Stack Based Architecture (SBA-16)
Final Report

Team 1T: Bill Fang, Reed Phillips, Sarthak Suri, Michael Zhao

Section 1 Introduction

We have created an architecture that is based on the stack, hence the name Stack Based Architecture. It uses 16-bit data bus and address bus. In the project, we also made a processor that implements this instruction set. It is multi-cycle and the average CPI is 3.5, running at 70MHz.

Section 2 Overview

2.1 Instruction Set Design

The best aspect of a stack is that you can efficiently manipulate the top of it. The worst aspect of a stack is that you can only manipulate the top of it. Our instruction set was designed to deal with both of these aspects.

For efficiently manipulating the top of the stack, we have a wide array of instructions. There are arithmetic operations like add, which do arithmetic on the values on top of the stack. We can have a wide variety of these instructions because the opcodes do not have to specify where their operands come from. For more mundane operations like moving things around, we have operations like swap (which swaps the order of the top two entries on the stack) and kill (which removes the top entry on the stack). We also included no-pop variations for various instructions (differentiated by the -“n” suffix), which allows for operations to proceed without removing operands from the stack. This saves computation time as values that will be reused do not have to be duplicated for storage.

We have several ways to work around the limitations of a stack. The most prominent are the registers, which are primarily used as places to set data aside while another computation is done. Memory can also be used for this purpose. The method we probably use most often when programming is the pushs instruction, which copies values from deep in the stack to the top.

We split our instruction set based on required arguments and intended use:

- A-Type - Arithmetic type instructions that use the ALU
- I-Type - Instructions that use immediate
- B-Type - Branch Instructions
- J-Type - Jump Instructions
- R-Type - Instructions that use registers

2.2 Implementation

The instruction set is implemented using a multi-cycle datapath. All instructions take at least 3 cycles, and most instructions take 4 cycles. One of the concerns during our implementation

process is that pushing and popping from the stack may be expensive because memory operation is involved. We solved this by letting the stack pop as soon as it knows that it should pop (i.e. when opcode is first available). Therefore, popping from the stack happens in parallel with other execution. Pushing is faster than popping because the stack operation doesn't depend on memory results, so it does not need to start early.
A complete datapath diagram and RTL can be found in the design document in the appendix.

2.3 Testing

We had a systematic testing procedure which helped us to speed up the testing process. First, we write unit test programs using the instruction set. For each instruction, the unit test program runs it and uses "addi" and "bz" to determine if the result is correct. The program outputs decimal number 233 (0xE9) if all instructions passed the test, or it outputs an instruction-specific failure code. This helps us to determine which instruction failed the test.
When the tests are written, we first used the interpreter to test the tests. This helped us to make sure the tests are correct before we apply it to our processor.

Section 3 Unique Features

3.1 Infinite Stack

Our stack seamlessly expands into memory to provide the illusion of infinite stack space, limited only by the size of the main memory. This allows programs to store a large number of intermediate values and recurse as deep as they need to (again, restricted only by free memory).

3.2 Interpreter

Our interpreter will help users debug their assembly instruction program. Users can check their instruction syntax, and step through their program and see what is on the stack or the registers during the program before assembling the instructions and running the CPU.

3.3 Assembler

Our assembler is a versatile assembler, users can config the assembler to assemble1 different types of instructions and use it to assemble machine code for different types of CPU. And the assembled output can be formatted in the desired manner.

3.4 Fixed Point Support

As part of the project, we implemented fixed-point addition and multiplication in software. The system supports fixed-point numbers from -2 (inclusive) to 2 (exclusive). It has a precision up to 1/16384.

Section 4 Conclusion

In conclusion, SBA-16 is an instruction set capable of handling any 16-bit integer arithmetic algorithm. However, it also has its downside, primarily because it is based on stack. For example, storing intermediate results and be able to use them much later in the program. Because of these downsides that SBA-16 must handle, it looks like a hybrid of stack architecture and load-store architecture. These limitations of stack architecture made this project interesting to work on, but also makes the architecture unsuitable for commercial use.

Appendices

Stack Based Processor
Design Document

Team 1T: Bill Fang, Reed Phillips, Sarthak Suri, Michael Zhao

Versions

Version	Date	Change Log
7.0	Nov. 15	Added how to for debugging tool Added how to run realprime on FPGA
6.0	Nov. 13	Added Details for System Tests Added Performance Benchmark Added Debugging Tools section
5.0	Nov. 6	Added another input to ALUsrcB, namely SE<<1, for calculating branch target. Modified state S2 due to the above change. Added diagrams for integration plan Added Integration Plan for Central Control Added system tests (section 8) Fixed a minor bug in state transition diagram (an extra wire)
4.1	Oct. 29	Added State Transition Diagram to Section 7.1 Added State Control Signals to Section 7.2
4.0	Oct. 29	Modified RTL: ● POP should take 2 cycles. So now it is split into POP-memRead and POP-stkWrite. See Section 4.1.1 ● POPing 1 cycle early. This reduces 1 cycle for most instructions ● If PUSH involves writing ALUOut to stack, it can happen in the same cycle. See Section 4.1.3 ● Corrections to pushi, pui and pushs Added Section 6: Integration Plan and Tests to replace Section 5.4 ● More detailed planning and tests Added Section 5.3.3: Branch Decider Removed Section: Push-Pop Control Unit Updated Component Specification in Section 4.2 Specified Possible Values for Mux Control Added Section 7: Control State Transition ● This version contains a table that is incomplete
3.0	Oct. 23	Added Section 5
2.2	Oct. 19	Checked RTL plug in real example with pictures
2.1	Oct. 15	Checked RTL.
2.0	Oct. 14	Added Executive Summary Section Added table for each instruction type after the description

		Merged assembly, object code, and high-level language. Added Section 4 Added Versions
1.0	Oct. 8	Added Section 0-3

Table of Contents

Executive Summary

In this document we describe the design of our processor with a stack architecture. In this architecture, the program will store values and perform calculations on the stack instead of an array of temporary registers.

Section 0 Registers

We have two register files, one for the stack and the other for other values such as return address, assembler temporary, kernel register and temporary registers. We will have 8 registers in the first register file and our stack register file will have 32 registers. The figure below explains the non-stack register file.

Register	Index	Description
`$at`	0	Reserved For Assembler Use.
`$ra`	1	Return Address.
`$t0-$t3`	2-5	Temporaries.
`$mem`	6	Memory address base used by some instructions.
`$k0`	7	Reserved For Kernel Use.

Of those 8 registers above, only $k0 and $at are not supposed to be used by programmers. The stack registers are invisible to the programmer as well and should only be accessed via stack instructions. Furthermore, there are two more registers PC and SP that are for internal use only. PC (program counter) stores the address for the next instruction and SP (stack pointer) points to the spill-over portion of the stack in memory.

Section 1 Instructions

We tried to design the instruction set as small as possible while having the maximized functionality. For example, we decided to include 4 variants for most arithmetic (A-Type) instructions and they either allow the programmer to turn on popping or use immediate values.

1.1 Instruction Formats

We have 5 formats in our instruction set: A-Type, I-Type, B-Type, J-Type and R-Type. Each of them use a different address mode. The table below lists the layout of the bits and address mode operations for each instruction format.

	Layout	**Address Mode**	**Operations**
A	0 | 0 | 0 | 0 | 0 | NP | FUNC(10)	N/A	N/A
I	opcode(5) | NP | IMM(10)	Immediate	SignExt(IMM)={6'IMM[9],IMM} ZeroExt(IMM)={6'b0, IMM} ZeroPad(IMM)={IMM, 6'bo}
B	opcode(5) | NP | IMM(10)	Relative	PCBase(OFF)=PC+2+2*SignExt(OFF) MemBase(OFF)=$mem+2*SignExt(OFF)
J	opcode(5) | ADDR(11)	Pseudo-direct	JpAddr(ADDR)={PC[15-12],ADDR,0}
R	opcode(5) | NP | *Unused* | REG(3)	Direct	N/A

1.1.1 A-Type Instructions

A-Type (Arithmetic) instructions share the same opcode, namely zero, and use the FUNC field to determine what to do. All A-Type instructions get their operands from the stack. The NP field stands for "No-Pop" and determines if the operands should be popped off the stack.

Instr	**Operation/Description**	**NP**	**FUNC**
add	STK(1) = STK(0) + STK(1); SP--; **Add:** Pop the top two elements of the stack and push the sum back.	0	0
addn	SP++; STK(0) = STK(1) + STK(2);	1	0

	Add Nopop: Push the sum of the top two elements on the stack.		
`sub`	`STK(1) = STK(0) - STK(1); SP--;` **Sub:** Pop the top two elements of the stack and push the result of first element minus the second element.	`0`	`1`
`subn`	`SP++; STK(0) = STK(1) - STK(2);` **Sub Nopop:** Push the result of first element minus the second element.	`1`	`1`
`bus`	`STK(1) = STK(1) - STK(0); SP--;` **Inverse Sub:** Pop the top two elements of the stack and push the result of second element minus the first element.	`0`	`2`
`busn`	`SP++; STK(0) = STK(2) - STK(1);` **Inverse Sub Nopop**: Push the result of second element minus the first element.	`1`	`2`
`and`	`STK(1) = STK(0) & STK(1); SP--;` **And**: Pop the top two elements of the stack and push the bit-wise AND result back.	`0`	`3`
`andn`	`SP++; STK(0) = STK(1) & STK(2);` **And Nopop**: Push the bit-wise AND result to the stack.	`1`	`3`
`or`	`STK(1) = STK(0) | STK(1); SP--;` **Or**: Pop the top two elements of the stack and push the bit-wise OR result back.	`0`	`4`
`orn`	`SP++; STK(0) = STK(1) | STK(2);` **Or Nopop**: Push the bit-wise OR of the top two elements to the stack.	`1`	`4`
`slt`	`STK(1) = (STK(0) < STK(1))? 1 : 0; SP--;` **Set Less Than**: Pop the top two elements of the stack and push 1 if first element is less than the second one, otherwise push 0 to the stack.	`0`	`5`
`sltn`	`SP++; STK(0) = (STK(1) < STK(2))? 1 : 0;` **Set Less Than Nopop**: Push 1 if first element is less than the second one, otherwise push 0 to the stack.	`1`	`5`
`sll`	`STK(1) = STK(0) << STK(1); SP--;` **Shift Left Logical**: Pop the top two elements of the stack and push the result of first element shifted left by the value of the second element to the stack, carrying in zeros.	`0`	`6`
`slln`	`SP++; STK(0) = STK(1) << STK(2);` **Shift Left Logical Nopop**: Push the result of first element shifted left by the value of the second element to the stack, , carrying in zeros.	`1`	`6`
`srl`	`STK(1) = STK(0) >>> STK(1); SP--;` **Shift Right Logical**: Pop the top two elements of the stack and push the result of first element shifted right by the value of the second element to the stack, , carrying in zeros.	`0`	`7`

`srln`	`SP++; STK(0) = STK(1) >>> STK(2);` **Shift Right Logical Nopop**: Push the result of first element shifted right by the value of the second element to the stack, carrying in zeros.	`1`	`7`
`sra`	`STK(1) = STK(0) >> STK(1); SP--;` **Shift Right Arithmetic**: Pop the top two elements of the stack and push the result of first element shifted right by the value of the second element to the stack, carrying in the most significant bit.	`0`	`8`
`sran`	`SP++; STK(0) = STK(1) >> STK(2);` **Shift Right Arithmetic Nopop**: Push the result of first element shifted right by the value of the second element to the stack, carrying in the most significant bit.	`1`	`8`

1.1.2 I-Type Instructions

I-Type (Immediate) instructions are similar in layout to A-Type instructions, but they use the last 10 bits as an immediate. The 10-bit immediate will be zero- or sign-extended depending on the opcodes, which are different for each I-Type instruction. The NP field functions the same as A-Type instructions.

Instr	**Operation/Description**	**NP**	**OP**
`addi IMM`	`STK(0) = STK(0) + SignExt(IMM);` **Add Immediate**: Pop the top element of the stack and push back the sum of the top element and the immediate.	`0`	`1`
`addin IMM`	`SP++; STK(0) = STK(1) + SignExt(IMM);` **Add Immediate Nopop**: Push the sum of the top element and the immediate.	`1`	`1`
`andi IMM`	`STK(0) = STK(0) & ZeroExt(IMM);` **And Immediate**: Pop the top element of the stack push the bit-wise AND result of the top element and the immediate.	`0`	`2`
`andin IMM`	`SP++; STK(0) = STK(1) & ZeroExt(IMM);` **And Immediate Nopop**: Push the bit-wise AND result of the top element and the immediate.	`1`	`2`
`ori IMM`	`STK(0) = STK(0) | ZeroExt(IMM);` **Or Immediate**: Pop the top element of the stack push the bit-wise OR result of the top element and the immediate.	`0`	`3`
`orin IMM`	`SP++; STK(0) = STK(1) | ZeroExt(IMM);` **Or Immediate Nopop**: Push the bit-wise OR result of the top element and the immediate.	`1`	`3`
`slti IMM`	`STK(0) = (STK(0) < SignExt(IMM))? 1 : 0;` **Set Less Than Immediate**: Pop the top element of the stack and push 1	`0`	`4`

	if the element is less than the sign-extended immediate, otherwise push 0 to the stack.		
`sltin IMM`	`SP++; STK(0) = (STK(1) < SignExt(IMM))? 1 : 0;` **Set Less Than Immediate Nopop**: Push 1 if the top element is less than the sign-extended immediate, otherwise push 0 to the stack.	`1`	`4`
`slli IMM`	`STK(0) = STK(0) << ZeroExt(IMM);` **Shift Left Logical Immediate**: Pop the top element of the stack and push the result of first element shifted left by the value of the sign-extended immediate to the stack, carrying in zeroes	`0`	`5`
`sllin IMM`	`SP++; STK(0) = STK(1) << ZeroExt(IMM);` **Shift Left Logical Immediate Nopop**: Push the result of first element shifted left by the value of the sign-extended immediate to the stack, carrying in zeroes	`1`	`5`
`srli IMM`	`STK(0) = STK(0) >>> ZeroExt(IMM);` **Shift Right Logical Immediate**: Pop the top element of the stack and push the result of first element shifted right by the value of the sign-extended immediate to the stack, carrying in zeroes	`0`	`6`
`srlin IMM`	`SP++; STK(0) = STK(1) >>> ZeroExt(IMM);` **Shift Right Logical Immediate Nopop**: Push the result of first element shifted right by the value of the sign-extended immediate to the stack, carrying in zeroes	`1`	`6`
`srai IMM`	`STK(0) = STK(0) >> ZeroExt(IMM);` **Shift Right Arithmetic Immediate**: Pop the top element of the stack and push the result of first element shifted right by the value of the sign-extended immediate to the stack, carrying in the most significant bit	`0`	`7`
`srain IMM`	`SP++; STK(0) = STK(1) >> ZeroExt(IMM);` **Shift Right Arithmetic Immediate Nopop**: Push the result of first element shifted right by the value of the sign-extended immediate to the stack, carrying in the most significant bit	`1`	`7`
`pushi IMM`	`SP++; STK(0)=SignExt(IMM)` **Push Immediate**: Pushes the sign-extended immediate to the stack.	`1`	`17`
`pui IMM`	`SP++; STK(0)=ZeroPad(IMM);` **Push Upper Immediate**: Pushes the immediate, left-shifted 6, to the stack.	`1`	`18`
`pushs IMM`	`SP++; STK(0)=STK(ZeroExt(IMM[4-0])+1);` **Push Stack:** Puts a copy of the (immediate)th item in the stack on top. Can only access the part of the stack in registers, the first 32 items.	`1`	`19`

1.1.3 B-Type Instructions

B-Type instructions are similar in layout to A- and I-Type instructions. The last 10 bits are used as offsets. Although the name comes from “branch type”, the offset can either form a branch address or memory address depending on the instruction. The NP field is the same as in A- and I-Types.

