Milestone 1

Date Created: 1/24/2022

Date Last Updated: 2/23/2022

Group Members: Daisuke Chon, Angelica Consengco, Matthew Larkins

Component 1: Introduction

Our group's processor, named "3BC" is designed to perform the following 3 programs:

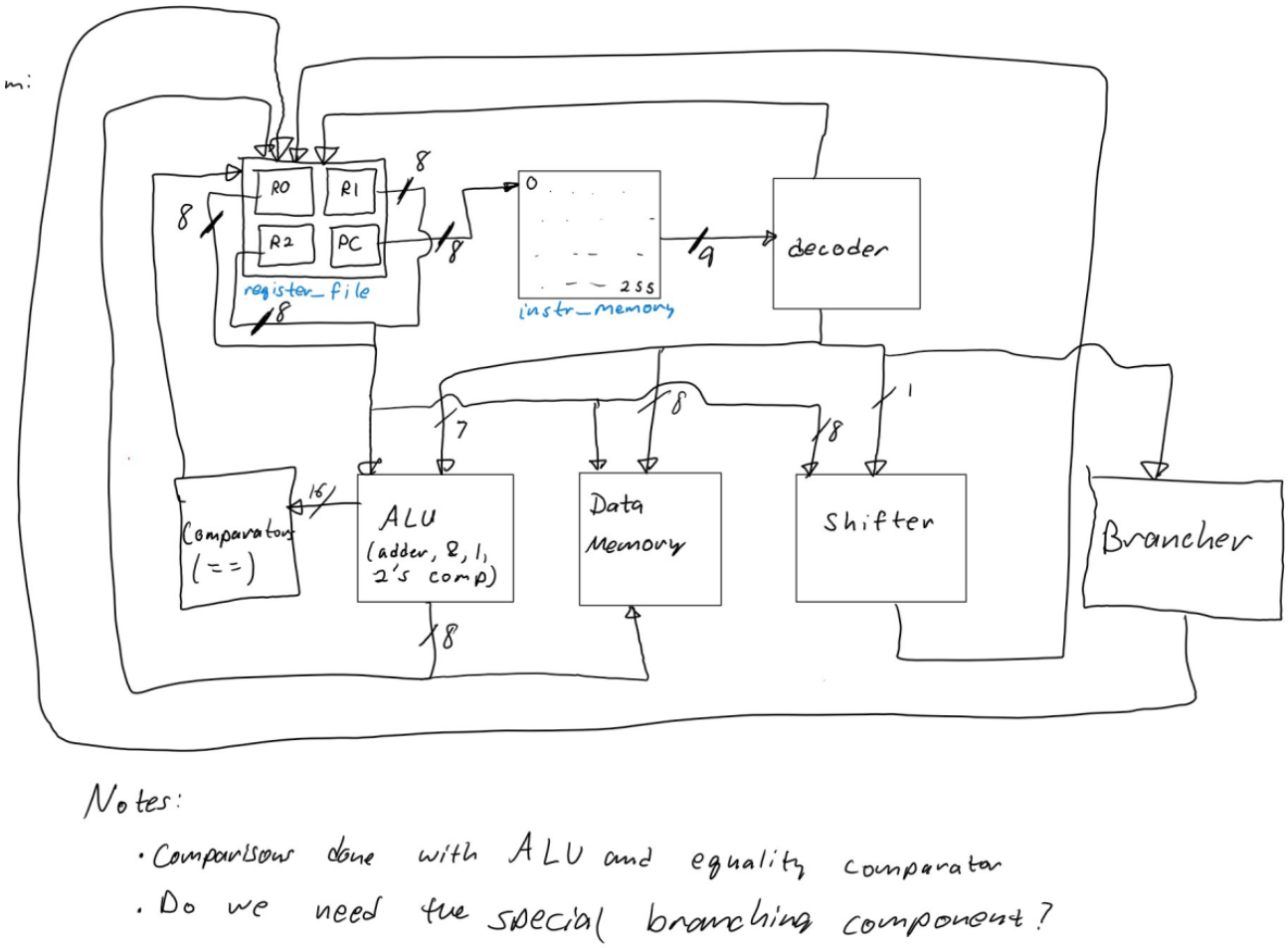
1. Hamming Encoding of 11-bit strings
2. SECDED Hamming Decoding of 11-bit strings
3. Pattern matching 5-bit strings on 11-bit strings

The machine takes major influence from typical load-store architectures. Although being constrained to 9 bits in our ISA, we strived to come up with a processor that would make writing the three assigned programs a relatively simple endeavor. As such, we agreed we should try to have 4 general purpose registers. This should also help make the programs run faster.

To achieve this, one of the things we do differently from a typical load-store architecture is that when we load and store registers, we use one of the two as both a source and a destination. This allows for a more efficient use of space per instruction because we only need 4 bits to specify which registers we will use during execution.

Another thing we do differently is that our I-Type instructions have 3 bits dedicated to representing immediate values. We represent any 8 bit number in the system via shifting and adding (see demonstration in Component 4).

Component 2: Architectural Overview



Full image should also be available in the submission

Component 3: Machine Specification

Instruction Formats:

R-type: 4-Bits Opcode | 2-Bits Rd | 2-Bits Rs | 1-Bit Unused

Example:

```
add r0, r1   <--> 1011 00 01 x
r0 = r0 + r1 <--> add  r0 r1 x
x = unused
```

I-type: 4-Bits Opcode | 2-Bits Rs | 3-Bits Immediate

Example:

```
ldi r0, 0 <--> 0100 00 000
r0 = 0    <--> 0100 00 000
```

Note: The max value we can represent naively with 3 bits is 7. We can represent values larger than this by repeatedly operating on values stored in 8-bit registers until they contain our desired values.

Operations:

Opcode	Format	Operation	Assembly	Example	Machine Code
0000	I	left shift	ls	ls r0, #2	0000 00 010
0001	I	right shift	rs	rs r0, #2	0001 00 010
0010	R	AND	and	and r0, r1	0010 00 01 X
0011	R	OR	or	or r0, r1	0011 00 01 X
0100	I	load immediate	ldi	ldi r0, #7	0100 00 111
0101	R	load from datamem into register	ldr	ldr r2, (r0)	0101 10 00 X
0110	R	store from register into datamem	str	str (r2), r0	0110 10 00 X
0111	R	branch if not equal to zero	bnz	bnz r0, r1	0111 00 01 X
1000	R	>=	geq	geq r0, r1	1000 00 01 X
1001	R	==	eq	eq r0, r1	1001 00 01 X
1010	R	2's Comp Negation	neg	neg r1, r1	1010 01 01 X
1011	R	add	add	add r0, r1	1011 00 01 X
1100	R	add immediate	addi	addi r0, #3	1100 00 011
1101	R	!=	neq	neq r0, r1	1101 00 01 X
1110	X	nop			
1111	X	nop			

Internal Operands/Registers:

- r0 : 00 - general purpose
- r1 : 01 - general purpose
- r2 : 10 - general purpose
- r3 : 11 - general purpose
- PC : special (Program counter)

Control Flow (Branches):

We will support one branching instruction, branch if not zero (bnz). This will be an R-Type instruction of the form

```
bnz destination, source
```

e.g.

`bnz R2, R1`

where `R2` is the *destination* in the sense that it holds the index to the destination address in a lookup table and `R1` is the *source* in the sense that it is the source value to compare with zero for equality.

Given that our assembly code doesn't employ too many branching and the parts that do do not jump more than 50 - 100 instructions, all jumps will be written as an offset addition to where the program counter is currently at. For example, if we want to jump from line 300 to 250, we would specify for the PC register to have its value subtracted by 50 (that is, by adding by a negative number using the NEG instruction). If we want to jump from there to line 350, we would add 100 to the value stored in the PC register. There are labels written in our assembly code but that should be thought of as a comment for the programmer to know what block/section the code is jumping to.

