits
Register File write: regwrite - 1b	Chooses Whether or not to write to the register file.
Memory write: memw - 1b	Chooses whether or not to write to Memory.
Memory read: memr - 1b	Chooses whether or not to read from Memory.
PC Write PCWrite 1b	Chooses whether or not to write to the PC
PC Write Conditional PCWriteCond 1b	Chooses whether a conditional will allow the PC to write
Instruction Register Write IRwrite 1b	Chooses whether or not to write to the IR.
ALU control 3b	Decides the operation of the ALU
EorNE	Chooses whether the comparator should compare equals or not equals for the two inputs to the comparator

RTL Test

This is code and RTL written by Anderson and Yuankai. Anderson was a little bit involved with creating the original RTL, but most of the RTL design was done by Caleb and Brian. We made this test code for them to think about what the RTL did and to find any errors we may have made when designing the RTL.

```
          RA=0x6666
          PC=0x0000
          SP=0x7FFF

          test:
0x0000         lui  0
               ori  0
               sacc $t0
               lui  0
               ori  1
               add  $t0
               sacc $a0
               lui  (upper)-4
               ori  (lower)-4
               add  $sp
               sacc $sp
               lacc $ra
               smem $sp, 0
               lui  (upper)branch
               ori  (lower)branch
               jal  $acc
               lmem $sp, 0
               sacc $ra
               lui  0
               ori  4
               add  $sp
               sacc $sp
               jr   $ra

          branch:
0x0100         lui  0
               sacc $t0
```

```
                sub  $a0
                slt  $t0, 1
                sacc $t1
                lui  0
                beq  $t1, $acc
                jr   $ra

start:
     IR=0x0000
     PC=0x0000
lui 0:
     0xA000
     PC=0x0004
     Reg[1]=0x0000=>(sll 8) 0x0000
ori 2:
     IR=0xD020
     PC=0x0008
     A=>0x02=>(ZE) 0x0002
     ALUOut=A or Reg[1]=0x0002
     Reg[1]=ALUOut=0x0002
sacc $t0:
     IR=0xF800
     PC=0x000C
     Reg[8]=Reg[1]=0x0002
lui 0:
     IR=0xA000
     PC=0x0010
     Reg[1]=0x0000=>(sll 8) 0x0000
Ori 1:
     IR=0x0010
     PC=0x0014
     A=>0x01=>(ZE) 0x0001
     ALUOut=A or Reg[1]=0x0001
     Reg[1]=ALUOut=0x0001
add $t0:
     IR=0x0800
     PC=0x0018
     A=Reg[8]
     ALUout=Reg[1]+Reg[8]=0x0001+0x0002=0x0003
```

```
     Reg[1]=ALUOut=0x0003
sacc $a0:
     IR=0xF200
     PC=0x001C
     Reg[2]=Reg[1]=0x0003
lui (upper)-4:
     IR=0xAFF0
     PC=0x0020
     Reg[1]=0x00FF=>(sll 8) 0xFF00
ori (lower)-4:
     IR=0xDFC0
     PC=0x0024
     A=>0xFC=>(ZE) 0x00FC
     ALUout=A or Reg[1]=0xFFFC
     Reg[1]=ALUOut=0xFFFC
add $sp:
     IR=0x0000
     PC=0x0028
     A=Reg[13]
     ord=0
     ALUOut=Reg[1]+A=0xFFFC+0x7FFF=0x7FFB
sacc $sp:
     IR=0xFD00
     PC=0x002C
     Reg[13]=Reg[1]=0x7FFB

lacc $ra:
     IR=0xEC00
     PC=0x0030
     A=Reg[12]
     Reg[1]=Reg[13]=0x6666
smem $sp, 0:
     IR=0x3D00
     PC=0x0034
     A=Reg[13]=0x7FFB
     ALUOut=A+(SE)0=0x7FFB
     MEM[0x7FFB]=Reg[1]=0x6666
lui (upper)branch:
     IR=0xA010
```

```
     PC=0x0038
     Reg[1]=0x0001=>(sll 8) 0x0100
ori (lower) branch?
     IR=0xD000
     PC=0x003C
     A=0x00
     ALUOut=Reg[1] or ZE(A)=0x????
     Reg[1]=0x0100
jal $acc:
     IR=0xC100
     PC=0x0040
     Reg[12]=0x0040
     PC=Reg[1]=0x0100
lmem $sp, 0:
     IR=0x2D00
     PC=0x0044
     A=Reg[13]
     Reg[1]=A=
sacc $ra:
     IR=0xFC00
     PC=0x0048
     A=0x0C=12
     Reg[12]=Reg[1]=0x6666
lui 0:
     IR=0xA000
     PC=0x004C
     Reg[1]=0x00=>(sll 8) 0x0000
ori 4:
     IR=0xD400
     PC=0x0050
     A=0x04
     Reg[1]=Reg[1] or ZE(A)=0x0004
add $sp:
     IR=0x0D00
     PC=0x0054
     Reg[1]=Reg[1]+Reg[13]=0x0004+0x7FFB=0x7FFF
sacc $sp:
     IR=0xFD00
     PC=0x0058
```

```
     A=0x0D=13
     Reg[13]=Reg[1]=0x7FFF
jr   $ra:
     IR=0xBC00
     PC=0x005C
     PC=Reg[1]=0x6666

branch:
lui 0:
     IR=0xA000
     PC=0x0100
     Reg[1]=0x0000=>sll 8 0x0000
sacc $t0:
     IR=0xF800
     PC=0x0104
     Reg[8]=reg[1]=0x0000
sub $a0, 1:
     IR=0x2201
     PC=0x0108
     A=Reg[2]=0x0003
     ord=1
     ALUOut=Reg[2]-Reg[1]=0x0003-0x0000=0x0003
     Reg[1]=ALUOut=0x0003
slt $t0, 1:
     IR=0x7801
     PC=0x010C
     ord=1
     ALUOut=Reg[8]<Reg[1]=0x0000<0x0000=0
     Reg[1]=ALUOut=0x0000
sacc $t1:
     IR=0xF900
     PC=0x0110
     Reg[9]=Reg[1]=0x0000
lui 0:
     IR=0xA000
     PC=0x0114
     Reg[1]=0x0000=>(sll 8) 0x0000
lui (upper)test:
     IR=0xA000
```

```
     PC=0x0118
     Reg[1]=0x0000=>(sll 8) 0x0000
ori (lower)test:
     IR=0xD000
     PC=0x001C
     A=0x00
     Reg[1]=Reg[1] or ZE(A)=0x0000
beq $t1, $acc:
     IR=0x4910
     PC=0x0120
     PC=Reg[1]=>0x0000
jr $ra:
     IR=0xBC00
     PC=0x0124
     PC=Reg[1]=0x0000
```

Comments for RTL design: When Anderson and I (Kevin) wrote the test for rtl, we feel the instructions are easy to understand and write. We can easily understand the way it works. The RTL combines many similar instructions, so it is easy to know the pattern. The test is mainly show every small part works as expected. It does not purposefully calculate anything.