Instr	Operation/Description	NP	OP
`bz Label`	`if(STK(0)==0) { PC=PCBase(OFF); } SP--;` **Branch Zero**: Pop the top off. If it is zero, branch to Label.	`0`	`8`
`bzn Label`	`if(STK(0)==0) { PC=PCBase(OFF); }` **Branch Zero Nopop**: If the top of the stack is zero, branch to Label.	`1`	`8`
`bnz Label`	`if(STK(0)!=0) { PC=PCBase(OFF); } SP--;` **Branch Not Zero**: Pop the top off. If it is not zero, branch to Label.	`0`	`9`
`bnzn Label`	`if(STK(0)!=0) { PC=PCBase(OFF); }` **Branch Not Zero Nopop**: If the top of the stack is not zero, branch to Label.	`1`	`9`
`pushm OFF`	`SP++; STK(0)=Mem[MemBase(OFF)];` **Push Memory**: Load the memory at OFF words from $mem and push it on stack	`1`	`13`
`popm OFF`	`Mem[MemBase(OFF)]=STK(0); SP--;` **Pop Memory**: Pop the top off the stack and store the word to memory at OFF words from $mem	`0`	`14`
`peekm OFF`	`Mem[MemBase(OFF)]=STK(0);` **Peek Memory**: Store the top of the stack to memory at OFF words from $mem	`1`	`14`

1.1.4 J-Type Instructions

J-Type instructions are used to move the PC. It has an 11-bit field ADDR which is used to form a 16-bit jump address by appending the 4 most significant bits of the PC on the left and a zero bit on the right. J-Type instructions do not interact with the stack, so there is no NP field.

Instr	Operation/Description	OP
`j Label`	`PC=JpAddr(ADDR);` **Jump**: Unconditional branch to Label.	`10`
`jal Label`	`$ra=PC+2; PC=JpAddr(ADDR);` **Jump And Link**: Set return address to next instruction and branch to Label. Used in procedure calls.	`11`

1.1.5 R-Type Instructions

R-Type (register) instructions can interact with both the stack and the registers. The last 3 bits are used to specify the register number. The NP field has the same function as the ones in other types. There are also some R-Types instructions that do not use the register field.

Instr	Operation/Description	NP	OP
`jr REG`	`PC=R[REG];` **Jump Register**: Unconditional branch to the address stored in a register.	`1`	`12`
`pushr REG`	`SP++; STK(0)=R[REG];` **Push Register**: Pushes the value in a register to the stack.	`1`	`15`
`popr REG`	`R[REG]=STK(0); SP--;` **Pop Register**: Pops the item on top of the stack to a register.	`0`	`16`
`peekr REG`	`R[REG]=STK(0);` **Peek Register**: Copies the item on top of the stack to a register.	`1`	`16`
`swap`	`STK(0)<=>STK(1);` **Swap**: Switch the positions of the top two items on the stack.	`1`	`20`
`kill`	`SP--;` **Kill**: Remove the top item from the stack.	`0`	`21`

1.2 Machine Language Translation

Instructions are translated to machine language according to the format. The first 5 bits will always be the opcode. We currently randomly assigned number for opcode and funct. We are planning on change it later since we want to make the opcode can be easily implemented to the hardware implementation. Examples can be found in section 3.

1.2.1 A-Type and R-Type

For these two types, the NP field is set to 1 if the instruction requires a pop from the stack. For A-Types, the FUNC field is specified in Section 1.1.1. For R-Types, a 3-bit number representing the register index is put into the last 3 bits of the instruction.

1.2.2 I-Type

The immediate supplied to each I-Type instruction must fit in 10 bits and it will be put at the last 10 bits of the instruction. The NP field is set the same way in A- and R-Types.

1.3.3 B-Type

B-Type instructions have a label to jump to. The address of the label is first fetched, then PC+2 is subtracted from it. The result is divided by 2 and put into the last 10 bits of the instruction Similar to I-Types, this result must fit within 10 bits for the instruction to be valid. The NP field is set the same way as in A-, R- and I-Types.

1.3.4 J-Type

J-Type instructions also jump to a label. The addr field will contain the 1-11th bits (the least significant bit is the 0th bit) of the label address. The label must have the same first 4 bits as the PC.

Section 2 Procedure Call Convention

To call a procedure, the caller must put the arguments on top of the stack and jal to the callee's address. The callee is not supposed to access the part of stack below the arguments. When the callee is finished, it will jump back with the return value(s) in place of the arguments on the stack. The caller is responsible for saving the values of all registers it would like to use later, either on the stack or to the memory. Note that we don't have a stack for immediate variables in the memory. It is recommended for the programmer to use the register stack to save all temporaries. The register stack itself will eventually spill over to memory.

Section 3 Assembly Implementation

In this section there are some sample programs and fragments implemented using our instruction set. We also shows the result object code and the corresponding addresses.

Addr	Obj	Assembly	High Level Language
			`// Find m that is relatively prime to n.`
		`relPrime:`	
`0x00`	`01111 1 00000 00001`	`  pushr $ra  # Save $ra`	`int relPrime(int n)`
`0x02`	`10100 1 00000 00000`	`  swap`	`{`
`0x04`	`10001 1 00000 00010`	`  pushi 2    # int m = 2`	`  int m = 2;`
		`While:`	`  while (gcd(n, m) != 1) {`
`0x06`	`10011 1 00000 00001`	`  pushs 1    # copy m and n`	`// n is the input from the outside world`
`0x08`	`10011 1 00000 00001`	`  pushs 1`	
`0x0a`	`01011 1 00000 01111`	`  jal GCD    # compute GCD`	
`0x0c`	`00001 0 11111 11111`	`  addi -1`	
`0x0e`	`01000 0 00000 00010`	`  bz Return  # if it's 1,`	
		`#   return`	
`0x10`	`00001 0 00000 00001`	`  addi 1     # otherwise,`	`    m = m + 1;`
		`#   m = m + 1`	`  }`
`0x12`	`01010 1 00000 00011`	`  j While`	
		`Return:`	
`0x14`	`10100 1 00000 00000`	`  swap`	
`0x16`	`10101 0 00000 00000`	`  kill       # remove n`	
`0x18`	`10100 1 00000 00000`	`  swap`	
`0x1a`	`10000 0 00000 00001`	`  popr $ra   # restore ra`	`  return m;`
`0x1c`	`01100 1 00000 00001`	`  jr $ra     # return m`	`}`
			`// The following method determines the Greatest Common Divisor of a and b`
			`// using Euclid's algorithm.`
		`GCD:`	
`0x1e`	`01000 1 00000 01100`	`  bzn Done   # if a = 0 go to`	`int gcd(int a, int b)`
		`#   done`	`{`
`0x20`	`10100 1 00000 00000`	`  swap       # swap b on top`	`  if (a == 0) {`
		`#   of a`	`    return b;`
			`  }`
		`Loop:`	
`0x22`	`01000 1 00000 01010`	`  bzn Done   # if b = 0 go to`	`  while (b != 0) {`
		`#   done`	
`0x24`	`00000 1 00000 00101`	`  sltn       # Put 1 on top`	`    if (a > b) {`
		`#   if b<a`	`      a = a - b;`
`0x26`	`01000 0 00000 00101`	`  bz  Else   # if b>=a go to`	`    }`
		`#   else`	
`0x28`	`10000 1 00000 00010`	`  peekr $t0  # store b in $t0`	
`0x2a`	`00000 0 00000 00010`	`  bus        # replace a, b`	

```
                                  #   with a-b
0x2c  01111 1 00000 00010   pushr $t0  # put b on top
                                  #   of stack
0x2e  01010 1 00000 10001   j Loop
                          Else:
0x30  00000 1 00000 00001   subn       # put b-a on top      else {
                                  #   of stack              b = b - a;
0x32  10100 1 00000 00000   swap       # put b on top        }
0x34  10101 0 00000 00000   kill       # remove b on       }
                                  #   top
0x36  01010 1 00000 10001   j Loop
                          Done:
0x38  10101 0 00000 00000   kill
0x3a  01100 1 00000 00001   jr $ra                         return a;
                                                          }
```

Sample For Loop

Addr	Obj	Assembly	High Level Language
			//Simple for loop
0x00	10001 1 00000 00001	pushi 1 # int x = 1	int x = 1;
0x02	10001 1 00000 00000	pushi 0 # int i = 0	for (int i = 0; i < 10;
		Loop:	i++) {
0x04	00100 1 00000 01010	sltin 10 # if i >= 10...	
0x06	01000 0 00000 00101	bz End # end loop	
0x08	10100 1 00000 00000	swap # put x on top	
0x0a	00001 0 00000 00010	addi 2 # x = x + 2	x = x + 2;
0x0c	10100 1 00000 00000	swap # put i on top	
0x0e	00001 0 00000 00001	addi 1 # i++	
0x10	01010 0 00000 00010	j Loop	
		End:	}
0x12	10101 0 00000 00000	kill # i has served its	
		# purpose	

More Complex For Loop

Addr	Obj	Assembly	High Level Language
			//More complex for loop; calculates Fibonacci numbers
0x00	10001 1 00000 00001	pushi 1 # int x = 1	int x = 1;
0x02	10001 1 00000 00001	pushi 1 # int y = 1	int y = 1;
0x04	10001 1 00000 00000	pushi 0 # int i = 0	
		Loop:	for (int i = 0; i < 10;
0x06	00100 1 00000 01010	sltin 10 # if i >= 10…	i++) {
0x08	01001 0 00000 01001	bnz End # end loop	
0x0a	10000 0 00000 00010	popr $t0 # store i	

```
0x0c   10011 1 00000 00001   pushs 1    # push a copy of
                             #   x to the
                             #   stack          y = x + y;
0x0e   00000 0 00000 00000   add        # y = x + y
0x10   10100 1 00000 00000   swap
0x12   10011 1 00000 00001   pushs 1    # push a copy of
                             #   y to the       x = y + x;
                             #   stack
0x14   00000 0 00000 00000   add
0x16   10100 1 00000 00000   swap
0x18   01111 1 00000 00010   pushr $t0  # i is back
0x1a   00001 0 00000 00001   addi 1     # i++;           }
0x1c   01010 0 00000 00011   j Loop
                           End:
0x1e   10101 0 00000 00000   kill       # i has served
                             #   its purpose
```

Sample Procedure Call

Addr	Obj	Assembly	High Level Language
		`#Suppose we want to save $t0, #$ra, and $mem; put them on #the stack`	`// Sample procedure call`
0x00	01111 1 00000 00010	`pushr $t0`	`int i = doStuff(x, y);`
0x02	01111 1 00000 00001	`pushr $ra`	`//doStuff is at address 0x0100`
0x04	01111 1 00000 00110	`pushr $mem`	
		`#Get arguments on top of the #stack. Suppose, worst-case #scenario, they were on top #before we backed registers #up.`	
0x06	10011 1 00000 00100	`pushs 4  #put y on top…`	
0x08	10011 1 00000 00100	`pushs 4  #and put an x on it`	
0x0a	01011 0 01000 00000	`jal doStuff`	
		`#Now i is on top of the #stack, followed by the #backed-up registers.`	
0x0c	10000 0 00000 00011	`popr $t1  # set it aside`	
0x0e	10000 0 00000 00110	`popr $mem`	
0x10	10000 0 00000 00001	`popr $ra`	
0x12	10000 0 00000 00010	`popr $t0`	
		`# restore our registers #Now i is in $t1 and the #stack is as it was before #the procedure call.`	

Sample fetch from memory

Addr	Obj	Assembly	High Level Language
	//assume $mem = 0x0040	# If $mem is already in a # useful place (e.g. a data # bank), then you can just # load it:	// Sample fetch data from memory
0x00	01101 1 00000 00000	pushm 0 # $mem stores ptr	int i = *ptr; //ptr = 0x0040
		# Otherwise, the value of ptr # has to be loaded:	
0x02	10010 1 11111 11111	pui -1 # highest 10 bits	int j = *ptr2;
0x04	00011 0 00001 11111	ori 63 # low 6 bits	//ptr2 = 0xffff
0x06	10000 0 00000 00110	popr $mem	
0x08	01111 1 00000 00000	pushm 0	

Sample Switch Statement

Addr	Obj	Assembly	High Level Language
0x00	10001 1 00000 00111	pushi 7 # int i = 7	// Sample switch statement
0x02	10011 1 00000 00000	pushs 0 # copy i so it # doesn't go away	int i = 7;
0x04	00001 0 11111 11111	addi -1	switch (i) {
0x06	01001 1 00000 00010	bnzn Check_2 # case 1	case 1:
0x08	10101 0 00000 00000	kill # i's still on top # so remove it	
		# Code 1 Here #	// Code 1
0x0a	01010 0 00000 01101	j Done # break	break;
		Check_2:	case 2:
0x0c	00001 0 11111 11111	addi -1	
0x0e	01001 1 00000 00010	bnzn Check_7 # case 2	
0x10	10101 0 00000 00000	kill	
		# Code 2 Here #	// Code 2
0x12	01010 0 00000 01101	j Done	break;
		Check_7:	case 7:
0x14	00001 0 11111 11011	addi -5	
0x16	01001 0 00000 00001	bnz Default # case 7	
		# Code 7 Here #	// Code 7
		# no need to remove i since # the bnz just removed it	break;
0x18	01010 0 00000 01101	j Done	
		Default:	default:
		# Code default #	// Code default
		Done:	}
0x1a			

Section 4 Register Transfer Language

4.1 List of RTL

4.1.1 PUSH and POP

To save space, we will use the following shorthands POP and PUSH in Section 4.1.3. Notice that POP takes 2 cycles to complete because it has to wait for the memory to read before it can write to the stack. PUSH, on the other hand, will take 2 cycles only if pushing from memory or register. Pushing the ALUOut on to the stack can happen in the same cycle.

POP-memRead	STK[0]=STK[1], STK[1]=STK[2]... STK[30]=STK[31] MDR=Mem[SP+2] SP = SP + 2
POP-stkWrite	STK[31]=MDR
PUSH	STK[31]=STK[30],STK[30]=STK[29]...STK[1]=STK[0] Mem[SP]=old STK[31] SP = SP - 2

4.1.2 Cycle 1 & 2: Fetch and Decode

Since all instructions have the same fetch and decode cycles, we had separated these two stages from the rest of the RTL. These are the first two cycles to process the instruction.

	Cycle 1 - Fetch	**Cycle 2 - Decode**
All Instructions	newPC = PC + 2 PC = newPC IR = Mem[PC]	M=Reg[6] ALUOUT=SE(IR[9-0])<<1+newPC A=STK[0] B=STK[1] R=Reg[IR[2-0]] STKR = STK[IR[4-0]]

4.1.3 Cycles 3-5

Since our processor is multi-cycle, instructions may take different number of cycles to finish. In our design, 7 instructions (bzn, bnzn, j, jal, jr, peekr, swap) takes 3 cycles; pushm takes 5 cycles; the rest of the instructions all take 4 cycles.

