

Processor Final Report

Stack Architecture



Group 2C

Rias Hilliard, Benjamin Hall, Luke Evard, Dominic Grazioli

# Executive Summary

This processor is a stack-based, multicycle architecture. The programmer cannot directly modify any specific registers, rather, there is a stack in which all operations are completed. The main, or data stack holds the written and saved data from instruction computations and assignments. This design also has a second stack within memory, the return stack. The return stack is used for storing arguments and return values and addresses across function calls.

The architecture includes 16-bit I/O as well as a 1-bit reset input. The input value can be pushed onto the data stack using the instruction *in*. The top of the stack can be popped to the output register using the *out* instruction, which continuously outputs the value popped.

This processor has three different instruction types: S, I, and J. S-Types modify the data or return stack. I-Types have immediate values in the first 8 bits of the instruction which are utilized in different ways. J-Types are all jump instructions. Each has a 4-bit opcode for the control unit to output the proper control signals. S and I Types also have an additional 4-bits to specify their functionality.

# Report

**Introduction**

This processor is a stack-based, multicycle processor. The programmer is able to use an assembly language made for this processor using all the instruction names and descriptions. With these instructions, they are able to modify the stack for storing data (DataStack) while also calling functions and jumping to other lines of code and back (using the ReturnStack). There are 3 types of instructions they can use to do calculations, logical operations, jumping lines, and putting immediates on the DataStack. The programmer is also able to use a simple I/O system to reset or give data to the processor, then output some data at any specified time.

**Instruction Set Design**

The instructions chosen were based on MIPs instruction set with additional instructions useful for handling the stack as well as I/O use. They include basic arithmetic and logical instructions for building any program or algorithm as well as other instructions for handling conveniences. There are 3 types of instructions: S-Type (DataStack and ReturnStack modification), I-Type (instructions passing in an 8-bit immediate), and J-Types (use the ReturnStack to go to a different line of instruction). The instructions are in 16-bits primarily using the layout:

|  |  |  |
| --- | --- | --- |
| opcode (4-bits) | function (4-bits) | immediate (8-bits) |

with exception to J-Types, which have 12-bit immediates to use for a larger jump in the instructions.

The MSB, [15], is a 1 if the instruction uses an immediate value; it is only 0 for S-Type instructions. The next bit, [14], is a 1 if the instruction uses a jump, and so is only set for 1 for J-Type instructions. The last 2-bits of the opcode are call the function type bits. For S-Types, these are specified by the following:

00 function type pertains to logical, bitwise, and arithmetic instructions,

01 to memory instructions,

10 to DataStack manipulating functions, and

11 to the ‘ja’ instruction

I-Type function bits are always 00 because they do not have to specify anything in particular because the function code handles it. J-Type uses the last 2-bits of the opcode to specify the instruction itself, because the rest of the instruction is used for a 12-bit immediate address.

**Implementation**

After refining our instructions and having the design of our processor completely thought out, we started to put everything together. First, we began by creating RTL for each instruction, keeping in mind that our processor would be multicycle. We simplified the RTL by combining cycles that overlapped between instructions. For example, every single instruction shared the same first two cycles. With a fully functional RTL were able to start building our datapath. We built the datapath step by step starting with the ALU operations. From there we would pick instructions and add the necessary wires, registers, and components so that the datapath could support that instruction. With a completed datapath we could finally start building the basic components in Verilog. We created all of the components that were used in the datapath including, a single register, a register file, memory, ALU, and an adder. Once all of the basic components had been created and tested, we wired certain parts that interacted directly with each other in the datapath to form logical subsystems. The single register component, along with a few other parts, were used multiple times in the Verilog as generic components for specific parts to minimize code duplication. We finally wired all of the subsystems together and added the control unit to provide all of the control signals for a fully functional datapath. Some things were later revised in the wiring of the datapath, specifically to provide support for input and output as well as a fixing a few bugs we encountered.

**Testing Methodology**

Tests were written before implementation for individual parts, like the ALU, multiplexors, etc. Then, after the parts were implemented, they were tested exhaustively to ensure that all possible combinations output the correct results. However, as we scaled up to larger subsystems of the datapath, we had to test for normal behavior and edge cases, since exhaustive testing is impractical with so many possible inputs.

**Conclusion**

By using a top-down design approach, we were able to design a functional processor. This processor supports all simple programs, but does not fully support large call depths. By designing the RTL for each instruction, we were able to design our datapath and control units robustly. The level of detail provided by generating the requirements for the individual pieces of our datapath allowed us to create detailed tests to ensure the proper functionality of each piece. Once the entire datapath was wired correctly, very few individual components required editing.

# Test Results

1. Number of Bytes: 96
2. Number of Instructions: 132694
3. Number of Cycles: 540869
4. CPI: 4.076
5. Cycle time: 8.841 ns
6. Execution Time: 4.782 ms
7. Gate Count: <N/A>
8. Device Utilization Summary:

Number of Slices: 3734 out of 4656 80%

Number of Slice Flip Flops: 4263 out of 9312 45%

Number of 4 input LUTs: 3174 out of 9312 34%

Number of IOs: 36

Number of bonded IOBs: 36 out of 232 15%

Number of BRAMs: 15 out of 20 75%

Number of GCLKs: 2 out of 24 8%

# Appendix A: Design Document

# Structures

## DataStack

The DataStack is our processor’s main stack in our stack architecture. The DataStack is register based in the register file contained within the CPU. We currently only use 8 registers in the register file to act as the stack. More will be added when we have more time. Exceeding this will cause a stack overflow exception, thus it is the programmer’s responsibility for stack maintenance and moving values into memory when necessary. The DP register keeps track of the top of the DataStack.

However, the top of the DataStack actually is the second value in the programming stack. This is because the TR register is a buffer that acts as the top programming stack. The stack is as follows:

|  |
| --- |
| **DataStack** |
| TR |
| DP |
| DP-1 |
| DP-2 |
| : |
| DP-n |

## ReturnStack

The ReturnStack is our processor’s memory stack in our stack architecture. This stack functions as a call stack, but the programmer may use it in creative ways as well. The ReturnStack is located in memory and thus has an enormous capacity. The RP register keeps track of the top of the ReturnStack.

## Registers

The programmer cannot use or modify any register explicitly. The instruction does all the register modifications and manipulations internally.

## Input/Output

Our processor has basic input/output functionality. There are two user inputs to the datapath: reset [1-bit], and input [16-bits]. Reset will trigger a full reset of the datapath and set all registers to default values. Input is a value that can be pushed to the stack using the *in* instruction. This input value can then be used within the program for whatever the programmer wants. There is one output from the datapath: output [16-bits]. The programmer may use the *out* instruction to pop the top value from the stack to the output register, which will continuously output that value until *out* is called again or the datapath is reset.

# Call Conventions

## Return

We use the ‘ja’ instruction to jump to the 16-bit address at the top of the DataStack. The programmer uses the ‘ja’ instruction as to return from a procedure. Conventions call for the DataStack to be preserved across the procedure call, meaning that the programmer is responsible for removing all values used in the DataStack, if any, before finally using the ‘ja’ to return.

The programmer must push the return values onto the ReturnStack in reverse order so that the value at the top of the ReturnStack is R1 and at the bottom of the function return frame is Rn. There is no limit to the number of return values for a function as long as there is room.

## Function Calling

We use the ‘call’ instruction, which uses PC-relative addressing, to jump to a procedure address. To keep up with conventions, the programmer must push all the arguments onto the ReturnStack in reverse order. This is so the first argument is at the top of the ReturnStack and the last argument is at the bottom of the function argument frame. The ‘call’ instruction will place the return address at the top of the ReturnStack. It is the programmer’s responsibility to move the return address from the ReturnStack to the DataStack with the ‘fromr’ instruction so that the programmer can use ‘ja’ instruction to return.