Component Test Plan

Step1: Individual test

1. ALU Test

Input	Output
ALUcontrol:000(and) input1[7:0]:from 0000 0000 to 1111 1111 input2[7:0]:from 0000 0000 to 1111 1111	[7:0]:input1 and input2
ALUcontrol:001(or) input1[7:0]:from 0000 0000 to 1111 1111 input2[7:0]:from 0000 0000 to 1111 1111	[7:0]:input1 or input2
ALUcontrol:010(add) input1[7:0]:from 0000 0000 to 1111 1111 input2[7:0]:from 0000 0000 to 1111 1111	[7:0]:input1+input2 (may overflow)
ALUcontrol:110(sub) input1[7:0]:from 0000 0000 to 1111 1111 input2[7:0]:from 0000 0000 to 1111 1111	[7:0]:input1-input2 (may overflow)

ALUcontrol:111(slt)
input1[7:0]:from 0000 0000 to 1111 1111
input2[7:0]:from 0000 0000 to 1111 1111
	[7:0]:
	if (input1<input2)
	1
	else
	0

2. Mem Test

Input	Output
save things into the memory (once at a time)	load from memory to see if it the same
save things into the memory (several data at a time)	load from memory to see if it the same

3. Register file Test

Input	Output
save things into register (once at a time)	load from register to see if it the same
save things into register (several data at a time)	load from register to see if it the same

4. SE Test

Input	Output
input[x:0]:from 0000 0000 to 1111 1111	[15:x+1]=input[x] [x:0]=input

5. ZE Test

Input	Output
input[x:0]:from 0000 0000 to 1111 1111	[15:x+1]=0 [x:0]=input

6. Left Shift Test

Input	Output
number to shift[7:0]:from 0000 0000 to 1111 1111 bits to shift[1:0]: from 0 to 8	[7:0]:number to shift<<bits to shift

7. == Test

Input	Output
input1[7:0]:from 0000 0000 to 1111 1111 input2[7:0]:from 0000 0000 to 1111 1111	input1==input2 1 for true, 0 for false

8. mux Test

Input	Output
mux control:from 000 to 101 input0, input1, input2, input3, input4, input5	mux control=000, input0 mux control=001, input1 mux control=010, input2 mux control=011, input3 mux control=100, input4 mux control=101, input5

Step 2: Integration test

1. PC+1 Test

Components	Input	Output
PC, ALU	PC (include edge cases like 0000 0000, FFFF FFFF)	PC=PC+1

2. IR=Mem[PC] Test

Components	Input	Output
Mem, IR, PC	PC (include edge cases like 0000 0000, FFFF FFFF)	IR=Mem[PC]

3. sacc Test

Components	Input	Output
Reg, IR, C	C IR	Reg[IR[11-8]]=C Check if the register has stored the value

4. lacc Test

Components	Input	Output
Reg, A	A	Reg[1]=A load the value in accumlator out to see the value matches

5. Reg and Other calculation Test

Components	Input	Output
IR, Reg, A, B, C, ==, SE, ZE, <<	A=Reg[IR[11-8]] B=Reg[IR[7-4]] C=Reg[1]	Get the data from instruction and register file, and run them in the == or SE or ZE or << to do some calculation

6. RegToMem Test

Components	Input	Output
ALU, ALUOut, A, Mem, C, IR	IR, A, C	ALUOut=A+SE(IR[7-0]) Mem[ALUOut]=C Check if the memory has stored the value at required address

7. MemToReg Test

Components	Input	Output
ALU, ALUOut, MDR, A, IR, Reg, Mem	ALU, A	ALUOut=A+SE(IR[7-0]) MDR=Mem[ALUOut] Reg[1]=MDR Check if the $acc has stored the changed value

8. PCReg Test

Components	Input	Output
PC, IR, ALU, ALUOut, Reg	IR, PC	Write the value of the PC into Register Load the value stored in Register into PC

9. beq Test

Components	Input	Output
A, C, PC, B	A, C, PC, B	if(A==C) PC=B Check if PC is equal to B when A is equal to C

10. bne Test

Components	Input	Output
A, C, PC, B	A, C, PC, B	if(A!=C) PC=B Check if PC is equal to B when A is not equal to C

11. lui Test

Components	Input	Output
Left Shift Unit(LSU), SE Unit, Reg, IR	IR	Reg[1]=IR[7-0]<<8 Check if $acc has the correct value in register

12. jr Test

Components	Input	Output
PC, Reg, IR	PC, IR	PC=Reg[IR[11-8]] Check if PC jumps to the correct value

13. jal Test

Components	Input	Output
PC, Reg, IR	PC, IR	Reg[12]=PC PC=Reg[IR[11-8]] Check if PC jumps to the correct value Check if $ra is equal to the place it should jump back

14. PCMemReg Test

Components	Input	Output
PC, Mem, IR, Reg	PC, IR	Store PC into Register ,and then store this value into Memory Load from Memory into Register, then set its value to PC

15. PCMemRegALU Test

Components	Input	Output
PC, Mem, IR, Reg, ALU	PC, IR	Store PC into Register, do some calculation, and then store this value into Memory Load from Memory into Register, do some calculation, and then set its value to PC

16. Reg ALU Test

Components	Input	Output
Reg, A, B, C, ALU	A=Reg[IR[11-8]] B=Reg[IR[7-4]] C=Reg[1]	Get the data from register file and run them in the ALU to do some calculation

Step 3: Full datapath test

When we test it, we will need to save all the instructions to memory first, edit any possible inputs in the test bench so that the input will be set to $a0.

1. A small piece of test code to just combine some instructions together.

Column1	Cout	t0	t1	s0	instructions in hex
lui 0	0	X	X	X	a000
ori 12	12	X	X	X	d00c
sacc $t2	12	X	X	12	fa00
lui 0	0	X	X	12	a000

ori 4	4	X	X	12	d004
sacc $t1	4	X	4	12	f900
lui 0	0	X	4	12	a000
ori 1	1	X	4	12	d001
sacc $t0	1	1	4	12	f800
lacc $t0	1	1	4	12	e800
add $t0	2	1	4	12	800
sacc $t0	2	2	4	12	f800
lacc $t1	4	2	4	12	e900
bne $t0 $t2	4	2	4	12	58a0
lacc $t0	2	2	4	12	e800
add $t0	4	2	4	12	800
sacc $t0	4	4	4	12	f800
lacc $t1	4	4	4	12	e900
bne $t0 $t2	4	4	4	12	58a0
lui 0	0	4	4	12	a000
ori 3	3	4	4	12	d003
jr $acc	3	4	4	12	b100

We look into the waves shown in the simulation and it all works as expected. We manually select the address we need to branch or jump to in this test code.

2. The Eculid's algorithm

We mainly test it first with an input of 4 to see how it works on a small number. After it works, we try large numbers. We mainly debug by seeing the position of our PC is in the right place or not.

If it is in the right place, then we have a large chance that it did the correct calculation, and therefore branch or jump to the right place.