	Cycle 3	**Cycle 4**	**Cycle 5**
add	ALUOut = A + B POP-memRead	STK[0]=ALUOut POP-stkWrite	
addn	ALUOut = A + B	PUSH STK[0]=ALUOut	
sub	ALUOut = A - B POP-memRead	STK[0]=ALUOut POP-stkWrite	
subn	ALUOut = A - B	PUSH STK[0]=ALUOut	
bus	ALUOut = B - A POP-memRead	STK[0]=ALUOut POP-stkWrite	
busn	ALUOut = B - A	PUSH STK[0]=ALUOut	
and	ALUOut = A & B POP-memRead	STK[0]=ALUOut POP-stkWrite	
andn	ALUOut = A & B	PUSH STK[0]=ALUOut	
or	ALUOut = A | B POP-memRead	STK[0]=ALUOut POP-stkWrite	
orn	ALUOut = A | B	PUSH STK[0]=ALUOut	
slt	ALUOut = A slt B POP-memRead	STK[0]=ALUOut POP-stkWrite	
sltn	ALUOut = A slt B	PUSH STK[0]=ALUOut	
sll	ALUOut = A << B POP-memRead	STK[0]=ALUOut POP-stkWrite	
slln	ALUOut = A << B	PUSH STK[0]=ALUOut	
srl	ALUOut = A >>> B POP-memRead	STK[0]=ALUOut POP-stkWrite	
srln	ALUOut = A >>> B	PUSH	

		STK[0]=ALUOut	
sra	ALUOut = A >> B POP-memRead	STK[0]=ALUOut POP-stkWrite	
sran	ALUOut = A >> B	PUSH STK[0]=ALUOUT	
addi	ALUOut = A + SE(IR[9-0])	STK[0]=ALUOut	
addin		PUSH STK[0]=ALUOut	
slti	ALUOut = A slt SE(IR[9-0])	STK[0]=ALUOut	
sltin		PUSH STK[0]=ALUOut	
pushi	ALUOut = 0 + SE(IR[9-0])	PUSH STK[0]=ALUOut	
pui	ALUOut = 0 + ZP(IR[9-0])	PUSH STK[0]=ALUOut	
pushs	ALUOut = 0 + STKR	PUSH STK[0]=ALUOut	
andi	ALUOut = A & ZE(IR[9-0])	STK[0]=ALUOut	
andin		PUSH STK[0]=ALUOut	
ori	ALUOut = A | ZE(IR[9-0])	STK[0]=ALUOut	
orin		PUSH STK[0]=ALUOut	
slli	ALUOut = A << ZE(IR[9-0])	STK[0]=ALUOut	
sllin		PUSH STK[0]=ALUOut	
srli	ALUOut = A >>> ZE(IR[9-0])	STK[0]=ALUOut	
srlin		PUSH STK[0]=ALUOut	
srai	ALUOut = A >> ZE(IR[9-0])	STK[0]=ALUOut	
srain		PUSH	

		STK[0]=ALUOut	
bz	if (A==0) PC=ALUOut POP-memRead	POP-stkWrite	
bzn	if (A==0) PC=ALUOut		
bnz	if (A!=0) PC=ALUOut POP-memRead	POP-stkWrite	
bnzn	if (A!=0) PC=ALUOut		
j	PC=newPC[15-12] || IR[10-0] || 0		
jal	PC=newPC[15-12] || IR[10-0] || 0 Reg[1]=newPC		
jr	PC=R		
pushm	ALUOut = M + SE(IR[9-0])<<1	PUSH MDR=Mem[ALUOut]	STK[0]=MDR
peekm		Mem[ALUOut]=A	
popm	ALUOut = M + SE(IR[9-0])<<1 POP-memRead	Mem[ALUOut]=A POP-stkWrite	
pushr	ALUOut = 0 + R	PUSH STK[0]=ALUOut	
popr	Reg[IR[2-0]]=A POP-memRead	POP-stkWrite	
peekr	Reg[IR[2-0]]=A		
kill	POP-memRead	POP-stkWrite	
swap	STK[0]<=>STK[1]		

4.1.4 Checking the RTL

For checking the RTL, our method is checking with a systematic way. Going over by instruction type, checking for each instruction: Does the instruction do all it needs to do? Are we using the right instructions bits? Is the order of execution proper? Is there any conflicting on reading and writing with in the same cycle? Are there any syntax errors? Is the symbol consistent?

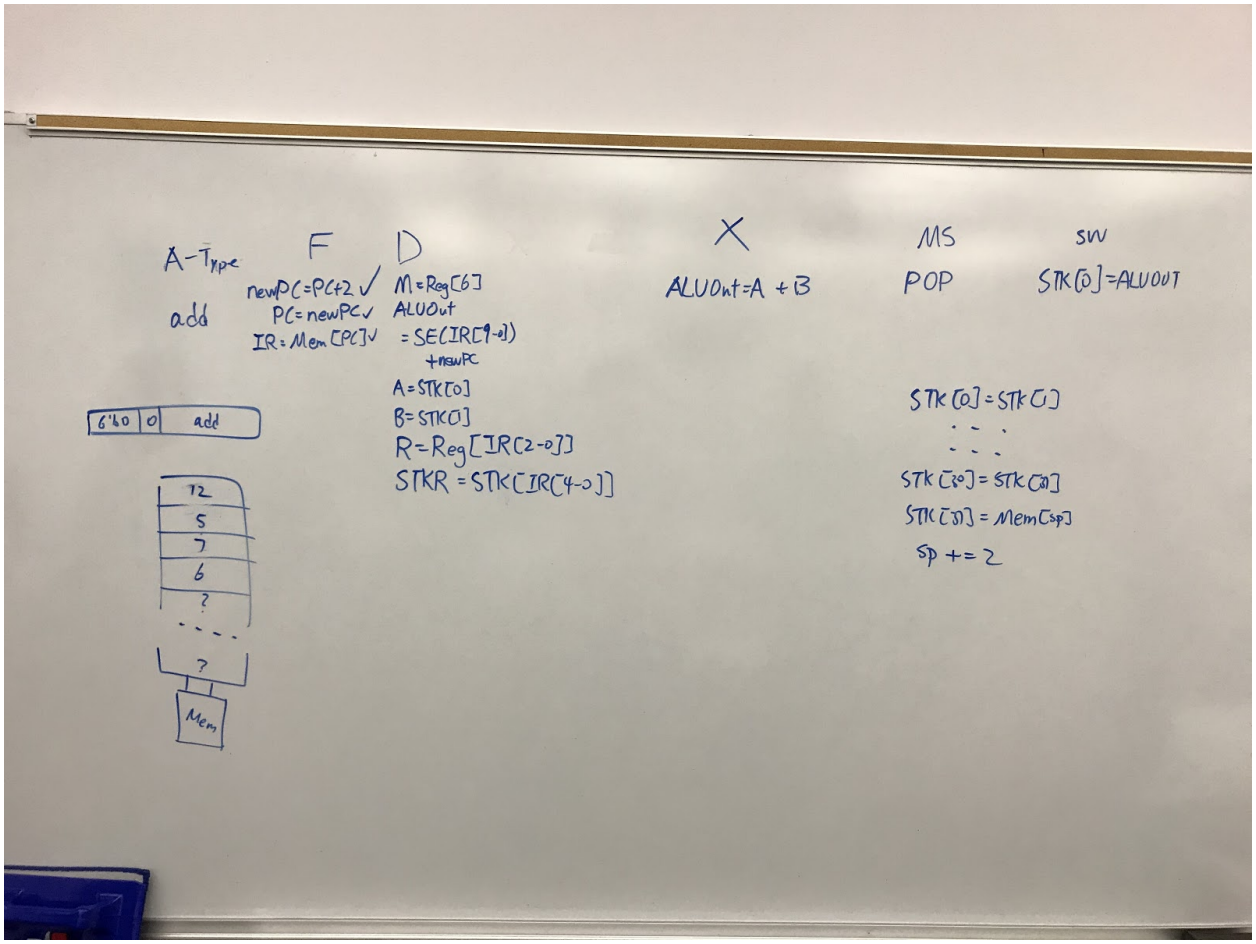


Figure 1: RTL Verification Snapshot

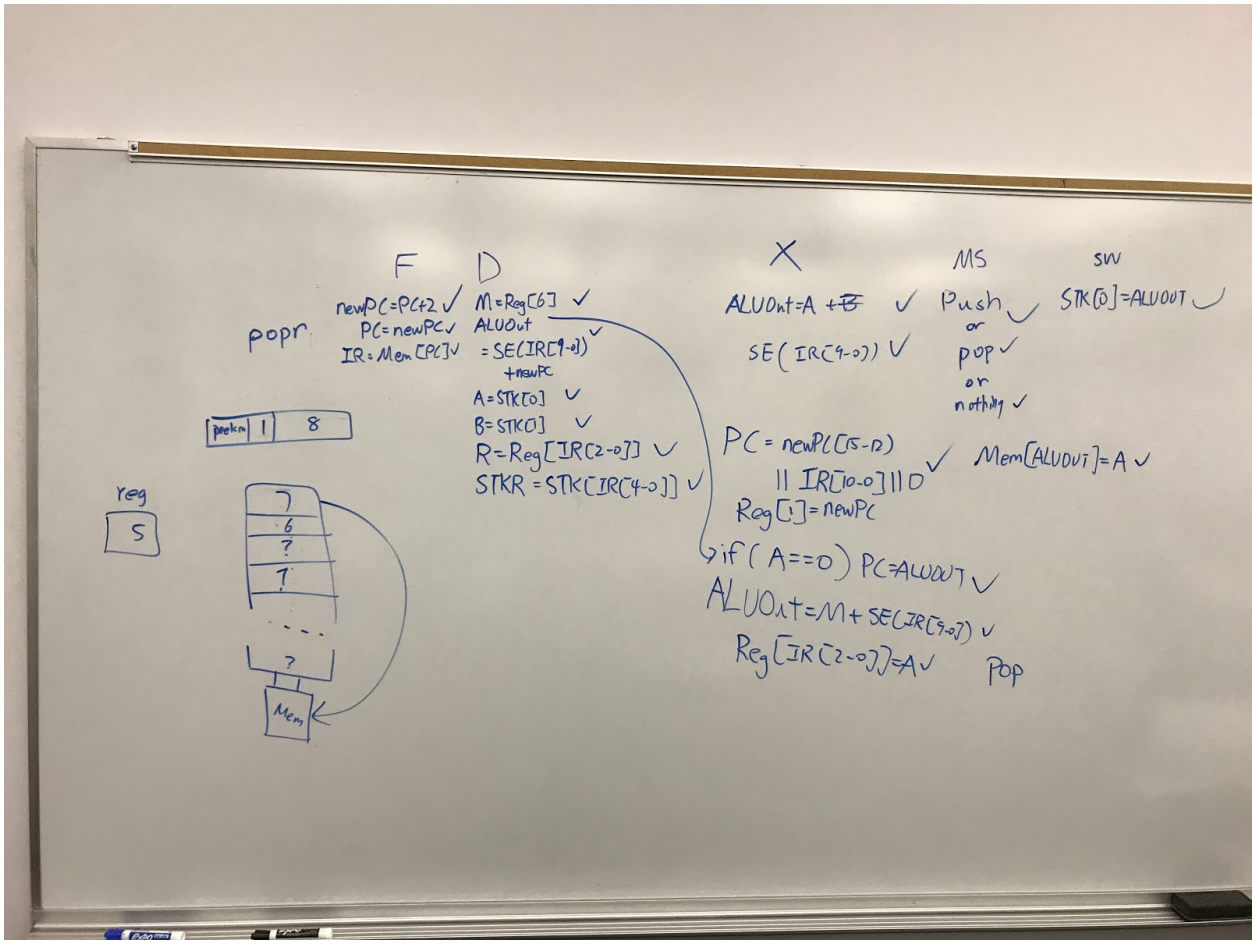


Figure 2: RTL Verification Snapshot

4.2 List of Components

In this section we discuss the building blocks of the datapath and which parts of the RTL each component implements.

4.2.1 Arithmetic Logic Unit

The Arithmetic Logic Unit, or ALU, will have two 16-bit input signals A and B, one control signal OP of 4 bits and 1 output signal: a 16-bit result

The ALU has 8 operations available: and, or, add, subtract, set less than, shift left logical, shift right logical and shift right arithmetic. Below is a table of OP and the corresponding operation

OP(3:0)	1000	1010	1011	0111	0101	0110	0010	0001	0000
Operation	<<	>>>	>>	slt	bus	sub	add	or	and

This ALU will implement 8 symbols in the RTL, namely +, -, & ,|, slt, <<, >>> and >>.

4.2.2 16-Bit Register

Registers are 16 flip-flops that store all the values within the CPU. They have one 16-bit input signal (writing value), one 16-bit output (value), and two control signals (clock and writing enabled)

When the clock signal rises and their writing enabled flag is set to 1, the value stored in the register is replaced by the writing value.

Registers are used to build the stack, the register file, PC, and miscellaneous signals within the CPU (ALUOut, SP, A, B, R, STKR, M, MDR).

4.2.3 Stack File

The stack file consists of 32 registers. There are two inputs: a new value (16 bits) and the item currently being read (5 bits). There are three 1-bit control signals: pop, push, and clock. There are seven outputs: STK[0] (16 bits), STK[1] (16 bits), the value currently being read (16 bits), the value to add to SP (16 bits), STK[31] (16 bits), and flags for when the stack wants to read or write from memory (1 bit each). The flags become control signals for the memory.

STK[0] is always the value in the first of the 32 registers. Similarly, STK[1] is always the value in the second register and STK[31] is always the value in the last register. The value currently being read is STK[item currently being read]. The value to add to SP is 2 if pop is set, -2 if push is set, and 0 if neither is set. The “want to read” flag is pop, and the “want to write” flag is push.

When the stack file receives a clock tick, different things happen if pop or push is set.
If pop is set, each register writes its value to the one below it. So STK[0] = STK[1], STK[1] = STK[2], and so on. Then, the new value (which should be Mem[SP]) is written into STK[31].

If push is set, then each register writes its value to the one above it. So STK[31] = STK[30], STK[30] = STK[29], and so on. STK[0] becomes 0. Note that the value in STK[31] should already have been written to memory.

The stack file implements any STK, POP, and PUSH in the RTL.

4.2.4 Register File

Register file is in charge of all the register read and write. It will have RegWrite enable signal (1 bit) to control writing of register file. It will also have the write address (16 bits) and write data

(16 bits). The Register file also have read address (16 bits) input and corresponding read data (16 bits) output for reading register data.

The only control signal RegWrite will allow the register file to change the register at the write address when RegWrite is set to 1. Otherwise, we can only read from registers.

The Register file will implement all the Reg symbols such as Reg[0], Reg[IR[2-0]].

4.2.5 Memory

The memory has 3 16-bit signals: writeData, writeAddr and readAddr. It has 2 control signals memWrite and memRead, and one 16-bit output signal. It also takes in clock signal.

When memWrite is 1, the memory will replace the data at writeAddr with writeData. When memRead is 1, the memory will output the data at readAddr. Write will happen on the rising edge of the clock cycle and read will be on the falling edge, so that the memory always writes before it reads.

The two control signals are independent. When they are both on, memory does both write and read; when they are both off, the memory doesn't do anything. This component implements Mem[writeAddr]=writeData and Mem[readAddr].

4.2.6 Sign Extender

The sign extender will take one 10-bit input immediate and has one 16-bit Sign Extended output. There is no control signal. This component implements the SE() symbol.

4.2.7 Zero Padder

The zero padder takes one 10-bit input immediate and produces a 16-bit output signal with the most significant 10 bits identical to the input followed by 6 zero bits. There is no control signal. This component implements the ZP() symbol.

4.2.8 Zero Extender

The zero extender takes one 10-bit input and has one 16-bit output. The output is always six bits of zeroes followed by the 10-bit input. This component implements the ZE() symbol.