Programatically, jumping from line 300 to line 250 would look like the following:

```
// Create the number 50
add r1, #6
lsl r1, #3
add r1, #2
// Make it negative
neg r1, r1
// Now make sure r2 is equal to 1 (or rather, not equal to zero) to make the jump.
For demonstration purposes, this will be hardcoded in but should be done via
comparison operations.
ldi r2, #1
// Perform the bnz operation. bnz adds the offset in r1 to the value stored in the
program counter.
bnz r1, r2
```

Addressing Modes:

We support addressing exclusively via indexing to a lookup table. This is reasonable for the 3BC processor because it is designed to perform 3 static procedures where addresses are the same for every use. Thus, there is no need for supporting generalized addressing - we know in advance the finite number and values of the addresses the programs will be using. It will be simpler to just look them up, then, and not implement unnecessary functionality like calculating a destination address from any offset that can be given by an instruction.

The three instructions in the 3BC ISA that use addressing are `str`, `ldr`, and `bnz`. In the cases of `ldr` and `str`, addresses are indexes to bytes in the 256-byte data memory, whereas in the case of `bnz`, addresses represent addresses of instructions in the separate instruction memory. Both sets of addresses can be stored in a lookup table and accessed/used similarly.

Component 4: Programmer's Model

As mentioned in the introduction, our machine takes inspriation from a typical load-store architecture. As such, if a programmer wants to do anything, they must load values into a register, conduct the operation, and

then store it back.

If a programmer wants to use a specific 8-bit value, then they can use ldi and conduct shifting and masking operations to create any desired 8-bit value. For example:

Task: put value 01110101 in r1

```
ldi r1, #3    -- r1: 00000011
ls r1, #3     -- r1: 00011000
addi r1, #5   -- r1: 00011101
ls r1, #2     -- r1: 01110100
addi r1, #1   -- r1: 01110101
```

Component 5: Program Implementations

Program 1:

```
// Describe Hardware to assembler
.arch armv6
.cpu  cortex-a53
.syntax unified

// Constants


Func1_start:
    // Registers available: r0 - r2

    // load from data mem[0:29] store to data mem[30:59]
    ldi r0, #0          // Prepare to load data mem[0]
    str r0, [215]       // store data mem[0] for later
    ldi r0, #30         // load data mem[30]
    str r0, [216]       // store data mem[30] for later

Func1_loop:
    ldr r0, [215]       // Load current input address
    ldr r1, (r0)        // Load LSW of input string into r1

    addi r0, #1         // Prepare to load MSW of input string
ldr r2, (r0)    // Load MSW of input string into r2

    addi r0, #1         // Increment input address for the next input if
applicable
    str r0, [215]   // store input address to use later

    // store LSW and MSW vals into memory. From here, we'll just call everything
by its decimal number for brevity
```

```
str r1, [195]
str r2, [196]

// store each individual bit of the string into its own memory address
// b1 (We're now on the LSB)
ls r1, #7
rs r1, #7
str r1, [197]
ldr r1, [195]

//b2
ls r1, #6
rs r1, #7
str r1, [198]
ldr r1, [195]

//b3
ls r1, #5
rs r1, #7
str r1, [199]
ldr r1, [195]

//b4
ls r1, #4
rs r1, #7
str r1, [200]
ldr r1, [195]

//b5
ls r1, #3
rs r1, #7
str r1, [201]
ldr r1, [195]

//b6
ls r1, #2
rs r1, #7
str r1, [202]
ldr r1, [195]

//b7
ls r1, #1
rs r1, #7
str r1, [203]
ldr r1, [195]

//b8
rs r1, #7
str r1, [204]

//b9 (We're now on the MSB)
```

```
    ls r2, #7
    rs r2, #7
    str r2, [205]
    ldr r2, [196]

    //b10
    ls r2, #6
    rs r2, #7
    str r2, [206]
    ldr r2, [196]

    //b11
    rs r2, #2
    str r2, [207]
    ldr r2, [196]

    // parity bit creation
    // p8 = ^(b11:b5)
p8_loop:
    // b11^b10
    ldi r2, #207
    ldr r0, (r2)
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // (b11^b10)^b9
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ((b11^b10)^b9)^b8
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b8)^b7
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b7)^b6
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b6)^b5
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1

p8_end:
    //store p8 for when we process p16
    str r0, [210]
    //start building the final string
    // b11
```

```
    ldr r1, [207]
    ls r1, #1
    // b10
    ldr r2, [206]
    or r1, r2
    ls r1, #1
    // b9
    ldr r2, [205]
    or r1, r2
    ls r1, #1
    // b8
    ldr r2, [204]
    or r1, r2
    ls r1, #1
    // b7
    ldr r2, [203]
    or r1, r2
    ls r1, #1
    // b6
    ldr r2, [202]
    or r1, r2
    ls r1, #1
    // b5
    ldr r2, [201]
    or r1, r2
    ls r1, #1
    // insert p8 here. Now the MSW is finished
    or r1, r0
    ls r1, #1
    str r1, [208]

p4_loop:
    // p4 = ^(b11:b8, b4, b3, b2)
    // b11^b10
    ldi r2, #207
    ldr r0, (r2)
    add r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // (b11^b10)^b9
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ((b11^b10)^b9)^b8
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b8)^b4
    ldi r2, #200
    ldr r1, (r2)
    neq r0, r1
```

```
    // ^(b11:b8)^b4^b3
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b8)^b4^b3^b2
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1

p4_end:
    // store p4 for when we process p16
    str r0, [211]
    // b4
    ldr r1, [200]
    ls r1, #1
    // b3
    ldr r2, [199]
    or r1, r2
    ls r1, #1
    // b2
    ldr r2, [198]
    or r1, r2
    ls r1, #1
    // insert p4 here
    or r1, r0
    ls r1, #1
    str r1, [209]

p2_loop:
    // p2 = ^(b11, b10, b7, b6, b4, b3, b1)
    // b11^b10
    ldi r2, #207
    ldr r0, (r2)
    add r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // (b11^b10)^b7
    ldi r2, #203
    ldr r1, (r2)
    neq r0, r1
    // ((b11^b10)^b7)^b6
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // (b11^b10^b7^b6)^b4
    ldi r2, #200
    ldr r1, (r2)
    neq r0, r1
    // (b11^b10^b7^b6)^b4^b3
    addi r2, #(-1)
    ldr r1, (r2)
```

```
    neq r0, r1
    // (b11^b10^b7^b6)^b4^b1
    ldi r2, #197
    ldr r1, (r2)
    neq r0, r1

p2_end:
    // store p2 for when we process p16
    str r0, [212]
    // b1
    ldr r1, [209]
    ldr r2, [197]
    or r1, r2
    ls r1, #1
    // insert p2 here
    or r1, r0
    ls r1, #1
    str r1, [209]

p1_loop:
    // p1 = ^(b11, b9, b7, b5, b4, b2, b1)
    // b11^b9
    ldi r2, #207
    ldr r0, (r2)
    ldi r2, #205
    ldr r1, (r2)
    neq r0, r1
    // (b11^b9)^b7
    ldi r2, #203
    ldr r1, (r2)
    neq r0, r1
    // ((b11^b9)^b7)^b5
    ldi r2, #201
    ldr r1, (r2)
    neq r0, r1
    // (b11^b9^b7^b5)^b4
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // (b11^b9^b7^b5^b4)^b2
    ldi r2, #198
    ldr r1, (r2)
    neq r0, r1
    // (b11^b9^b7^b5^b4^b2)^b1
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1

p1_end:
    // store p1 for when we process p16
    str r0, [213]
```

```
    // insert p1 here
    ldr r1, [209]
    or r1, r0
    ls r1, #1
    str r1, [209]

p16_loop:
    // p16 = ^(b11:1, p8, p4, p2, p1)
    // b11^b10
    ldi r2, #207
    ldr r0, (r2)
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // (b11^b10)^b9
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ((b11^b10)^b9)^b8
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b8)^b7
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b7)^b6
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b6)^b5
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b5)^b4
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b4)^b3
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b3)^b2
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b2)^b1
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b1)^p8
```

```
    ldi r2, #210
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b1)^p8^p4
    addi r2, #1
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b1)^p8^p4^p2
    addi r2, #1
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b1)^p8^p4^p2^p1
    addi r2, #1
    ldr r1, (r2)
    neq r0, r1

p16_end:
    // insert p16
    ldr r1, [209]
    or r1, r0
    str r1, [209]

gen_word:
    // load memory address we need to load finalized output into
    // store lsw
    ldr r0, [216]
    ldr r1, [208]
    str r1, (r0)
    // store msw
    addi r0, #1
    ldr r1, [209]
    str r1, (r0)
    // increment output mem storage address for the next string
    addi r0, #1
    str r0, [216]
    // check if we're done
    ldi r1, #59
    ldr r2, Func1_loop
    neq r0, r1
    ldi r1, #1
    beq r0, r1

end:
// yay we're done
    fin
```

