CSE 141L Milestone 1

Esther Xiong, A17043893; Eban Covarrubias, A16743935; Charlie Trinh, A18061636

Academic Integrity

Your work will not be graded unless the signatures of all members of the group are present beneath the honor code.

To uphold academic integrity, students shall:

- Complete and submit academic work that is their own and that is an honest and fair representation of their knowledge and abilities at the time of submission.
- Know and follow the standards of CSE 141L and UCSD.

Please sign (type) your name(s) below the following statement:

I pledge to be fair to my classmates and instructors by completing all of my academic work with integrity. This means that I will respect the standards set by the instructor and institution, be responsible for the consequences of my choices, honestly represent my knowledge and abilities, and be a community member that others can trust to do the right thing even when no one is watching. I will always put learning before grades, and integrity before performance. I pledge to excel with integrity.

Esther Xiong

Eban Covarrubias

Charlie Trinh

Team

Esther Xiong

Eban Covarrubias

Charlie Trinh

Introduction

TODO. Name your architecture. What is your overall philosophy? What specific goals did you strive to achieve? Can you classify your machine in any of the standard ways (e.g., stack machine, accumulator, register-register/load-store, register-memory)? If so, which? If not, devise a name for your class of machine. Word limit: 200 words.

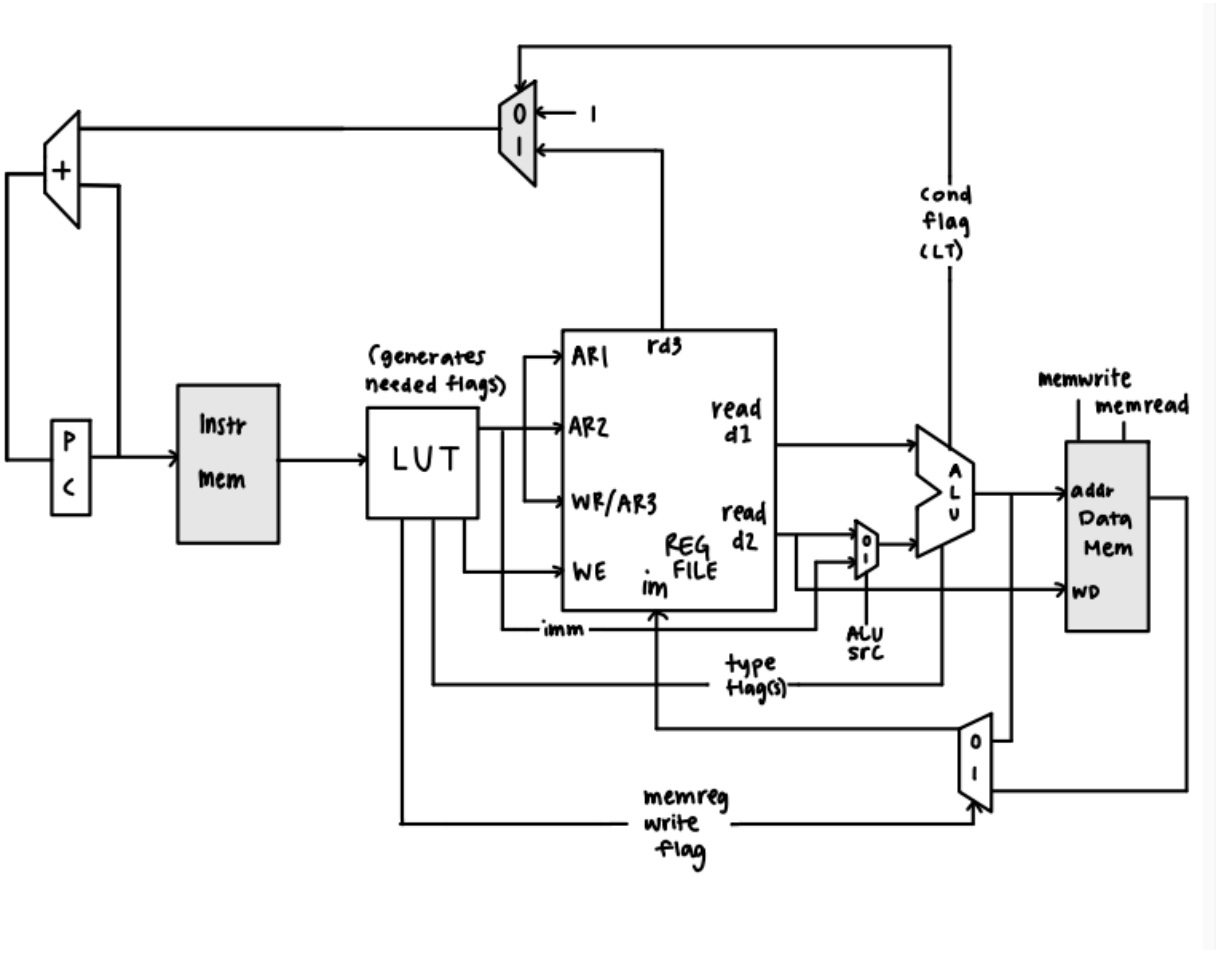
What we want:

Simplified Mips Architecture

Our goal is to make sure we can support all essential commands while working with a limited number of bits. We want to minimize both the size of our instruction set, and try to keep a short list of available registers. We will limit ourselves to only 4 general use

registers, each of which have 8 bits of memory. This is to minimize the number of bits used up on the instructions for referencing registers. We will have a total of 8 operations to pick from, which will take up the first 3 bits of each instruction, using different opcodes. The instructions we will use will be specified lower on this document.

Architectural Overview



Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
Mem	3 bits opcode, 2 bits send/destination register, 2 bits address register, 2 bits offset.	str/ldr
B	3 bits opcode, 2 bits destination line register, 2 bits register to compare 1, 2 bits register to compare 2.	blt
R	3 bits opcode, next 6 bits are used for 3 registers. Each register takes 2 bits. We have Register output, register input 1, register input 2.	xor , and

Data Transfer	3 bits opcode, 2 bits for destination register, 4 bits for immediate input.	Mov, shift
Arithmetic	3 bits opcode, 1 bit to denote signed addition or not, 1 bit for carry in, 2 bits for destination register, 2 bits for register holding added value.	Add, Sub, Abs, Copy
Comp	5 bits opcode, 2 bits register 1, 2 bits register 2.	cmp
SimpleA	7 bits opcode, 2 bit register that is both input and output reg.	Add1, add2, sub1, sub2

#Simple A operations can be replaced at any time, they are very case specific

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES

Xor = exclusive bitwise or.	R	3 bits for opcode (110), 2 bits for output register(XX), 2 bits for input 1 register(XX), 2 bits for input 2 register(XX).	#Assume R0 has 0b0001_0001 #Assume R1 has 0b0010_0011 Xor r2, r0, r1 110 10 00 01 #after instruction r2 will hold 0b0011_0010	
And = bitwise and function	R	3 bits for opcode(111), 2 bits for output register(XX), 2 bits for input 1 register(XX), 2 bits for input 2 register(XX).	#Assume R0 has 0b0001_0001 #Assume R1 has 0b0010_0011 And r2, r0, r1 111 10 00 01 #after instruction r2 will hold 0b0000_0001	
Add = addition	Arithmetic	5 bits for opcode(10000), 2 bits for the first register (input and output register), 2 bits for the second register (input)	#Assume R0 has 0b0001_0001 #Assume R1 has 0b0010_0011 Add r0, r1 0b0011_0100 #after instruction r0 will hold	We will add r0 and r1 together and the output of this will be stored in r0