# Addressing

## Memory Addressing

Our processor will use 16-bit word addressing.

## Addressing Modes

We will only be using PC-relative addressing modes when necessary. The instructions ‘j,’ ‘ja,’ ‘jeq,’ ‘jneq,’ and ‘call,’ are the only instruction requiring the use of PC-relative addressing modes.

# Instruction Format and Syntax

## S-Type

All S-Type instructions will manipulate the DataStack or ReturnStack in some way. This includes all arithmetic, logical, and memory instructions. The S-Type has a patterned 4-bit opcode and a 4-bit function code. We decided to pattern the opcode for several reasons, but ultimately the pattern will simplify the logic for our control unit. The other 8-bits in the instruction are unused by S-Type instructions. The S-type format is as follows:

|  |  |  |
| --- | --- | --- |
| opcode (4-bits) | function (4-bits) | unused (8-bits) |

The S-Type opcode has a pattern. The most significant bit (MSB), [15], will always be a 0 to signify that the instruction does not use an immediate value and is thus an S-type instruction. The next bit, [14], is the jump bit, which indicates if the instruction is a jump instruction. If this bit is 0, then a jump will not happen, and if it is 1, then a jump will happen. Within the S-Type, the only instruction with a 1 jump bit is ‘ja.’ The final two bits in the opcode, [13:12], specify the function type of the instruction. This is so that we can use the same values in the function section, but the instruction is different based on these two bits. The 00 function type pertains to logical, bitwise, and arithmetic instructions, 01 to memory instructions, 10 to DataStack manipulating functions, and 11 to the ‘ja’ instruction. The following four bits, [11:8], are the function codes that specify which instruction is being performed based on the function type. The function type and functions are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | function type (2-bit) | | | |
|  | 00 | 01 | 10 | 11 |
| function code |  | | | |
| 0x0 | reset | | | |
| 0x1 | add | load | burn | ja |
| 0x2 | sub | store | dup |  |
| 0x3 | and | tor | over |  |
| 0x4 | or | fromr | swap |  |
| 0x5 | slt |  | out |  |
| 0x6 | band |  | in |  |
| 0x7 | bor |  |  |  |
| 0x8 | bnor |  |  |  |
| 0x9 | bxor |  |  |  |

## I-Type

All I-type instruction use immediate values in their operations. The I-Type instructions have an 8-bit immediate in addition to a patterned 4-bit opcode and 4-bit function code. We decided to pattern the opcode for several reasons, but ultimately the pattern will simplify the logic for our control unit. The format is as follows:

|  |  |  |
| --- | --- | --- |
| opcode (4-bits) | function (4-bits) | immediate (8-bits) |

The I-Type opcode has a similar pattern to the S-Type opcode. The MSB, [15], will always be 1 to indicate that this instruction uses an immediate value. The next bit, [14], is the jump bit, which indicates if the instructions uses a jump. In an I-Type instruction, this bit will always be 0. The final two bits in the opcode, [13:12], specify the function type of the instruction. This is so that we can use the same values in the function section, but the instruction is different based on these two bits. The I-Type instructions will always have 00 in these final two bits. This is because there are not enough I-Type instructions to warrant splitting up functions and function types. The following four bits, [11:8], are the function codes that specify which instruction is being performed based on the function type. After these bits is the final 8-bits that is the immediate value for the operation. The function type and functions are as follows:

|  |  |
| --- | --- |
| function code | Instruction |
| 0x0 | push |
| 0x1 | pushu |
| 0xA | sll |
| 0xB | srl |
| 0xC | sra |

## J-Type

J-Type instructions are all jump instructions that use an immediate value. The J-Type instructions have a patterned 4-bit opcode and a 12-bit immediate value. We decided to pattern the opcode for several reasons, but ultimately the pattern will simplify the logic for our control unit. The format is as follows:

|  |  |
| --- | --- |
| opcode (4-bit) | immediate (12-bit) |

The J-Type opcode has a slightly different pattern than the previous types. The MSB, [15], is a 1 again to signify that it uses an immediate value. The next bit, [14], is the jump bit, which indicates if the instructions uses a jump. In an J-Type instruction, this bit will always be 1. The final two bits in the opcode, [13:12], are the jump operation codes, which specify which type of jump instruction that we perform. Since every jump instruction uses a PC-relative addressing mode, the last 12-bits are the offset immediate. The jump operation codes are as follows:

|  |  |
| --- | --- |
| jump operation code | Instruction |
| 00 | jeq |
| 01 | jneq |
| 10 | j |
| 11 | call |

# Instructions

## S-Type

**Nop**

|  |  |  |  |
| --- | --- | --- | --- |
| *nop* | 0x0 | 0x0 | 0x00 |

The no operation instruction. This is the default instruction for the instruction register. This should never be called by the programmer.

**Addition**

|  |  |  |  |
| --- | --- | --- | --- |
| *add* | 0x0 | 0x1 | 0x00 |

Pops off the top two values on the DataStack, adds them together, and puts the result onto the top of the DataStack.

**Subtraction**

|  |  |  |  |
| --- | --- | --- | --- |
| *sub* | 0x0 | 0x2 | 0x00 |

Pops off the top two values on the DataStack, subtracts the top value from the second value, and puts the result onto the top of the DataStack.

**Logical AND**

|  |  |  |  |
| --- | --- | --- | --- |
| *and* | 0x0 | 0x3 | 0x00 |

Pops off the top two values on the DataStack, performs a logical AND, then puts the result onto the top of the DataStack.

**Logical OR**

|  |  |  |  |
| --- | --- | --- | --- |
| *or* | 0x0 | 0x4 | 0x00 |

Pops off the top two values on the DataStack, performs a logical OR, then puts the result onto the top of the DataStack.

**Set Less Than**

|  |  |  |  |
| --- | --- | --- | --- |
| *slt* | 0x0 | 0x5 | 0x00 |

Pops the two values from the Data Stack and compares them. Pushes a 1 onto the top of the DataStack if the top value is less than the second value, and 0 otherwise.

**Bitwise AND**

|  |  |  |  |
| --- | --- | --- | --- |
| *band* | 0x0 | 0x6 | 0x00 |

Pops the top two values from the DataStack, does a bitwise AND comparison, and pushes the result onto the top of the DataStack.

**Bitwise OR**

|  |  |  |  |
| --- | --- | --- | --- |
| *bor* | 0x0 | 0x7 | 0x00 |

Pops the top two values from the DataStack, does a bitwise OR comparison, and pushes the result onto the top of the DataStack.

**Bitwise NOR**

|  |  |  |  |
| --- | --- | --- | --- |
| *bnor* | 0x0 | 0x8 | 0x00 |

Pops the top two values from the DataStack, does a bitwise NOR comparison, and pushes the result onto to the top of the DataStack.

**Bitwise XOR**

|  |  |  |  |
| --- | --- | --- | --- |
| *bxor* | 0x0 | 0x9 | 0x00 |

Pops the top two values from the DataStack, does a bitwise XOR comparison, and pushes the result onto the top of the DataStack.

**Load**

|  |  |  |  |
| --- | --- | --- | --- |
| *load* | 0x1 | 0x1 | 0x00 |

Pops the address at the top of the DataStack and then pushes the value from that address in memory onto the top of the DataStack.

**Store**

|  |  |  |  |
| --- | --- | --- | --- |
| *store* | 0x1 | 0x2 | 0x00 |

The value to store is at the top of the DataStack and the memory address is the second value in the DataStack. Pop the value and address from the Data Stack. Store the value into the address in memory.