Program 2: // Describe Hardware to assembler .arch armv6 .cpu cortex-a53 .syntax unified

```
// Constants

// For brevity, we'll just call everything by its decimal number
Func2_start:
    // Registers available: r0 - r2

    // load from data mem[64:93] store to data mem[94:123]
    ldi r0, #64         // Prepare to load data mem[64]
    str r0, [224]       // store data mem[0] for later
    ldi r0, #94         // load data mem[94]
    str r0, [225]       // store data mem[30] for later

Func2_loop:
    ldr r0, [221]       // Load current input address
    ldr r1, (r0)        // Load LSW of input string into r1

    addi r0, #1         // Prepare to load MSW of input string
  ldr r2, (r0)    // Load MSW of input string into r2

    addi r0, #1         // Increment input address for the next input if
applicable
    str r0, [221]   // store input address to use later

    // store LSW and MSW vals into memory.
    str r1, [195]
    str r2, [196]

    // store each individual bit of the string into its own memory address
    // p16 (We're now on the LSB)
    ls r1, #7
    rs r1, #7
    str r1, [197]
    ldr r1, [195]

    //p1
    ls r1, #6
    rs r1, #7
    str r1, [198]
    ldr r1, [195]

    //p2
    ls r1, #5
    rs r1, #7
    str r1, [199]
    ldr r1, [195]

    //b1
    ls r1, #4
    rs r1, #7
    str r1, [200]
```

```
ldr r1, [195]

//p4
ls r1, #3
rs r1, #7
str r1, [201]
ldr r1, [195]

//b2
ls r1, #2
rs r1, #7
str r1, [202]
ldr r1, [195]

//b3
ls r1, #1
rs r1, #7
str r1, [203]
ldr r1, [195]

//b4
rs r1, #7
str r1, [204]

//p8 (We're now on the MSB)
ls r2, #7
rs r2, #7
str r2, [205]
ldr r2, [196]

//b5
ls r2, #6
rs r2, #7
str r2, [206]
ldr r2, [196]

//b6
ls r2, #5
rs r2, #7
str r2, [207]
ldr r2, [196]

//b7
ls r2, #4
rs r2, #7
str r2, [208]
ldr r2, [196]

//b8
ls r2, #3
rs r2, #7
```

```
    str r2, [209]
    ldr r2, [196]

    //b9
    ls r2, #2
    rs r2, #7
    str r2, [210]
    ldr r2, [196]

    //b10
    ls r2, #1
    rs r2, #7
    str r2, [211]
    ldr r2, [196]

    //b11
    rs r2, #7
    str r2, [212]

p8_exp:
    // b11^b10
    ldi r2, #212
    ldr r0, (r2)
  ldi r1, #1
    eq r0, r1

    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // (b11^b10)^b9
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ((b11^b10)^b9)^b8
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b8)^b7
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b7)^b6
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b6)^b5
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1

p8_comparison:
```

```
    str r0, [213] //store expected p8 in address 213
    //compare actual parity with expected parity
    // load p8
    ldr r1, [205]
    eq r0, r1 //check if parity bits match
    str r0, [220] //store result in address 220

p4_exp:
    // p4 = ^(b11:b8, b4, b3, b2)
    // b11^b10
    ldi r2, #212
    ldr r0, (r2)
    add r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // (b11^b10)^b9
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ((b11^b10)^b9)^b8
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b8)^b4
    ldi r2, #204
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b8)^b4^b3
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b8)^b4^b3^b2
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1

p4_comparison:
    str r0, [214] //store expected p4 in address 214
    //compare actual parity with expected parity
    // load p4
    ldr r1, [201]
    eq r0, r1 //check if parity bits match
    str r0, [221] //store result in address 221

p2_exp:
// p2 = ^(b11, b10, b7, b6, b4, b3, b1)
    // b11^b10
    ldi r2, #212
    ldr r0, (r2)
    add r2, #(-1)
    ldr r1, (r2)
```

```
    neq r0, r1
    // (b11^b10)^b7
    ldi r2, #208
    ldr r1, (r2)
    neq r0, r1
    // ((b11^b10)^b7)^b6
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // (b11^b10^b7^b6)^b4
    ldi r2, #204
    ldr r1, (r2)
    neq r0, r1
    // (b11^b10^b7^b6)^b4^b3
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // (b11^b10^b7^b6)^b4^b1
    ldi r2, #200
    ldr r1, (r2)
    neq r0, r1

p2_comparison:
    str r0, [215] //store expected p2 in address 215
    //compare actual parity with expected parity
    // load p8
    ldr r1, [199]
    eq r0, r1 //check if parity bits match
    str r0, [222] //store result in address 222


p1_exp:
    // p1 = ^(b11, b9, b7, b5, b4, b2, b1)
    // b11^b9
    ldi r2, #212
    ldr r0, (r2)
    ldi r2, #210
    ldr r1, (r2)
    neq r0, r1
    // (b11^b9)^b7
    ldi r2, #208
    ldr r1, (r2)
    neq r0, r1
    // ((b11^b9)^b7)^b5
    ldi r2, #206
    ldr r1, (r2)
    neq r0, r1
    // (b11^b9^b7^b5)^b4
    ldi r2, #204
    ldr r1, (r2)
    neq r0, r1
```

```
    // (b11^b9^b7^b5^b4)^b2
    ldi r2, #202
    ldr r1, (r2)
    neq r0, r1
    // (b11^b9^b7^b5^b4^b2)^b1
    ldi r2, #200
    ldr r1, (r2)
    neq r0, r1



p1_comparison:
    str r0, [216] //store expected p1 in address 216
    //compare actual parity with expected parity
    // load p1
    ldr r1, [198]
    eq r0, r1 //check if parity bits match
    str r0, [223] //store result in address 223



p16_loop:
    // p16 = ^(b11:1, p8, p4, p2, p1)
    // b11^b10
    ldi r2, #212
    ldr r0, (r2)
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // (b11^b10)^b9
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ((b11^b10)^b9)^b8
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b8)^b7
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b7)^b6
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b6)^b5
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b5)^b4
    addi r2, #(-1)
```

```
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b4)^b3
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b3)^b2
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b2)^b1
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b1)^p8
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b1)^p8^p4
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b1)^p8^p4^p2
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1
    // ^(b11:b1)^p8^p4^p2^p1
    addi r2, #(-1)
    ldr r1, (r2)
    neq r0, r1



p16_comparison:
    str r0, [220] //store expected p16 in address 220
    //compare actual parity with expected parity
    // load p16
    ldr r1, [197]
    eq r0, r1 //check if parity bits match
    str r0, [224] //store result in address 224

check_error:
    //r2 will store number of errors
    ldi r2, #0
    //p8
    ldr r0, [220] //get result of comparing expected parity with actual parity
    add r2, r0
    //p4
    ldr r0, [221] //get result of comparing expected parity with actual parity
    add r2, r0
    //p2
```

```
    ldr r0, [222] //get result of comparing expected parity with actual parity
    add r2, r0
    //p1
    ldr r0, [223] //get result of comparing expected parity with actual parity
    add r2, r0

    // check number of errors
    ldi r1, #4
    // if number of errors == 0
    ldr r0, r2
    ldr r2, output
    beq r0, r1
    // if number of errors == 1
    ldi r1, #3
    ldr r2, error_correction
    beq r0, r1
    // if number of errors == 2
    ldi r1, #2
    ldr r2, error_detection
    beq r0, r1

error_correction:
    // calculate which bit is incorrect
    ldr r0, [220]
    ls r0, #1
    ldr r1, [221]
    or r0, r1
    ls r0, #1
    ldr r1, [222]
    or r0, r1
    ls r0, #1
    ldr r1, [223]
    or r0, r1

    // Find the incorrect bit and flip it
    ldi r1, #197
    add r1, r0
    ldr r2, (r0)
    ldi r1, #0
    // if incorrect bit = 1, eq 1, 0 gives us 0. If incorrect bit = 0, eq 0, 0
gives us 1
    eq r2, r1
    str r2, (r0)
    // jump to output
    ldi r0, #0
    ldr r2, output
    beq r0, r1

error_detection:
    //if p16 is correct (as expected) then there is a two-bit error
    ldi r2, #1
```

```
    //load result of comparing expected parity with actual parity
    ldr r0, [224]
    //if p16_exp == p16
    eq r2, r0
    //we assume we do nothing if two errors are detected (from a Piazza post)

output:
    //LSW
    //b8
    ldr r0, [209]
    ls r0, #1
    //b7
    ldr r1 [208]
    or r0, r1
    ls r0, #1
    //b6
    ldr r1, [207]
    or r0, r1
    ls r0, #1
    //b5
    ldr r1, [206]
    or r0, r1
    ls r0, #1
    //b4
    ldr r1, [204]
    or r0, r1
    ls r0, #1
    //b3
    ldr r1, [203]
    or r0, r1
    ls r0, #1
    //b2
    ldr r1, [202]
    or r0, r1
    ls r0, #1
    //b1
    ldr r1, [200]
    or r0, r1
    // store LSW
    // load output address for LSW
    ldr r1, [225]
    // store LSW in output address
    str r0, (r1)
    // increment output address
    addi r1, #1

    //MSW
    //b11
    ldr r0, [212]
    ls r0, #1
    //b10
```

```
    ldr r1, [211]
    or r0, r1
    ls r0, #1
    //b9
    ldr r1, [210]
    or r0, r1
    ls r0, #1


    //store MSW
    // we already have output address in r1
    // store MSW in output address
    str r0, (r1)
    // increment output address
    addi r1, #1
    // store output address
    str r1, [225]

    // check if we're done
    ldi r1, #123
    ldr r2, Func2_loop
    neq r0, r1
    ldi r1, #1
    beq r0, r1

end:
    // we're done, yay
    fin
```