			0b0011_0100	
Sub = subtract (signed)	Arith metic	5 bits for opcode(10001), 2 bits for the first register (input and output register), 2 bits for the second register (input)	#Assume R0 has 0b0001_0001 #Assume R1 has 0b0010_0011 (r1 will filp signs to negative so 0b1101_1101) Sub r0, r1 #after instrution r0 will hold 0b1110_1110	We will subtract r0-r1, to do this we need to flip r1's sign by using an xor and adding 1 to it to satisfy 2's complement on the value stored within r1. Then after we can add r0 and r1 together to get the difference.
Copy = copy	Arith metic	5 bits for opcode(10010), 2 bits for the first register (input and output register), 2 bits for the second register (input)	#Assume R0 has 0b0001_0001 #Assume R1 has 0b0010_0011 copy r0, r1 #after instrution r0 will hold 0b0010_0011	We take r1's value and copy it into r0.
Abs =	Arith	5 bits for opcode(10011), 2	#Assume R0 has 0b1111_1111	R0 will be the the most significant bytes

absolute value	metic	bits for the first register (input and output register), 2 bits for the second register (input and output register)	#Assume R1 has 0b1111_1111 abs r0, r1 #after instruction r0 will hold 0b0000_0000 #after instruction r1 will hold 0b0000_0001	and R1 will be the least significant bytes of a 16 bit number. Then we will flip all the bits in this 16 bit number then add 1 to it to satisfy 2's complement when changing the sign of a number. Then r0 will store the first 8 bits (MSB), and r1 will store the last 8 bits (LSB) giving us the absolute value of the 16 bit number split between r0 and r1.
Mov = move instruction copies data from one location to another	Data Transfer	3 bits opcode (010), 2 bits for destination register(XX), 4 bits for immediate input (XXXX). This will allow value assignment from 0 to 15.	Mov r0, 0001 010 00 0001 #after instruction r0 will hold 0b0000_0001	Note that we can only represent up to the number 15 in binary. If we want other number we must use this function in combination with both saddto and shift.

Shift = Shift some register value a specified number of bits.	Data Transfer	3 bits opcode(011), 2 bits destination register(XX), 4 bits signed shift amount.	#assume r0 hold 0b0000_1010 Shift r0, 0001 #after instruction r0 will hold 0b0000_0101.	Note that this shift is logical shift only, we can change the direction by using a negative value for the shift offset value. Positive is logical right shift, negative is logical left shift.
Cmp = compare	comp	5 bits opcode (10100), 2bits reg1 (XX), 2 bits reg2(XX).	#assume r0 holds 0, r1 holds 1 Cmp r0 r1 #all comparison flags are set, in this case, less than = 1, equal = 0, greater than = 0	
Str = store value in memory	mem	3 bits opcode(000), 2 bits register with data to send(XX), 2 bits register with memory address(XX),	#assume r0 hold 0b0000_0011 #assume r1 hold 0b0001_0000 Str r0, r1, 00 #after this command, at memory	Note that the offset in this example is 0. The offset is added to the specified memory address to store data at. We can only store one byte at a time.

		2 bits for offset.	address 16 we will store the byte 0b0000_0011 as specified by r0.	
ldr= load value from memory	mem	3 bits opcode(001), 2 bits register to receive data (XX), 2 bits register with memory address(XX), 2 bits for offset.	#assume memory address 16 holds a value of 0b'0011_1100 #assume r1 holds the value 15, or 0b0000_1111 Ldr r0, r1, 01 #after this r0 will hold 0b'0011_1100	Note that we used the offset to increment r1 from 15 to 16 in order to load from address 16. The reason I did this is because in practice we probably will use a mov statement to get the correct address into a register, and mov only has the range 0-15 which makes the offset useful for reducing lines of instructions.
Ble = branch if less than equal to	B	7 bits opcode(1010100), 2 bit input register (XX)	After cmp statement if r0 <= r1 then we will branch based on: #r2 has 0b1 Ble r2 We will branch forwards if r0 <= r1	Limit is -128 to 127 where a negative value will go back lines and positive values will go forward lines.

Blt = branch if less than	B	7 bits opcode(1010101), 2 bit input register (XX)	After cmp statement if r0 < r1 then we will branch based on: #r2 has 0b1 Blt r2 We will branch forwards if r0 < r1	Limit is -128 to 127 where a negative value will go back lines and positive values will go forward lines.
Beq = branch if equal	B	7 bits opcode(1010110), 2 bit input register (XX)	After cmp statement if r0 == r1 then we will branch based on: #r2 has 0b1 Beq r2 We will branch forwards if r0 == r1	Limit is -128 to 127 where a negative value will go back lines and positive values will go forward lines.
Bne = branch if not equal	B	7 bits opcode(1010111), 2 bit input register (XX)	After cmp statement if r0 != r1 then we will branch based on: #r2 has 0b1 bne r2	Limit is -128 to 127 where a negative value will go back lines and positive values will go forward lines.

			We will branch forwards if r0 != r1	
Bge = branch if greater than or equal to	B	7 bits opcode(1011000), 2 bit input register (XX)	After cmp statement if r0 >= r1 then we will branch based on: #r2 has 0b1 Bge r2 We will branch forwards if r0 >= r1	Limit is -128 to 127 where a negative value will go back lines and positive values will go forward lines.
Bgt = branch if greater than	B	7 bits opcode(1011010), 2 bit input register (XX)	After cmp statement if r0 > r1 then we will branch based on: #r2 has 0b1 Bgt r2 We will branch forwards if r0 > r1	Limit is -128 to 127 where a negative value will go back lines and positive values will go forward lines.
B = forced branch	B	7 bit opcode (1011011), 2 bit input register (XX)	This is a force branch where we will branch no matter what, even without the cmp statement.	Limit is -128 to 127 where a negative value will go back lines and positive values will go forward lines.

			#r2 has 0b1 B r2 We will branch 1 line forwards	
add1	simpl eA	7 bit opcode (1011100), 2 bit input register (XX)	We will add 1 like a ++ function in c. #r0 has 0b0000 Add1 r0 #after this function call r0 has 0b0001	
add2	simpl eA	7 bit opcode (1011101), 2 bit input register (XX)	We will add 2 to the register. #r0 has 0b0000 Add2 r0 #after this function call r0 has 0b0010	
sub1	simpl eA	7 bit opcode (1011110), 2 bit input register (XX)	We will subtract 1 like a - - function in c.	

			#r0 has 0b0010 sub1 r0 #after this function call r0 has 0b0001	
sub2	simpl eA	7 bit opcode (1011111), 2 bit input register (XX)	We will subtract 2 to the register. #r0 has 0b0010 sub2 r0 #after this function call r0 has 0b0000	

Internal Operands

We will have 4 registers, represented by 00, 01, 10, 11. Each of these registers will be general purpose registers.

Control Flow (branches)

We are only supporting one branch statement, which is branch if less than, (blt) The max branch distance we can support is anywhere in the range [-128, 127]. Whatever number is held in the first argument register is how far from our current program position we will move. In order to accommodate larger jumps, we must use multiple blt statements that jump to eachother.