**Pop to ReturnStack**

|  |  |  |  |
| --- | --- | --- | --- |
| *tor* | 0x1 | 0x3 | 0x00 |

Pops the top value of the DataStack and push it onto the ReturnStack.

**Push from ReturnStack**

|  |  |  |  |
| --- | --- | --- | --- |
| *fromr* | 0x1 | 0x4 | 0x00 |

Pops the top value of the ReturnStack and push it onto the DataStack.

**Burn**

|  |  |  |  |
| --- | --- | --- | --- |
| *burn* | 0x2 | 0x1 | 0x00 |

Gets rid of the top value of the DataStack. Essentially pops the top value into nothingness.

**Duplicate**

|  |  |  |  |
| --- | --- | --- | --- |
| *dup* | 0x2 | 0x2 | 0x00 |

Duplicates the top value of DataStack and then pushes it onto the top of the DataStack.

**Over**

|  |  |  |  |
| --- | --- | --- | --- |
| *over* | 0x2 | 0x3 | 0x00 |

Duplicates the value of the second element in the DataStack and then pushes it onto the top of the DataStack.

**Swap**

|  |  |  |  |
| --- | --- | --- | --- |
| *swap* | 0x2 | 0x4 | 0x00 |

Swaps the top two values of the DataStack. The top value becomes the second value and the second value is now at the top of the Data Stack.

**Out**

|  |  |  |  |
| --- | --- | --- | --- |
| *out* | 0x2 | 0x5 | 0x00 |

Pops the value from the top of the data stack to the OUT register, which continuously outputs from the datapath.

**In**

|  |  |  |  |
| --- | --- | --- | --- |
| *in* | 0x2 | 0x6 | 0x00 |

Pushes the user input value onto the data stack.

**Jump to Address**

|  |  |  |  |
| --- | --- | --- | --- |
| *ja* | 0x3 | 0x1 | 0x00 |

Pops the address at the top of the DataStack and then jumps to that address in the program.

## I-Type

**Push Immediate**

|  |  |  |  |
| --- | --- | --- | --- |
| *push imm* | 0x8 | 0x0 | imm (8-bit) |

Pushes the value of the immediate onto the DataStack.

**Push Upper Immediate**

|  |  |  |  |
| --- | --- | --- | --- |
| *pushu imm* | 0x8 | 0x1 | imm (8-bit) |

Pushes the upper 8 bits of an immediate onto the DataStack.

**Shift Left Logical**

|  |  |  |  |
| --- | --- | --- | --- |
| *sll shamt* | 0x8 | 0xA | imm (8-bit) |

Pops the top value in the DataStack, shifts the value left by the shamt (shift amount), and pushes the new value onto the top of the DataStack.

**Shift Right Logical**

|  |  |  |  |
| --- | --- | --- | --- |
| *srl shamt* | 0x8 | 0xB | imm (8-bit) |

Pops the top value in the DataStack, zero extends the value right by the shamt (shift amount), and pushes the new value onto the top of the DataStack.

**Shift Right Arithmetic**

|  |  |  |  |
| --- | --- | --- | --- |
| *sra shamt* | 0x8 | 0xC | imm (8-bit) |

Pops the top value in the DataStack, sign extends the value right by the shamt (shift amount), and pushes the new value onto the top of the DataStack.

## J-Type

**Jump if Equal**

|  |  |  |
| --- | --- | --- |
| *jeq label* | 0xC | offset (12-bit) |

Pop and compare the top two values in the DataStack. If they are equal, then jump to the label using PC-relative addressing.

**Jump if Not Equal**

|  |  |  |
| --- | --- | --- |
| *jneq label* | 0xD | offset (12-bit) |

Pop and compare the top two values in the DataStack. If they are not equal to zero, then jump to the label using PC-relative addressing.

**Jump to Label**

|  |  |  |
| --- | --- | --- |
| *j label* | 0xE | offset (12-bit) |

Unconditionally jump to the label using PC-relative addressing.

**Procedure Call**

|  |  |  |
| --- | --- | --- |
| *call label* | 0xF | offset (12-bit) |

Jump to the procedure address given by the label, using PC relative addressing. Place the return address onto the top of the ReturnStack.

# Assembly Language Example Fragments

|  |  |  |
| --- | --- | --- |
| Assembly Language Fragment Examples | | |
| Code | Assembly Code | Machine Code |
| 1+2 | push 1  push 2  add | 0001011100000001  0001011100000010  0000000100000000 |
| if (1 == 2)  goto addr | push 1  push 2  sub  jez addr | 0001011100000001  0001011100000010  0000001000000000  00010101 offset |
| if (1 > 2)  goto addr | push 1  push 2  slt  jnez addr | 0001011100000001  0001011100000010  0000100000000000  00010110 offset |
| Mem[addr] = TOP | store | 0001001000000000 |
| TOP = Mem[addr] | load | 0001000100000000 |
| !1 | push 1  push 0  bnor | 0001011100000001  0001011100000000  0000101100000000 |

# Euclid’s Algorithm

|  |  |
| --- | --- |
| Assembly | Machine Code |
| relPrime:  fromr // Pop n from top of return stack AS AN INPUT and push it to the top of the data stack  push 2 // Push 2 to stack (m)  over // Copy n to top  over // Copy m to top  primeLoop:  tor // Move m to return stack  tor // Move n to return stack  call gcd // Put fp and ra onto return stack  fromr // Get return value from rs  push 1 // Push 1 to stack  jeq endPrime //If return value == 1 goto return  push 1  add // Increment m  over // Copy n to top  over // Copy m to top  j primeLoop // Jump to start of loop  endPrime:  tor // Move m to top of rs  burn // Remove n from stack  out // Output m from relPrime  j end // Jump to memory at Address at top of stack (ra)  end:  j end // Loop | 0001010000000000 // relPrime: fromr  1000000000000010 // push 2  0010001100000000 // over  0010001100000000 // over  0001001100000000 // primeLoop: tor  0001001100000000 // tor  1111000000001101 // call gcd  0001010000000000 // fromr  1000000000000001 // push 1  1100000000000101 // jeq endPrime  1000000000000001 // push 1  0000000100000000 // add  0010001100000000 // over  0010001100000000 // over  1110111111110101 // j primeLoop  0001001100000000 // endPrime: tor  0010000100000000 // burn  0000101000000000 // end: out  1110000000000000 // j end  1110111111111110 // j end |
| gcd:  fromr // Pop ra from top of return stack  fromr // Pop b from top of return stack  fromr // Pop a from top of return stack  dup // Make a copy of a  push 0  jneq gcd\_a // if a != 0, don't return b  burn // Burn a  tor // Push b to return stack  ja // Branch to ra  gcd\_a:  over // copy b to top of stack  push 0  sub  jeq gcd\_return // if b == 0, return a  over  over  swap // b is now on top of the stack  slt // evaluates (b < a)  push 1  sub  bez gcd\_if // branch to the if clause if b<a  swap // swap a and b  over // copy a to top of stack  sub // b = b - a  swap  j gcd\_a // go back to top of loop    gcd\_if:  over // copy b to top of stack  sub // a = a - b  j gcd\_a // go back to top of the loop  gcd\_return:  tor // Push a to return stack  burn // Burn b  ja // return | 0001010000000000 // gcd: fromr  0001010000000000 // fromr  0001010000000000 // fromr  0010001000000000 // dup  1000000000000000 // push 0  1101000000000011 // jneq gcd\_a  0010000100000000 // burn  0001001100000000 // tor  0011000100000000 // ja  0010001100000000 // gcd\_a: over  1000000000000000 // push 0  1100000000001111 // jeq gcd\_return  0010001100000000 // over  0010001100000000 // over  0000010100000000 // slt  1000000000000001 // push 1  0000001000000000 // sub  0010010000000000 // swap  1100000000000101 // jeq gcd\_if  0010010000000000 // swap  0010001100000000 // over  0000001000000000 // sub  0010010000000000 // swap  1110111111110001 // j gcd\_a  0010001100000000 // gcd\_if: over  0000001000000000 // sub  1110111111101110 // j gcd\_a  0001001100000000 // tor  0010000100000000 // burn  0011000100000000 // ja |

# Register Transfer Language (RTL)

\* Note: the DS in the RTL refers to the portion of the DataStack implemented using a register file.