Program 3: // string in data memory bytes [128:159] // 5-bit pattern in data memory byte [160] bits 7:3 // // 1) store # of occurences of 5-bit pattern: // a) within string, byte boundaries ON in data memory byte [192] // b) within string, byte boundaries OFF in data memory byte [194] // 2) store # of bytes it occurs in in data memory [192]

```
// data_mem map:
// byte         value
// 128          s[0] where s is the stored input string
// .            s[x] where s is the stored input string
// 159          s[31] where s is the stored input string
// 160          5-bit pattern to recognize in bits 7:3, XXX (don't care) in bits
2:0
// .            X (don't care)
// 192          # of times 5-bit pattern recognized within byte boundaries
// 193          # of bytes in which the 5 bit pattern is recognized at least once
// 194          # of times the 5-bit pattern recognized in string without
regarding byte boundaries
// 195          index of the current byte of memory we are referencing while
scanning the input string
```

```
// 196          flag indicating the patter was recognized within the current
byte...set to 1 if yes, 0 if not

/***************
Initializations
***************/
// # of times 5-bit pattern recognized within byte boundaries initially 0
ldi R2, 0
ldi R1, 192   // max immediate is 7, will do shifts and adds to get 192
str [R1], R2 // total number of times the 5-bit pattern occurs (byte boundaries
ON) is initially 0...192 will be loaded into a register with shifts and adds in
true implementation

// # of bytes in which the 5 bit pattern is recognized at least once initially 0
ldi R1, 193   // max immediate is 7, will do shifts and adds to get 193
str [R1], R2 // total number of bytes within which the 5-bit pattern occurs is
initially 0

// # of times the 5-bit pattern recognized in string without regarding byte
boundaries initially 0
ldi R1, 194   // max immediate is 7, will do shifts and adds to get 192
str [R1], R2 // total number of times the 5-bit pattern occurs (byte boundaries
OFF) is initially 0

// index of the current byte of memory we are referencing while scanning the input
string initially 128
ldi R2, 128   // max immediate is 7, will do shifts and adds to get 128...this is
the index of the first memory byte
ldi R1, 195   // max immediate is 7, will do shifts and adds to get 195...this is
the address we will store the index
str [R1], R2  // keeping track of current index by storing it in memory (because
we will need all three registers sometimes)

/***************
Procedure
***************/
L1:
    // flag indicating the pattern was recognized within the current byte
initially 0 (false)
    ldi R2, 0
    ldi R1, 196
    str [R1], R2

    // put 5-bit pattern to recognize in R0
    ldi R2, [160]
    ldr R0, [R2]  // get the byte with pattern stored in bits 7:3
    rs R0, 3      // shift the top 5 bits to the lower 5, padding msb's with 0's
to get 0007:3

    // data_mem[idx] bits 7:3
    ldi R2, [195] // R2 = index of current byte in string
```

```
    ldr R1, [R2]  // R1 = data_mem[index of current byte in string]
    rs R1, 3      // right shift R1 by 3, padding with 0s...we have bits 7:3
    eq R1, R0     // R1 = (R1 == R0); [1 if true, 0 if false]
    ldr R2, [194] // total number of times 5-bit pattern occurs with byte
boundaries OFF
    add R2, R1    // +1 if the pattern was recognized, +0 if not
    str [194], R2 // max immediate is 7, will do shifts and adds to get
194...store updated total

    // check whether R1 is 1, else go to NEXT_STEP_2
    ldi R2, [address of NEXT_STEP_1 instruction]
    ldi R0, 1
    beq           // if (R0==R1) PC = [R2] (jump to that instruction address),
else PC = PC + 1 (default PC increment)
    // if here, we didn't jump, therefore we know R1 == 1 (i.e. pattern found in
current byte)
    // first, increment number of times pattern found with byte boundaries ON
    ldr R2, [192]
    addi R2, 1
    str [192], R2

    // second, increment number of bytes containing pattern if necessary
    ldr R2, [196] // get flag indicating pattern was found in current byte
    neq R1, R2    // since we know R1 = 1, this will be 1 if R2 is 0, and 0
otherwise
    ldr R2, [193] // get number of bytes in which pattern was recognized
    add R2, R1    // recall that R1 is now 1 if the pattern was found and the
"pattern found in current byte" byte was false, 0 otherwise, so we increment in
the proper instance
    str [196], 1  // we found the pattern, so set the "patter found in current
byte" byte to 1

NEXT_STEP_1:
    // put 5-bit pattern to recognize in R0
    ldi R2, [160]
    ldr R0, [R2]  // get the byte with pattern stored in bits 7:3
    rs R0, 3      // shift the top 5 bits to the lower 5, padding msb's with 0's
to get 0007:3

    // data_mem[idx] bits 6:2
    ldi R2, [195] // max immediate is 7, will do shifts and adds to get 195...R2 =
current index
    ldr R1, [R2]  // R1 = data_mem[current_index]
    ls R1, 1      // left shift R1 by 1, fill lsb's with 0s...we have [6:0]0
    rs R1, 3      // right shift R1 by 2, padding msb's with 0s...we have 000[6:2]
    eq R1, R0     // R1 = (R1 == R0); [1 if true, 0 if false]
    ldr R2, [194] // total number of times 5-bit pattern occurs with byte
boundaries OFF
    add R2, R1    // +1 if the pattern was recognized, +0 if not
    str [194], R2 // max immediate is 7, will do shifts and adds to get
194...store updated total
```

```
    // check whether R1 is 1, else go to NEXT_STEP_2
    ldi R2, [address of NEXT_STEP_2 instruction]
    ldi R0, 1
    beq           // if (R0==R1) PC = [R2] (jump to that instruction address),
else PC = PC + 1 (default PC increment)
    // if here, we didn't jump, therefore we know R1 == 1 (i.e. pattern found in
current byte)
    // first, increment number of times pattern found with byte boundaries ON
    ldr R2, [192]
    addi R2, 1
    str [192], R2

    // second, increment number of bytes containing pattern if necessary
    ldr R2, [196] // get flag indicating pattern was found in current byte
    neq R1, R2    // since we know R1 = 1, this will be 1 if R2 is 0, and 0
otherwise
    ldr R2, [193] // get number of bytes in which pattern was recognized
    add R2, R1    // recall that R1 is now 1 if the pattern was found and the