Addressing Modes

We use direct addressing for our load and store operations. The addresses are represented by the 8 bits in our second register argument, which gives a range of [0, 256]. The addresses are calculated by using a standard binary representation. We also add the offset argument value, which has a range of [0, 3].

Programmer's Model

TODO. 4.1 How should a programmer think about how your machine operates? Provide a description of the general strategy a programmer should use to write programs with your machine. For example, one could say that the programmer should prioritize loading in the necessary values from memory into as many registers as possible, then perform calculations. Another approach could be loading and writing to memory in between every calculation step. Word limit: 200 words.

Due to having very limited registers, (only 4) the programmer should use load and store to save values for future use. The registers will likely be changing values very frequently for larger more complex programs. We also have limited operations since our instruction bit string is so short, which means getting specific values into registers using mov can be challenging. In order to get larger values (eg over 15) into a register, the user must use the saddto operation or shift if they are able to reach this value by multiplying by multiples of 2, or a combination of the two operations.

TODO. 4.2 Can we copy the instructions/operation from MIPS or ARM ISA? If no, explain why not? How did you overcome this or how do you deal with this in your current design? Word limit: 100 words.

We are not able to copy due to the limited number of instruction bits. We are dealing with this by reducing both the amount of operations that we can use, having more general use operations, and reducing the number of registers to only 4. This current design also doesn't use any bits for type specification. Which means, we likely will use a lookup table to generate these bits based on our opcodes.

Milestone 2

Individual Component Specification

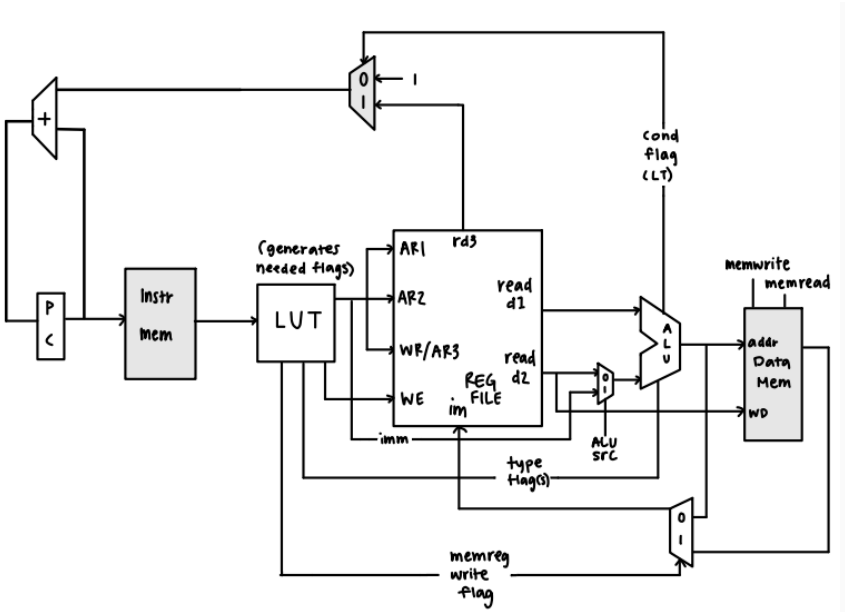
Top Level

Module file name: toplevel.sv

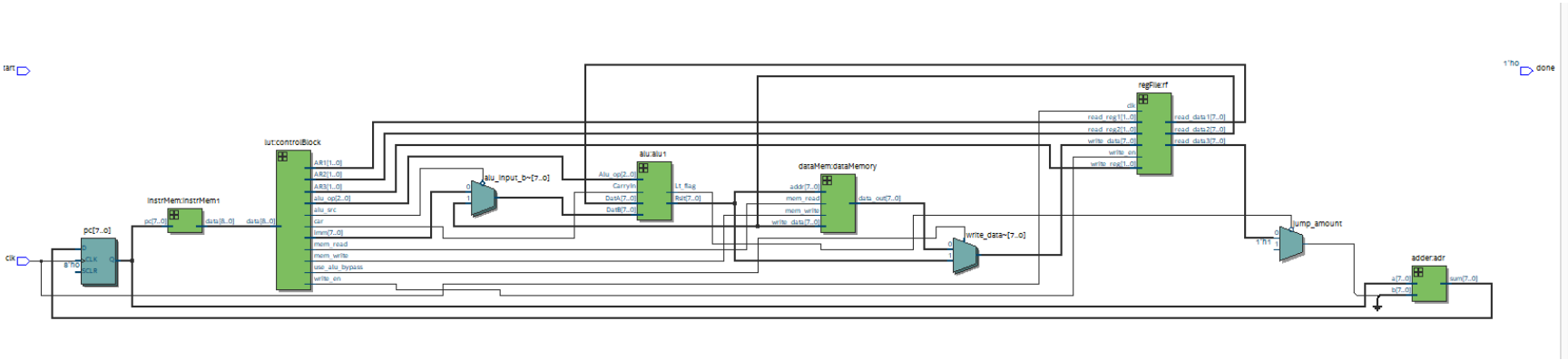
Functionality Description

The toplevel pieces together all other components of our architecture, it works by closely following the diagram that we designed in milestone 1. The Schematic below is what we followed, we also put the rtl view to display the similarity between the two. Note that there are still some changes that need to be made, not all of the input flags are fully integrated, and some components are a wip.

Schematic



Here is the RTL view of our hardware (which is still a wip, dataMem, regfile and alu are incomplete but started)



Program Counter

Module file name: within toplevel.sv
Module testbench file name: TODO

Functionality Description

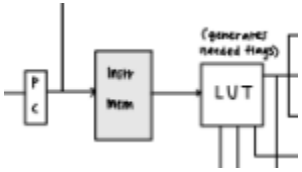
We have created the PC within the top level using a flip flop, Note that in our RTL it is blue, which implies that it is not an actual component, and is built directly into the top level. Since the PC was very simple we opted for this.

(Optional) Testbench Description

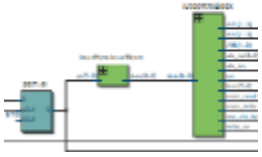
TODO. Describe your testbench. How does it work? What test cases does it test?
N/A

Schematic

TODO. Show us your schematic for the fetch unit.



This is the RTL of the same unit:



(Optional) Timing Diagram

TODO. Show us a screenshot of the timing diagram that demonstrates all relevant functions of the fetch unit.

N/A

Instruction Memory

Module file name: dataMem.sv

Functionality Description

TODO. This module simply acts as memory to be accessed using the pc, it holds only 9 bit instructions. We are working on functionality to write to it as well using txt files.

Schematic

TODO. Show us your schematic for the fetch unit.

Control Decoder

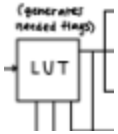
Module file name: lut.sv

Functionality Description

TODO. This control decoder takes in a 9 bit instruction, and then outputs all of the appropriate registers and flags based on the specific operation that we are dealing with. This was by far the most tedious to create in hardware, but incredibly important for getting the right data to the right places.

Schematic

TODO. Show us your schematic for the control decoder.



Register File

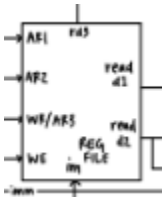
Module file name: regFile.sv

Functionality Description

TODO. This holds register data, and given inputs it will give the data held by specified registers, and may write to a register.