PC	Lable	Pseudo Instructions	Assembly Code	Hex	Binary
0		lui b00000000	lui b00000000	a000	1010000000000000
1		ori b01110100	ori b01110100	d074	1101000001110100
2		sacc $sp	sacc $sp	fd00	1111110100000000
3		lacc $0	lacc $0	e000	1110000000000000
4	relPrime:	li $s0 2	sacc $at0	fe00	1111111000000000
5			lui b00000000	a000	1010000000000000
6			ori b00000010	d002	1101000000000010
7			sacc $s0	fa00	1111101000000000
8			lacc $at0	ee00	1110111000000000
9	looop:	lacc $s0	lacc $s0	ea00	1110101000000000
10		sacc $a1	sacc $a1	f300	1111001100000000
11		addi $sp $sp3	sacc $at0	fe00	1111111000000000
12			lui b00000000	a000	1010000000000000
13			ori b00000011	d003	1101000000000011
14			add $sp	0d00	0000110100000000
15			sacc $sp	fd00	1111110100000000
16			lacc $at0	ee00	1110111000000000
17		lacc $ra	lacc $ra	ec00	1110110000000000
18		smem $sp0	smem $sp0	3d00	0011110100000000
19		lacc $a0	lacc $a0	e200	1110001000000000
20		smem $sp-1	smem $sp-1	3dff	0011110111111111

21		lacc $s0	lacc $s0	ea00	1110101000000000
22		smem $sp-2	smem $sp-2	3dfe	0011110111111110
23		jal pgcd	sacc $at0	fe00	1111111000000000
24			lui b00000000	a000	1010000000000000
25			ori b00111100	d03c	1101000000111100
26			sacc $at1	ff00	1111111100000000
27			lacc $at0	ee00	1110111000000000
28			jal $at1	cf00	1100111100000000
63	gcd:	addi $sp $sp3	sacc $at0	fe00	1111111000000000
64			lui b00000000	a000	1010000000000000
65			ori b01011111	d05f	1101000001011111
66			sacc $at1	ff00	1111111100000000
67			lacc $a0	e200	1110001000000000
68			beq $0 $at1	40f0	0100000011110000
69			lacc $at0	ee00	1110111000000000
70	loop:	beqp $a1 $0 acase	sacc $at0	fe00	1111111000000000
71			lui b00000000	a000	1010000000000000
72			ori b01101001	d069	1101000001101001
73			sacc $at1	ff00	1111111100000000
74			lacc $a1	e300	1110001100000000
75			beq $0 $at1	40f0	0100000011110000
76			lacc $at0	ee00	1110111000000000
77		lacc $a0	lacc $a0	e200	1110001000000000
78		slt $a1	slt $a1	7300	0111001100000000
79		sacc $t0	sacc $t0	f800	1111100000000000

80		li $acc1	sacc $at0	fe00	1111111000000000
81		beqp $t0 $acc asub	lui b00000000	a000	1010000000000000
82			ori b00000001	d001	1101000000000001
83			sacc $acc	f100	1111000100000000
84			sacc $at0	fe00	1111111000000000
85			lui b00000000	a000	1010000000000000
86			ori b01100001	d061	1101000001100001
87			sacc $at1	ff00	1111111100000000
88			lacc $t0	e800	1110100000000000
89			beq $at0 $at1	4ef0	0100111011110000
90			lacc $at0	ee00	1110111000000000
103	asub:	lacc $a0	lacc $a0	e200	1110001000000000
104		sub $a1 1	sub $a1	1301	0001001100000001
105		sacc $a0	sacc $a0	f200	1111001000000000
106		j loop	sacc $at0	fe00	1111111000000000
107			lui b00000000	a000	1010000000000000
108			ori b01000010	d042	1101000001000010
109			sacc $at1	ff00	1111111100000000
110			lacc $at0	ee00	1110111000000000
111			jr $at1	bf00	1011111100000000
70	loop:	beqp $a1 $0 acase	sacc $at0	fe00	1111111000000000
71			lui b00000000	a000	1010000000000000
72			ori b01101001	d069	1101000001101001
73			sacc $at1	ff00	1111111100000000
74			lacc $a1	e300	1110001100000000

75		beq $0 $at1	40f0	0100000011110000
76		lacc $at0	ee00	1110111000000000
77	lacc $a0	lacc $a0	e200	1110001000000000
78	slt $a1	slt $a1	7300	0111001100000000
79	sacc $t0	sacc $t0	f800	1111100000000000
80	li $acc1	sacc $at0	fe00	1111111000000000
81	beqp $t0 $acc asub	lui b00000000	a000	1010000000000000
82		ori b00000001	d001	1101000000000001
83		sacc $acc	f100	1111000100000000
84		sacc $at0	fe00	1111111000000000
85		lui b00000000	a000	1010000000000000
86		ori b01100001	d061	1101000001100001
87		sacc $at1	ff00	1111111100000000
88		lacc $t0	e800	1110100000000000
89		beq $at0 $at1	4ef0	0100111011110000
90		lacc $at0	ee00	1110111000000000
91	lacc $a1	lacc $a1	e300	1110001100000000
92	sub $a0 1	sub $a0	1201	0001001000000001
93	sacc $a1	sacc $a1	f300	1111001100000000
94	j loop	sacc $at0	fe00	1111111000000000
95		lui loop		
96		ori loop		
97		sacc $at1		
98		lacc $at0		
99		jr $at1		

70	loop:	beqp $a1 $0 acase	sacc $at0	fe00	1111111000000000
71			lui b00000000	a000	1010000000000000
72			ori b01101001	d069	1101000001101001
73			sacc $at1	ff00	1111111100000000
74			lacc $a1	e300	1110001100000000
75			beq $0 $at1	40f0	0100000011110000
76			lacc $at0	ee00	1110111000000000
112	acase:	lacc $a0	lacc $a0	e200	1110001000000000
113		sacc $v0	sacc $v0	f600	1111011000000000
114		jr $ra	jr $ra	bc00	1011110000000000
29		lmem $sp0	lmem $sp0	2d00	0010110100000000
30		sacc $ra	sacc $ra	fc00	1111110000000000
31		lmem $sp-1	lmem $sp-1	2dff	0010110111111111
32		sacc $a0	sacc $a0	f200	1111001000000000
33		lmem $sp-2	lmem $sp-2	2dfe	0010110111111110
34		sacc $s0	sacc $s0	fa00	1111101000000000
35		addi $sp $sp-3	sacc $at0	fe00	1111111000000000
36			lui b11111111	a0ff	1010000011111111
37			ori b11111111	d0ff	1101000011111111
38			add $sp	0d00	0000110100000000
39			sacc $sp	fd00	1111110100000000
40			lacc $at0	ee00	1110111000000000
41		li $acc 1	sacc $at0	fe00	1111111000000000
42			lui b00000000	a000	1010000000000000

43			ori b00000001	d001	1101000000000001
44			sacc $acc	f100	1111000100000000
45		beqp $v0 $acc output	sacc $at0	fe00	1111111000000000
46			lui b00000000	a000	1010000000000000
47			ori b00111010	d03a	1101000000111010
48			sacc $at1	ff00	1111111100000000
49			lacc $v0	e600	1110011000000000
50			beq $at0 $at1	4ef0	0100111011110000
51			lacc $at0	ee00	1110111000000000
52		add $s0	add $s0	0a00	0000101000000000
53		sacc $s0	sacc $s0	fa00	1111101000000000
54		j looop	sacc $at0	fe00	1111111000000000
55			lui b00000000	a000	1010000000000000
56			ori b00001000	d008	1101000000001000
57			sacc $at1	ff00	1111111100000000
58			lacc $at0	ee00	1110111000000000
59			jr $at1	bf00	1011111100000000
9	looop:	lacc $s0	lacc $s0	ea00	1110101000000000
10		sacc $a1	sacc $a1	f300	1111001100000000
11		addi $sp $sp3	sacc $at0	fe00	1111111000000000
12			lui b00000000	a000	1010000000000000
13			ori b00000011	d003	1101000000000011
14			add $sp	0d00	0000110100000000
15			sacc $sp	fd00	1111110100000000