4.2.9 Multiplexer

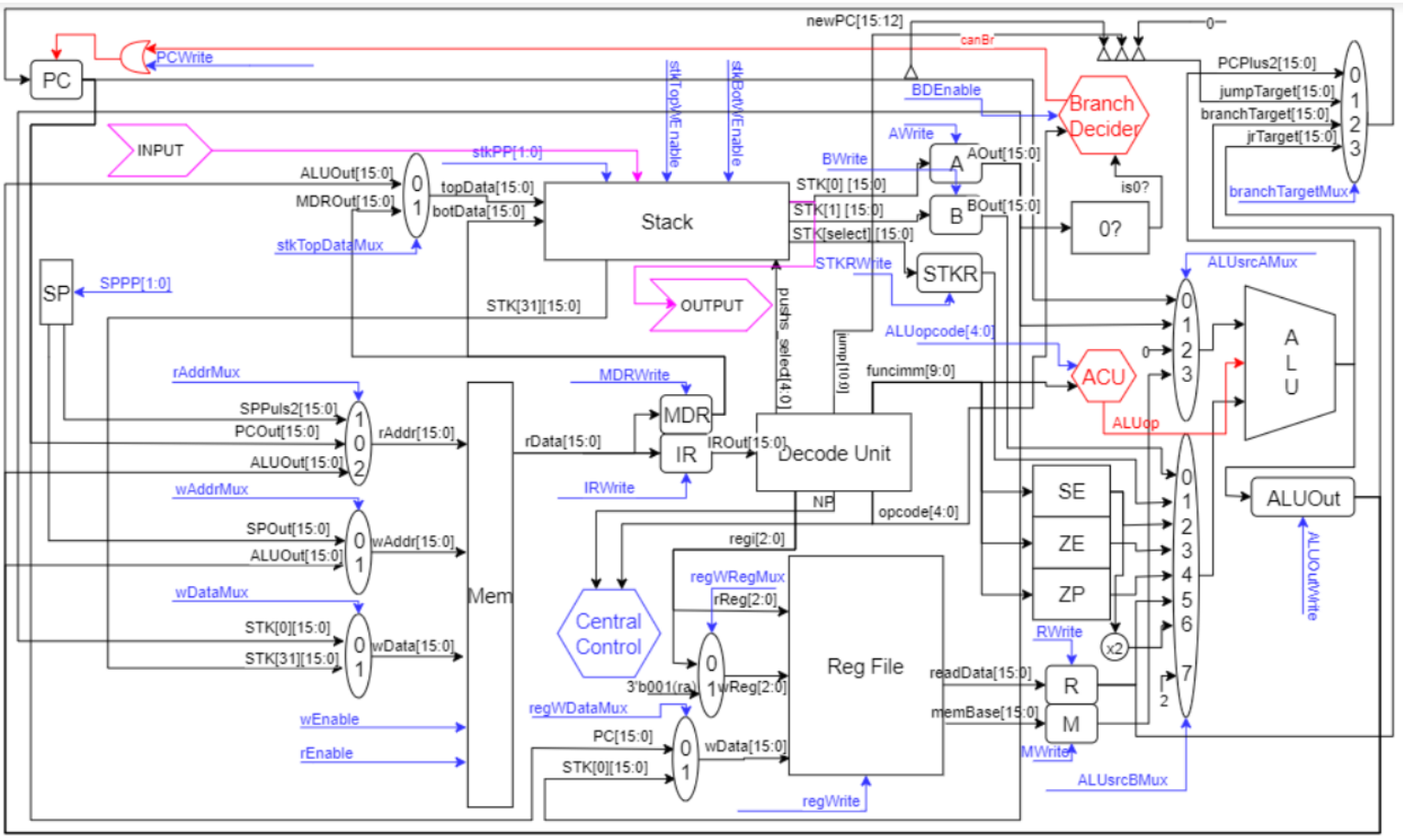
Multiplexers (muxes) have many inputs and one output. Both inputs and the output are 16-bit buses. There will be one control signal to decide which input to pick as the output. The size of the control signal depends on how many inputs the mux can take.

For example, for a 4-input mux, the control signal needs to be 2 bits, and the mux picks the first input if the control signal is 0; second input if the control is 1; third input if 2 and fourth input if 3.

Muxes are used to implement datapaths and other components. They don’t show up as symbols in RTL.

Section 5 Datapath

5.1 Datapath Diagram



5.2 Component Specification

5.2.1 Arithmetic Logic Unit

The ALU will be able to do and, or, add, subtract, set less than, shift left logical, shift right logical and shift right arithmetic operations with its inputs, including the two 16-bit input signals A and B, one control signal OP of 4 bits and 1 output signals: a 16-bit result.

The component will be implemented in Verilog using cases for different ALU opcodes. The ALU will be done with a 16-bit version directly instead of implementing 1-bit then extend it to the full 16-bit.

Below is a list of opcodes and the corresponding operations.

OP(3:0)	1000	1010	1011	0111	0101	0110	0010	0001	0000
Operation	<<	>>>	>>	slt	bus	sub	add	or	and

For unit testing, it will be exhaustive testing looping through different input combinations to test outputs for different opcodes. For shifting the test won't be exhaustive since we just need to prove its ability to shift 1- 16 bits.

5.2.2 Stack File

This component is a collection of registers that are connected into a stack. It takes five control signals: push, pop, topwe, bottomwe, and (5 bits) select. It takes two 16-bit inputs: topData and bottomData. It has four 16-bit outputs: STK[0], STK[1], STK[select], and STK[31].
The stack has an array of 32 registers. The zeroth is wired to STK[0], the first to STK[1], and the one indexed by the select input to STK[select]. STK[31] is the value of the thirty-first register. On the rising edge of the clock, the contents of the registers can be updated. If push (but not pop) is set, each stack register writes to the one above it, so STK[1] = STK[0], STK[2] = STK[1], and so on. Zero gets written to STK[0]. If pop (but not push) is set, each stack register writes to the one below it, so STK[0] = STK[1], STK[1] = STK[2], and so on. Zero gets written to STK[31]. If both pop and push are set, then the values in STK[0] and STK[1] get swapped. If topwe is set, then topData gets written to STK[0]. If bottomwe is set, then bottomData gets written to STK[31].

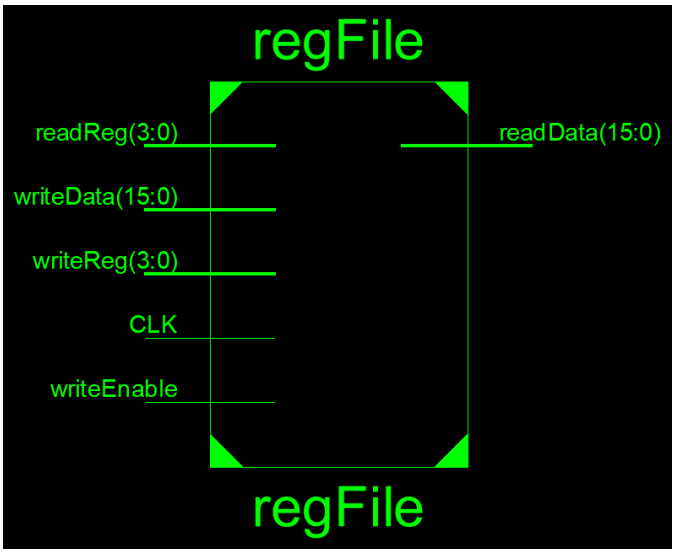
Tests for the stack file cannot be exhaustive because there are far too many possible states for the stack. Instead, they will be a simulated series of pushes and pops and the values in the stack will be monitored. The tests will include pushing data in the top, overflowing/underflowing the stack (where excess elements are just deleted), swapping, and clearing.

5.2.3 Register File

The register file is a collection of miscellaneous registers. It has three inputs: readReg (3-bit), writeData (16-bit) and writeReg (3-bit). It takes two control signal: writeEnable (1-bit) and reset (1-bit). It has one output: readData (16-bit).

Registers are read from and written to on the rising edge of the clock signal. For data to be written, writeEnable must be 1. Data is read from an intermediate register connected to the

register to be read from. When a reset signal is received, all registers are reinitialized to the default value of 0x0000.



Exhaustive tests for the register file can take a long time due to the number of possible values that may be stored. However as we do not perform reads and writes in the same clock cycle, we will not miss any critical issues due to this limitation. The test will simulate a number of writes to each register and verify the value stored in each register after every write.

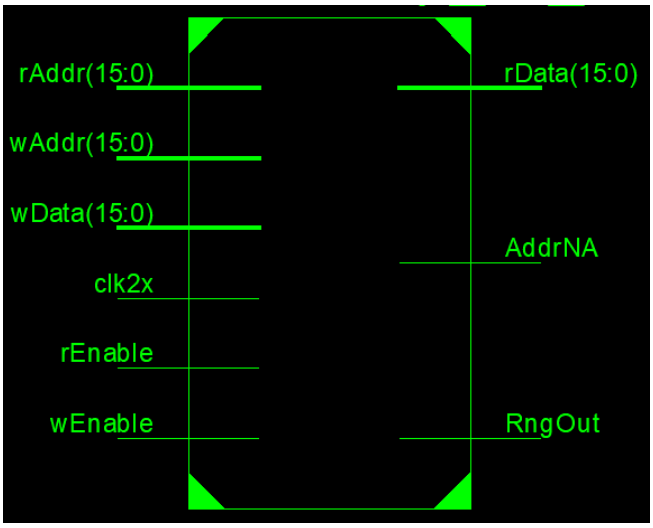
5.2.4 Memory

The memory unit is a dual-port block memory. It has three inputs: wAddr, rAddr and wData which are all 16-bit buses. It has three outputs: two 1-bit flags RngOut and AddrUA (Address Unaligned) and 16-bit bus rData. It also has two control signals wEnable and rEnable and a clock input.

This memory writes on rising edge of the clock signal and reads on falling edge. wEnable needs to be 1 for the memory to write. rEnable needs to be 1 for the memory to read.

The memory is first tested with some simple read and writes, to see if the same data can be retrieved. Then, an exhaustive write-read test is applied to every address to see if the memory has consistent and expected behavior during write and read collisions.

The two output flags are for exception detection. RngOut is set when either of wAddr or rAddr is out of the memory range. AddrUA is set when either of wAddr or rAddr is unaligned. An exhaustive test is needed for all possible inputs to address pins (0 to 65535). In our implementation, a 1 in the lowest bit of the address indicates that it is unaligned and a 1 in the highest bit indicates the address is out of range.



5.2.5 Stack Pointer Register

The stack pointer register holds the value of the stack pointer in memory. It has three

control signals: push, pop, and clear. It has two 16-bit outputs: the value of the stack pointer, and two plus the same value.

On the rising edge of the clock, the value of the stack pointer can change. If clear is set, then it is set to 0. If push (but not pop) is set, then it increases by 2. If pop (but not push) is set, then it decreases by 2. Nothing happens in all other cases.

The tests for this component consisted of setting only clear, then setting only push, then setting only pop. When clear is set, the value should become 0. Each clock cycle push is set, the value should increase by 2. Each clock cycle pop is set, the value should decrease by 2.

5.2.6 Decode Unit

The decode unit takes the 16-bit instruction IR and decodes it into 6 parts. Each part is taken directly from the instruction input bus. Details are in the table below.

In		Out->	opcode	np	funcimm	addr	pushs_select	regi
IR			IR[15:11]	IR[10]	IR[9:0]	IR[10:0]	IR[4:0]	IR[2:0]

The test branch used to test the decode unit was an exhaustive test that inputs all 65536 possibilities of IR. It then verifies that, for each IR, the six outputs are expected.

5.2.7 Zero Checker

The zero checker checks to see if a 16-bit bus contains all zeros. If it does then the checker will output 1. Otherwise, the checker will output 0.

The hardware implementation will be an if statement in Verilog to check if it’s zero and assign the output. The schematic will be a 16-input OR gate to or the bits together with an inverter in the end.

This module doesn’t require any control signals. For testing this component, exhaustive testing will be easy to implement with a loop to test all the possible 16-bit value. And the test bench only needs to output result when detected zero for ease of checking.

5.2.8 Sign Extender

The sign extender takes one 10-bit input and produces one 16-bit output that consists of 6 bits identical to the most significant bit (the sign bit) followed by the 10-bit input. It will be implemented by extending the 15th wire of the bus to the 16th through the 31st wires. An exhaustive test is used for every 10-bit input (0-1023 in unsigned decimal, 1024 possibilities) and make sure the output is as expected.

5.2.9 Zero Padder

The zero padder takes one 10-bit input and produces one 16-bit output signal that consists of the 10-bit input followed by 6 zero bits. It will be implemented by connecting the 10-bit bus to the 15th through the 6th wires, and connecting the 5th through the 0th wire in the output bus to ground. A test similar to that of the sign extender is used to test this part.

5.2.10 Zero Extender

The zero extender takes one 10-bit input and produces one 16-bit output signal that consists of 6 zero bits followed by the 10-bit input. The implementation is similar to the zero padder, but with the ground connecting to the most significant 6 bits in the output bus and the input connecting to the least significant 10 bits. A test similar to that of zero padder is used to test this part.

5.3 Control Unit Specification

There are three control units in the datapath: Central Control, ALU Control and Push-Pop Control.

5.3.1 Central Control

The Central Control Unit (CCU) controls 20 signals listed in the table below.

Control Signal	**Description**
PCWrite	If set, the value of PC is allowed to change.
regWrite	If set, the register file can be written to.
wEnable	If set, memory can be written to.
rEnable	If set, memory can be read from.
IRWrite	If set, the instruction register is allowed to change.
MDRWrite	If set, the memory data register is allowed to change.
AWrite	If set, the A register (which holds the output of STK[0]) is allowed to change.
BWrite	If set, the B register (which holds the output of STK[1]) is allowed to change.
STKRWrite	If set, the STKR register (which holds the output of STK[select]) is

	allowed to change.
RWrite	If set, the R register (which holds the output of the register file) is allowed to change.
MWrite	If set, the M register (which holds the value of the $mem register) is allowed to change.
ALUsrcAMux	Selects which input source goes into the first input of the ALU. 00: PC 01: A (STK[0]) 10: 0 11: memBase
ALUsrcBMux	Selects which input source goes into the second input of the ALU 000: B (STK[1]) 001: STKR 010: SE(imm) 011: ZE(imm) 100: ZP(imm) 101: regRead 110: <Unused> 111: 2
ALUOutWrite	If set, the ALUOut register is allowed to change
rAddrMux	Selects which address to read from memory 00: PC 01: SP+2 10: ALUOut 11: <Unused>
wAddrMux	Selects which address to write to memory 0: SP 1: ALUOut
wDataMux	Selects which data to write to memory 0: STK[0] 1: STK[31]
regWDataMux	Selects which data to write to register 0: PC 1: STK[0]
regWRegMux	Selects which register to write data to 0: regi 1: 1 (return address)

branchTargetMux	Selects which target to branch to 00: PC+2 01: Target of j and jal 10: Target of bz/bnz/bzn/bnzn 11: Target of jr
stkTopDataMux	Selects which data to write to stack
stkTopWEnable	If set, the stack writes data to STK[0] 0: ALUOut 1: MDR
stkBotWEnable	If set, the stack writes data to STK[31].
SPPP	A two-bit control signal for the stack pointer push and pop: 01: pop 10: push 00, 11: nothing
stkPP	A two-bit control signal for the stack push and pop: 01: pop 10: push 11: swap 00: nothing
ALUopcode	Sets which operation the ALU should do. 1XXXX: Use Func 0XXXX: Use the least significant four bits as ALUop
BDEnable	If set, the Branch Decider will output 1 when the branch condition is met. If not set, the Branch Decider always outputs 0.

5.3.2 ALU Control

The ALU Control Unit (ACU) controls the operation of ALU. It takes the ALUopcode signal from central control and func field of the instruction and uses combinational logic to determine the actual opcode for ALU. The ALUopcode will either tell ACU to use a specific ALUop, or use the ALUop specified by funct.

Below is a table of how this works in detail.

ALUopcode[5:0]	Funct[4:0]	ALUop[4:0]
1XXXX	Any	Funct[4:0]

0XXXX	Any (Unused)	ALUopcode[4:0]

5.3.3 Branch Decider

The branch decider takes the BDEnable control signal from central control and the 5-bit opcode, and it has a 1-bit isZero input. It outputs whether or not a conditional branch should be taken.

If BDEnable is not set, it always outputs zero. If it is, then it looks at the opcode and isZero. If the opcode is bz (8) and isZero is set, it returns 1. If the opcode is bnz (9) and isZero is not set, it returns 1. Otherwise, it returns zero.