Schematic

TODO. Show us your schematic for the register file.



ALU (Arithmetic Logic Unit)

Module file name: alu.sv
Module testbench file name: TODO

Functionality Description

The alu is capable of doing several operations, including shifting, (left and right) adding, xor, and. We will be adding in subtraction as well very soon, this hardware is still being developed.

(Optional) Testbench Description

TODO. Describe your testbench. How does it work? What test cases does it test?

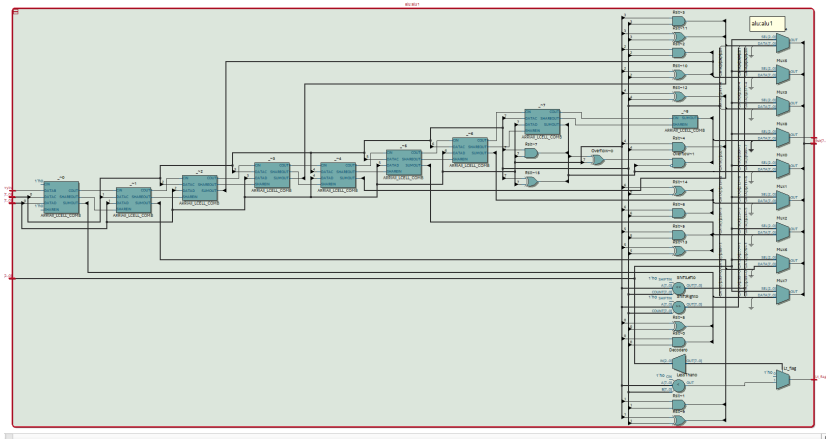
ALU Operations

The operations that the alu will be able to perform is logical left shift, logical right shift, xor, and, add, subtract and comparison (specifically less than comparison)

Schematic

TODO. Show us your schematic for the alu.

RTL:



(Optional) Timing Diagram

TODO. Show us a screenshot of the timing diagram that demonstrates all relevant operations you mentioned in the ALU Operations section.

Data Memory

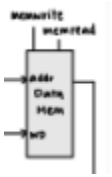
Module file name: dataMem.sv

Functionality Description

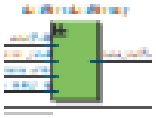
TODO. This

Schematic

TODO. Show us your schematic for the data memory.



RTL:



Lookup Tables

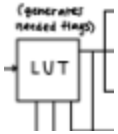
Module file name: lut.sv

Functionality Description

This module is used as our control decoder, it takes in our instruction and generates all needed flags and places all the registers where they need to be.

Schematic

TODO. Show us your schematic(s) for the lookup table(s).



Muxes (Multiplexers)

Module file name: within toplevel.sv

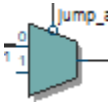
Functionality Description

We opted to build the muxes within the toplevel, since we can simply using an always_comb block to generate the 3 needed muxes in our design. The original use of modules for the muxes seemed overkill, and I like the look the the rtl has for the if statements when they are made in the top level.

Schematic

TODO. Show us your schematic for your mux(es).

RTL:



Milestone 3

Program Implementation

TODO. Program 1 software strategies for your hardware
We have only 4 registers, so we have a load/store style of software to make up for this.

Program 1 C code

```
//first compute the max by checking each pair
//max = 0;
//min = 255
//for(int i = 0; i < 64; i += 2){
//    for(int j = i+2; j < 64; j+=2){
//          sum = 0;
//          part = xor(arr[i], arr[j]);
//          mask = 00000001
//          for(int count = 0; count < 8; count ++){
//            sum += and(mask, part);
//            shiftLeft(part, 1);
//          }
//          part = xor(arr[i+1], arr[j+1]);
//          mask = 00000001
//          for(int count = 0; count < 8; count ++){
//            sum += and(mask, part);
//            shiftLeft(part, 1);
//          }
//        if(max < sum){
```

```
//            max = sum;
//        }
//        if(sum < min){
//            min = sum;
//        }
//    }
//}
//SETTING I
```

Program 1 Assembly

```
mov r0 0
str r0 r1 2 //store i val of 0 in 66
//
mov r0 b0000
mov r1 b1000 // 8
shift r1 b0011 // 8*8 = 64
str r0 r1 b01 //max dist set to 0 (memory slot 65)
mov r0 15 //15
shift r0 1 //shift by 1 to set r0 to 30
str r0 r1 b00 //min dist set to 30 (mem slot 64)
//START OF INNER FOR LOOP, SET J
mov r0 8// 8
shift r0 3 // 64
ldr r0 r0 2 //load in i
mov r1 2
saddto 0 0 r0 r1 //hold j in r0
mov r1 8
shift r1 3 // 64
str r0 r1 3 //begin with j = i+2
//
//INSIDE THE LOOP
```

```
mov r0 0 //sum = 0
mov r1 1 //mask = 00000001
mov r3 8
shift r3 3 //r3 = 64
ldr r2 r3 2 //load in i
ldr r3 r3 3 //load in j
ldr r2 r2 0 //load in arr[i]
ldr r3 r3 0 //load in arr[j]
xor r2 r2 r3 //xor(arr[i], arr[j])
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit0
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit1
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit2
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit3
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit4
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit5
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit6
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit7
```

```
ldr r2 r3 2 //load in i
ldr r3 r3 3 //load in j
ldr r2 r2 1 //load in arr[i+1]
ldr r3 r3 1 //load in arr[j+1]
xor r2 r2 r3 //xor(arr[i+1], arr[j+1])
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit0
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit1
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit2
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit3
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit4
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit5
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit6
shift r2 1
and r3 r2 r1 //and(part, mask)
saddto 0 0 r0 r3//sum += and(part, mask)//check bit7, sum = r0
mov r3 8 //at this point sum = r0
shift r3 3 // 64
ldr r1 r3 1 //load in max = r1
mov r2 2
```

```
blt r2 r0 r1 //skip update if sum < max
str r0 r3 1 //update max = sum
ldr r1 r3 0 //load in min = r1
blt r2 r1 r0 //skip update if min < sum
str r0 r3 0 //update min = sum
//INNER FOR LOOP ON J
mov r1 8
shift r1 3 // r1 = 64
ldr r0 r1 3 //load in j from slot 67
mov r3 2 //r3 = 2
saddto 0 0 r0 r3
str r0 r1 3 //j += 2
mov r3 b1011
shift r3 4 //get 10110000 in r3 aka -128+32+16 = -80
mov r2 1
saddto 0 0 r3 r2 //r3 = -80+1=-79
blt r3 r0 r1//branch -79 lines (up 79 lines) if j<64
//OUTER FOR LOOP ON I
mov r1 8
shift r1 3 // r1 = 64
ldr r0 r1 2 //load in i from slot 66
mov r3 2 //r3 = 2
saddto 0 0 r0 r3
str r0 r1 2 //i += 2
mov r3 b1010
shift r3 4 //get 10100000 in r3 aka -96
blt r3 r0 r1//branch -96 lines (up 96 lines) if i<64
```

TODO. Assembler

This assembler is within our github
TODO. Compiler, if you wrote one
N/A

Program 1 Machine code

010000000
010011000
011010011
000000101
010001111
011000001
000000100
010000000
000000110
010001000
011000011
001000010
010010010
100000001
010011000
011010011
000000111
010000000
010010001
010111000
011110011
001101110
001111111
001101000
001111100
110101011