16			lacc $at0	ee00	1110111000000000
17		lacc $ra	lacc $ra	ec00	1110110000000000
18		smem $sp0	smem $sp0	3d00	0011110100000000
19		lacc $a0	lacc $a0	e200	1110001000000000
20		smem $sp-1	smem $sp-1	3dff	0011110111111111
21		lacc $s0	lacc $s0	ea00	1110101000000000
22		smem $sp-2	smem $sp-2	3dfe	0011110111111110
23		jalp gcd	sacc $at0	fe00	1111111000000000
24			lui b00000000	a000	1010000000000000
25			ori b00111100	d03c	1101000000111100
26			sacc $at1	ff00	1111111100000000
27			lacc $at0	ee00	1110111000000000
28			jal $at1	cf00	1100111100000000
63	gcd:	beqp $a0 $0 bcase	sacc $at0	fe00	1111111000000000
64			lui b00000000	a000	1010000000000000
65			ori b01011111	d05f	1101000001011111
66			sacc $at1	ff00	1111111100000000
67			lacc $a0	e200	1110001000000000
68			beq $0 $at1	40f0	0100000011110000
69			lacc $at0	ee00	1110111000000000
70	loop:	beqp $a1 $0 acase	sacc $at0	fe00	1111111000000000
71			lui b00000000	a000	1010000000000000
72			ori b01101001	d069	1101000001101001
73			sacc $at1	ff00	1111111100000000
74			lacc $a1	e300	1110001100000000

75			beq $0 $at1	40f0	0100000011110000
76			lacc $at0	ee00	1110111000000000
77		lacc $a0	lacc $a0	e200	1110001000000000
78		slt $a1	slt $a1	7300	0111001100000000
79		sacc $t0	sacc $t0	f800	1111100000000000
80		li $acc1	sacc $at0	fe00	1111111000000000
81		beqp $t0 $acc asub	lui b00000000	a000	1010000000000000
82			ori b00000001	d001	1101000000000001
83			sacc $acc	f100	1111000100000000
84			sacc $at0	fe00	1111111000000000
85			lui b00000000	a000	1010000000000000
86			ori b01100001	d061	1101000001100001
87			sacc $at1	ff00	1111111100000000
88			lacc $t0	e800	1110100000000000
89			beq $at0 $at1	4ef0	0100111011110000
90			lacc $at0	ee00	1110111000000000
103	asub:	lacc $a0	lacc $a0	e200	1110001000000000
104		sub $a1 1	sub $a1	1301	0001001100000001
105		sacc $a0	sacc $a0	f200	1111001000000000
106		j loop	sacc $at0	fe00	1111111000000000
107			lui b00000000	a000	1010000000000000
108			ori b01000010	d042	1101000001000010
109			sacc $at1	ff00	1111111100000000
110			lacc $at0	ee00	1110111000000000
111			jr $at1	bf00	1011111100000000

70	loop:	beqp $a1 $0 acase	sacc $at0	fe00	1111111000000000
71			lui b00000000	a000	1010000000000000
72			ori b01101001	d069	1101000001101001
73			sacc $at1	ff00	1111111100000000
74			lacc $a1	e300	1110001100000000
75			beq $0 $at1	40f0	0100000011110000
76			lacc $at0	ee00	1110111000000000
77		lacc $a0	lacc $a0	e200	1110001000000000
78		slt $a1	slt $a1	7300	0111001100000000
79		sacc $t0	sacc $t0	f800	1111100000000000
80		li $acc1	sacc $at0	fe00	1111111000000000
81		beqp $t0 $acc asub	lui b00000000	a000	1010000000000000
82			ori b00000001	d001	1101000000000001
83			sacc $acc	f100	1111000100000000
84			sacc $at0	fe00	1111111000000000
85			lui b00000000	a000	1010000000000000
86			ori b01100001	d061	1101000001100001
87			sacc $at1	ff00	1111111100000000
88			lacc $t0	e800	1110100000000000
89			beq $at0 $at1	4ef0	0100111011110000
90			lacc $at0	ee00	1110111000000000
91		lacc $a1	lacc $a1	e300	1110001100000000
92		sub $a0 1	sub $a0	1201	0001001000000001
93		sacc $a1	sacc $a1	f300	1111001100000000
94		j loop	sacc $at0	fe00	1111111000000000

95			lui b00000000	a000	1010000000000000
96			ori b01000010	d042	1101000001000010
97			sacc $at1	ff00	1111111100000000
98			lacc $at0	ee00	1110111000000000
99			jr $at1	bf00	1011111100000000
70	loop:	beqp $a1 $0 acase	sacc $at0	fe00	1111111000000000
71			lui b00000000	a000	1010000000000000
72			ori b01101001	d069	1101000001101001
73			sacc $at1	ff00	1111111100000000
74			lacc $a1	e300	1110001100000000
75			beq $0 $at1	40f0	0100000011110000
76			lacc $at0	ee00	1110111000000000
77		lacc $a0	lacc $a0	e200	1110001000000000
78		slt $a1	slt $a1	7300	0111001100000000
79		sacc $t0	sacc $t0	f800	1111100000000000
80		li $acc1	sacc $at0	fe00	1111111000000000
81		beqp $t0 $acc asub	lui b00000000	a000	1010000000000000
82			ori b00000001	d001	1101000000000001
83			sacc $acc	f100	1111000100000000
84			sacc $at0	fe00	1111111000000000
85			lui b00000000	a000	1010000000000000
86			ori b01100001	d061	1101000001100001
87			sacc $at1	ff00	1111111100000000
88			lacc $t0	e800	1110100000000000
89			beq $at0 $at1	4ef0	0100111011110000