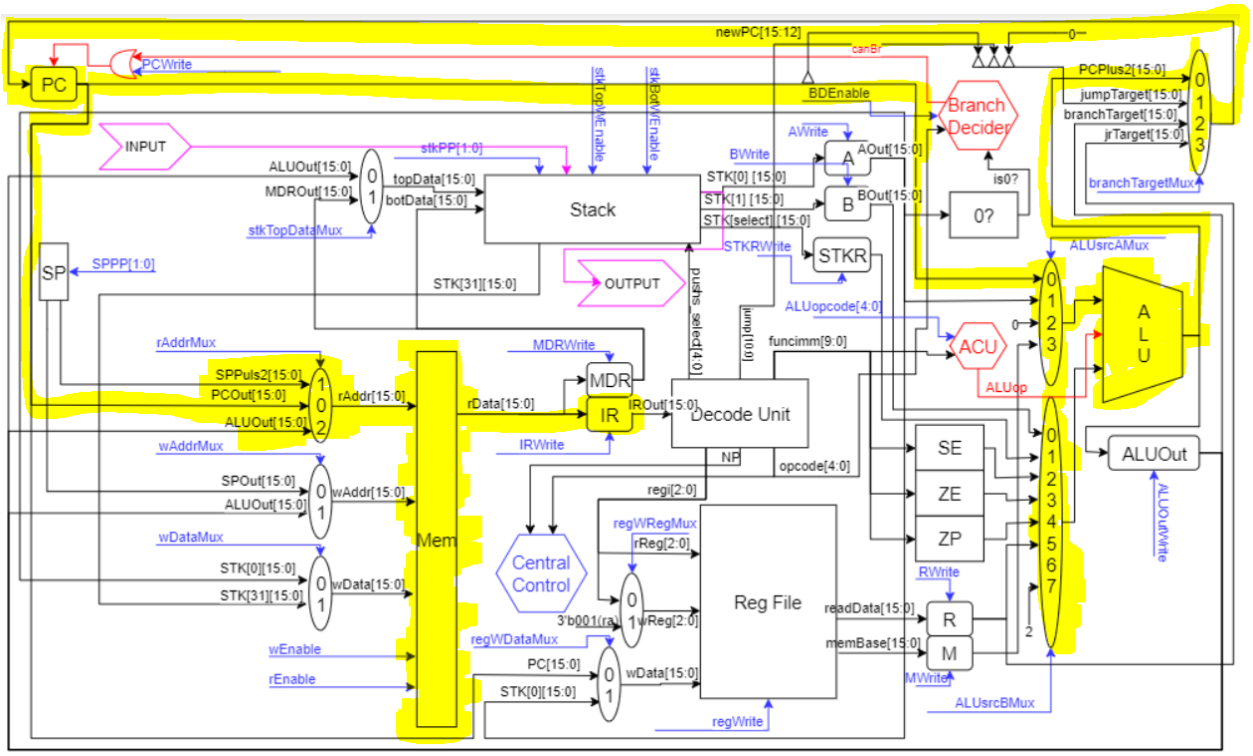
This component can be tested exhaustively, for all combinations of opcode, BDEnable, and isZero.

Section 6 Integration Plan and Tests

In the paragraphs below, “simulated” instructions are run by manually setting various inputs and outputs that would normally be set by other, yet-to-be-integrated parts.

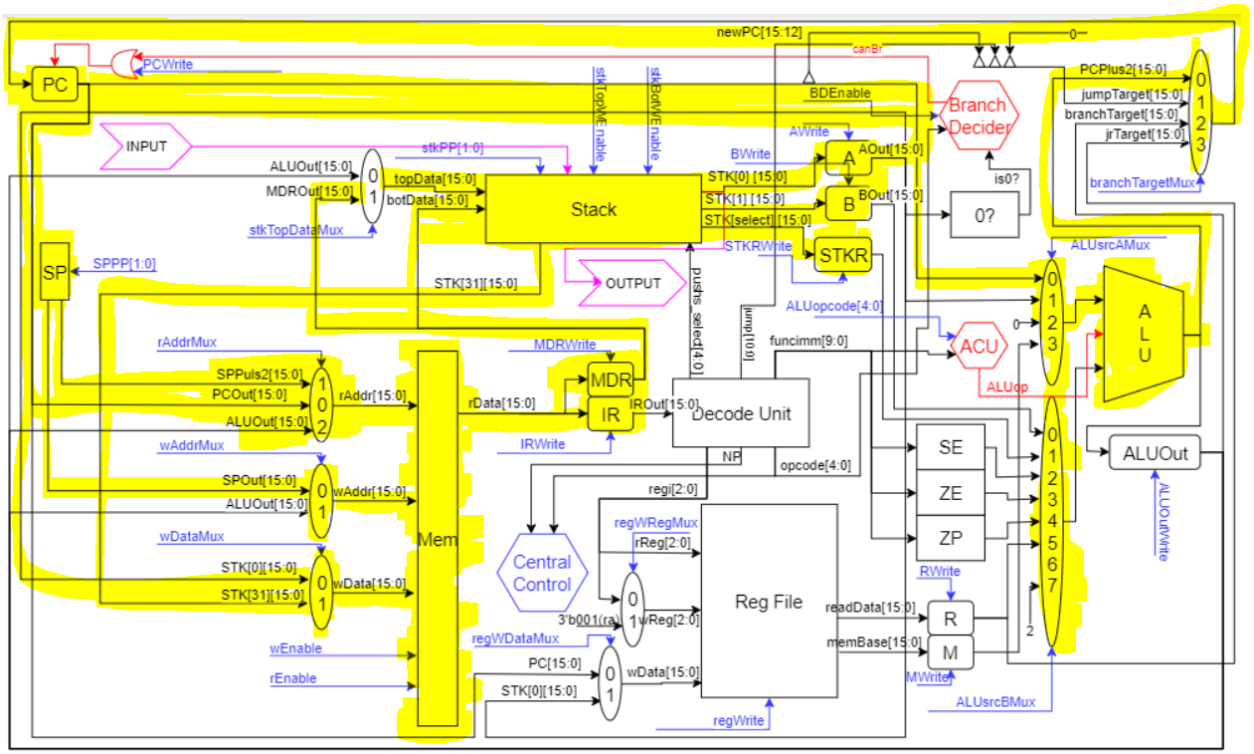
6.1 Fetch

The first parts we plan to integrate are the PC register, memory, the IR register, and the ALU. Together, they should be able to complete the fetch cycle of an instruction. Tests will consist of initializing some values in memory, letting PC increment through them, and monitoring IR to make sure it outputs the correct value. The parts integrated in this step are highlighted in yellow in the diagram below.



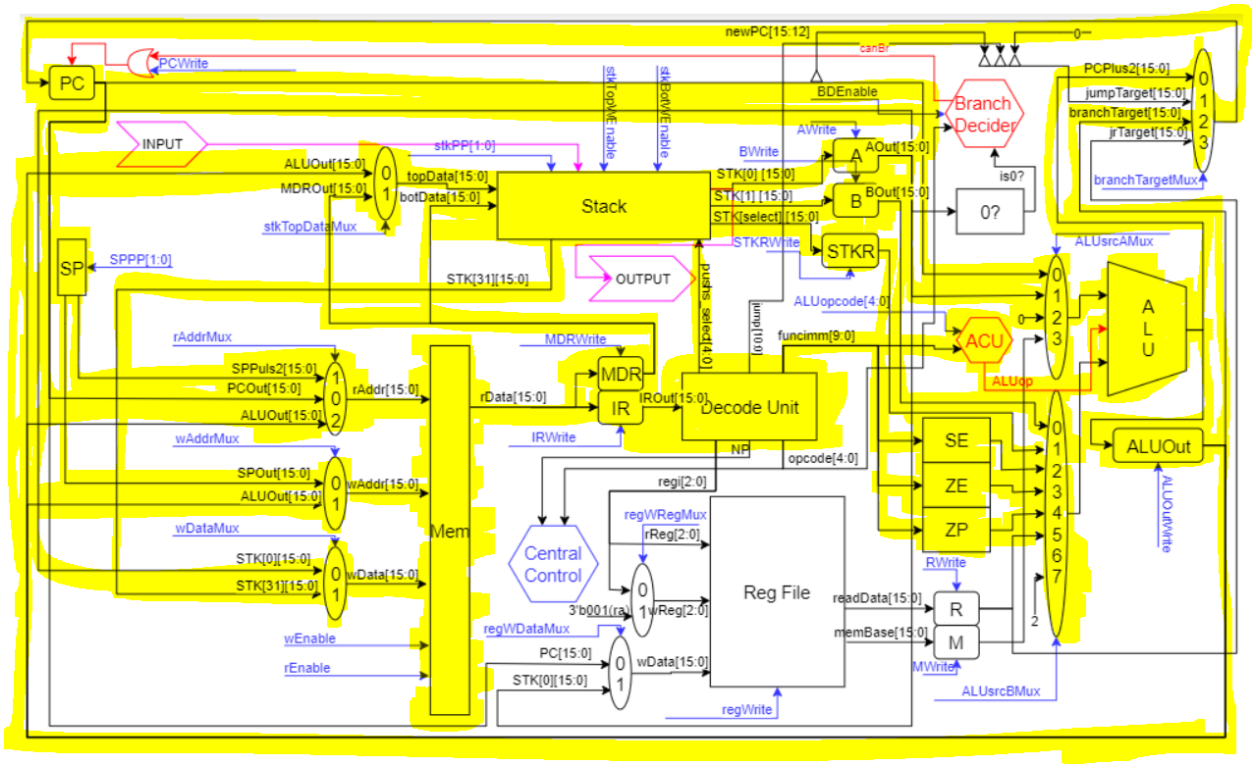
6.2 Add Stack

Add the stack file, SP, A, B, STKR, and MDR to the previous parts. This will allow full tests of pushing and popping, with the spillover to memory. Tests will consist of simulating repeated pushes and pops and monitoring the values in the stack and in memory. The diagram below highlights all the parts that have been integrated so far.



6.3 Add Decode Unit, SE, ZE, ZP, ALUOut

Add the decode unit to the previous parts. The decode unit can run the stack file's select input. Tests will be similar to those of the previous part, but they will also test the output of STK[select] and get stack output from A and B. We will also add SE, ZE, ZP inputs to ALU source B in this step, as well as ACU.



6.4 Add Register File

Add the register file, R, and M to the previous parts. Tests will consist of simulated pushr, popr, and peekr instructions, as well as the values in R and M after simulated decode stages. The only things left will be zero detector, branch decider and central control unit.

6.5 Add Branches

Add the zero detector, and branch decider. At this point, the CPU should be able to run all instructions with manual control signals. Tests will consist of simulated branches and jumps, as well as complete instructions of other types. The only unintegrated part will be the control unit.

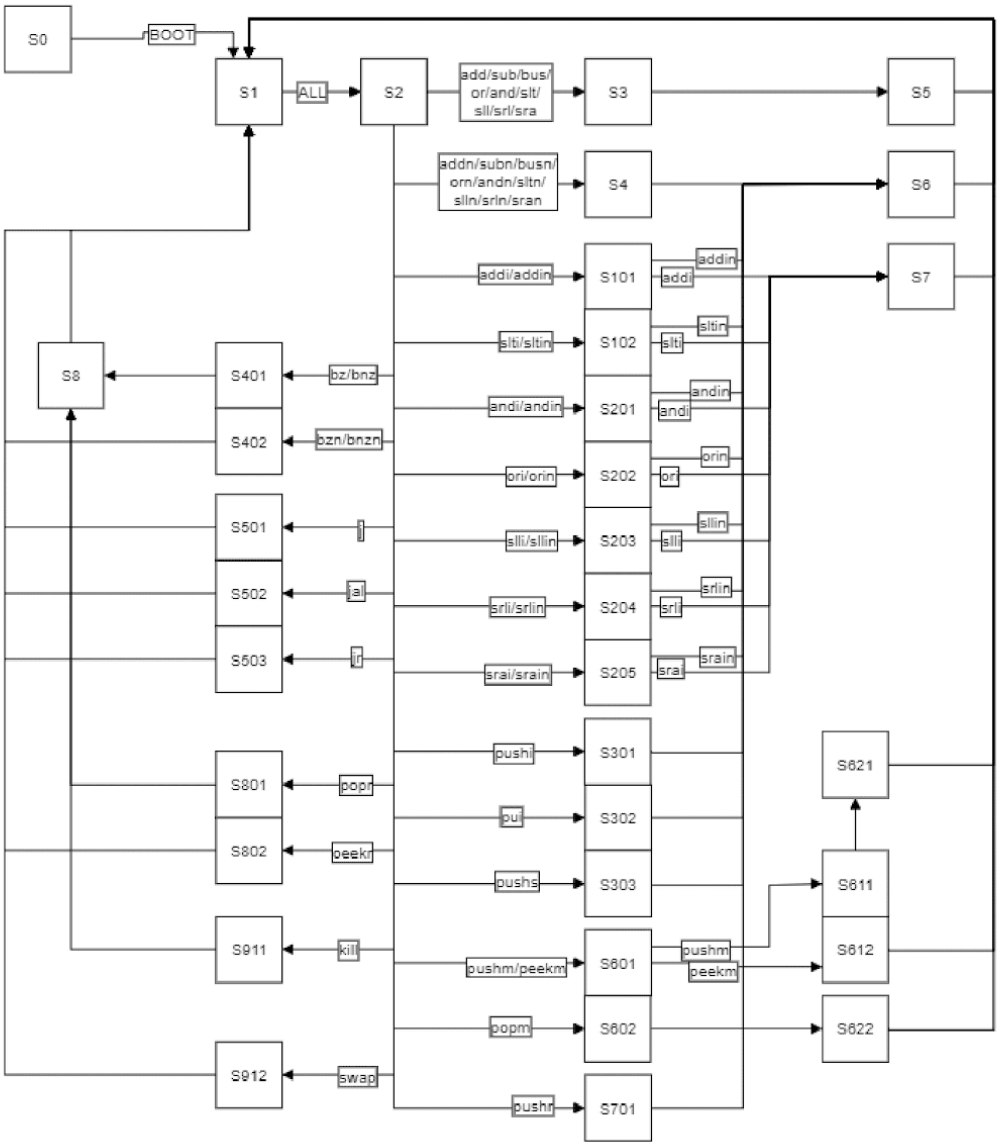
6.6 Added Central Control (Complete processor)

The integrated part in 6.5 should only have control signal inputs and opcode-NP outputs besides I/O, clock and reset pins. In this step, the central control will be integrated. Tests will involve running branch instructions.

Section 7 Control State Transition

In this section we provide the state transition for our processor. There are two subsections: the state transition diagram and control signal table. The diagram shows the flow between states and the table shows what the controls are set to in each state.

Section 7.1 State Transition Diagram



Section 7.2 Control Signals For States

In the following table we provide control signals for each state in Section 7.1. For enable flags, we always explicitly state whether it should be a 1 or 0. For mux select, we put an explicit value if it matters or “-” if it doesn’t matter. For example, branchTargetMux only has meaningful input if we are writing to PC.