"pattern found in current byte" byte was false, 0 otherwise, so we increment in
the proper instance
    str [196], 1  // we found the pattern, so set the "patter found in current
byte" byte to 1

NEXT_STEP_2:
    // put 5-bit pattern to recognize in R0
    ldi R2, [160]
    ldr R0, [R2]  // get the byte with pattern stored in bits 7:3
    rs R0, 3      // shift the top 5 bits to the lower 5, padding msb's with 0's
to get 0007:3

    // data_mem[idx] bits 5:1
    ldi R2, [195] // max immediate is 7, will do shifts and adds to get 195...R2 =
current index
    ldr R1, [R2]  // R1 = data_mem[current_index]
    ls R1, 2      // left shift R1 by 2, fill lsb's with 0s...we have [5:0]00
    rs R1, 3      // right shift R1 by 3, padding with 0s...we have 000[5:1]
    eq R1, R0     // R1 = (R1 == R0); [1 if true, 0 if false]
    ldr R2, [194] // total number of times 5-bit pattern occurs with byte
boundaries OFF
    add R2, R1    // +1 if the pattern was recognized, +0 if not
    str [194], R2 // max immediate is 7, will do shifts and adds to get
194...store updated total

    // check whether R1 is 1, else go to NEXT_STEP_3
    ldi R2, [address of NEXT_STEP_3 instruction]
    ldi R0, 1
    beq           // if (R0==R1) PC = [R2] (jump to that instruction address),
else PC = PC + 1 (default PC increment)
    // if here, we didn't jump, therefore we know R1 == 1 (i.e. pattern found in
current byte)
```

```
    // first, increment number of times pattern found with byte boundaries ON
    ldr R2, [192]
    addi R2, 1
    str [192], R2

    // second, increment number of bytes containing pattern if necessary
    ldr R2, [196] // get flag indicating pattern was found in current byte
    neq R1, R2    // since we know R1 = 1, this will be 1 if R2 is 0, and 0
otherwise
    ldr R2, [193] // get number of bytes in which pattern was recognized
    add R2, R1    // recall that R1 is now 1 if the pattern was found and the
"pattern found in current byte" byte was false, 0 otherwise, so we increment in
the proper instance
    str [196], 1  // we found the pattern, so set the "patter found in current
byte" byte to 1

NEXT_STEP_3:
    // put 5-bit pattern to recognize in R0
    ldi R2, [160]
    ldr R0, [R2]  // get the byte with pattern stored in bits 7:3
    rs R0, 3      // shift the top 5 bits to the lower 5, padding msb's with 0's
to get 0007:3

    // data_mem[idx] bits 4:0
    ldi R2, [195] // max immediate is 7, will do shifts and adds to get 195...R2 =
current index
    ldr R1, [R2]  // R1 = data_mem[current_index]
    ls R1, 3      // left shift R1 by 3, fill lsb's with 0s...we have [4:0]000
    rs R1, 4      // right shift R1 by 3, padding with 0s...we have 0000[4:0]
    eq R1, R0     // R1 = (R1 == R0); [1 if true, 0 if false]
    ldr R2, [194] // total number of times 5-bit pattern occurs with byte
boundaries OFF
    add R2, R1    // +1 if the pattern was recognized, +0 if not
    str [194], R2 // max immediate is 7, will do shifts and adds to get
194...store updated total

    // check whether R1 is 1, else go to NEXT_STEP_4
    ldi R2, [address of NEXT_STEP_4 instruction]
    ldi R0, 1
    beq           // if (R0==R1) PC = [R2] (jump to that instruction address),
else PC = PC + 1 (default PC increment)
    // if here, we didn't jump, therefore we know R1 == 1 (i.e. pattern found in
current byte)
    // first, increment number of times pattern found with byte boundaries ON
    ldr R2, [192]
    addi R2, 1
    str [192], R2

    // second, increment number of bytes containing pattern if necessary
    ldr R2, [196] // get flag indicating pattern was found in current byte
    neq R1, R2    // since we know R1 = 1, this will be 1 if R2 is 0, and 0
```

```
otherwise
    ldr R2, [193] // get number of bytes in which pattern was recognized
    add R2, R1    // recall that R1 is now 1 if the pattern was found and the
"pattern found in current byte" byte was false, 0 otherwise, so we increment in
the proper instance
    str [196], 1  // we found the pattern, so set the "patter found in current
byte" byte to 1

NEXT_STEP_4:
    // put 5-bit pattern to recognize in R0
    ldi R2, [160]
    ldr R0, [R2]  // get the byte with pattern stored in bits 7:3
    rs R0, 3      // shift the top 5 bits to the lower 5, padding msb's with 0's
to get 0007:3

    // data_mem[idx] bits 3:0 , data_mem[idx+1] bit 7
    ldr R2, [195] // R2 is the index of the current byte in the string
    ldr R1, [R2]  // R1 is the current byte in the string
    ls R1, 4      // R1 is [3:0]0000
    rs R1, 4      // R1 is 0000[3:0]
    rs R0, 1      // R0 is 0000[first four bits of the pattern]
    eq R1, R0     // R1 is 1 if the last four bits of the current string byte are
the first four bits of the pattern, 0 otherwise
    ldi R0, 0
    ldi R2, [address of NEXT_STEP_5]
    beq // go to NEXT_STEP_5 if the pattern is not already a partial match,
otherwise PC=PC+1 (default increment)

    // put 5-bit pattern to recognize in R0
    ldi R2, [160]
    ldr R0, [R2]  // get the byte with pattern stored in bits 7:3
    rs R0, 3      // shift the top 5 bits to the lower 5, padding msb's with 0's
to get 0007:3

    // data_mem[idx+1] bit 7
    ldr R2, [195] // R2 is the index of the current byte in the string
    addi R2, 1    // R2 is the index of the next byte in the string
    ldr R1, [R2]  // R1 is the next byte in the string
    rs R1, 7      // R1 is 00000007 (bit 7 of the next byte in the string)
    ls R0, 7      // R0 is P0000000 (P is last bit in pattern)
    rs R0, 7      // R0 is 0000000P (P is last bit in pattern)
    eq R1, R0     // true if bit 7 in next string byte is last bit in the pattern
    ldi R0, 0
    ldi R2, [address of NEXT_STEP_5]
    beq           // go to NEXT_STEP_5 if the remainder of the pattern is not a
match, otherwise PC=PC+1 (default increment)

    // pattern was found crossing over bytes, increment the # of times pattern was
found without regard to byte boundaries
    ldi R1, [194]
    addi R1, 1
```

```
    str [194], R1

NEXT_STEP_5:
    // put 5-bit pattern to recognize in R0
    ldi R2, [160]
    ldr R0, [R2]  // get the byte with pattern stored in bits 7:3
    rs R0, 3      // shift the top 5 bits to the lower 5, padding msb's with 0's
to get 0007:3

    // data_mem[idx] bits 2:0 , data_mem[idx+1] bits 7:6