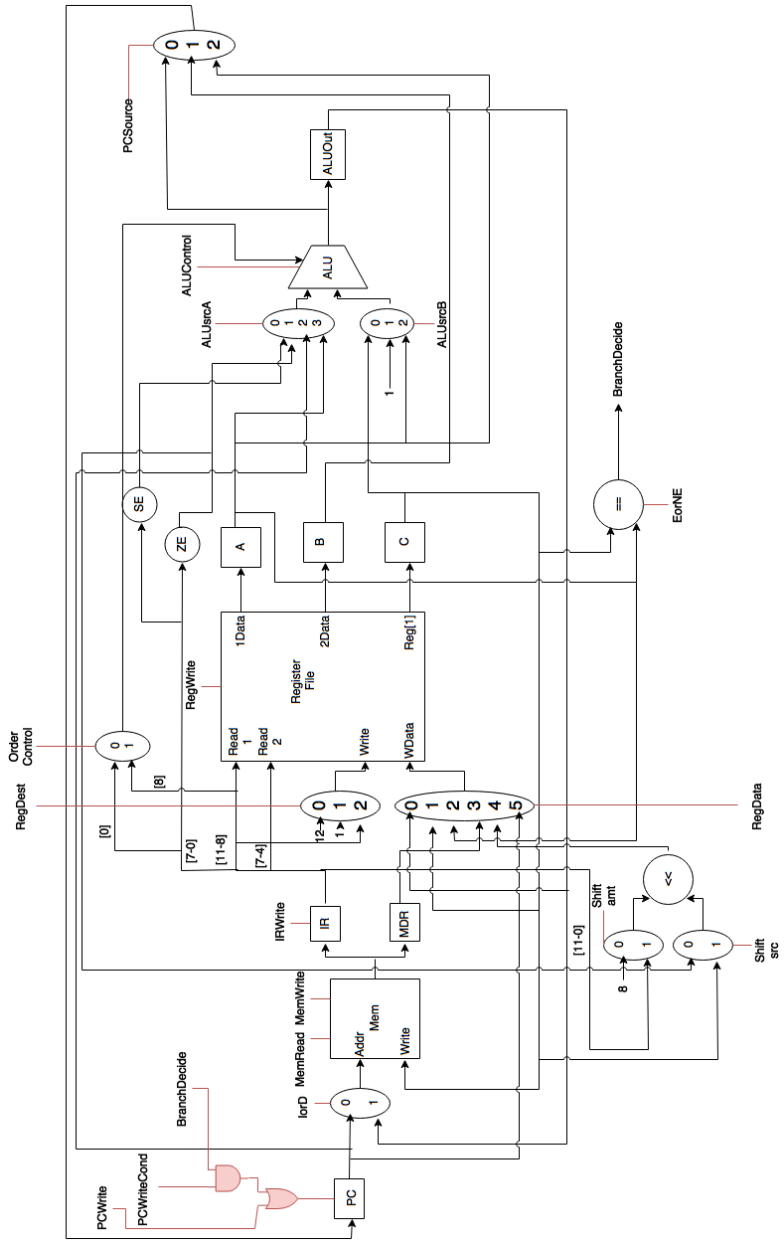
90			lacc $at0	ee00	1110111000000000
91		lacc $a1	lacc $a1	e300	1110001100000000
92		sub $a0 1	sub $a0	1201	0001001000000001
93		sacc $a1	sacc $a1	f300	1111001100000000
94		j loop	sacc $at0	fe00	1111111000000000
95			lui b00000000	a000	1010000000000000
96			ori b01000010	d042	1101000001000010
97			sacc $at1	ff00	1111111100000000
98			lacc $at0	ee00	1110111000000000
99			jr $at1	bf00	1011111100000000
70	loop:	beqp $a1 $0 acase	sacc $at0	fe00	1111111000000000
71			lui b00000000	a000	1010000000000000
72			ori b01101001	d069	1101000001101001
73			sacc $at1	ff00	1111111100000000
74			lacc $a1	e300	1110001100000000
75			beq $0 $at1	40f0	0100000011110000
76			lacc $at0	ee00	1110111000000000
77		lacc $a0	lacc $a0	e200	1110001000000000
78		slt $a1	slt $a1	7300	0111001100000000
79		sacc $t0	sacc $t0	f800	1111100000000000
80		li $acc1	sacc $at0	fe00	1111111000000000
81		beqp $t0 $acc asub	lui b00000000	a000	1010000000000000
82			ori b00000001	d001	1101000000000001
83			sacc $acc	f100	1111000100000000
84			sacc $at0	fe00	1111111000000000

85			lui b00000000	a000	1010000000000000
86			ori b01100001	d061	1101000001100001
87			sacc $at1	ff00	1111111100000000
88			lacc $t0	e800	1110100000000000
89			beq $at0 $at1	4ef0	0100111011110000
90			lacc $at0	ee00	1110111000000000
91		lacc $a1	lacc $a1	e300	1110001100000000
92		sub $a0 1	sub $a0	1201	0001001000000001
93		sacc $a1	sacc $a1	f300	1111001100000000
94		j loop	sacc $at0	fe00	1111111000000000
95			lui b00000000	a000	1010000000000000
96			ori b01000010	d042	1101000001000010
97			sacc $at1	ff00	1111111100000000
98			lacc $at0	ee00	1110111000000000
99			jr $at1	bf00	1011111100000000
70	loop:	beqp $a1 $0 acase	sacc $at0	fe00	1111111000000000
71			lui b00000000	a000	1010000000000000
72			ori b01101001	d069	1101000001101001
73			sacc $at1	ff00	1111111100000000
74			lacc $a1	e300	1110001100000000
75			beq $0 $at1	40f0	0100000011110000
76			lacc $at0	ee00	1110111000000000
112	acase:	lacc $a0	lacc $a0	e200	1110001000000000
113		sacc $v0	sacc $v0	f600	1111011000000000

114	jr $ra	jr $ra	bc00	1011110000000000
29	lmem $sp0	lmem $sp0	2d00	0010110100000000
30	sacc $ra	sacc $ra	fc00	1111110000000000
31	lmem $sp-1	lmem $sp-1	2dff	0010110111111111
32	sacc $a0	sacc $a0	f200	1111001000000000
33	lmem $sp-2	lmem $sp-2	2dfe	0010110111111110
34	sacc $s0	sacc $s0	fa00	1111101000000000
35	addi $sp $sp-3	sacc $at0	fe00	1111111000000000
36		lui b11111111	a0ff	1010000011111111
37		ori b11111111	d0ff	1101000011111111
38		add $sp	0d00	0000110100000000
39		sacc $sp	fd00	1111110100000000
40		lacc $at0	ee00	1110111000000000
41	li $acc1	sacc $at0	fe00	1111111000000000
42		lui b00000000	a000	1010000000000000
43		ori b00000001	d001	1101000000000001
44		sacc $acc	f100	1111000100000000
45	beqp $v0 $acc output	sacc $at0	fe00	1111111000000000
46		lui b00000000	a000	1010000000000000
47		ori b00111010	d03a	1101000000111010
48		sacc $at1	ff00	1111111100000000
49		lacc $v0	e600	1110011000000000
50		beq $at0 $at1	4ef0	0100111011110000
51		lacc $at0	ee00	1110111000000000

60	output:	lacc $s0	lacc $s0	ea00	1110101000000000
61		sacc $v0	sacc $v0	f600	1111011000000000
62		jr $ra	jr $ra	bc00	1011110000000000

Datapath



Control
IorD (1 bit)

Value	Effect
0	The memory address is set to PC
1	The memory address is set to ALUOut

MemRead

Value	Effect
0	None
1	Data memory contents designated by address input are put on Read data output.

MemWrite

Value	Effect
0	None
1	Data memory contents designated by the address input are replaced by the value on the Write data input.

IRWrite

Value	Effect
0	None
1	Value in IR is replaced by the value readed from memory (which is current instruction).

RegDest

Value	Effect
00	The register destination number for the write register is set to 12.
01	The register destination number for the write register is set to 1.
10	The register destination number for the write register comes from the IR field (Bits [11:8].
11	The register destination number for the write register is set to 3

RegData

Value	Effect
000	The data for writing into register is set to ALUOut.
001	The data for writing into register is set to value in C ($Accumulator).
010	The data for writing into register is set to value in A.
011	The data for writing into register is set to value in MDR.
100	The data for writing into register is set to value after shift
101	The data for writing into register is set to value PC
110	The data for writing into register is set to value in Input

Shiftamt

Value	Effect
0	Shift offset is set to 8
1	Shift offset is set to value in IR (bits [11:0])

Shiftsrc

Value	Effect
0	Value for shiftment is set to value in IR (Zero extended bits[7:0]).
1	Value for shiftment is set to value in C.

RegWrite

Value	Effect
0	None
1	The register on the Write register input is written with the value on the Write data input.

ALUsrcA

Value	Effect
00	Value inputting into first source of ALU is set to value in IR (Sign extended bits[7:0]).
01	Value inputting into first source of ALU is ZE(IR[7-0])
10	Value inputting into first source of ALU is set to value in PC
11	Value inputting into first source of ALU ifs set to the value in A

ALUsrcB

Value	Effect
00	Value inputting into second source of ALU is set to value in C
01	Value inputting into second source of ALU is set to 1
10	Value inputting into second source of ALU is set to value in A

ALUControl

Value	Effect

000	The operation performed in ALU will be and The output of the ALU will be ALUSrcA and ALUSrcB.
001	The operation performed in ALU will be or. The output of the ALU will be ALUSrcA or ALUSrcB.
010	The operation performed in ALU will be add The output of the ALU will be ALUSrcA add ALUSrcB.
110	The operation performed in ALU will be sub The output of the ALU will be ALUSrcA sub ALUSrcB.
111	The operation performed in ALU will be slt The output of the ALU will be ALUSrcA slt by ALUSrcB bits.