States > Controls	S0	S1	S2	S3	S4	S5	S6	S7	S8	S101	S102	S201
PCWrite	0	1	0	0	0	0	0	0	0	0	0	0
regWrite	0	0	0	0	0	0	0	0	0	0	0	0
wEnable	0	0	0	0	0	0	1	0	0	0	0	0
rEnable	0	1	0	1	0	0	0	0	0	0	0	0
IRWrite	0	1	0	0	0	0	0	0	0	0	0	0
MDRWrite	0	0	0	1	0	0	0	0	0	0	0	0
AWrite	0	0	1	0	0	0	0	0	0	0	0	0
BWrite	0	0	1	0	0	0	0	0	0	0	0	0
STKRWrite	0	0	1	0	0	0	0	0	0	0	0	0
RWrite	0	0	1	0	0	0	0	0	0	0	0	0
MWrite	0	0	1	0	0	0	0	0	0	0	0	0
ALUsrcAMux	--	00	00	01	01	--	--	--	--	01	01	01
ALUsrcBMux	---	111	110	000	000	---	---	---	---	010	010	011
ALUOutWrite	0	0	1	1	1	0	0	0	0	1	1	1
rAddrMux	--	00	--	01	--	--	--	--	--	--	--	--
wAddrMux	-	-	-	-	-	-	0	-	-	-	-	-
wDataMux	-	-	-	-	-	-	1	-	-	-	-	-
regWDataMux	-	-	-	-	-	-	-	-	-	-	-	-
regWRegMux	-	-	-	-	-	-	-	-	-	-	-	-
branchTargetMux	--	00	--	--	--	--	--	--	--	--	--	--
stkTopDataMux	-	-	-	-	-	0	0	0	0	-	-	-
stkTopWEnable	0	0	0	0	0	1	1	1	0	0	0	0
stkBotWEnable	0	0	0	0	0	1	0	0	1	0	0	0
SPPP	00	00	00	01	00	00	10	00	00	00	00	00
stkPP	00	00	00	01	00	00	10	00	00	00	00	00
ALUopcode[4]	0	0	0	1	1	0	0	0	0	0	0	0
ALUopcode[3:0]	0000	0010	0010	0000	0000	0000	0000	0000	0000	0010	0111	0000
BDEnable	0	0	0	0	0	0	0	0	0	0	0	0

States > Controls	S202	S203	S204	S205	S301	S302	S303	S401	S402	S501	S502	S503
PCWrite	0	0	0	0	0	0	0	0	0	1	1	1
regWrite	0	0	0	0	0	0	0	0	0	0	1	0
wEnable	0	0	0	0	0	0	0	0	0	0	0	0
rEnable	0	0	0	0	0	0	0	1	0	0	0	0
IRWrite	0	0	0	0	0	0	0	0	0	0	0	0
MDRWrite	0	0	0	0	0	0	0	1	0	0	0	0
AWrite	0	0	0	0	0	0	0	0	0	0	0	0
BWrite	0	0	0	0	0	0	0	0	0	0	0	0
STKRWrite	0	0	0	0	0	0	0	0	0	0	0	0
RWrite	0	0	0	0	0	0	0	0	0	0	0	0
MWrite	0	0	0	0	0	0	0	0	0	0	0	0
ALUsrcAMux	01	01	01	01	10	10	10	--	--	--	--	--
ALUsrcBMux	011	011	011	011	010	100	001	---	---	---	---	---
ALUOutWrite	1	1	1	1	1	1	1	0	0	0	0	0
rAddrMux	--	--	--	--	--	--	--	01	--	--	--	--
wAddrMux	-	-	-	-	-	-	-	-	-	-	-	-
wDataMux	-	-	-	-	-	-	-	-	-	-	-	-
regWDataMux	-	-	-	-	-	-	-	-	-	-	0	-
regWRegMux	-	-	-	-	-	-	-	-	-	-	1	-
branchTargetMux	--	--	--	--	--	--	--	10	10	01	01	11
stkTopDataMux	-	-	-	-	-	-	-	-	-	-	-	-
stkTopWEnable	0	0	0	0	0	0	0	0	0	0	0	0
stkBotWEnable	0	0	0	0	0	0	0	0	0	0	0	0
SPPP	00	00	00	00	00	00	00	01	00	00	00	00
stkPP	00	00	00	00	00	00	00	01	00	00	00	00
ALUopcode[4]	0	0	0	0	0	0	0	0	0	0	0	0
ALUopcode[3:0]	0001	1000	1010	1011	0010	0010	0010	0000	0000	0000	0000	0000
BDEnable	0	0	0	0	0	0	0	1	1	0	0	0

States > Controls	S601	S602	S611	S612	S621	S622	S701	S801	S802	S911	S912	
PCWrite	0	0	0	0	0	0	0	0	0	0	0	
regWrite	0	0	0	0	0	0	0	1	1	0	0	
wEnable	0	0	1	1	0	1	0	0	0	0	0	
rEnable	0	1	1	0	0	0	0	1	0	1	0	
IRWrite	0	0	0	0	0	0	0	0	0	0	0	
MDRWrite	0	1	1	0	0	0	0	1	0	1	0	
AWrite	0	0	0	0	0	0	0	0	0	0	0	
BWrite	0	0	0	0	0	0	0	0	0	0	0	
STKRWrite	0	0	0	0	0	0	0	0	0	0	0	
RWrite	0	0	0	0	0	0	0	0	0	0	0	
MWrite	0	0	0	0	0	0	0	0	0	0	0	
ALUsrcAMux	11	11	--	--	--	--	10	--	--	--	--	
ALUsrcBMux	110	110	---	---	---	---	101	---	---	---	---	
ALUOutWrite	1	1	0	0	0	0	1	0	0	0	0	
rAddrMux	--	01	10	--	--	--	--	01	--	01	--	
wAddrMux	-	-	0	1	-	1	-	-	-	-	-	
wDataMux	-	-	1	0	-	0	-	-	-	-	-	
regWDataMux	-	-	-	-	-	-	-	1	1	-	-	
regWRegMux	-	-	-	-	-	-	-	0	0	-	-	
branchTargetMux	--	--	--	--	--	--	--	--	--	--	--	
stkTopDataMux	-	-	-	-	1	-	-	-	-	-	-	
stkTopWEnable	0	0	0	0	1	0	0	0	0	0	0	
stkBotWEnable	0	0	0	0	0	1	0	0	0	0	0	
SPPP	00	01	10	00	00	00	00	01	00	01	00	
stkPP	00	01	10	00	00	00	00	01	00	01	11	
ALUopcode[4]	0	0	0	0	0	0	0	0	0	0	0	
ALUopcode[3:0]	0010	0010	0000	0000	0000	0000	0010	0000	0000	0000	0000	
BDEnable	0	0	0	0	0	0	0	0	0	0	0	

Section 8 System Test

In this section we describe a series of tests that we will apply to our complete processor to see if it works.

8.1 Unit Instruction Tests

For these tests, we will run each instruction twice with different arguments and inspect the output. We will first test the instructions that we need to use to load other tests, like pushi.

All the unit tests output the number 233 as the success code, so we just need to look at the final output to determine if the tests pass.

8.2 Small Segment Tests

After we are sure that every instruction works, we will run 3 small code segments: the simple for-loop described in section 3, fibonacci number, and recursive sum. The later two are detailed below. Fibonacci is ran with the largest input possible to not cause overflow, which is fib(23)=28657

Fibonacci

Addr	Object Code	Assembly	Comment
			//Simple for loop
		# Given an input n, returns the nth Fibonacci number.	
0x00	1000000000000010	popr $t0	Store input
0x02	1000110000000001	pushi 1	Initial values
0x04	1000110000000000	pushi 0	
		loop:	
0x06	0111110000000010	pushr $t0	
0x08	0100010000000110	bzn done	
0x0a	0000101111111111	addi -1	Decrement counter
0x0c	1000000000000010	popr $t0	
0x0e	1001110000000001	pushs 1	
0x10	0000000000000000	add	
0x12	1010010000000000	swap	
0x14	0101000000000011	j loop	
		done:	
0x16	1010100000000000	kill	Clean up stack
0x18	1010010000000000	swap	
0x20	1010100000000000	kill	

Recursive Sum - Test procedure calls by recursing n levels. The output should be the same as the input

Addr	Object Code	Assembly	Comment
		# Given a number n, it recurses n levels deep and returns n. start:	//Simple for loop
0x00	0101100000000010	jal sum	Initial branch
0x02	0101000000001011	j done	
		sum:	
0x04	0100010000000111	bzn return	Base case
0x06	0000101111111111	addi -1	
0x08	0111110000000001	pushr $ra	Backup ra
0x0a	1010010000000000	swap	
0x0c	0101100000000010	jal sum	
0x0e	1010010000000000	swap	
0x10	1000000000000001	popr $ra	Restore ra
0x12	0000100000000001	addi 1	
		return:	
0x14	0110010000000001	jr $ra	Return
		done:	

8.3 Large Program Tests

We will run relPrime for input = 178, 2310 and 5040. We should get output = 3, 13, 11.

Section 9 Performance Benchmark

We ran the relPrime with input=5040, reached the expected output and collected the following data:

Program Size: 62 bytes
Number of instructions: 71435
Number of cycles: 255165
Clock period: 14.254ns (frequency: 70.156MHz)

Based on these data, we computed the execution time of the program

CPI = 255165 / 71435 = 3.572 cpi
ET = 255165 * 14.254ns = 3637121.91ns = 3.64ms

Below is the Device utilization summary from Xilinx Synthesis Report
Device utilization summary:

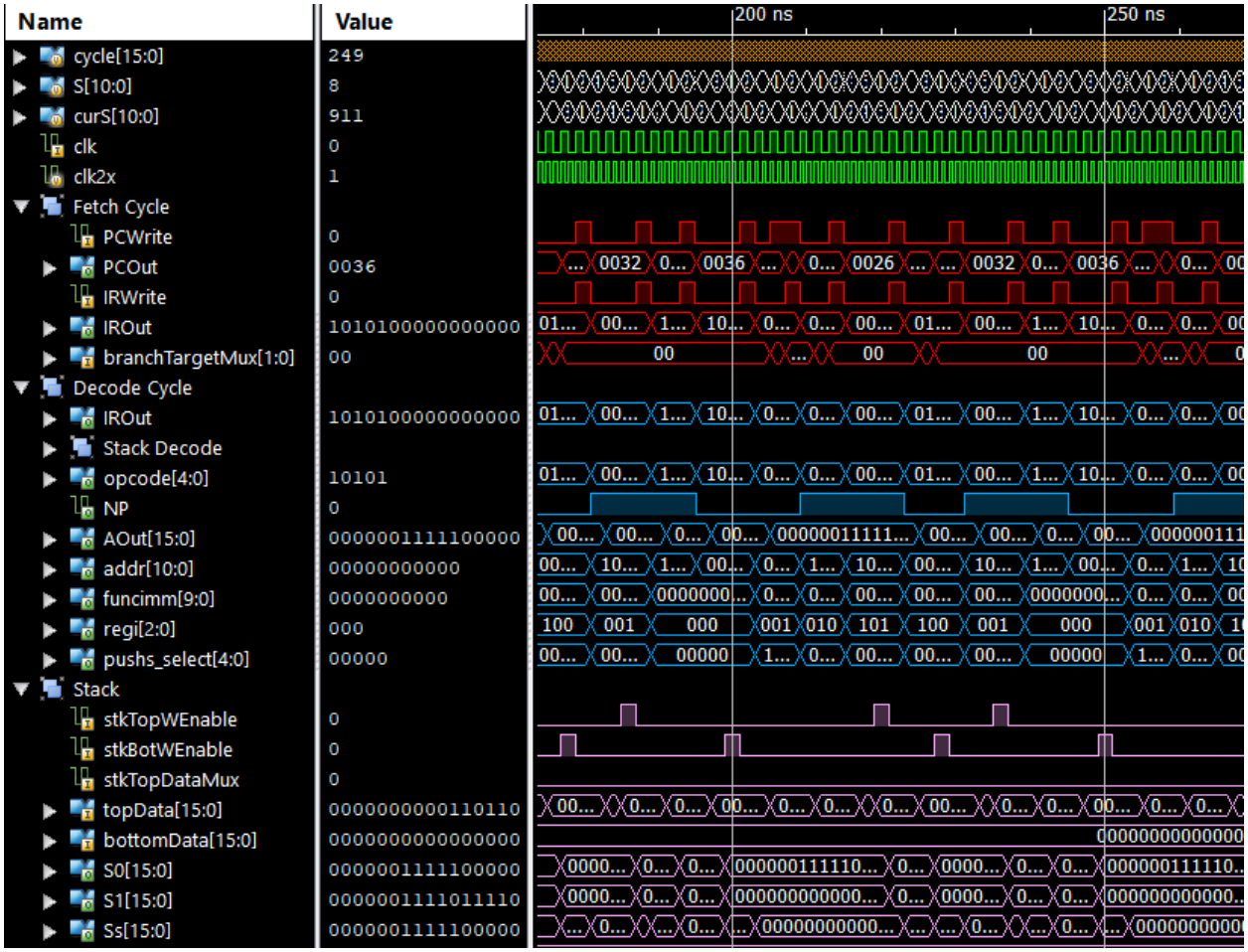
Selected Device : 3s500efg320-4

Number of Slices:	1150	out of	4656	24%
Number of Slice Flip Flops:	844	out of	9312	9%
Number of 4 input LUTs:	2083	out of	9312	22%
Number of IOs:	34			
Number of bonded IOBs:	34	out of	232	14%
Number of BRAMs:	15	out of	20	75%
Number of GCLKs:	2	out of	24	8%

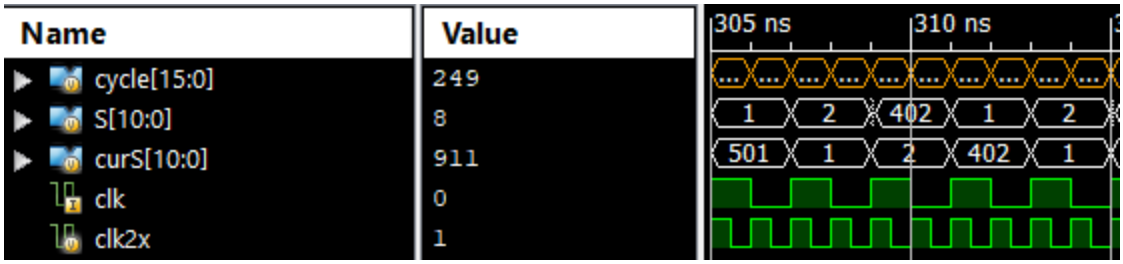
Section 10 Debugging Tools

10.1 Wave Configuration

To assist in the debugging process, we made nice looking wave configurations with different colors indicate different parts of the data path. Also, the signals are group in both cycles and components. For example, PC is used in fetch cycle and as one of the inputs to the ALU, so it appears in both fetch cycle block and ALU block. This way it is very easy to find a signal when we need it.



Part of the wave form. The red signals are fetch stage. The blue signals are decode stage. The pink signals are all related to stack. Other groups are not shown.



Zoomed in view of the clock and state indicators.

10.2 Interpreter

The interpreter is a Java program that reads a text file with assembly code and runs it. It provides breakpoints and debugging tools, which were useful for making sure our tests were doing what we thought they were.

It runs in the command prompt. Putting “break” instructions in the assembly code places breakpoints. When execution reaches a breakpoint, it stops and prompts for debug commands to do things like view the contents of the stack, registers, where it is in the program, change where breakpoints are, and step through the program.

```
$ java -jar interpreter.jar TESTS/i-asm.txt
Execution paused at line 17: pushi -1
>help
List of valid commands:
step: step one instruction
run: resume execution
stack: view the stack
reg: view the registers
program: view the program
break: add a breakpoint
clear: clear a breakpoint
help: view this list
```

We used the interpreter to test our tests before we put them on the processor.

10.3 Assembler

```
# Start Your Program Here!


pushiTest:
        pushi 100
        bz pushiFail
        pushi 0
        bnz pushiFail
        j addiTest
pushiFail:
        # need some way of writing to reg or output without stack to check
pushi
        pushi 201 # error code
        j reallyDone


addiTest:
        pushi -1
        addi 1
        bnz addiFail
        pushi 0
        addi 1
        bz addiFail
        j addinTest
addiFail:
        pushi 202 # error code
        j reallyDone


addinTest:
        pushi -1
        addin 1
        bnz addinFail
        pushi 0
        addin 1
        bz addinFail
        kill
        kill
        j andiTest
addinFail:
```

```
1000110001100100
0100000000000011
1000110000000000
0100100000000001
0101000000000111
1000110011001001
0101000011000110
1000111111111111
0000100000000001
0100100000000100
1000110000000000
0000100000000001
0100000000000001
0101000000010000
1000110011001010
0101000011000110
1000111111111111
0000110000000001
0100100000000110
1000110000000000
0000110000000001
0100000000000011
1010100000000000
1010100000000000
0101000000011011
1000110011001011
0101000011000110
1000110000001001
0001000000011011
0000101111110111
0100100000000001
0101000000100010
1000110011001100
0101000011000110
1000110000001001
0001010000011011
0000101111110111
0100100000000010
1010100000000000
```

The assembler is written in JavaScript and was a huge help in the testing stage. We are able to write the assembly program in a human readable syntax and the assembler takes care of the branch labels automatically. The assembler UI is very simple - two text areas and a button to assemble. It actually parses instruction definitions as plain text. So it can be used to assemble any instruction set.