110111001
100000011
011100001
110111001
100000011
011100001
110111001
100000011
011100001
110111001
100000011
011100001
110111001
100000011
011100001
110111001
100000011
011100001
110111001
100000011
011100001
110111001
100000011
001101110
001111111
001101001
001111101
110101011
110111001
100000011
011100001
110111001

100000011
011100001
110111001
100000011
011100001
110111001
100000011
011100001
110111001
100000011
011100001
110111001
100000011
011100001
110111001
100000011
011100001
110111001
100000011
010111000
011110011
001011101
010100010
101100001
000001101
001011100
101100100
000001100
010011000
011010011
001000111
010110010

100000000
000000111
010111011
011110100
010100001
100101110
101110001
010011000
011010011
001000110
010110010
100000000
000000110
010111010
011110100
101110001

TODO. Same for Programs 2 and 3

Program 2 C code

```
// Initialize min and max differences
minMSB = 255;
minLSB = 255;
maxMSB = 0;
maxLSB = 0;

// Loop through the array in pairs
for (i = 0; i < 64; i += 2) {
    for (j = i + 2; j < 64; j += 2) {
```

```
sign_mask = 0x80;
signA = arr[i] & sign_mask;
signB = arr[j] & sign_mask;

if (signA != signB) {
    // Different signs, addition
    diffLSB = arr[i + 1] + arr[j + 1];
    carry = (diffLSB < arr[i + 1]) ? 1 : 0;
    diffMSB = arr[i] + arr[j] + carry;
} else {
    // Same sign, subtraction
    if (arr[i] != arr[j]) {
        if (arr[i] > arr[j]) {
            diffLSB = arr[i + 1] - arr[j + 1];
            borrow = (diffLSB < 0) ? 1 : 0;
            diffLSB += (borrow ? 256 : 0);
            diffMSB = arr[i] - arr[j] - borrow;
        } else {
            diffLSB = arr[j + 1] - arr[i + 1];
            borrow = (diffLSB < 0) ? 1 : 0;
            diffLSB += (borrow ? 256 : 0);
            diffMSB = arr[j] - arr[i] - borrow;
        }
    } else {
        diffLSB = arr[i + 1] - arr[j + 1];
        if (diffLSB < 0) {
            diffLSB += 256;
            diffMSB = -1;
        } else {
            diffMSB = 0;
        }
    }
```

```
        }

        // Update max difference
        if ((diffMSB > maxMSB) || (diffMSB == maxMSB && diffLSB > maxLSB)) {
            maxMSB = diffMSB;
            maxLSB = diffLSB;
        }

        // Update min difference
        if ((diffMSB < minMSB) || (diffMSB == minMSB && diffLSB < minLSB)) {
            minMSB = diffMSB;
            minLSB = diffLSB;
        }
    }
}
```