\* Note 2: The table is an excel file, but was split up to fit comfortably into this document







# Datapath Components

**ALU:** The Arithmetic Logic Unit. The ALU will take in two 16-bit inputs, A and B, and will output a result based on the ALUop control signal. The isZero output will be a 1 if the result is zero, and 0 otherwise. The overflow output will be a 1 when an arithmetic overflow occurred, and 0 otherwise.

* **Inputs**
  + [16-bit] A – operand
  + [16-bit] B – operand
* **Outputs**
  + [16-bit] R – result
  + [1-bit] isZero – zero signifier
  + [1-bit] overflow – signifies if arithmetic overflow
* **Control Signals**
  + [4-bits] ALUop – Determines the ALU operation
    - 0000 – add
    - 0001 – sub
    - 0010 – && (logical and)
    - 0011 – || (logical or)
    - 0100 – slt
    - 0101 – & (bitwise and)
    - 0110 – | (bitwise or)
    - 0111 – !| (bitwise nor)
    - 1000 – XOR
    - 1001 – sll
    - 1010 – srl
    - 1011 – sra
* **Symbols**
  + ALU

**Adder:** The Adder. The adder will take in two 16-bit inputs, A and B, and add them together. The output is their sum, R.

* **Inputs**
  + [16-bit] A – operand
  + [16-bit] B – operand
* **Outputs**
  + [16-bit] R – result
* **Control Signals**
  + N/A
* **Symbols**
  + **‘+’**

**Sign Extender:** The zero extender will accept an 8-bit immediate value and sign extend it by taking the immediate’s MSB and placing it before the immediate it eight times to sign extend the immediate.

* **Inputs**
  + [8-bit] ImmIN – immediate value
* **Outputs**
  + [16-bit] ImmOUT - sign extended immediate value
* **Control Signals**
  + N/A
* **Symbols**
  + SE

**Zero Extender:** The zero extender will accept an 8-bit immediate value and zero extend it by placing eight zeroes before the immediate to zero extend the immediate.

* **Inputs**
  + [8-bit] ImmIN – immediate value
* **Outputs**
  + [16-bit] ImmOUT - zero extended immediate value
* **Control Signals**
  + N/A
* **Symbols**
  + ZE

**8-bit Left Shifter:** The 8-bit left shifter will accept an 8-bit immediate value and tack on eight zeroes after the immediate to shift it left by 8 bits. It will output a 16-bit immediate left shifted version of the immediate.

* **Inputs**
  + [8-bit] ImmIN – immediate value
* **Outputs**
  + [16-bit] ImmOUT - immediate value left shifted by 8 bits
* **Control Signals**
  + N/A
* **Symbols**
  + <<8

**Register:** The Register. This stores and holds values. It has a 16-bit input for a new value that will only overwrite the current register value if enabled (1) by the enabled bit. It will always output its current value.

* **Inputs**
  + [16-bit] NewVal – the new value of the register
* **Outputs**
  + [16-bit] Val – the current value in the register
* **Control Signals**
  + [1-bit] Enabler – enables/disables the changing of the register value
* **Symbols**
  + **B** – Stores the output from ALUsrc2 mux. The ALU uses the value of B
  + **TR** **(Top Register)** – Holds the value at the top of the DataStack
  + **DP (DataStack Pointer)** – Holds the pointer value to the top of the register stack
  + **RP (ReturnStack Pointer)** – Holds the pointer value to the top of the ReturnStack
  + **BR (Branch Register)** – Stores the jump target address from the ALU across cycles
  + **PC (Program Counter)** – The program counter pointing to instructions in memory
  + **IR (Instruction Register)** – Holds the current instruction

**Register File:** A Register File. This is a series of registers in the CPU. The Register File will take the inputs of the Addr and WriteData. If RegWrite is enabled, then there will be no output and the value from the WriteData input will be stored into memory at the given address. If RegRead is enabled, then the register file will output, in ReadData, the value at the given memory address.

* **Inputs**
  + [16-bit] Addr – memory address
  + [16-bit] WriteData – the data to write at a memory address
* **Outputs**
  + [16-bit] ReadData – the data read from a memory address
* **Control Signals**
  + [1-bit] RegWrite – enables/disables writing to memory
  + [1-bit] RegRead – enables/disables reading from memory
* **Symbols**
  + Reg File

# Integration

## Unit Tests

Exhaustively test each datapath component unit by looping through all possible combinations of inputs and checking that the output is as expected. We also test our muxes to make sure our hand-made muxes worked as expected. The basic description for our unit-testing plan is as follows:

**Memory:** For the Memory tests, we will start with a hardcoded set of memory values. When the MemWrite enabler is 1, then we expect the value at the given input memory address to change to the value of the input WriteData in the memory unit. If the MemWrite is disabled then nothing should be changed.

**Register File:** We will test the register file similar to how the memory was tested. We will hard code in some values, and then test writing and reading with different enable bit combinations.

**Register:** Test the register by checking the value over clock cycles. We will test writing to the register with the enabled bit on and expect a change and then with it off and expect the value to stay the same.

**Adder:** We test all combinations of inputs for the adder and check for the right sum and overflow values.

**ALU:** Exhaustively test all possible combination of input values with every possible ALUop to ensure that the ALU outputs the expected result, overflow, and isZero values.

**Sign Extender:** We input all possible 28 bit combinations into the sign extender and expect each one to be sign extended properly.

**Zero Extender:** We input all possible 28 bit combinations into the zero extender and expect each one to be zero extended properly.

**8-bit left shifter**: We will test left and right shifting by moving around a 1 with zeroes and moving around 0 through 1s. Essentially, check that shifts are actually happening.

**Mux:** Iterate through possible mux codes, ensuring that the output switches to the relevant inputs correctly.

## Integration Plan and Tests

The components used in the datapath are placed into their own smaller subsystems. We test these subsystems are by comparing the inputs and outputs of the subsystems with the unit tests for each individual component. Once each subsystem has been exhaustively and successfully tested using the unit tests as reference, the subsystems will be combined into larger systems, with tests performed on them in a similar fashion. This process will continue until we end up successfully testing the entire datapath as a whole. Below are the descriptions of each tests input with expected outputs:

|  |  |  |
| --- | --- | --- |
| **Parts in Basic Subsystem** | **Inputs** | **Outputs** |
| TR subsystem:   * TR, TRSrc MUX | Uses inputs from unit tests for the TRSrc MUX. | Compares the output of the subsystem with the output of the unit test for the TR register. |
| DP Adder subsystem:   * DP, Basic Adder, DP MUX | Uses inputs from unit tests for the DP MUX and the one input in the adder not reliant on the DP MUX. | Compares the output of the subsystem with the output of the unit test for the DP register. |
| RP Adder subsystem   * RP, Basic Adder, RP MUX | Uses inputs from unit tests for the RP MUX and the one input in the adder not reliant on the RP MUX. | Compares the output of the subsystem with the output of the unit test for the RP register. |
| PC Adder subsystem   * PC, Basic Adder, PC MUX | Uses input from unit tests for the basic adder, the input into the MUX not reliant on the adder, the control signal into the PC MUX and PC register. | Compares the output of the subsystem with the output of the unit test for the PC register. |
| ALU subsystem   * ALU, ALUSrc MUX, BSrc MUX B register, BR register | Uses inputs from unit tests for the ALUSrc MUX and the B register as well the control signal to decide the operation in the ALU. | Compares the output of the subsystem with the three outputs of the unit test for the ALU and BR register. |
| Mem subsystem   * Mem, MemAddr MUX, MemData MUX | Uses inputs from unit tests for the MemAddr MUX and MemData MUX as well as the two control signals for the Mem. | Compares the output of the subsystem with the output of the unit test for Mem. |
| B Subsystem   * B register, BSrc MUX | Uses inputs from unit tests for the BSrc MUX | Compares the output of the subsystem with the output of the unit test for the TR register. |