Section 11How to use our debug tool

11.1 Wave Configuration

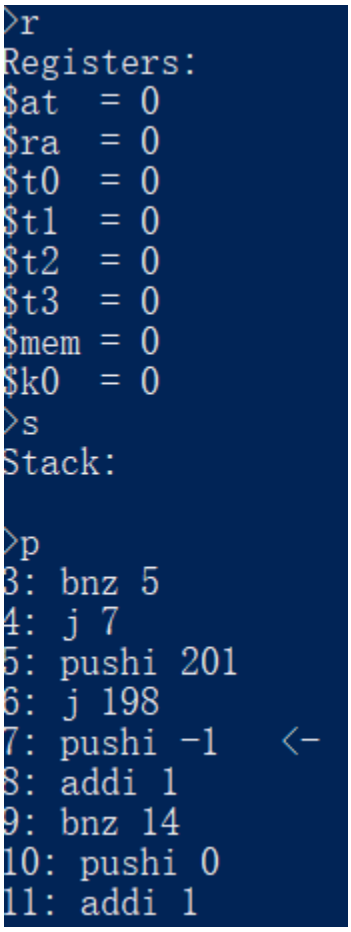
We have configured waveform files for each part of our tests. The formated waveform files have all the color coded and grouped signals.

When simulate the project right click **Simulate Behavioral Model** Choose corresponding *Custom Waveform Configuration File* under **Process property**

11.2 Interpreter

The interpreter is in the repo, the **README.md** provide instruction of how to use the Interpreter.

break in your asm.txt file will add a breakpoint for your program
In command prop **r** for register, **s** for stack, **p** for program.
More details reference the Interpreter documentation or type **help**



11.3 Assembler

Assembler code can be accessed from the git repo. To access the online assembler visit https://online-asm.firebaseapp.com/index.html login with Rose account.

You can setup Instruction set in the assembler with the designed instruction format. Then you can create your own assembly code documents and set up the assembler to assemble the program in various formats such as binary, hex, with other format wrap around like , or () and etc.

There are also choose of line number, comment and etc. For more detail reference the assembler documentation

Zhengshan(Bill) Fang, Work Log

https://docs.google.com/document/d/1GhhjKKGrFmy5iXPUSxbDNmEiF5HvZqmb2tHR8h05kFU/edit?usp=sharing

Saturday, November 16, 2019

Met with the team [1hr]

- Practiced again

Looking ahead:

- Dress up for tomorrow
- Presentation tomorrow!

Sunday, November 17, 2019

Met with the team [1hr]

- Practice for the presentation

Looking ahead:

- Practice for the presentation again

Saturday, November 16, 2019

Met with the team [2.5hr]

- Finished the final report
- Final presentation

Looking ahead:

- Practice for the presentation

Looking ahead post M6

- We going to continue to debug the FPGA program so we can use our CPU design on the board
- Documentation of how to use our processor and our debug tool
- Final report writing
- Final presentation slide and practice
- Potential cool demo program of using decimal computation

Friday, November 15, 2019

Met with Reed [1hr]

- Test and write how to run the real-prime program
- Documented how to for debugging tools

Thursday, November 14, 2019

Met with Dr.Stamm + team meeting [1hr]

- Spend a long time did an unsuccessful demo
- Investigate more on the FPGA implementation program could be issues relating to reset

Wednesday, November 13, 2019

Met with the team in the class [1hr]

- Debug FPGA implementation sample program works
- Register works our CPU doesn't work

Tuesday, November 12, 2019

Met with the team in the class [1hr]

- Documentation and FPGA grogram modifying the sample ALU program
- The program symbol is misaligned some connection is missed

Monday, November 11, 2019

Met with the team [2hr]

- Finish debug all the tests for our CPU instruction testing
- Get rid of input dependency for the test. (input doesn't matter, we can still test all the BJR instructions.
- Found a fail in our CPU for logical shift and arithmetic shift reversed

Monday, November 11, 2019

Met with the team in the class [1hr]

- Debug BJR tests (Assembling issue)
- The program assembles in machine code figure out putting it in memory properly

Sunday, November 10, 2019

Met with the team [2hr]

- Finish most of the testing run real prime with our CPU

Sunday, November 10, 2019

Individual work time [1.5hr]

- Test for all the B, J, R type of instructions

Friday, November 8, 2019

Met with the team [0.5hr]

- Met with Dr. Stamm
- Plan next meeting and plan for next week

Thursday, November 7, 2019

Met with the team [1hr]

- System test

Wednesday, November 6, 2019

Met with the team [2hr]

- Finished Integration and testing Part 5, 6 Branch decider and Control

Tuesday, November 5, 2019

Met with the team [3 hr]

- Finished Part4 integration File register
- Finished register file test

Tuesday, November 5, 2019

In-class meeting [1 hr]

- Finish testing of decode and extenders

Monday, November 4, 2019

Met with the team [3 hr]

- Finished Part2 of the stack integration and testing
- Tested all push pop function
- Integrated decode and extender units

Thursday, October 31, 2019

Met with the Reed [1 hr]

- Partially integrated of part2 processor (Part1, STK, MDR)

Wednesday, October 30, 2019

Met with the team [2 hr]

- Integrated part processor (Men, PC, ALU)
- Start testing for the partial integration

Monday, October 28, 2019

Met with the team [2 hr]

- Finished ALU control
- Change ALU control to ACU
- Finished ACU
- Complete ALU control section in the doc

Monday, October 28, 2019

In-class work time [2 hr]

- Finished ALU testing
- Fixed zero checker test bench
- Start ALU control

Sunday, October 27, 2019

Worked [2.5 hr]

- Revise the ALU test bench

Wednesday, October 23, 2019

In-class work time [2 hr]

- Finished 4-bit ALU testing
- Demo lab6
- Wrote integration plan with the team

Tuesday, October 22, 2019

Worked [3.5 hr]

- Worked on lab 6 ALU finished the 4-bit ALU
- Wrote component descriptions for ALU and zero checker
- brief plan for implement.
- the basic specs for your control unit
- A brief description of the unit tests to verify the implementation is correct.

Monday, October 21, 2019

Worked [1.5 hr]

- Worked on lab 6 ALU finished the 1-bit ALU
- Finished the test bench for 1-bit ALU
- Start working on the 4bit ALU
- Implemented zeroChecker

Saturday, October 19, 2019

Met with the team [1.5hr]

- Verified the RTLs
- Implemented the datapath
- Assign components to build on our own (ALU and Zero detect) estimated [1.5 hr]

Friday, October 18, 2019

Worked [0.5 hr]

- Worked on lab 6 ALU

Tuesday, October 15, 2019

Worked [.5hr]

- Added checking RTL to the design doc
- Propose everyone do the block diagram of datapath and description, update the list of components and RTL description accordingly of the type we do last time
  Time Estimated [2hr]

Monday, October 14, 2019

Met with the team [2.5hr]

- Generated table for each instruction type
- Added what has changed section
- Reorganize sample code section
- Organized RTL table
- Put down Signal needed

Friday, October 11, 2019

Work on RTL for I type instruction [45 min]

- Figure out the RTL and the signal needed for the instruction
- Ready to share it in the next meeting
- Added table of content and page number to the report

Monday, October 7, 2019

Met with the team [1.5 hr]

- Discuss and finish the procedure call convention
- Assign opcode/funct to instructions
- Have the example machine language write out

- For Next meeting do RTL for I type instruction [Estimated time1hr]

Sunday, October 6, 2019

Met with the team [3hr]

- Finalized register table
- Finalized instructions types and have word description for each type
- Finalized instructions table and description for each instruction
- Extend the assembly implementations
- Changed opcode to 5 bits and extend imm and address to 10 bits
- We have decided to have our stack been register cascaded and the last one goes to memory. When we do push/pop we enable write. Supposedly we doing multicycle.

Saturday, October 5, 2019

Met with the team [2hr]

- We decided to make a stack-based processor. We have decided to have some special registers and then a register stack. We each shared the instructions we thought of before the meeting then generated a list of different types of instructions. Have the draft of the structure for different types of instructions.
- Implemented GCD function in assembly

Reed Phillips Work Log

Friday, October 4:

Brainstorm instructions (~10 minutes)

Saturday, October 5:

Meet with team (~2.5 hours)

We combined our lists of instructions into one cohesive list. We're making a stack architecture, so we needed ways around the primary disadvantage of a stack: you can only access the top one or two things on it. We decided to have several registers to store values in, as well as an instruction to read values from deeper in the stack.

Sunday, October 6:

Meet with team (~2.5 hours)

We finalized the format of our instructions, except for the specific opcodes. We decided to figure those out when we had hardware to wire up with them. We wrote some code fragments and the required relPrime function.

Monday, October 7:

Meet with team (~1.5 hours)

We assigned dummy opcodes to each instruction for the purpose of writing machine code. We don't have an assembler yet, so we manually assembled each of the code fragments. We finalized procedure call convention. We divvied up tasks to start milestone 2; mine is to draft RTL for our R-type instructions: pushr, popr, peekr, and jr.

Draft R-type RTL (~15 minutes)

I wrote out a draft for the RTL driving pushr, popr, peekr, and jr.

Monday, October 14 :

Meet with team(~3 hours)

Unfortunately, Sarthak went AWOL. The other three of us merged our draft RTL into steps of a multicycle datapath. We fixed the formatting issues in our design document and made a list of circuit components we'd need. Hopefully we'll be able to double-check our RTL when we have all of it.

Saturday, October 19:

Meet with team (~2.5 hours)

We double-checked our RTL by going through the steps for add, addn, addi, pushm, bz, and jal. Everything went well, so we drew out our datapath on a whiteboard. We split up some of the parts we'd need and agreed to have them built in Xilinx by Monday; I got the stack and the stack pointer control. I estimated the stack to take an hour or two and the stack pointer control to take about half an hour.

Sunday, October 20:

Xilinx is stupid (~1.5 hours)

I built the stack pointer control, which has a register that takes control signals to increment or decrement by 2. The Verilog was correct within fifteen minutes, and the other hour or so was spent reformatting it to get around all sorts of syntax errors that Xilinx refused to tell me about. For instance, rather than complain when I wrote a 16-bit value into an (accidentally) 1-bit register, it just decided to write 0 every time instead.

Monday, October 21 – Friday, October 15:

Class time (6 hours)

I didn't write this part while doing it, so I've forgotten details of what I did when. The big thing was making the stack file in Verilog. We also made decisions about Milestone 3-related things as a group.

Saturday, October 26:

Fix RTL (~15 minutes)

While reviewing the RTL, I realized we'd forgotten to write it for two of our instructions, swap and kill. I wrote RTL for both and added some logic to the stack file to handle swaps.

Monday, October 28:

Class time (1 hour)

I went over the control signals in the datapath and rectified it with our design document, which had a slightly different list. We discussed removing the Push-Pop Control Unit since it was just annoying to take out of the main control, and we did.

Meet with team (~2 hours)

We noticed that we had forgotten about the branch decider, so I implemented it. I expanded on our integration plan and tests. I worked with Bill to make sure his plan for ALU control fit with the rest of the CPU.

Wednesday, October 30:

Meet with team (~2 hours)

We did the first step of our integration: PC, memory, and ALU. We made a file with the parts and tests. The tests aren't very exhaustive, but they cover the things that these parts can do alone.

Thursday, October 31:

Class time (1 hour)

Bill and I started our second step of integrating, the stack file.

Monday, November 4:

Meet with team (~3.5 hours)

We finished integrating the stack file. There were several timing-related problems to debug with transferring data from the stack to memory and vice versa, but we worked them out. We also wrote some tests to simulate pushing and popping and they passed. We then integrated the decode unit and immediate-extenders, which was fairly straightforward. Tests for those have not been written yet.

Tuesday, November 5:

Class time (1 hour)

We wrote tests for the decode unit and immediate-extenders, which they passed.

Meet with team (~3 hours)

We integrated and tested the register file.

Wednesday, November 6:

Meet with team (~2 hours)

We integrated and tested the branch logic. Then we attached the control, which had some timing issues to work out but we figured it out. We then tested a simple program with the completed CPU, and it worked.

Saturday, November 9:

Document interpreter (~15 minutes; making was ~4 hours)

I had made an interpreter/debugger for programs running on the processor for fun, which is not in the previous work logs because I forgot exactly when I worked on it. I added a readme, packaged it neatly in a .jar file, and included some sample programs. It's located in the /implementation/interpreter directory.

Monday, November 11:

Debug interpreter (~40 minutes)

After releasing the interpreter to the team, they found several bugs like "it doesn't think busn is an instruction" and "it crashed". I fixed those.

Class time (1 hour)

We worked on our instruction tests, fixed some bugs in them, and ran the one for B/J/R types on the processor. It passed.

Make FPGA interface (~1.5 hours)

I modified the ALUIO project from the course webpage so that we could put the processor into it. A lot of time was spent making it look nice; now the lcd displays "[input] Push W [output]". I tested it by putting a register in where the CPU should go and it worked as expected.

Tuesday, November 12:

Meet with group (~3.5 hours)

We finished running our system tests and attempted to import the CPU into the board project. But Xilinx doesn't work, so it failed. We deleted everything and decided to try again later.

Wednesday, November 13:

Class time (~2.5 hours)

We tried importing the CPU into the board project again, with moderate success. We managed to get the lcd to display input and output, but the output wasn't right.

Saturday, November 16:

Meet with team (~2.5 hours)

We finished working out the kinks in the FPGA, except for the fact that it skips the first instruction each time it resets and everything can break if the reset button gets pushed really fast. We made a rough draft of the slides. I finished making a program that computes sine.

Sunday, November 17:

Prettify the FPGA (~1 hour)

I modified our CPUAndBoard project to make a version that looks nicer. It only has four hex digits of input and says “Push E” between the input and output.

Meet with team (~3 hours)

We got the sine program to run on the FPGA, discovering a bug in our processor related to the offset for accessing memory. We fixed it, and it worked. We got the pretty CPUAndBoard to run on the FPGA, and got sine running on it. We finished the slides and split up parts to talk about. Unfortunately, Sarthak wasn’t there; hopefully he can come to our meeting tomorrow.

Monday, November 18:

Meet with team (~1 hour)

We all met and went through the presentation, and it worked pretty well. We combined all our files into one final report.

Sarthak Suri
Design Journal
CSSE232

Milestone 1 Work:

Saturday, October 5, 2019

Brainstorm potential instructions and architecture [30 minutes]

Saturday, October 5, 2019

Met with team [2 hours]

- Decided architecture of stack-based design, including separation of stack register file from special registers.
- Added temporary registers and peek (with integer argument) to allow faster access to frequently used variables without keeping them at the top of the stack
- Drafted a list of instructions and their arguments to figure out instruction type details
- Decided to use opcode/funct system for ALU instructions to minimize size of opcode while retaining all proposed instructions
- Went over procedure call convention
- Implemented GCD subroutine for relPrime

Sunday, October 6, 2019

Reviewed work from meeting [30 minutes]

- Couldn't attend team meeting due to a last-minute time change (from 6pm to 9am at 10pm the previous night)
- Looked over updates and made minor edits

Monday, October 7, 2019

Met with team [1.5 hours]

- Assigned opcode/funct values
- Converted assembly samples into machine code
- Documented procedure call convention
- Decided on tasks for Milestone 2

Tasks for milestone 2:

1. Draft RTL for A-type instructions (30 minutes)
2. Review other RTL drafts (60 minutes)
3. Brainstorm components and layout (30-60 minutes)

Milestone 2 Work:

Friday, October 11, 2019

Draft RTL for A-Type instructions [30 minutes]

Tuesday, October 15, 2019

Review RTLs and layout [40 minutes]

Milestone 3 Work:

Saturday, October 19, 2019

Met with team [1.5 hours]

- Verified RTL
- Designed datapath and discussed control signals.
- Divided component implementation work for Milestone 3.