Program 2 Assembly

```
//START OF ASSEMBLY CODE
//Set min and max MSB and LSB
mov r0 8
shift r0 3 //64
mov r1 0
saddto 1 1 r1 r1//subtract 1 from 0 to get 1111_1111
str r1 r0 2 //store minMSB in 66
str r1 r0 3 //store minLSB in 67
mov r1 4
saddto 0 0 r0 r1 //r0 = 68
mov r1 0
str r1 r0 0 //store maxMSB in 68
```

```
str r1 r0 1 //store maxLSB in 69
//Use memory slot 70 and 71 for i and j
str r1 r0 2 //i = 0
//set j = i+2
mov r0 8
shift r0 3 // 64
mov r1 6
saddto 0 0 r0 r1 //r0 = 70
ldr r1 r0 0 //get value of i
mov r2 2
saddto 0 0 r1 r2
str r1 r0 1 //j = i+2
//INSIDE LOOP
mov r0 b1000
shift r0 4 //create sign mask
//Loading in arr[i] arr[j]
mov r1 8
shift r1 3
mov r2 4
saddto 0 0 r1 r2 //70
ldr r2 r1 0 //r2 holds I
ldr r3 r1 1 //r3 holds j
ldr r2 r2 0 //r2 holds arr[i]
ldr r3 r3 0 //r3 holds arr[j]
and r0 r2 r0 //signA
mov r1 0
and r1 r3 r1 //signB
mov r3 15
shift r3 3 //120
mov r2 4
saddto 0 0 r3 r2 //124
blt r3 r0 r1 //skip down 124 lines if signA < signB
```

```
mov r3 15
mov r3 3 //120
mov r2 1
saddto 1 0 r3 r2 //119
blt r3 r1 r0 //skip down 119 lines if signB < signA
//IF THE SIGNS ARE EQUAL DO THIS:
mov r1 8
shift r1 3
mov r2 4
saddto 0 0 r1 r2 //70
ldr r2 r1 0 //r2 holds I
ldr r3 r1 1 //r3 holds j
ldr r2 r2 0 //r2 holds arr[i]
ldr r3 r3 0 //r3 holds arr[j]
mov r0 8
shift r0 3 //64
mov r1 7
saddto 0 0 r0 r1 //71
blt r0 r2 r3 //skip down 71 lines if arr[i] < arr[i]
mov r0 6
shift r0 3 //24
mov r1 3
saddto 0 0 r0 r1 //27
blt r0 r3 r2 //skip down 27 lines if arr[j] < arr[i]
//CASE WHERE: arr[j] == arr[i] SO diffMSB = 0
mov r0 8
shift r0 3 //64
mov r1 6
saddto 0 0 r0 r1 //r0=70
mov r1 0
str r1 r0 3 //UPDATE diffMSB = 0
ldr r2 r0 0 //load in i
```

```
ldr r3 r0 1 //load in j
ldr r2 r2 1 //load in arr[i+1]
ldr r3 r3 1 //load in arr[j+1]
mov r1 9
shift r1 3 //hold r1 = 72
mov r0 6
blt r0 r2 r3 //skip down 6 lines if arr[i+1] < arr[j+1]
saddto 1 0 r2 r3
str r2 r1 1 // update diffLSB = arr[i+1] - arr[j+1]
mov r1 3
mov r0 0
blt r1 r0 r1 //forced skip down 3
saddto 1 0 r3 r2
str r3 r1 1 //update diffLSB = arr[j+1] - arr[i+1]
//LARGE FORCED SKIP TO AVOID FALL THRU
mov r0 8
shift r0 3
mov r1 4
saddto 0 0 r0 r1 //68
blt r0 r0 r1 //Forced skip down 68 (r0) lines (to J for loop)
//
//
//IF SIGNS ARE EQUAL AND arr[j] < arr[i]
mov r1 8
shift r1 3
mov r0 6
saddto 0 0 r1 r0 //70
ldr r2 r1 0 //r2 = i
ldr r3 r1 1 //r3 = j
ldr r2 r2 1 //arr[i+1]
ldr r3 r3 1 //arr[j+1]
mov r0 13
```

```
blt r0 r2 r3 //jump down 13 lines if arr[i+1] < arr[j+1]
saddto 1 0 r2 r3 //this is fine because we know arr[j+1] <= arr[i+1]
str r2 r1 3 //update diffLSB to be arr[i+1] - arr[j+1]
ldr r2 r1 0 //r2 = i
ldr r3 r1 1 //r3 = j
ldr r2 r2 0 //arr[i]
ldr r3 r3 0 //arr[j]
saddto 1 0 r2 r3 //arr[i]-arr[j]
str r2 r1 2 //update diffMSB to be arr[i] - arr[j]
mov r0 9
mov r2 2
mov r3 3
blt r0 r2 r3 //forced jump down 9 lines
// if arr[i+1] < arr[j+1]
saddto 1 0 r2 r3
str r2 r1 3 //update diffLSB to be arr[i+1] - arr[j+1]
ldr r2 r1 0 //r2 = i
ldr r3 r1 1 //r3 = j
ldr r2 r2 0 //arr[i]
ldr r3 r3 0 //arr[j]
saddto 1 1 r2 r3 //arr[i]-arr[j]-1
str r2 r1 2 //update diffMSB to be arr[i] - arr[j] - 1
//Done with SIGNS ARE EQUAL AND arr[j] < arr[i]
mov r0 6
shift r0 2 //24
mov r1 7
saddto 0 0 r0 r1 //31
mov r2 2
mov r3 3
blt r0 r2 r3 //force skip down 31 lines until J for loop
//
//
```

```
//IF SIGNS ARE EQUAL AND arr[i] < arr[j]
//
//
mov r1 8
shift r1 3
mov r0 6
saddto 0 0 r1 r0 //70
ldr r2 r1 0 //r2 = i
ldr r3 r1 1 //r3 = j
ldr r2 r2 1 //arr[i+1]
ldr r3 r3 1 //arr[j+1]
mov r0 13
blt r0 r3 r2 //jump down 13 lines if arr[j+1] < arr[i+1]
//we know arr[j+1] >= arr[i+1]
saddto 1 0 r3 r2 //this is fine because we know arr[i+1] <= arr[j+1]
str r3 r1 3 //update diffLSB to be arr[i+1] - arr[j+1]
ldr r2 r1 0 //r2 = i
ldr r3 r1 1 //r3 = j
ldr r2 r2 0 //arr[i]
ldr r3 r3 0 //arr[j]
saddto 1 0 r3 r2 //arr[j]-arr[i]
str r3 r1 2 //update diffMSB to be arr[j]-arr[i]
mov r0 9
mov r2 2
mov r3 3
blt r0 r2 r3 //forced jump down 9 lines
//arr[j+1] < arr[i+1]
saddto 1 0 r3 r2
str r3 r1 3 //update diffLSB to be arr[j+1] - arr[i+1]
ldr r2 r1 0 //r2 = i
ldr r3 r1 1 //r3 = j
ldr r2 r2 0 //arr[i]
```

```
ldr r3 r3 0 //arr[j]
saddto 1 1 r3 r2 //arr[j]-arr[i]-1
str r3 r1 2 //update diffMSB to be arr[j] - arr[i] - 1
//Done with SIGNS ARE EQUAL AND arr[i] < arr[j]
mov r0 9
shift r0 2 //36
mov r1 8
saddto 0 0 r0 r1 //44
mov r2 2
mov r3 3
blt r0 r2 r3 //force skip down 44 lines until J for loop
//
//IF THE SIGNS ARE NOT EQUAL DO THIS:
mov r0 8
shift r0 3 //64
mov r1 6
saddto 0 0 r0 r1 //r0=70
ldr r2 r0 0 //load in i
ldr r3 r0 1 //load in j
ldr r2 r2 1 //load in arr[i+1]
ldr r3 r3 1 //load in arr[j+1]
saddto 0 0 r3 r2 //calculate diffLSB
str r3 r0 2 //diffLSB = arr[i + 1] + arr[j + 1];
mov r0 11
blt r0 r3 r2 //if diffLSB < arr[i+1] we have overflow error
mov r1 9 //INSERTED RANGE EXTENDER
mov r0 0
blt r1 r0 r1 //forced skip over range extender
mov r0 8 //RANGE EXTENDER FOR J LOOP
shift r0 4//-128
blt r1 1
blt r0 r0 r1//END OF RANGE EXTENDER FOR J LOOP
```

```
mov r0 8 //RANGE EXTENDER FOR I LOOP
shift r0 4//-128
blt r1 1
blt r0 r0 r1//END OF RANGE EXTENDER FOR I LOOP
//NO OVERFLOW ISSUES
mov r0 8 //If we dont skip, we add as normal
shift r0 3 //64
mov r1 6
saddto 0 0 r0 r1 //r0=70
ldr r2 r0 0 //load in i
ldr r3 r0 1 //load in j
ldr r2 r2 0 //load in arr[i]
ldr r3 r3 0 //load in arr[j]
saddto 0 0 r2 r3
str r2 r0 2 //diffMSB = arr[i] + arr[j]
//OVERFLOW ERROR
mov r0 8 //If we do skip, we add and include a carry
shift r0 3 //64
mov r1 6
saddto 0 0 r0 r1 //r0=70
ldr r2 r0 0 //load in i
ldr r3 r0 1 //load in j
ldr r2 r2 0 //load in arr[i]
ldr r3 r3 0 //load in arr[j]
saddto 0 1 r2 r3 //arr[i] + arr[j] + 1
str r2 r0 2 //diffMSB = arr[i] + arr[j] + 1
//DONE WITH if(signA != signB)
//
//DONE WITH ALL UPDATES OF diffMSB and diffLSB, update min and max
//update MAX difference
mov r0 8
shift r0 3 //64
```

```
mov r1 4
saddto 0 0 r0 r1 //r0 = 68
ldr r2 r0 0 //load r2 = maxMSB
saddto 0 0 r0 r1 //r0 = 72
ldr r3 r0 0// load r3 = diffMSB
mov r1 5
shift r1 3 //20
mov r0 4
saddto 0 0 r1 r0 //26
blt r1 r2 r3 // skip down 26 lines if maxMSB < diffMSB and update
mov r1 5
shift r1 3 // 20
mov r0 1
saddto 0 0 r1 r0 //21
blt r1 r3 r2 // skip down 21 lines if diffMSB < maxMSB and dont update
//fall thru means maxMSB == diffMSB
mov r0 8
shift r0 3 //64
mov r1 4
saddto 0 0 r0 r1 //r0 = 68
ldr r2 r0 1 //load r2 = maxLSB
saddto 0 0 r0 r1 //r0 = 72
ldr r3 r0 1// load r3 = diffLSB
mov r1 12
blt r1 r3 r2 //skip down 12 lines if diffLSB < maxLSB and dont update
//update MAX to diff
mov r0 8
shift r0 3 //64
mov r1 4
saddto 0 0 r0 r1 //r0 = 68
saddto 0 0 r0 r1 //r0 = 72
ldr r3 r0 0// load r3 = diffMSB
```

```
saddto 1 0 r0 r1 //r0 = 68
str r3 r0 0 //update maxMSB = diffMSB
saddto 0 0 r0 r1 //r0 = 72
ldr r3 r0 1 //load r3 = diffLSB
saddto 1 0 r0 r1 //r0 = 68
str r3 r0 1 //update maxLSB = diffLSB
//DONE WITH UPDATING MAX
//
//update MIN difference//////////////////
mov r0 8
shift r0 3 //64
mov r1 2
saddto 0 0 r0 r1 //r0 = 66
ldr r2 r0 0 //load r2 = minMSB
mov r1 6
saddto 0 0 r0 r1 //r0 = 72
ldr r3 r0 0// load r3 = diffMSB
mov r1 5
shift r1 3 //20
mov r0 4
saddto 0 0 r1 r0 //26
blt r1 r3 r2 // skip down 26 lines if diffMSB < minMSB and update
mov r1 5
shift r1 3 // 20
mov r0 1
saddto 0 0 r1 r0 //21
blt r1 r2 r3 // skip down 21 lines if minMSB < diffMSB and dont update
//fall thru means minMSB == diffMSB
mov r0 8
shift r0 3 //64
mov r1 2
saddto 0 0 r0 r1 //r0 = 66
```

```
ldr r2 r0 1 //load r2 = minLSB
mov r1 6
saddto 0 0 r0 r1 //r0 = 72
ldr r3 r0 1// load r3 = diffLSB
mov r1 12
blt r1 r2 r3 //skip down 12 lines if minLSB < diffLSB and dont update
//update MIN to diff
mov r0 8
shift r0 3 //64
mov r1 8
saddto 0 0 r0 r1 //r0 = 72
ldr r3 r0 0// load r3 = diffMSB
saddto 1 0 r0 r1 //r0 = 64
str r3 r0 2 //update minMSB = diffMSB
saddto 0 0 r0 r1 //r0 = 72
ldr r3 r0 1 //load r3 = diffLSB
saddto 1 0 r0 r1 //r0 = 64
str r3 r0 3 //update minLSB = diffLSB
//DONE WITH UPDATING MIN
//
//INNER LOOP IF J<64
mov r0 8
shift r0 3 //r0 = 64
mov r1 7
saddto 0 0 r0 r1 //r0 = 71
ldr r1 r0 0 //load j into r1
mov r2 2
saddto 0 0 r1 r2
str r1 r0 0 //j+=2 (update j)
mov r2 8
shift r2 3 //64
mov r3 b1000
```

```
shift r3 4 //-128
mov r0 9
saddto 0 0 r3 r0 //-119
blt r3 r1 r2 //branch up to line 191 by moving up 119 lines (-119) if j < 64
//OUTERLOOP IF I<64
mov r0 8
shift r0 3 //r0 = 64
mov r1 6
saddto 0 0 r0 r1 //70
ldr r1 r0 0 //load in i
mov r2 2
saddto 0 0 r1 r2
str r1 r0 0 //update i+=2
mov r2 8
shift r2 3 //64
mov r3 b1000
shift r3 4 //-128
blt r3 r1 r2 //branch up to extender at line 207 (-128) lines if i < 64
```