PCSource

Value	Effect
00	Output value of mux is set to value come from ALU
01	Output value of mux is set to value in B
10	Output value of mux is set to value in A

PCWriteCond

Value	Effect
0	PC will only update if PCWrite is 1.
1	If ALUOut is zero (or say, ***zero*** is 1), PC's value is updated

PCWrite

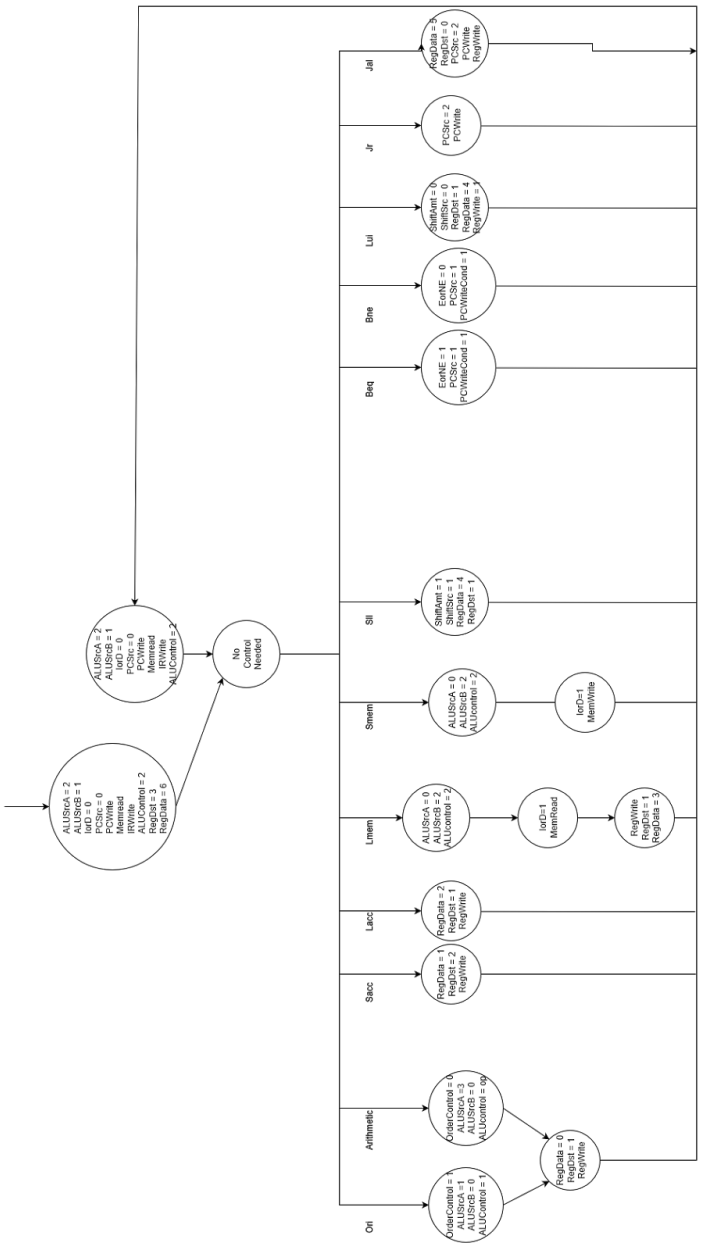
Value	Effect
0	PC will only update if the value of PCWriteCond and zero are both 1.
1	PC's value is updated

EorNE

Value	Effect
0	Value of BranchDecide is 1 if two inputs to the comparator are not equal. If two inputs are equal, value of BranchDecide is set to 0.
1	Value of BranchDecide is 0 if two inputs to the comparator are not equal. If two inputs are equal, value of BranchDecide is set to 1.

OrderControl

Value	Effect
0	Order bits for ALU are IR[7-0]
1	Order bits for ALU are ZE(IR[11-8])


ALUSrcA = 2
ALUSrcB = 1
IorD = 0
PCSrc = 0
PCWrite
Memread
IRWrite
ALUControl = 2
RegData = 3
RegData = 6
ALUSrcA = 2
ALUSrcB = 1
IorD = 0
PCWrite
Memread
IRWrite
ALUControl = 2
No Control Needed
Jal
RegData = 3
RegDst = 0
PCSrc = 2
PCWrite
RegWrite
Jr
PCSrc = 2
PCWrite
Lui
ShiftAmt = 0
ShiftSrc = 0
RegDst = 1
RegData = 4
RegWrite = 1
Bne
EorNE = 0
PCSrc = 1
(PCWriteCond = 1)
Beq
EorNE = 1
PCSrc = 1
(PCWriteCond = 1
Sll
ShiftAmt = 1
ShiftSrc = 1
RegData = 4
RegDst = 1
Smem
ALUSrcA = 0
ALUSrcB = 2
ALUcontrol = 2
IorD=1
MemWrite
Lmem
ALUSrcA = 0
ALUSrcB = 2
ALUcontrol = 2
IorD=1
MemRead
RegWrite
RegDst = 1
RegData = 5
Lacc
RegData = 2
RegDst = 1
RegWrite
Sacc
RegData = 1
RegDst = 2
RegWrite
Arithmetic
OrderControl = 0
ALUSrcA = 1
ALUSrcB = 0
ALUcontrol = op
Ori
OrderControl = 1
ALUSrcA = 1
ALUSrcB = 0
ALUControl = 1
RegData = 0
RegDst = 1
RegWrite

Control Test Plan
Cycle One

Control Signal	What it ends up doing
ALUSrcA = 2	Puts PC into the ALU
ALUSrcB = 1	Puts 1 into the ALU
PCWrite = 1	Allows a value to be written into PC
MemRead = 1	Allows a value to be read from memory
IRWrite = 1	Allows an instruction to be written to IR
PCSrc = 0	Sets the value going to PC the output of the ALU
ALUControl = 2	Allows 1 to be added to the PC in the ALU

In combination, these signals add one to PC and get the instruction at PC from memory. This cycle's control works.

Cycle Two

No control signals needed for cycle two.
Cycle Three
Ori

Control Signal	What it ends up doing
OrderControl = 1	Makes the order bits from the correct part of the instruction
ALUSrcA = 1	Sets the top input to the ALU the output of AorO
ALUSrcB = 0	Sets the bottom input to the ALU to C

This combination of these control signals gets C and the immediate from the instruction and ors them using the correct order(not that it really matters).

Arithmetic/Slt

Control Signal	What it actually does
OrderControl = 0	Sets the order bits to the last 8 bits of the instruction, which is were the order bits are for every arithmetic instruction.
ALUSrcA = 1	Sets the top input of the ALU to A
ALUSrcB = 0	Sets the bottom input of the ALU to C

This combination of control signals is very similar to the ones for ori, but it chooses A and a different set of order bits instead of an immediate and a different set of order bits.

Sacc

Control Signals	What they actually do
RegData = 1	Sets the value to put into the register file the value in C
RegDst = 2	Sets the register to write to the register in bits 11-8 of the instruction
RegWrite = 1	Allows for the register file to be written to

This combination of control signals sort of bypasses the ALU and writes directly to another register. It chooses to put C into whatever register is specified by the instructions

Lacc

Control Signals	What they actually do
RegData = 2	Sets the write data from the register file to the data in A
RegDst = 1	Sets the register that will be written to to the accumulator
RegWrite = 1	Allows for the register file to be written to

Exactly the same as sacc but register gets written to acc instead of the other way around. The changes in the control bits are what allow for this.