|  |  |  |
| --- | --- | --- |
| **Parts in Gen2 Subsystems** | **Inputs** | **Outputs** |
| DataStack subsystem:   * Reg File, TR subsystem, DP Adder subsystem | Uses input from the integrated tests for the TR subsystem and DP Adder subsystem as well as the control signals for the Reg File from its unit tests. | Compares the output of the subsystem with the output of the unit test for the Reg File. |
| AdvancedMemory subsystem:   * Mem subsystem * PC Adder subsystem * RP Adder subsystem | Uses input from the integrated tests for the PC Adder subsystem and RP Adder subsystem, and where necessary from the Mem subsystem. | Compares the output of the subsystem with the output of the integrated test for the Mem subsystem. |

|  |  |  |
| --- | --- | --- |
| **Final Datapath** | **Inputs** | **Outputs** |
| Datapath:   * Mem Gen2 subsystem, Reg File subsystem, ALU subsystem, IR, B Subsystem, Control Unit, OUT register | Determines input based on each individual instruction. | Compares output to what is expected to happen from the description of the instruction. |

## System Test Plan

We will test our system in three phases: datapath test without control, datapath test with control, small program test. In the first phase, we plan to run instructions without the aid of the control unit. This is to make sure that our datapath actually works correctly with the right control signals. We will manually enter the control signals for a couple instructions to make sure everything is properly connected. The next phase is a system test with control unit. This phase will test the control unit with the datapath. This phase we will load singular instructions into memory and have the datapath execute them. If everything works as planned, we should get the same results seen with same instructions for phase 1. The third and final phase of our system test plan is to run a small program. This phase will build off the previous one by simply having more instructions in memory for the system to execute. This phase makes sure that our entire datapath runs programs correctly. The program that the system will run is specially designed to test every instruction in our processor (seen as separate columns in the RTL file).

# Control

## Control Signals

We have 18 total control signals that the control unit uses. These signals are as follows:

1. MemAddr – [2-bit] Mux control signal that determines which memory address to access
2. MemData – [2-bit] Mux control signal that determines the data to write to memory
3. MemWrite – [1-bit] Enable/Disable control signal for writing to memory
4. DPInc – [2-bit] Mux control signal for incrementing, decrementing, or ignoring the DP
5. RPInc – [2-bit] Mux control signal for incrementing, decrementing, or ignoring the RP
6. RegWrite – [1-bit] Enable/Disable control signal for writing to the register file
7. IRWrite – [1-bit] Enable/Disable control signal for writing to the instruction register
8. TRSrc – [3-bit] Mux control signal that determines the value to write into the TR register
9. TRWrite – [1-bit] Enable/Disable control signal for writing to the TR register
10. PCSrc – [2-bit] Mux control signal for determining the value to write into the PC register
11. PCWrite – [1-bit] Enable/Disable control signal for an OR gate enabling PC writing
12. Jump – [1-bit] Enable/Disable control signal for an AND gate enabling PC writing
13. JumpCond – [1-bit] Indicates whether the instruction is a jump equal or jump not equal
14. ALUSrc – [1-bit] Mux control signal that determines the A value for the ALU to use
15. BSrc – [3-bit] Mux control signal that determines the value to mux into B
16. BWrite [1-bit] Enable/Disable control signal for writing to the B register
17. ALUop – [4-bit] Control signal that determines the operation that the ALU performs
18. OUTWrite – [1-bit] Enable/Disable control signal for writing to the OUT register

## Control Unit

The logic of the Control Unit controls all the control signals. Below are the input and output signals of the Control Unit:

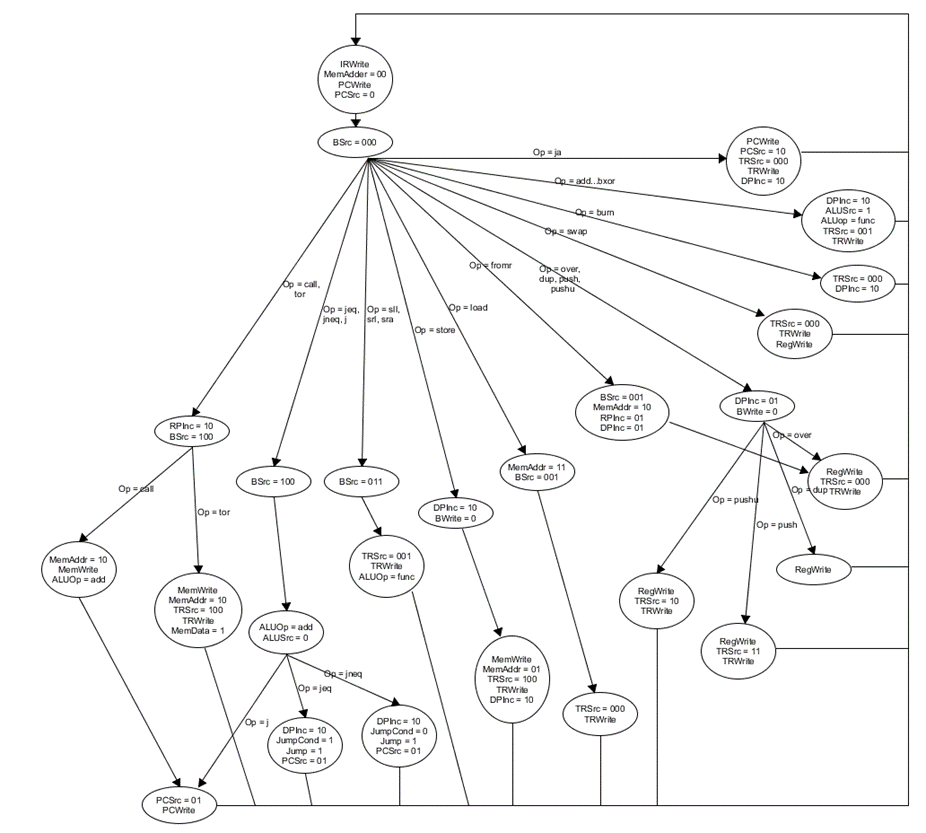
**Input:**

* [8-bit] The last 8 bits of the instruction, which include the 4-bit opcode and the 4-bit function code.
* [1-bit] Reset, which signals the datapath to reset entirely.2