    ldr R2, [195] // R2 is the index of the current byte in the string
    ldr R1, [R2]  // R1 is the current byte in the string
    ls R1, 5      // R1 is [2:0]0000
    rs R1, 5      // R1 is 00000[2:0]
    rs R0, 2      // R0 is 00000[first 3 bits of the pattern]
    eq R1, R0     // R1 is 1 if the last four bits of the current string byte are
the first four bits of the pattern, 0 otherwise
    ldi R0, 0
    ldi R2, [address of NEXT_STEP_6]
    beq // go to NEXT_STEP_5 if the pattern is not already a partial match,
otherwise PC=PC+1 (default increment)

    // put 5-bit pattern to recognize in R0
    ldi R2, [160]
    ldr R0, [R2]  // get the byte with pattern stored in bits 7:3
    rs R0, 3      // shift the top 5 bits to the lower 5, padding msb's with 0's
to get 0007:3

    // data_mem[idx+1] bits 7:6
    ldr R2, [195] // R2 is the index of the current byte in the string
    addi R2, 1    // R2 is the index of the next byte in the string
    ldr R1, [R2]  // R1 is the next byte in the string
    rs R1, 6      // R1 is 0000007:6 (bits 7:6 of the next byte in the string)
    ls R0, 6      // R0 is PP000000 (P's are last two bits in pattern)
    rs R0, 6      // R0 is 000000PP (P's are last two bit in pattern)
    eq R1, R0     // true if bits 7:6 in next string byte is last bit in the
pattern
    ldi R0, 0
    ldi R2, [address of NEXT_STEP_6]
    beq           // go to NEXT_STEP_6 if the remainder of the pattern is not a
match, otherwise PC=PC+1 (default increment)

    // pattern was found crossing over bytes, increment the # of times pattern was
found without regard to byte boundaries
    ldi R1, [194]
    addi R1, 1
    str [194], R1

NEXT_STEP_6:
    // put 5-bit pattern to recognize in R0
    ldi R2, [160]
```

```
    ldr R0, [R2]  // get the byte with pattern stored in bits 7:3
    rs R0, 3      // shift the top 5 bits to the lower 5, padding msb's with 0's
to get 0007:3

    // data_mem[idx] bits 1:0 , data_mem[idx+1] bits 7:5
    ldr R2, [195] // R2 is the index of the current byte in the string
    ldr R1, [R2]  // R1 is the current byte in the string
    ls R1, 6      // R1 is [1:0]00000
    rs R1, 6      // R1 is 000000[1:0]
    rs R0, 3      // R0 is 000000[first 2 bits of the pattern]
    eq R1, R0     // R1 is 1 if the last four bits of the current string byte are
the first four bits of the pattern, 0 otherwise
    ldi R0, 0
    ldi R2, [address of NEXT_STEP_7]
    beq // go to NEXT_STEP_7 if the pattern is not already a partial match,
otherwise PC=PC+1 (default increment)

    // put 5-bit pattern to recognize in R0
    ldi R2, [160]
    ldr R0, [R2]  // get the byte with pattern stored in bits 7:3
    rs R0, 3      // shift the top 5 bits to the lower 5, padding msb's with 0's
to get 0007:3

    // data_mem[idx+1] bits 7:5
    ldr R2, [195] // R2 is the index of the current byte in the string
    addi R2, 1    // R2 is the index of the next byte in the string
    ldr R1, [R2]  // R1 is the next byte in the string
    rs R1, 5      // R1 is 000007:5 (bits 7:5 of the next byte in the string)
    ls R0, 5      // R0 is PPP00000 (P's are last three bits in pattern)
    rs R0, 5      // R0 is 00000PPP (P's are last three bit in pattern)
    eq R1, R0     // true if bit 7:5 in next string byte is last bit in the
pattern
    ldi R0, 0
    ldi R2, [address of NEXT_STEP_7]
    beq           // go to NEXT_STEP_7 if the remainder of the pattern is not a
match, otherwise PC=PC+1 (default increment)

    // pattern was found crossing over bytes, increment the # of times pattern was
found without regard to byte boundaries
    ldi R1, [194]
    addi R1, 1
    str [194], R1

NEXT_STEP_7:
    // put 5-bit pattern to recognize in R0
    ldi R2, [160]
    ldr R0, [R2]  // get the byte with pattern stored in bits 7:3
    rs R0, 3      // shift the top 5 bits to the lower 5, padding msb's with 0's
to get 0007:3

    // data_mem[idx] bit 0 , data_mem[idx+1] bits 7:4
```

```
    ldr R2, [195] // R2 is the index of the current byte in the string
    ldr R1, [R2]  // R1 is the current byte in the string
    ls R1, 7      // R1 is [bit 0 of current byte of string]0000000
    rs R1, 7      // R1 is 0000000[bit 0 of current byte of string0]
    rs R0, 4      // R0 is 0000000[first bit of the pattern]
    eq R1, R0     // R1 is 1 if the last four bits of the current string byte are
the first four bits of the pattern, 0 otherwise
    ldi R0, 0
    ldi R2, [address of NEXT_STEP_8]
    beq // go to NEXT_STEP_8 if the pattern is not already a partial match,
otherwise PC=PC+1 (default increment)

    // put 5-bit pattern to recognize in R0
    ldi R2, [160]
    ldr R0, [R2]  // get the byte with pattern stored in bits 7:3
    rs R0, 3      // shift the top 5 bits to the lower 5, padding msb's with 0's
to get 0007:3

    // data_mem[idx+1] bits 7:4
    ldr R2, [195] // R2 is the index of the current byte in the string
    addi R2, 1    // R2 is the index of the next byte in the string
    ldr R1, [R2]  // R1 is the next byte in the string
    rs R1, 4      // R1 is 00007:4 (bits 7:4 of the next byte in the string)
    ls R0, 4      // R0 is PPPP00000 (P's are last 4 bits in pattern)
    rs R0, 4      // R0 is 00000PPP (P's are last 4 bits in pattern)
    eq R1, R0     // true if bits 7:4 in next string byte is last bit in the
pattern
    ldi R0, 0
    ldi R2, [address of NEXT_STEP_8]
    beq           // go to NEXT_STEP_8 if the remainder of the pattern is not a
match, otherwise PC=PC+1 (default increment)

    // pattern was found crossing over bytes, increment the # of times pattern was
found without regard to byte boundaries
    ldi R1, [194]
    addi R1, 1
    str [194], R1

NEXT_STEP_8
    // increment index of current byte in string
    ldi R1, [195]
    addi R1, 1
    str [195], R1

    // branch back to L1 if idx < 160
    ldi R0, 160     // 1 byte past the string (string is on bytes 128:159)
    ldi R1, [195]   // get the current index
    neq R1, R0      // true if current index has not yet reached 160
    ldi R2, [address of L1] // preparing to conditionally branch here to repeat
the operation
    ldi R0, 1
```

```
    beq // go to address of L1 if R0 != R1 (i.e. if R1, the current index, is not
equal to 160)...otherwise, continue, i.e. program is finished

    end
```