Monday, October 21, 2019

Worked on components with team [1 hour]

Reviewed Verilog syntax and standard practices [30 minutes]

Implemented SE, ZP, ZE [15 minutes]

Initial register file design [30 minutes]

Tuesday, October 22, 2019

Updated design document [5 minutes]

Redrew datapath draft (not added to design document) [30 minutes]

Updated register file design [1 hour]

Wednesday, October 23, 2019

Updated design document [15 minutes]

Updated register file design [10 minutes]

Milestone 4 Work:

Friday, October 25, 2019

Divided tasks for M4 [30 minutes]

- Will draft state transition diagram for A, I and J type instructions

Saturday, October 26, 2019

Draft state transition diagram for A, I and J type instructions [1 hour]

Monday, October 28, 2019

Drew state transition diagram on whiteboard with Michael [2 hours]

- Productive to draw and discuss it as we found inaccuracies in the RTL
- Decided to format FSM as a table in design document for clarity

Updated Design Document [30 minutes]

- Translated rough diagram into table
- Michael asked me to stop as he noticed we could speed most instructions by popping early which required changes to the FSM

Wednesday, October 30, 2019

Updated testbench for SE, ZE, ZP [15 minutes]

Worked on first stage of integration plan with team [1 hour]

Milestone 5 Work:

Tuesday, November 5, 2019

We continued integrating the processor [3.3 hours]

- Finished step 3 test
- Integrated step 4
- Started step 4 test
- Made small edits to datapath based on tests

Wednesday, November 6, 2019

Worked on step 5 test [15 minutes]

- I caught a severe cold and couldn't meet at noon

Milestone 6 Work:

Sunday, November 10, 2019

Implemented I-type tests [1.5 hours]

Monday, November 11, 2019

Ran tests in class [1 hour]

Fixed bugs in tests [25 minutes]

Tuesday, November 12, 2019

Worked on miscellaneous tasks in class [1 hour]

Worked on possible interrupt implementations [45 minutes]

Sarthak Suri
Design Journal
CSSE232

Wednesday, November 13, 2019

Tried to run processor with relPrime on FPGA [2.5 hours]

Saturday, November 16, 2019

Worked on final report and presentation [3 hours]

Monday, November 18, 2019

Practiced presentation and ensured updated documents were in repo [1 hour]

Michael Zhao's Log

Meeting #17

Complete Processer 100%

Date	Duration
Nov 18	1 hr

Description

> Practiced presentation. Finalized everything.

Looking Ahead

> Presentation Tomorrow 8am

Meeting #16

Complete Processer 100%

Date	Duration
Nov 17	5 hrs

Description

> Finished putting sine on the board. Practiced presentation.

Looking Ahead

> We will practice presentation again on Monday

Meeting #15

Complete Processer 100%

Date	Duration
Nov 16	3 hrs

Description

> Fixed relPrime on board. Made final report and presentation slides.

Looking Ahead

> We will practice presentation on Sunday

Individual Work #14.75

Complete Processer 99%

Date	Duration
Nov 14	2 hrs

Description

> Reed and I figured out the problem with the CPU when it is on board. When the board is first programmed, the CPU works fine. However, every subsequent run with the reset button will skip the first instruction. We are considering two possible fixes: try to find the cause or force programmers to put nop as the first instruction.

Looking Ahead

> We will discuss this issue and work on the final report during our next meeting on Saturday.

Individual Work #14.5

Date	Duration
Nov 13	2 hrs

Description

> We tried to put the CPU on the board in class and I continued after class. It turns out as soon as we update the schematic files the project will break, because the project downloaded from the course website is using old xilinx. In the end I got the CPU on the board, but it doesn't produce the correct output -- It outputs 352 no matter what the input is.

Looking Ahead

> I will try other programs on the board and see if it works.

Meeting #14

Date	Duration
Nov 11	3 hrs

Description

> We ran all the tests for the processor. They all work now. What's interesting is that most of the time we are fixing bugs in the test, not the CPU it self. Also, verilog uses ">>>" for signed shift and ">>" for unsigned shift. Java and JavaScript do the opposite. We also collected the performance data and worked on updating the design doc.

Looking Ahead

> We will work on putting the processor on the FPGA board next meeting

Meeting #13

Date	Duration
Nov 11	1 hr

Description

> We assembled our test codes in class. Fixed a lot of bugs in the assembler.

Looking Ahead

> Tonight we will run the tests on our processor

Individual Work #12.5

Date	Duration
Nov 10	1.5 hrs

Description

> I implemented the A-type unit tests. It uses pushi, pui, ori to load the values and test them. The test code will output a 0 if fails, and a 1 if passes. We will use tomorrow's class time to do the unit tests.

Looking Ahead

> Hopefully tomorrow we can finish all unit tests in class, and we will work on larger tests tomorrow or Tuesday

Meeting #12

Date	Duration
Nov 10	2 hrs

Description

> We didn't plan it out very well. We were going to do unit tests for each instruction, but it will take too much group time. We decided to split up the unit tests and do them individually, while using the group time to do some bigger tests. So we tested a bigger program that involves a lot of recursive calls. (We used an input = 178, so the recursive level can go at least that. In theory it should go as deep as the memory can go) We are still going to do the unit tests though. The only bug we found was that pushr didn't work because register file was set to only read on the rising edge. It should always read.

Looking Ahead

> We will finish writing the unit tests and test them no later than Tuesday (we hope). I am assigned to do A-types. It will take about 1.5 hours.

Meeting #11

Date	Duration
Nov 6	2 hrs

Description

> We completed all integration and tested one program that worked. That's all we can do in the given time contraint. For the next milestone we will execute the system tests to test the processor thoroughly.

Looking Ahead

> Next milestone we will start testing the entire process, beginning with unit tests for each instruction

Individual Work #10.5

Date	Duration
Nov 6	1.5 hrs

Description

> I made the wave config files for testing part 5 and part 6 of the integration. Also, I removed the latch in SP and combinatorial loop in ALU

Looking Ahead

> I will meet with my team at 12pm to finish integrating everything and do some tests.

Meeting #10

Date	Duration
Nov 5	3.25 hrs

Description

> We completed the test for step 3, finished integrating step 4, and is half-way through testing step 4. We also noticed thta we never added SE shifted left 2 to the ALU, so we did that (there was conveniently an empty slot in the ALU source B Mux).

Looking Ahead

> We will work 12pm - 2pm on Wednesday to finish step 4 and 5. I will probably also work 10am to 12pm to make the wave forms in advance so that it is easier to debug during the tests.

Individual Work #9.5

Date	Duration
Nov 5	2 hrs

Description

> I completed the control unit implementation and tested it.

Looking Ahead

> Team meeting tonight to complete integration plan and tests

Meeting #9

Date	Duration
Nov 4	3.5 hrs

Description

> We tested part 2 of the integration with push and pops to stack and see if they spill over to the memory correctly. During the test we discovered that registers have weird delays which should not be there. We decided to slow down the clock to deal with the delay. We also integrated part 3, which we will test tomorrow in class time.

Looking Ahead

> Next Meeting is tomorrow during class. I will work on B-type control and the other members will test part 3 of the integration.

Individual Work #8.67

Date	Duration
Nov 4	2 hrs

Description

> Implemented and tested control for A-type instructions and I-type arithmetic instructions (i.e. not pushi, pui, pushs)

Looking Ahead

> Add pushi, pui, pushs tonight, then work with team during meeting to continue integrating

Individual Work #8.33

Date	Duration
Nov 4	0.5 hrs

Description

> I started implementing the control unit. Finished State 0, 1 and 2.

Looking Ahead

> Meet in class on Monday (today after I sleep and wake up) to discuss meeting time and maybe do more integration

Meeting #8

productivity 100% M4 100%

Date	Duration
Oct 30	2 hrs

Description

> We integrated PC, Mem and ALU (for incrementing PC). For testing, we loaded 5 instructions in the memory, set the PC to 0, and let the clock ran. The PC incremented correctly and the correct instruction was pulled out of the memory.

Looking Ahead

> Next meeting we will start M5, making control units and continue integrating most of the components, as well as writing more tests for integration.

Individual Work #7.67

productivity 100% M4 80%

Date	Duration
Oct 30	3 hrs

Description

> Met with Sid and discussed changes made to RTL. After I got the approval, I changed the transition diagram to match with the new RTL.

Looking Ahead

> We will integrate PC, Memory, IR and I think ALU (for incrementing PC) on Wednesday.

Individual Work #7.33

productivity 100% M4 40%

Date	Duration
Oct 29	1.25 hrs

Description

> As I went through the RTL I realized that POPs can be moved earlier to speed up things. Most instructions will have 1 less cycle!!!!!!! The thing is we have to change the transition diagram significantly. For now I will just update other sections of the document and hopefully my teammates see my message and respond.

Looking Ahead

> The document should be good except for the transition diagram, which I'm going to see what my teammates say about the changes to RTL. It should take 1-2 hours to make the necessary changes. On Wednesday we are going to meet (without Sarthak - he won't be free) to execute Part I of the integration plan.

Meeting #7

productivity 100% M4 30%

Date	Duration
Oct 28	2 hrs

Description

> Sarthak and I worked on the state transition diagram while Bill worked on ALU and ACU and Reed worked on Branch Decider and writing integration plans and tests. We have compiled a draft for the transition diagram. We decided that the transition diagram is too messy if shown as bubbles and arrows, so we made a table of states in which each row tells the current state, the current controls and what the next state should be depending on the instruction. Also, we settled down on a datapath (without control wires going everywhere). We found a lot of problems in the RTL and datapath as we drew the transition diagram. For example, POP actually needs 2 cycles to complete because it always involve a memory read and a stack write that depends on the memory read, whereas PUSH doesn't always depend on the memory or stack result. We also made a new control signal BDEnable that enables the Branch Decider.

Looking Ahead

> I will update the RTL according to the errors we found when we drew the transition diagram. This should take less than 1 hour.

Individual Work # 6.5

Date	Duration
Oct 25	3.5 hrs

Description

> I drew the complete datapath. I found out that branch decider as a control was left out, probably because we drew it too small the first time.

Looking Ahead

> I will write the state transition diagram tomorrow and fix things in M3.

Meeting #6

Date	Duration
Oct 25	0.5 hrs

Description

> We split up what we are going to do for M4. I will draw the datapath digitally and do the state transition diagram for B and R type instructions before meeting on Monday.

Looking Ahead

> Probably also going to fix things in M3. Everything together should take about 6 hours.

Individual Work # 5.75

Date	Duration
Oct 23	2 hrs

Description

> I regenerated the memory IP core to add a read enable and tested again for the no-read-when-write issue. I think it is working fine now. I finished my part of M3.

Looking Ahead

> We will discuss M4 on Saturday since Bill is not available R and F. We may discuss task assignment during meeting with Sid.

Individual Work # 5.5

Date	Duration
Oct 22	3.25 hrs

Description

> I spent 10 minutes testing the IRS and 3 hours testing the memory. One-third of the time I was waiting for iSim to elaborate. I figured out how to let the block memory write on the rising edge and read on the falling edge (by stick an inverter to the read port clock and use the same clock signal.) I also figured that if the read address is available before the falling edge of the cycle before Memstack then the read latency problem is solved since data will be available before the rising edge of the Memstack stage. It is like start reading in the execute stage, which is very cool. What's not cool is the elaborating time and I can't figure out how to turn off read-write address collision warning. (I know I set it up when creating the memory, but there seems to be no way to change the configuaration.

> Oh yeah and also the exhaustive test for memory **never** finished.

Looking Ahead

> Tomorrow we will meet in class and show off the parts we made, finish the design doc, hopefully also draw a pretty version of the datapath.

Individual Work # 5.25

Date	Duration
Oct 21	1 hr

Description

> I implemented the memory unit using the simple dual port RAM. I also implemented the decode unit a.k.a IR Splitter (IRS) by connecting wires.

Looking Ahead

> I will make the tests before the meeting on Wednesday. It will take approximately 2 hours.

Meeting #5

productivity 90% M3 10%

Date	Duration
Oct 19	1.5 hrs

Description

> We drew the multi-cycle datapath for our processor.

Looking Ahead

> We split the jobs for implementing each component. I am doing memory and a instruction decoder. It will take approximately 1 hour.

Individual Work #4.5

productivity 80% M2 85%

Date	Duration
Oct 15	1.25 hrs

Description

> Finished modifying section 2 (adding instruction tables to each type). Also changed the formatting of RTL a little bit so that the table is not too big to read.

Looking Ahead

> Most likely won't be able to meet with team before milestone due. We will probably meet on Wednesday to sort things out.

Meeting #4

productivity 85% M2 80%

Date	Duration
Oct 14	2.5 hrs

Description

> We spent about 1.25 hours trying to modify the design doc according to suggestions made during meeting with instructor. We didn't finish changing all of it. Then we spent another 1.25 hours merging the RTLs we did over the break, Sarthak didn't show up so we don't have RTL for A-types yet. Then we brainstormed and wrote about components for about 45 minutes. It was running late so we didn't do the error double check.

Looking Ahead

> I will finish modifying the design doc according to suggestions. It will take approximately 0.75 hours. Hopefully Sarthak shows up tomorrow and meet with the other two people since I won't be free tomorrow at all before 5pm.

Individual Work #3.5

productivity 100% M2 10%

Date	Duration

Date	Duration
Oct 12	0.75 hrs

Description

> I wrote RTL for B- and J-types.

Looking Ahead

> Next meeting on Monday, we will merge our RTL into one giant table

Meeting #3

Date	Duration
Oct 7	1.25 hrs

Description

> We textized procedure call conventions and how to translate to machine language from instructions.

Looking Ahead

> We will start M2 next meeting. I am assigned to do RTL for B- and J-Type instructions (0.5-1 hr). We will meet on Wednesday

Meeting #2

Date	Duration
Oct 6	2.75 hrs

Description

> We discussed which instruction formats for which instructions. Then we wrote the green sheet for our instruction set. We also briefly talked about register stack spill over. There will be 32 stack registers wired to each other to make push and pop happen. The last register will read and write from memory with a stack pointer register that points to the stack in memory. The stack register can also be accessed by register index and this makes `pushs` very easy to implement in hardware (at least we think). We finished compiling `relPrime` into assembly and wrote some fragments. We already have a call convention but didn't write it in the doc yet. Also, we defined addressing modes for each type of instruction.

Looking Ahead

> We will textize procedure calls in the next meeting.

Meeting #1

Date	Duration
Oct 5	1.5 hrs

Description

> We started by brainstorming intructions. We decided that a big disadvantage of stack is that you can only access the topmost element. So we will have a few (3 for now) temporary registers. However, all operations like add, sub, etc will happen on the stack only. We will have two register files, one for stack and one for other stuff. We implemented `gcd(a, b)` using our instruction set and it was pretty simple. We are happy with our design (for now). Near the end of the meeting we started to design instruction formats. All instructions will be 16-bits. For now we have 6 bits of opcode and 4 types of instructions. We might reduce the number of types because a lot of them are similar. We are also going to use something like the R-type in MIPS where the opcodes are the same and funct tells the ALU what to do. There will be a no-pop flag because a lot of our instructions include variants that allows the programmer to choose between poping the operands or not. We also invented a procedure call convention.

Looking Ahead

> We will finish sorting out the instructions in next meeting and implement `relPrime(a,b)`