**Output**

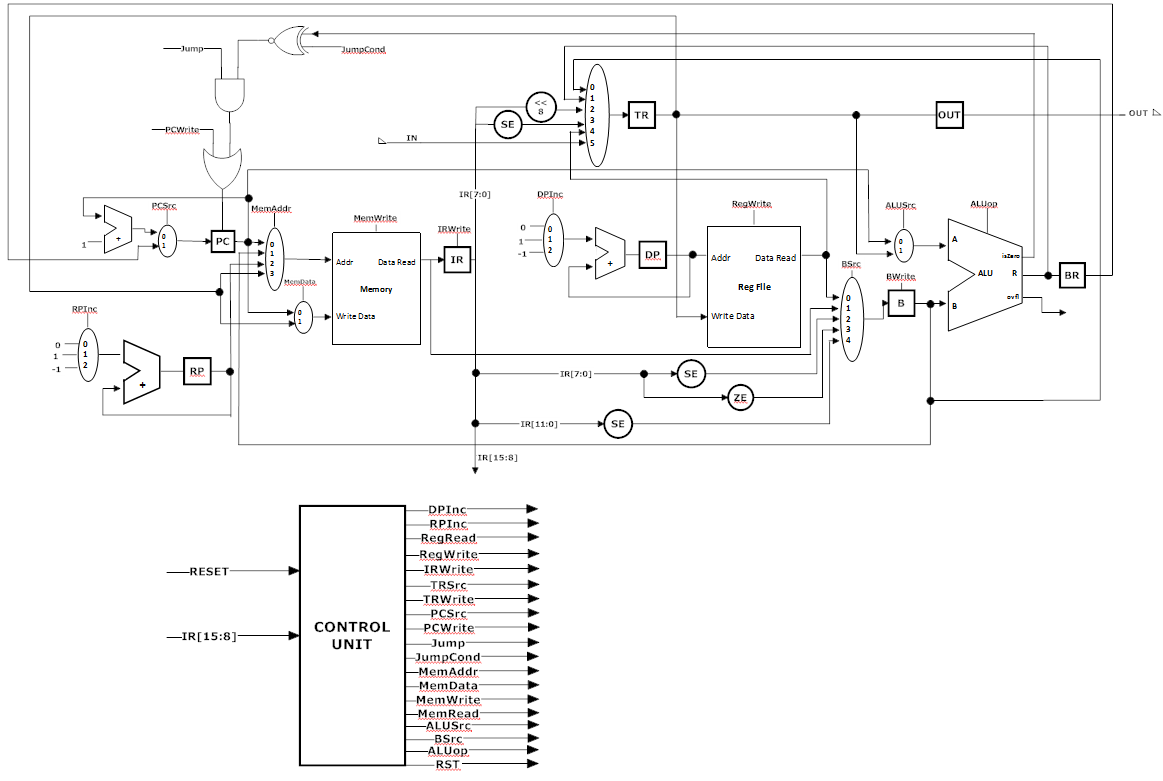
* **[**2-bit] MemAddr – 00 signals the mux to use the PC value, 01 signals the mux to use the RP register value, 10 signals the mux to use the MA register value, 11 signals the mux to use the TR register value. The output of this mux will be the address for the memory to use. Since memory is always accessed for incrementing PC, this value defaults to 00.
* [2-bit] MemData – 00 signals the mux to use the PC value, 01 signals the mux to use the TR value, 10 signals the mux to use the input value. The output of this mux is the data written into Memory. The default value for this mux does not matter.
* [1-bit] MemWrite – 0 disables writing to memory, 1 enables writing to memory. This value is defaulted to 0 and only changes if instructed.
* [1-bit] MemRead – 0 disables reading from memory, 1 enables reading from memory. This value is defaulted to 0 and only changes if instructed.
* [2-bit] DPInc – 00 signal decrements the pointer, 01 does not change the pointer value, 10 increments the pointer; This signal normally outputs 00 unless the instruction signals
* [2-bit] RPInc – 00 signal decrements the pointer, 01 does not change the pointer value, 10 increments the pointer; This signal normally outputs 00 unless the instruction signals
* [1-bit] RegRead – 0 disables reading a value from the register file, 1 will enable reading a value from the register file; This signal normally disables reading values from the register file and must be enabled by the instruction.
* [1-bit] RegWrite – 0 disables writing a value into the register file, 1 enables writing a value into the register file; This signal normally disables writing values to the register file and must be enabled by the instruction.
* [1-bit] IRWrite – 0 disables writing a value into the IR register, 1 enables writing a value into the IR register; This signal normally disables writing to the IR register and must be enabled by the instruction.
* [3-bit] TRSrc – 000 signals the mux to use the value from the B register, 001 signals the mux to use the value from the DataStack, 010 signals the mux to use the result of the ALU operation, 011 signals the mux to use the upper 8-bits of an immediate value, 100 signals the mux to use a sign extended immediate. 101 signals the mux to use the value being output by the user input wire. This is a mux signal that is dependent on every instruction and has no relevant default value
* [1-bit] TRWrite – 0 disables writing into the TR register, 1 enables writing a value into the TR register. This signal normally disables writing to the TR register and must be enabled by the instruction.
* [2-bit] PCSrc – 00 signals that the value to write into PC is PC + 1, 01 signals that the value to put into PC is a jump address from the ALU, 10 signals the value to mux into PC is from the TR register.
* [1-bit] PCWrite – 0 disables writing into the PC register, 1 enables writing a value into the PC register. This signal normally disables writing to the PC register and must be enabled by the instruction.
* [1-bit] Jump – 0 disables writing to the PC register. 1 will enable one side of the AND gate. The JumpCond XNOR gate determines the other input. When both are 1, the PC register will be enabled for writing. This signal is normally 0 and must be enabled by the instruction.
* [1-bit] JumpCond – Indicates whether the instruction is a jump equal or jump not equal for the XNOR gate. 0 indicates a jump not equal instruction, 1 indicates a jump equal instruction. The output of the XNOR gate is dependent on the isZero output from the ALU and the jump instruction type.
* [1-bit] ALUSrc – 0 signals that the A value for the ALU is from the TR register, 1 signals that the A value for the ALU is from the PC register. This mux signal is dependent on every instruction and has no relevant default value.
* [3-bit] BSrc – 000 signals the mux to use the value from the register file, 001 signals the mux to use the value is from Memory, 010 signals the mux to use a sign extended immediate, 011 signals the mux to use a zero extended immediate, 100 signals the mux to use a sign extended branch offset. The output of this mux will be the value to write to B. This mux signal is dependent on every instruction and has no relevant default value.
* [1-bit] BWrite – 0 disables writing into the B register, 1 enables writing a value into the B register. This signal normally enables writing to the B register and will only be disabled when B is necessary to save across instruction cycles.
* [4-bit] ALUop – this signals which operation the ALU is performing. There is no relevant default value. The operation signals are as described below:
  + 0000 – add
  + 0001 – sub
  + 0010 – && (logical and)
  + 0011 – || (logical or)
  + 0100 – slt
  + 0101 – & (bitwise and)
  + 0110 – | (bitwise or)
  + 0111 – !| (bitwise nor)
  + 1000 – XOR
  + 1001 – sll
  + 1010 – srl
  + 1011 – sra
* [1-bit] OUTWrite – 0 disables writing into the OUTWrite register, 1 enables writing a value into the TR register. This signal normally disables writing to the OUTWrite register and must be enabled by the instruction.

**Finite State Machine for the Control**

****

**Control Unit Test Description:** Test the control unit for every instruction. For each instruction opcode, test that the control unit outputs the right control signals for the correct states and cycles. The opcodes and states are shown in the control finite state diagram.

# Datapath



# Appendix B: Design Process Journal

# October 1, 2017

We decided on building a stack-based processor because of its simplicity in instruction and reduced data path. We discussed the instructions and registers we would use in our processor. No specific work was divvied out. It was agreed upon to finish working the next day to finalize everything.

# October 2, 2017

We came to a concrete conclusion on the method of using a stack as a processor. We would use a top-of-stack register as a sort of buffer for the top value and then a pointer to the top of the rest of the stack. This decision was made because it should speed up instructions instead of having to call two pops and a push for most instructions, we simply modify the register. We went through the MIPS green sheet and used instructions that seemed to be universally necessary for our stack as well. Rias was in charge of instructions, and he came up with some stack manipulation instructions, such as toR, fromR, burn, dup, over, and swap. Once we had our instructions decided, which we all relied upon for our own work, we then divvied up the workload so that we could get most of the work done at the same time. Once all of us had finished our assignments, we reviewed each other’s work as team.