MILESTONE 2

Date Created: 2/4/2022

Date Last Updated: 2/9/2022

Group Members: Daisuke Chon, Angelica Consengco, Matthew Larkins

PIDs: A15388691, A14113566, A16052530

Component 1: Changelog

1. Specified that addresses are absolute rather than relative under Component 3, Part iv. of Milestone 1: Machine Specification - Control Flow
2. Added assembly language instruction to machine code translation example under Component 4 of Milestone 1: Programmer's Model

---

Component 2: ALU Operations

Our ALU should be able to support the following operations:

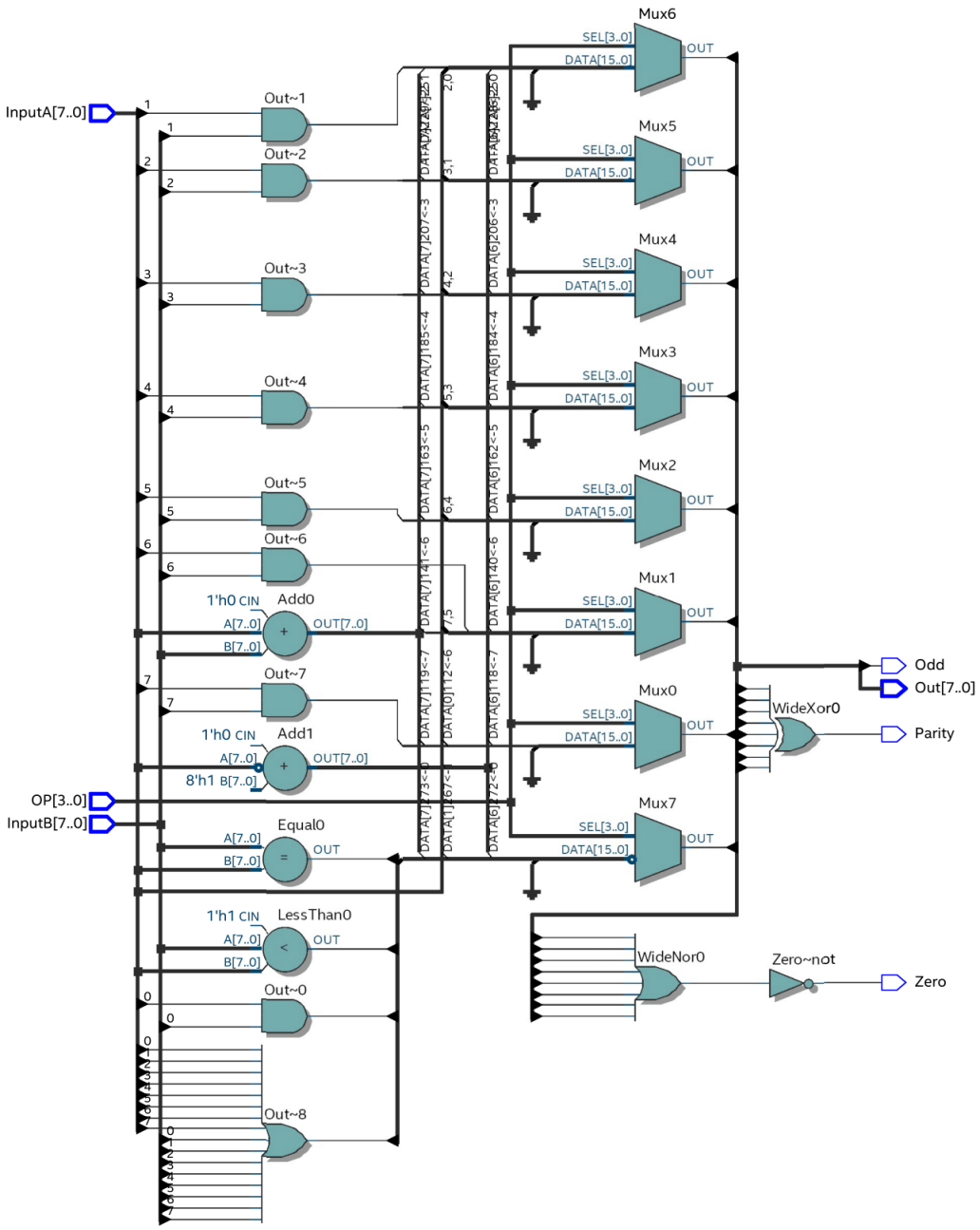
- logical left shift
- logical right shift
- bitwise and
- bitwise or
- Greater than or equal comparison
- Equals comparison
- 2's Comp Negation
- Addition
- Not Equals comparison

Our register file can read and write into registers. The basic implementation suits our design so we left it as is.

Component 3: Verilog Models

ALU Diagram

Date: February 08, 2022 Project: cse141L_ALU



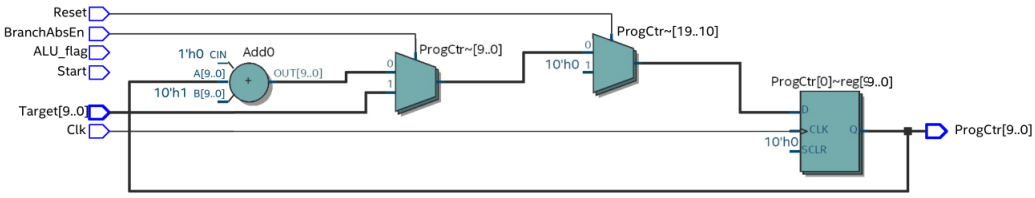
ALU Code

```
// Create Date:    2018.10.15
// Module Name:    ALU
// Project Name:   CSE141L
```

```
//
// Revision 2021.07.27
// Additional Comments:
//   combinational (unclocked) ALU
import definitions::*;                    // includes package "definitions"
module ALU #(parameter W=8, Ops=4)(
  input        [W-1:0]   InputA,      // data inputs
                         InputB,
  input        [Ops-1:0] OP,              // ALU opcode, part of microcode
  output logic [W-1:0]   Out,             // data output
  output logic           Zero,        // output = zero flag  !(Out)
                         Parity,      // outparity flag  ^(Out)
                         Odd                // output odd flag (Out[0])
// you may provide additional status flags, if desired
  );

  op_mne op_mnemonic;                     // type enum: used for convenient
waveform viewing

  always_comb begin
    Out = 0;                              // No Op = default
    case (OP)
      ADD : Out = InputA + InputB;        // add
      LSH : Out = {InputA[6:0], 1'b0};    // shift left, fill in with zeroes
        RSH : Out = {1'b0, InputA[7:1]};    // shift right
      AND : Out = InputA & InputB;        // bitwise AND
      OR  : Out = InputA || InputB;       // bitwise OR
      NEG : Out = ~InputA + 1;
      GEQ : Out = (InputA >= InputB);         // Greater than or Equal to
      EQ  : Out = (InputA == InputB);        // Equals to
      NEQ : Out = (InputA != InputB);         // Not Equals to
    endcase
  end

  assign Zero   = !Out;                   // reduction NOR
  assign Parity = ^Out;                   // reduction XOR
  assign Odd    = Out[0];                 // odd/even -- just the value of the LSB

  always_comb
    op_mnemonic = op_mne'(OP);            // displays operation name in waveform
viewer

endmodule
```

Program Counter Diagram:

Date: February 09, 2022 Project: cse141L_ProgCtr



Page 1 of 1 Revision: ProgCtr

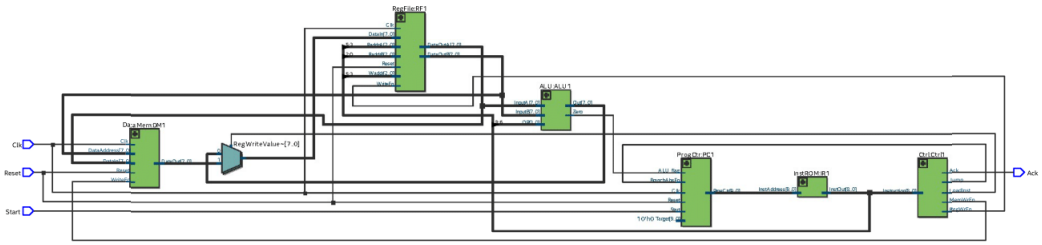
Program Counter Code:

```
// Design Name:    basic_proc
// Module Name:    InstFetch
// Project Name:   CSE141L
```

```
// Description:    instruction fetch (pgm ctr) for processor
//
// Revision:  2019.01.27
//
module ProgCtr #(parameter L=10) (
  input                Reset,      // reset, init, etc. -- force PC to 0
                       Start,      // Signal to jump to next program; currently
unused
                       Clk,        // PC can change on pos. edges only
                       BranchAbsEn,  // jump to Target
                       ALU_flag,   // Sometimes you may require signals from other
modules, can pass around flags if needed
  input        [L-1:0] Target,     // jump ... "how high?"
  output logic [L-1:0] ProgCtr     // the program counter register itself
  );


  // program counter can clear to 0, increment, or jump
  always_ff @(posedge Clk)         // or just always; always_ff is a linting
construct
    if(Reset)
      ProgCtr <= 0;
    else if(BranchAbsEn)                  // unconditional absolute jump
      ProgCtr <= Target;               //   how would you make it conditional
and/or relative?
    else
      ProgCtr <= ProgCtr+'b1;          // default increment (no need for ARM/MIPS
+4 -- why?)
endmodule
```