Program 2 Machine code

```
010001000
011000011
010010000
100110101
000010010
000010011
010010100
100000001
010010000
000010000
```

000010001
000010010
010001000
011000011
010010110
100000001
001010000
010100010
100000110
000010001
010001000
011000100
010011000
011010011
010100100
100000110
001100100
001110101
001101000
001111100
110001000
010010000
110011101
010111111
011110011
010100100
100001110
101110001
010111111
010110011
010100001
100101110

101110100
010011000
011010011
010100100
100000110
001100100
001110101
001101000
001111100
010001000
011000011
010010111
100000001
101001011
010000110
011000011
010010011
100000001
101001110
010001000
011000011
010010110
100000001
010010000
000010011
001100000
001110001
001101001
001111101
010011001
011010011
010000110

101001011
100101011
000100101
010010011
010000000
101010001
100101110
000110101
010001000
011000011
010010100
100000001
101000001
010011000
011010011
010000110
100000100
001100100
001110101
001101001
001111101
010001101
101001011
100101011
000100111
001100100
001110101
001101000
001111100
100101011
000100110
010001001

010100010
010110011
101001011
100101011
000100111
001100100
001110101
001101000
001111100
100111011
000100110
010000110
011000010
010010111
100000001
010100010
010110011
101001011
010011000
011010011
010000110
100000100
001100100
001110101
001101001
001111101
010001101
101001110
100101110
000110111
001100100
001110101

001101000
001111100
100101110
000110110
010001001
010100010
010110011
101001011
100101110
000110111
001100100
001110101
001101000
001111100
100111110
000110110
010001001
011000010
010011000
100000001
010100010
010110011
101001011
010001000
011000011
010010110
100000001
001100000
001110001
001101001
001111101
100001110

000110010
010001011
101001110
010011001
010000000
101010001
010001000
011000100
000000000
101000001
010001000
011000100
000000000
101000001
010001000
011000011
010010110
100000001
001100000
001110001
001101000
001111100
100001011
000100010
010001000
011000011
010010110
100000001
001100000
001110001
001101000
001111100

100011011
000100010
010001000
011000011
010010100
100000001
001100000
100000001
001110000
010010101
011010011
010000100
100000100
101011011
010010101
011010011
010000001
100000100
101011110
010001000
011000011
010010100
100000001
001100001
100000001
001110001
010011100
101011110
010001000
011000011
010010100
100000001

100000001
001110000
100100001
000110000
100000001
001110001
100100001
000110001
010001000
011000011
010010010
100000001
001100000
010010110
100000001
001110000
010010101
011010011
010000100
100000100
101011110
010010101
011010011
010000001
100000100
101011011
010001000
011000011
010010010
100000001
001100001
010010110

100000001
001110001
010011100
101011011
010001000
011000011
010011000
100000001
001110000
100100001
000110010
100000001
001110001
100100001
000110011
010001000
011000011
010010111
100000001
001010000
010100010
100000110
000010000
010101000
011100011
010111000
011110100
010001001
100001100
101110110
010001000
011000011

```
010010110
100000001
001010000
010100010
100000110
000010000
010101000
011100011
010111000
011110100
101110110
```

Program 3 C code

```
#include <stdint.h>
#include <stdio.h>

// Memory locations
uint8_t data_mem[128];

// Function prototype
void double_precision_multiply();

int main() {
    // Example: Initialize operands in memory (you can change these values for testing)
    data_mem[0] = 0x12; // A1 MSB
    data_mem[1] = 0x34; // A1 LSB
    data_mem[2] = 0x56; // B1 MSB
    data_mem[3] = 0x78; // B1 LSB
    // Add more initializations if needed
```

```
    // Perform multiplication
    double_precision_multiply();

    return 0;
}

void double_precision_multiply() {
    for (int i = 0; i < 16; ++i) {
        // Read operands
        uint16_t a = ((uint16_t)data_mem[4*i] << 8) | data_mem[4*i+1];
        uint16_t b = ((uint16_t)data_mem[4*i+2] << 8) | data_mem[4*i+3];

        // Perform shift-and-add multiplication
        uint32_t result = 0;
        uint32_t multiplicand = a;
        uint32_t multiplier = b;

        while (multiplier != 0) {
            if (multiplier & 1) {
                result += multiplicand;
            }
            multiplicand <<= 1;
            multiplier >>= 1;
        }

        // Store the result
        data_mem[64 + 4*i] = (result >> 24) & 0xFF;
        data_mem[64 + 4*i+1] = (result >> 16) & 0xFF;
        data_mem[64 + 4*i+2] = (result >> 8) & 0xFF;
        data_mem[64 + 4*i+3] = result & 0xFF;
    }
}
```

