

Design Document

Rias Hilliard, Benjamin Hall, Luke Evard, Katarina Chan, Dominic Grazioli



Group 2C

**Contents**

[Structure 3](#_Toc497899217)

[DataStack 3](#_Toc497899218)

[ReturnStack 3](#_Toc497899219)

[Registers 3](#_Toc497899220)

[Input/Output 3](#_Toc497899221)

[Call Conventions 3](#_Toc497899222)

[Return 3](#_Toc497899223)

[Function Calling 4](#_Toc497899224)

[Addressing 4](#_Toc497899225)

[Memory Addressing 4](#_Toc497899226)

[Addressing Modes 4](#_Toc497899227)

[Instruction Format and Syntax 4](#_Toc497899228)

[S-Type 4](#_Toc497899229)

[I-Type 5](#_Toc497899230)

[J-Type 6](#_Toc497899231)

[Instructions 6](#_Toc497899232)

[S-Type 6](#_Toc497899233)

[I-Type 9](#_Toc497899234)

[J-Type 10](#_Toc497899235)

[Assembly Language Example Fragments 11](#_Toc497899236)

[Euclid’s Algorithm 11](#_Toc497899237)

[Register Transfer Language (RTL) 12](#_Toc497899238)

[Datapath Components 13](#_Toc497899239)

[Integration 16](#_Toc497899240)

[Unit Tests 16](#_Toc497899241)

[Integration Plan and Tests 17](#_Toc497899242)

[Control 19](#_Toc497899243)

[Control Signals 19](#_Toc497899244)

[Control Unit 19](#_Toc497899245)

[Datapath 22](#_Toc497899246)

# Structure

## 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:

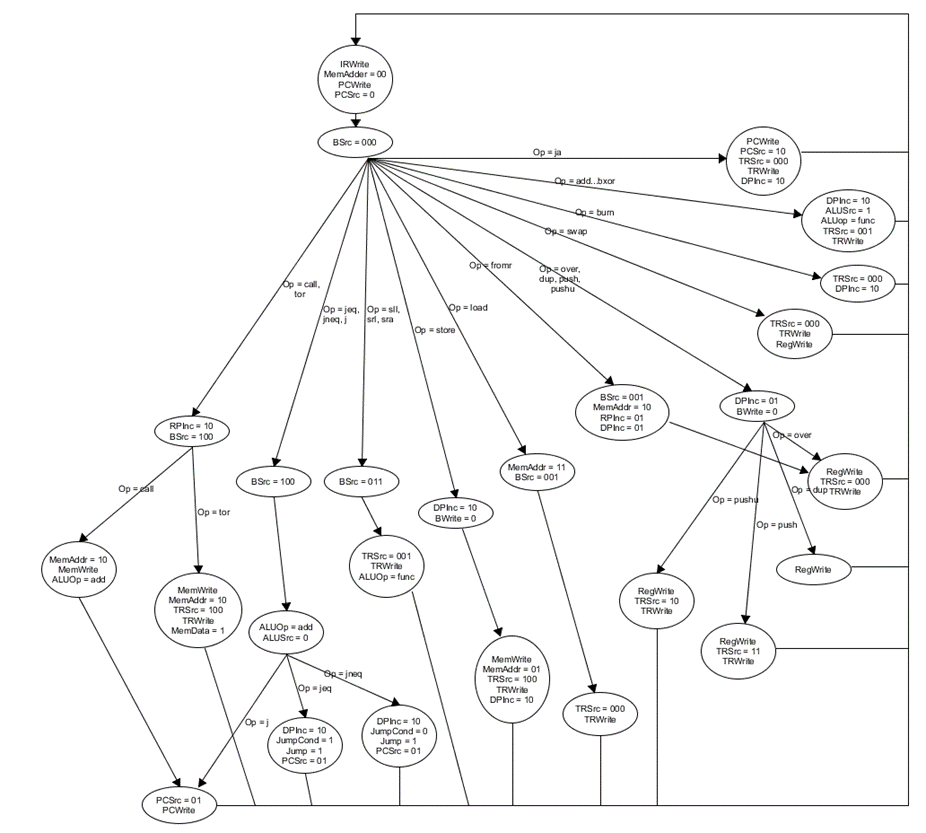
**Input:**

* [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