Lmem

Control Signal	What it actually does
ALUSrcA = 0	Same as above
ALUSrcB = 2	Puts A as the bottom input to the ALU

This set of control signals adds the immediate to the register specified.

Smem

Control Signal	What it actually does
AorO = 1	Puts the imm as the top input to the ALU
ALUSrcA = 0	Same as above
ALUSrcB = 2	Puts A as the bottom input to the ALU
IorD = 1	Makes memory location the value in ALUOut
MemWrite = 1	Allows for memory to be written to

This set of control signals does the exact same thing as lmem, but instead of reading from memory, it writes the accumulator value to memory at the calculated address.

Sll

Control Signal	What it actually does
ShiftAmt = 1	Sets the shift amount to the shifter to whatever is in the 11-0 imm
ShiftSrc = 1	Sets the thing to shift to C
RegData = 4	Sets the reg write data to the output of the shifter
RegDst = 1	Sets the register to write to to the accumulator

Operates with the shifter to shift the accumulator a certain amount

Beq

Control Signal	What it actually does
EorNE = 1	Sets the comparator to perform equals
PCSrc = 1	Sets the PC to value in B
PCWriteCond = 1	Makes sure the PC can be written to only if the condition is right

This puts the correct inputs into the comparator and jumps if correct

Bne

Control Signal	What it actually does

EorNE = 0	Sets the comparator to perform not equals
PCSrc = 1	Sets the PC to value in B
PCWriteCond = 1	Makes sure the PC can be written to only if the condition is right

This puts the correct inputs into the comparator and jumps if correct

Lui

Control Signal	What it actually does
ShiftAmt = 0	Sets the shift amount to the shifter to 8
ShiftSrc = 1	Sets the thing to shift to ZE(order)
RegData = 4	Sets the reg write data to the output of the shifter
RegDst = 1	Sets the register to write to to the accumulator

Very similar to sll, except it shifts an imm left 8 every time.

Jr

Control Signal	What it actually does
PCSrc = 2	Sets the value to put in PC to A
PCWrite = 1	Allows the PC to be written to

This basically just puts A into PC

Jal

Control Signal	What it actually does
RegData = 5	Sets the info to put in the register file to PC
RegDst = 0	Sets the register to write to to $ra
RegWrite = 1	Allows $ra to be written to
PCWrite = 1	Allows PC to be written to
PCSrc = 2	Sets PC to the jump target

This writes the PC into $ra. Basically just sets the controls for the Reg input stuff

Cycle 4

Ori/Arithmetic/Slt

RegData = 0	Sets the data that you are writing to the register to ALUOut
RegDst = 1	Sets the register you are writing to the accumulator
RegWrite = 1	Allows you to write to the register file

Writes the value in ALUOut to the accumulator
Lmem

Control Signal	What is actually does
IorD = 1	Makes memory location the value in ALUOut
MemRead = 1	Allows the memory to be read from

This writes to MDR
Smem

IorD = 1	Makes memory location the value in ALUOut
MemWrite = 1	Allows for memory to be read from

Cycle 5
Imem

Control Signal	What it actually does
RegWrite	Allows the register to be written to
RegDst = 1	Set the accumulator as the register to be written to.
RegData = 3	Writes MDR to the selected register.

Performance

While we test our datapath with 0x13B0 (5040 in decimal) as input, we get these results.
Total used memory: 274bytes
Total instructions: 316509
Total cycles: 949662
Average cycles per instruction: 3.0004265
Cycle time: 14.794ns
Total execution time: 15.194744ms
Gate count: 1800
Device utilization summary:

Selected Device : 3sd1800afg676-4

Number of Slices:	398	out of	16640	2%
Number of Slice Flip Flops:	374	out of	33280	1%
Number of 4 input LUTs:	796	out of	33280	2%
Number of IOs:	147			
Number of bonded IOBs:	147	out of	519	28%
Number of BRAMs:	1	out of	84	1%
Number of GCLKs:	1	out of	24	4%

Journal

Group 2D

Milestone 1 Meeting Sunday, October 1, 2017
Location: F217
Agenda:

- General design idea of the our assembly code

Work Done:

- Have a general design of our assembly code

Our design for assembly code: We decided to build a 16-bit accumulator and load-store based processor. We chose this design since it has nice structure and can take advantage of both load-store and accumulator design. Then, we decided to include 4 accumulators in our design for data storing and few other registers: $0, $ra, and $sp, etc.

Milestone 1 Meeting Monday, October2, 2017
Location: F217
Agenda:

- Write assembly code for relprime
- Write machine code for relprime
- Improve our design

Work Done:

- Write assembly code for relprime
- Write machine code for relprime
- Add lra, sra and ja to our design

Work for the week:
Decide design we are going to use, decide call conventions and instructions we are going to use, write down description of the instructions, develop rules for translating assembly language instructions to machine language estimated 1 day. [All group members, done in 1].
Write assembly code and machine code for relprime and Euclid's algorithm. [All group members, done in 2].

Milestone2 Meeting Thursday, October 5, 2017
Location: CS Labs F217

Agenda:

- Redesign instruction set
- Redesign method call procedure

- Redesign registers

Work done

- Need to talk to Micah during 7th hour tomorrow(~12:30)
- Got instruction type, registers, and instructions set in stone

Work to be done for next time

- Finish revising design doc
- Redo euclids
- M2

We chose to change our design to a mix of accumulator and load-store. This allows for short instructions and all the functionality that we need.

Milestone2 Meeting Sunday, October 8, 2017
Location: CS Lab F225
Agenda:

- Start RTL
- Fix any remaining stuff from bad design document

Work Done

- Constructed most of the RTL
- Started to find groupings of instructions based on the RTL

We wrote out the full RTL for each instruction individually, then we began to group together sections that were common among instructions.

Built the most of the RTL[Caleb and Brian, done in 2], revise the test code of milestone 1 with new instructions [Kevin and Anderson, done in 2], type everything to doc and transfer the assembly code to machine code [Kaiyu, done in 1]

Milestone2 Meeting Monday, October 9, 2017
Location: CS Lab F225
Time: 4pm
Agenda:

- Finish grouping RTL
- Record RTL in design doc
- Record parts/controls in design doc
- Make RTL tests

Work Done:

- RTL groupings all found
- RTL recorded in design doc

We finished the groupings of RTL instructions and broke it down into eight categories. We chose to test our RTL by having group members that didn't design the RTL write some assembly code and translate it out into our RTL to see if they thought it made sense.