## Work Distribution

* Rias will work on descriptions for assembly language programmer and each machine language instruction; estimate 1 day. [1 day]
* Luke will write an explanation of any procedure call conventions; estimate 1 day. [1 day]
* Dominic will write the rules for translating each assembly instruction into machine language and write the machine language translations for Euclid’s algorithm; estimate 1 day. [1 day]
* Ben will finish writing the sample assembly language program for Euclid’s algorithm; estimate 1 day. [1 day]
* Kat will write the descriptions of the instructions, example assembly fragments, and keep track of the Design Journal; estimate 1 day. [1 day]

# October 7, 2017

We were working on cleaning up our design document to fix our instructions to be 16-bit and so adding a few additional instructions. We also worked on our RTL design. Our main method of verification(“test”) for our RTL was just going through every single instruction, making sure it executes as intended (which values are what from the stack, where the result is stores, etc.).

## Work Distribution

* Rias will reorganize the instructions to be grouped more logically. [1 hour]
* Ben will change the Pseudo instructions (beq & bne) in the Machine Code

# October 9, 2017

Rias changed the size of the instructions due to them being too big to fit into the data bus. With a change in instruction length, Rias made new opcodes and changed the documents accordingly. We went through our RTL and saw where things changed and overlapped and created mux’s and other appropriate components to implement the RTL and create as simple a datapath as possible.

## Work Distribution

* Ben will change the machine code instructions (jeq & jne) in the Machine Code; estimated 1 hour [<1 hour]
* Luke write descriptions for each component; estimated 1 [1 hour]
* Rias will run through the RTL and try to have an unbiased viewpoint when checking validity; estimated 2 hours [3 hours]

# October 10, 2017

Rias and Luke went through the design documents and design journal to make sure everything was correct and ready for milestone 2 check with Micah tomorrow. Rias and a third party stepped through the operation to validate its logic. A third party was used because all of us had worked together on the RTL and could not provide an unbiased input on its logic.

## Work Distribution

* Rias performing checks and changes; estimated 1 hour [2 hours]
* Rias stepped through the RTL in order to verify proper operation; estimated 1 hour; [<1 hour]
* Luke performing checks and changes; estimated 1 hour [2 hours]

# October 17, 2017

The group worked together to determine new opcode patterns and updating documentation. This is so that our opcodes include more detail as to what the instruction does and ultimately simplifies control logic. Because the opcodes changed, the S-Type and I-Type instruction formats changed and a new J-Type format was introduced. The opcodes are now 4-bits long, thus the S- and I-Types need a function code to specify what to do, while the J-Type can now use a 12-bit immediate as an offset for PC-relative jumps.

# October 18, 2017

Luke and Kat worked on cleaning up the RTL and the rest of the document formatting. We also updated instruction bits now that some I-Type instructions have 12-bit immediates. This is so that we will be able to jump farther now. Rias and Ben worked on the datapath while Luke and Kat continued to modify the design document as the 2 gave us updated information on the implementation.

Rias and Luke worked on fixing the terrible RTL. We then created a datapath from the RTL. Datapath is affected a lot by the fact that two values from the stack are used almost everywhere, making the logic somewhat confusing. We changed the component specs and signals to match the datapath. We also have now documented our memory addressing and addressing modes. This took roughly 3 hours.

# October 21, 2017

Because Kat and Dominic were not able to attend tonight’s meeting, Rias, Luke, and Ben worked on fixing changes to previous Milestones. We started by having Luke and Ben make changes to the RTL to reflect the multicycle process. From there, they recreated a new and improved datapath. Meanwhile, Rias was making changes to the Design Document. By changes, we mean a complete overhaul of the entire document. We did this renovation for several reasons, such as a more organized document, better formatting, basic milestone changes, and the inclusion of the new RTL and Datapath.

## Work Distribution

* Luke and Ben – RTL changes; estimated 1 hour [1.5 hours]
* Luke and Ben – Datapath Recreation; estimated 3 hours [2 hours]
* Rias – Design Document complete overhaul; estimated 1 day [3 days]
* Luke – Rewriting Components list for M2 and M3; estimated 1 hour [1 hour]

# October 22, 2017

Luke was not able to attend today’s meeting because of an all-day fraternity event. Rias and Ben made some small changes to the RTL and datapath. Rias and Ben split the control unit into three separate units: the Memory Control Unit, the Register Control Unit, and the ALU Control Unit. We made this decision to simplify control logic and organize control signals. Kat began work on the FSM for each control unit’s control signals. Dominic began working on Lab 7.

## Work Distribution

* Rias and Ben – RTL, Datapath, Control Units; estimated 1 hour [1 hour]
* Kat – Multiple FSMs; estimate 1 day [Never finished]
* Dominic – Work on Lab 7; estimate 2 day [2 days]
* Rias – Control signals and Control Unit descriptions; estimated 1 day [1 day]
* Rias – More formatting work on the Design Document; estimated 2 days [2 days]

# October 23, 2017

In class today, we realized how far behind we were. Thus, Rias went back from M1 to the current Milestone noting all the things we need to change or include. Rias started catching us up by described the missing datapath components. Dominic started writing the unit test and Verilog test benches for our components. Luke helped us catch up by working on the integration and implementation plan. For work on M4, Ben is working on creating the Verilog components, Kat is still working on the FSM, and Rias is doing the control specs and updated signals. Ben made a Verilog unit test for the mux and checked Micah’s components to see what we could use for our architecture.

## Work Distribution

* Ben – Mux unit test and check; estimate 30 minutes [30 minutes]
* Ben and Luke – Making the ALU; estimate 1 day [<1 day]
* Luke – Integration plan; estimate 1 day [1 day]
* Kat – Still working on the FSMs; estimate 2 days [Never finished]
* Dominic – unit test descriptions for the control units and the parts; estimate 3 hours [3 hours]
* Dominic – Test-benches; estimate 2 day [Never finished]
* Dominic – Making the other Verilog Components; estimate 1 day [Never finished]
* Rias – Datapath Components; estimate 4 hours [5 hours]
* Rias – Control Unit descriptions; estimate 1 day [6 hours]
* Rias – Previous milestone fixes; estimate 2 days [To this day]

# October 24, 2017

Rias noticed a problem with the RTL and Datapath that required a moderate change in the Datapath and a slight change in the RTL. Rias re-drew the datapath, which introduced two new control signals, Jump and JumpCond, and fixed everything related accordingly. Dominic finished making our muxes. Rias made a shifter component and a test bench for it. Kat made a zero and sign extender and test benches for each respectively. Luke and Ben started working on the ALU for our datapath.

## Work Distribution

* Dominic – Muxes w/ Testbenches; estimated <1 hour [1 hour]
* Kat – Sign and Zero Extender; estimated <1 hour [2 hours]
* Ben and Luke – ALU; estimated 2 hours [<2 hours]
* Rias – Datapath re-draw and eight-bit-shifter; estimated 2 hours [2.5 hours]

# October 26, 2017

Kat and Dominic finished Lab 07 and demonstrated it to Micah. Rias noticed that the combinational components that Kat had built and tested were using a clock edge for some reason and thus fixed them. Rias also noticed that Dominic had made special muxes for three and five inputs for some reason and got rid of those, leaving only the two, four, and eight input muxes.