Program 3 Assembly

```
//start
//memory locations used
//counter (128), multiplier (129), multiplicand (130), result (140), (me using data_mem is a place holder for where the array starts)
ldr r0 data_mem 00//loads the data_mem address to r0
mov r3 #0x00 // loop counter
loop:
    mov r2 #0x80 //memory location of counter
    ldr r3 r2 00// ldr counter from memory location 128
    mov r2 #0b10000 //stores the number 16 into r2 to compare to the counter
    blt exit r2 r3 //r2(16) < counter then we branch to the exit
    ldr r0 data_mem 00//loads datamem address into r0
    shift r3 #0b1110 // shifts r3 to the left 2 for 4*i
    saddto 0 0 r0 r3 // creates the addres for data_mem of 4*i
    ldr r1 r0 00// load the value from memory location data_mem[4*i]
    mov r2 #0b01 // stores 1 into r2
    saddto 0 0 r0 r2 //add 1 to the memory location to create data_mem[4*i+1] to next the next value
    ldr r2 r0 00// load the value from memory location data_mem[4*i+1]
    mov r3 #0b1000 // -8 into r3
    shift r1 r3 //shifts left 8 times for r1
//
    //idk how to use orr
    //orr r1 r1 r2 //this combines the 2 together
    mov r3 #0x81 //store memory location 129 into r3
    str r1 r3 //store the combination of r1 and r2 into r3 (memory location 129)
//
```

```
    mov r2 #0x80 //memory location of counter
    ldr r3 r2 00// ldr counter from memory location 128
    ldr r0 data_mem 00//loads datamem address into r0
    shift r3 #0b1110 // shifts r3 to the left 2 for 4*i
    mov r1 0b10 // 2 into r1
    saddto 0 0 r3 r1 // 4*i + 2
    saddto 0 0 r0 r3 // creates the address for data_mem of 4*i+2
    ldr r1 r0 00// load the value from memory location data_mem[4*i+2]
    mov r2 #0b01 // stores 1 into r2
    saddto 0 0 r0 r2 //4*i+2 + 1 to create data_mem[4*i+3] to next the next value
    ldr r2 r0 00// load the value from memory location data_mem[4*i+1]
    mov r3 #0b1000 // -8 into r3
    shift r1 r3 //shifts left 8 times for r1
//
    //idk how to use orr
    //orr r1 r1 r2 //this combines the 2 together
    mov r3 #0x82 //store memory location 130 into r3
    str r1 r3 //store the combination of r1 and r2 into r3 (memory location 130)
//
    mov r3 0b0000 //initialize result
    mov r2 0x8c //memory location 140 for result location
    str r3 r2 00 //stores r3 into memory locatin 140
multiply_loop:
    mov r0 #0x81 //memory location 129 for the first number we got
    ldr r1 r0 00 //loads value from location 129 into r1
    mov r0 #0x82 // location 130 for the second number we got
    ldr r2 r0 00 //loads value from location 130 into r2
    mov r3 0b0001 // 1 into r3
    blt store r2 r3// r2 < 1 once r2 is less than 1 then we branch to the store branch
//
    //**check to see if the multiplier has an lsb of 1** <--write this part. This part is written under--> if so then result += multiplicand
    //figure out how to write this
```

```
    //if it doesnt then skip adding and branch to the shift
    mov r0 0x8c // result memory location
    ldr r3 r0 00 //load the value from the result memory location into r3
    saddto 0 0 r3 r2 // result += multiplicand
    str r3 r0 00 //store updated result back into result memory location
shift:
    mov r0 0b1111 // -1
    shift r2 r0 //shifts to the left 1 for the multiplicand
    mov r3 0x81 //memory location for multiplicand
    str r2 r3 00 //stores updated multiplicand
    mov r0 0b0001 // 1
    shift r1 r0 // shifts to the right 1 for the multiplier
    mov r3 0x82 //memory location for multiplier
    str r1 r3 00 //stores updated multiplier
//
    //brain not functioning so ill just write out what should be here its definitely not correct based on our processor lol
//   b multiply_loop
store:
    ldr r0 data_mem 00 //loads data_mem address into r0
    mov r1 0b01000000 // 64 into r1
    saddto 0 0 r0 r1 // creates data_mem[64]
    mov r1 0x80 // counter memory location
    shift r1 0b1110 //shift 2 to the left for the counter to multiply by 4
    saddto 0 0 r0 r1 // creates data_mem[64 + 4*1]
    //DONT CHANGE r0 UNTIL WE NEED TO GO TO THE NEXT MEMORY LOCATION TO STORE
    mov r2 0x8c //memory location for result
    ldr r1 r2 00 //gets value within result memory location
    str r1 r0 //stores result into memory location that is in r0 data_mem[64 + 4*i]
//
    shift r1 0b0100 //shifts right by 4
    shift r1 0b0100 //shifts right by 4
    mov r3 0b1 // 1 in r3
```

```
    saddto 0 0 r0 r3 //adds 1 to r0 to get to the next memory location to store result
    str r1 r0 00 //stores updated result into the next memory location data_mem[64 + 4*i + 1]
//
    shift r1 0b0100 //shifts right by 4
    shift r1 0b0100 //shifts right by 4
    saddto 0 0 r0 r3 //adds 1 to r0 to get to the next memory location to store result
    str r1 r0 00 //stores updated result into the next memory location data_mem[64 + 4*i + 2]
//
    shift r1 0b0100 //shifts right by 4
    shift r1 0b0100 //shifts right by 4
    saddto 0 0 r0 r3 //adds 1 to r0 to get to the next memory location to store result
    str r1 r0 00 //stores updated result into the next memory location data_mem[64 + 4*i + 3]
//
    //can change r0 now with no reprecussions
    mov r0 0x80 // loop counter memory location
    ldr r1 r0 00 // load the value that is stored in loop counter memory location
    mov r0 0b0001 // 1 in r0
    saddto 0 0 r1 r0 // r1 = r1 + r0
    mov r0 0x80 // loop counter memory location
    str r1 r0 00 //stores updated loop counter
//
    //idk how to force a branch so ill just leave this here for now
//    b loop
end:
```

Program 3 machine code

```
001000000
010110000
000000000
010100000
```

001111000
010100000
101001011
001000000
011110000
100000011
001010000
010100000
100000010
001100000
010110000
011010000
000000000
000000000
010110000
000000000
010100000
001111000
001000000
011110000
010010000
100001101
100000011
001010000
010100000
100000010
001100000
010110000
011010000
000000000
000000000
010110000

000000000
010110000
010100000
000111000
000000000
010000000
001010000
010000000
001100000
010110000
101001011
000000000
000000000
000000000
010000000
001110000
100001110
000110000
000000000
010000000
011100000
010110000
000101100
010000000
011010000
010110000
000011100
000000000
000000000
000000000
001000000
010010000

100000001
010010000
011010000
100000001
000000000
010100000
001011000
000000000
011010000
011010000
010110000
100000011
000010000
011010000
011010000
100000011
000010000
011010000
011010000
100000011
000010000
000000000
010000000
001010000
010000000
100000100
010000000
000010000
000000000
000000000
000000000

Changelog

TODO. have a bulleted list of your changes here. Example below:

- Milestone 3
    - We added the software based on our operations
        - Created program 1
        - Created program 2
        - Created program 3
- Milestone 2
    - We created the hardware based on our design from milestone 1
        - We decided to add in more functionality to our saddto, and have it work as both add and subtract
    - TODO: Finish writing all of the sv files and modules.
- Milestone 1
    - Initial version, all details can be found on the milestone 1 document.
    - Included milestone 1 information from previous doc into milestone 3 document