Finish all the RTLs and group them together, describe all the RTL in English and in a chart [Caleb and Brian, done in 2], write a piece of code in assembly and manually run them with the RTL instructions to test it [Kevin and Anderson, done in 2] check and revise all the codes and type codes to the doc [Kaiyu, done in 1]

Write Assembler and start documenting pseudo instructions [Caleb, assign for over fall break]

Milestone 3 Meeting Monday 10/16/17
Location: CSSE Lab F225
Time: 8pm
Agenda:

- Complete or nearly complete M3
- Fix any problems still unfixed from M2
- Caleb - continue working on assembler

Work Done:

- Completed prototype datapath
- Assembler almost working theoretically
- Design Document flaws from M2 fixed

Milestone 3 Meeting Tuesday 10/17/17
Location: O259
Time: Classtime
Agenda:

- Ask Micah questions
- Figure out plan for rest of M3

Work Done:

- Asked Micah questions

Work Assigned:
Write about datapath tests[Yuankai and Kaiyu, one day]
Revise component list[Caleb, one day]

Milestone 3 Meeting Tuesday 10/17/17
Location: CS Lab F225
Time: 4pm
Agenda:

- Draw digital datapath
- Finish datapath tests

Work Done:

- Write about datapath tests

Milestone 3 Meeting Wednesday 10/18/17
Location: O259
Time: Classtime

Agenda
- Write a table about each control signal in the design doc
- Update this journal with questions from milestone
- Finish digital datapath

Work Done:
- Draw datapath
- Complete table on control signals
- Updated journal to include design and testing decisions

Strategy for creating tests:
We first test every single component. We did an exhaustive test for every possible inputs. Then we test each instructions. For this one, we just test some main values and edge cases, because there are so many possibilities. We test from the easiest that only need two or three components to the more complicated ones.
Finally we use the test we wrote for RTL test to run it. If possible, we will use all the possible instruction inputs, which is from 0000 0000 0000 0000 to 1111 1111 1111 1111.

Errors during testing:
For addings, we are possible to meet an overflow. We may also jump to an invalid memory address. During branch or jump we may go to an address that is out of range. We think we need to throw exception at these cases.

Choice in architecture affected your datapath design and component specification:
We need to have a large mux at the data write, because we almost need to throw everything into the accumulator and work on that. We went about designing the whole processor by going cycle by cycle, starting with the two cycles shared amongst all the instructions. We then when across all the instructions and added to the datapath in order to facilitate for each instruction. We found that some instructions use very similar paths and some required a lot of separate work.

Milestone 4 Meeting: Monday 10/23/17
Location: O259
Time: Classtime
Agenda:
- Fix mistakes from last milestone
- Start on control state diagram
- Delegate tasks so that we don't have to all be in one place to work

Work Done

- Fixed Parts list and RTL table
- Started on Control Diagram
- Started on Control descriptions
- Started on fixing integration plan from last week

Milestone 4 Meeting: Monday 10/23/17
Location F225
Time: 6:40
Agenda
- Wrap up integration plan
- Finish changing control diagram due to change with order stuff

Work Done:
- Build most of the components, including all muxes and constants, 4 bit ze to 16 bit, 8 bit ze to 16 bit, 8 bit se to 16 bit, 16 bit register, 16 bit comparator(Anderson), register File (Brian), ALU(Kevin)

Milestone 4 Meeting: Tuesday 10/24/17
Location: F225
Time: 9:30
Agenda
- Finish control tests
- Work on parts

Work Done
- Design doc changes for M4 finished
- Parts worked on(some to completion)

Assembler having linker built in

During this milestone creating the control, our design process was as follows. We took each cycle from the RTL and drew out what it corresponded to in the datapath. We then set the control signals needed for the processor to take that path. We did this for each possible cycle to come up with our final finite state machine.

Milestone 5 Meeting:Monday 10/30/17
Location: O259
Time: Class time
Agenda
- Finish control tests
- Fix design document
- Work on assembling Datapath

Work Done
- Parts continued to update
- Document updated

Milestone 5 Meeting Monday 10/30/17
Location: F225
Time: 6:00
Agenda

- Finish fixing mistakes from previous milestone in design doc
- Finish up last components
- Continue putting data path together

Work Done

- Fixed all mistakes from past milestone
- Finished almost all parts
- Datapath has all parts that are finished

Meeting Tuesday, 10/31/17
Location: F225
Time: 6pm
Agenda

- Continue updating data path
- Finish remaining components
- Add information regarding datapath and tests to design doc

Work Done

- Finished Register File and comparator
- Adding more parts to datapath
- Started and Finished Control Unit

Meeting Wednesday, 11/1/17
Location: F225
Time: 1pm
Agenda

- Finish data path
- Finish remaining components
- Finish integration test
- Finish system test

Work Done

- Finish data path
- Finish remaining components
- Finish integration test

Meeting Monday, 11/6/2017
Location: Class
Time: 9:55am
Agenda

- Fixing synthesis bugs

Work Done
- Fixed most synthesis bugs

Meeting Monday, 11/6/2017
Location: F225
Time: 6pm
Agenda
- Fix remaining synthesis problems
- Try to locate synthesis report

Work Done
- Fixed remain synthesis problems
- Could not locate synthesis reports

Meeting Tuesday, 11/7/2017
Location: F225
Time: 6pm
Agenda
- Get instructions to go through data path
- Add I/O
- Fix bugs in assembler

Work Done
- Got some instructions to go through data path with correct values
- Edited Euclid's Algorithm
- Added I/O to prepare to put on board
- Found small bugs in the assembler
- After bugs fixed in assembler, ready to implement Euclid’s.

Meeting Wednesday, 11/8/2017
Location: Class
Time: 9:55
Agenda
- Fix bugs in Assembler associated with addi
- Fix old Eculid’s algorithms
- Update full datapath test plan with some small test cases used

Work Done
- Fixed old Eculid’s algorithms
- Finished modifying assembler
- Update full datapath test plan

Meeting Wednesday, 11/8/2017
Location: F225
Time: 1pm
Agenda

- Run Euclid's algorithm

Work Done
- Fixed some bugs in the datapath

Meeting Thursday 11/9/2017
Location: F225
Time: 1am
Agenda
- Run Euclid's algorithm

Work Done
- Fixed bugs in the assembler about the location of labels
- Fixed bugs in assembly code about the sub instruction

Meeting Thursday 11/9/2017
Location: F225
Time: 10am
Agenda
- Add invert to the memory clock cycle

Work Done
- Successfully run Euclid's algorithm

Meeting Saturday 11/11/2017
Location: F225
Time: 3pm
Agenda
- Finish M6
- Work on the final report and final presentation

Work Done
- Finished M6

Meeting Sunday 11/12/2017
Location: F225
Time: 3pm
Agenda
- Keep working on the final report and final presentation
- Fix the problems existed in previous design document

Work Done
- Finished Final report and final presentation
- Finished design document

Test Results

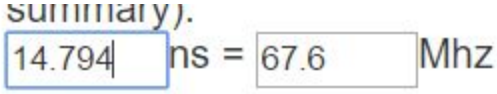
The total number of bytes required to store both Euclid's algorithm and relPrime as well as any memory variables or constants. 0x89->2*137=274

The total number instructions executed when relPrime is called with 0x13B0 (the result should be 0x000B using the algorithm specified in the project specifications). 316509

The total number of cycles required to execute `relPrime under the same conditions as Step 2. 949662

The average cycles per instruction based on the data collected in Steps 2 and 3. 949662/316509=3.0004265

The cycle time for your design (from the Xilinx Synthesis report – look for the Timing summary). 14.794ns



The total execution time for relPrime under the same conditions as Step 2. 15.194744ms

The gate count for your entire design (from the Xilinx Map report). This appears to have changed/is omitted in recent version. Extra credit for any group that finds a reasonable way to estimate the equivalent gate count from the data in the Xilinx reports.
About 1800. See video to show the video to show how we count.

The device utilization summary (from the Xilinx Synthesis report).

Device utilization summary:

Selected Device : 3sd1800afg676-4

Number of Slices:	398	out of	16640	2%
Number of Slice Flip Flops:	374	out of	33280	1%
Number of 4 input LUTs:	796	out of	33280	2%
Number of IOs:	147			
Number of bonded IOBs:	147	out of	519	28%

Number of BRAMs:	1	out of	84	1%
Number of GCLKs:	1	out of	24	4%