## Work Distribution

* Rias – Changes to the extenders and deletion of unnecessary muxes; estimated <1 hour

# October 27, 2017

We met with Micah today. Main detraction was that our FSMs were missing. He also mentioned that our RTL could have a few optimizations. With the exception of the Adder, Rias started and finished the entire lab 06 by 4:30. Rias then went and made RTL optimizations, now represented in the RTL\_Update.xlsx file.

## Work Distributions

* Rias – Optimizations and changes to the RTL estimated 1 hour [<2 hours]

# October 28, 2017

Rias made some edits to the existing Verilog parts. Rias made a register file with 256 registers and a new register component. Files changed to reflect naming convention of part[inputs]\_[input size]b\_[output size]b, where parts with the same input and output size will have the convention of part[inputs]\_[size]b. Rias started creating the FSM for the control unit. Rias also began implementing the control unit.

## Work Distribution

* Rias – Verilog edits for naming conventions and small changes; estimated <1 hour [<1 hour]
* Rias – Register file with test bench; estimated <1 hour [30 minutes]
* Rias – Finite State Machine for our single control unit; estimated 1 day [3 days]

# October 30, 2017

Kat went through the previous milestones and checked for inconsistencies and missing things in our design document. She made the list and checked it twice. Luke began Verilog integration and testing.

## Work Distribution

* Kat – design document maintenance and milestone checks; estimated 2 hours [<2 hours]
* Luke – Integration and testing as specified in the design document; estimated 1 day [1 day]

# October 31, 2017

Rias and Luke started burning the midnight oil at 8:17 pm. Rias finally finished the control FSM. Rias made the return stack implemented in memory start at the opposite side of memory from the program, meaning all increments to RP in the RTL became decrements and vice versa. Luke began making control unit tests. At 4:30 am, Rias finished implementing the control unit in Verilog. Luke finished the Control Unit tests at 5 in the morning. Rias added a memory unit at around 6. At 6:45, Rias and Luke integrated the RP, DP, and PC subsystems. Rias finished the rest of the Gen1 Subsystems and Luke has gotten considerably far in the integration tests. It is now 9:20 am. All-nighter success!

## Work Distribution

* Rias – Finish control FSM (written down on a white board); estimate 3 hours [3 hours]
* Luke – Control Unit Tests; estimate 2 hours [~1.5 hours]
* Rias – Control Unit Implementation; estimate 3 hours [~3.5 hours]
* Rias and Luke – Component Integration for complete datapath; estimate 1 hour [1 day]
* Luke – Integration tests; estimate 1 day [~1 day]

# November 1, 2017

Kat marked down the FSM in a portable form using Umlet. Rias and Kat went through and tested the state transition diagram with our datapath to make sure it works. At 4:04 pm, the complete datapath was fully implemented in Verilog. Unfortunately, Rias and Luke were unable to incorporate testing for the entire datapath by the deadline of 4:30 today.

## Work Distribution

* Rias and Luke – complete integration; estimate 1 day [1 day]

# November 2, 2017

Rias called a meeting after our team meeting in order to catch up Ben, Dominic, and Kat. Dominic was unable to make it, replying “Can’t make it tonight for other class but I’ll do whatever as far as work goes.” Kat was feeling out of it and had to leave early. Ben, Rias, and Luke worked on the integration tests. The FinalDatapath synthesizes with a severe amount of warnings that we do not yet know how to fix.

## Work Distribution

* Ben – Integration tests for TRSystem, ALUSystem, Memory; estimated 1 day [<1 day]
* Luke – Integration tests for RP and DP loops; estimated 1 hour [1 hour]
* Kat – Check our control unit signal outputs with the datapath; estimated 1 day [1 day]
* Rias – Tests for register file, 8-bit shifter, and DataStack; estimated 2 hours [<2 hours]
* Rias – Control Unit fixes; estimated 1 day [1 day]

# November 5, 2017

Rias, Ben, and Luke met at 1pm Sunday. Kat and Dominic were both unable to make it due to obligations in other classes. Rias and Ben are still investigating the synthesis warnings. Rias came up with a system test plan. Luke began performing system tests by first manually entering control signals into the datapath to check specific states of our datapath during each cycle of an instruction. Ben made a complete ALU test bench.

## Work Distribution

* Ben – Integration test of ALUSystem; estimated 2 hours [<2 hours]
* Luke and Rias – System tests for simple instructions; estimated 2 hours [2 hours]
* Ben and Rias – System tests follow up; estimated 1 hour [2 hours]

# November 6, 2017

With some changes from the past meeting made to the control unit, the control unit now successfully synthesizes. We intend to get the processor working by the end of today’s meeting. Rias moved some wires and Verilog components, which somehow allows our entire processor to synthesize correctly. Luke and Ben began working on the design and implementation of I/O, which include control changes, additional components, and additional instructions.

For I/O we decided to have an input wire that allows the user to push a 16-bit value to the stack, and a 16-bit output register OUT that continuously outputs the value it holds. To accommodate this, we added two instructions, *in* and *out*. *In* pushes the input value to the stack, and *out* pops the top value from the stack and pushes it to the OUT register.

Ben also worked on updating our Euclid’s Algorithm program for input and output. Dominic made the changes to the machine to reflect the updated algorithm. Rias began implementing the system tests and getting the processor to work, excluding basic input and output. Kat made necessary changes to the control unit FSM and to our datapath. We will need to update these once we fully figure out I/O. Kat also added some logic in control for counting cycles for milestone 6.

## Work Distribution

* Kat – Control unit FSM updates; estimated 1 hour [1 hour]
* Kat – New, redrawn datapath; estimated 2 hours [Not completed at meeting]
* Kat – M6 counters in the control unit; estimated <1 hour [<1 hour]
* Ben – Program changes for allowing I/O; estimated 1 hour [30 minutes]
* Luke and Ben – I/O design; estimated 3 hours [2 hours]
* Rias – System test implementation; estimated 2 hours [4 hours]
* Rias – Proper fixes to the datapath
* Ben – assisting with system tests and fixing
* Dominic – Checking the updated Euclid algorithm [2 hours]
* Ben –I/O Design; estimated 3 hours [3 hours]
* Luke – Begin implementing I/O; estimated 1 day [1 day]

# November 7, 2017

We met today at 8 pm to get everything working that needs to be working. Rias wrote the assembler, compiler, and linker for our processor. Luke implemented I/O into hardware in Verilog with some assistance from Ben. Ben also updated the design document for our I/O design. Kat updated the control unit FSM to reflect our I/O design and changes made previously. Ben fixed our Euclid’s algorithm program to work properly. Dominic performed some logic for counting instructions. Dominic did all the performance data collection/evaluation necessary for M6. Around 1 o’clock our processor and algorithm were fully operational, just like the fully armed and operation battle station! Go team, go!

## Work Distribution

* Rias – Assembler, compiler, linker; estimated 5 hours [3 hours]
* Ben and Luke – I/O design and documentation; estimated 3 hours [2 hours]
* Luke and Ben – I/O hardware implementation; estimated 2 hours [~1.5 hours]
* Kat – FSM and rebuilding of datapath; estimated 2 hours [>4 hours]
* Dominic – Performance evaluation and collection; estimated 1 hour [1 hour]
* Team – Used newly made assembler to test Euclid’s with our processor

# November 13, 2017

The group met at 10:30 pm to finalize our presentation and final report document. Kat insisted on working on the final report, Rias finalized the assembler, and Luke, Ben, and Dominic put the slideshow together.

After hours of work, Kat left to go sleep, and some progress on the final report had been made.

## Work Distribution

* Rias – Assembler, compiler, linker – 3 hours
* Ben, Luke, Dominic – Presentation – estimated 1 hour, actual 3 hours
* Kat – Final report – estimated 2 hours, actual 4 hours
* Rias, Ben, Luke – Final Report – estimated 90 minutes, actual 2 hours