Top Level Diagram

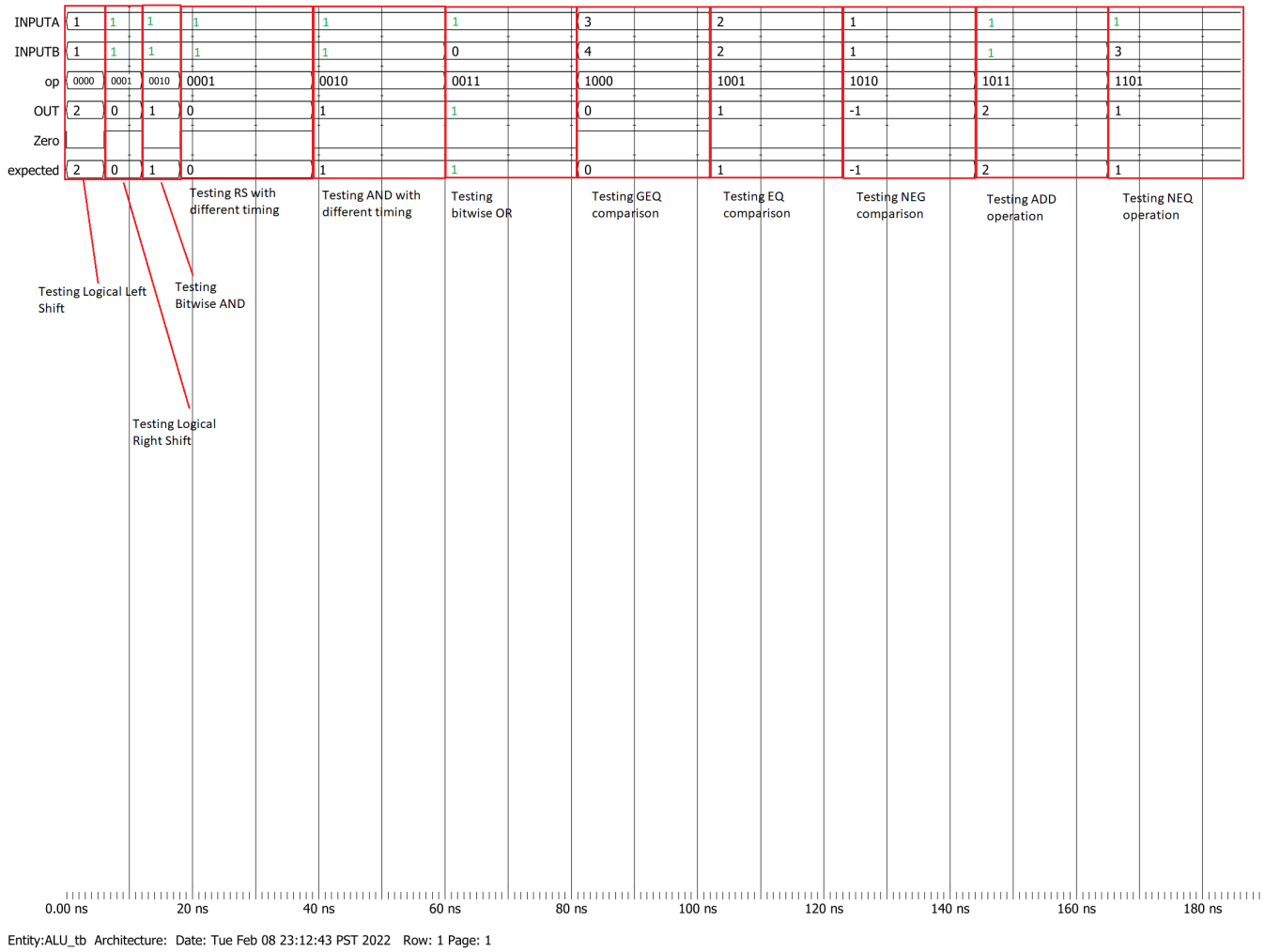
Date: February 09, 2022 Project: TopLevel



Page 1 of 1 Revision: TopLevel

Component 4: Timing Diagrams

ALU Timing Diagram

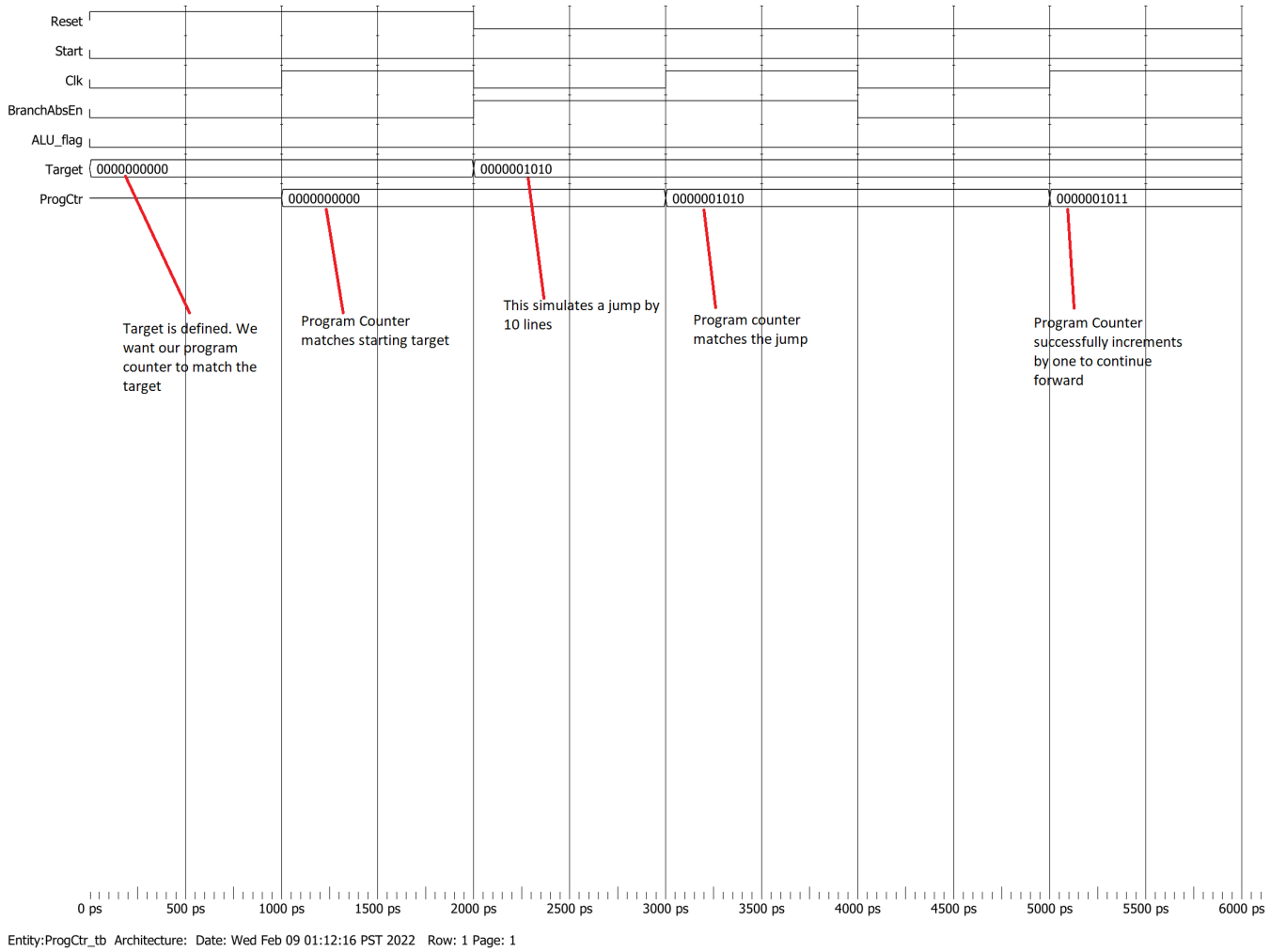


ALU Transcript

```
# Start time: 22:36:41 on Feb 08,2022
# Loading sv_std.std
# Loading work.definitions
# Loading work.ALU_tb_sv_unit
# Loading work.ALU_tb
# Loading work.ALU_sv_unit
# Loading work.ALU
# ** Warning: (vsim-3017) C:/Users/johna/Desktop/Angelica-CSE141L/CSE-141L-Project/WI22-csel41l-testbench-fixes/basic_proc/ALU_tb.sv(26): [TFMPC] - Too few port connections. Expected 7, found 5.
#    Time: 0 ps  Iteration: 0  Instance: /ALU_tb/uut File: C:/Users/johna/Desktop/Angelica-CSE141L/CSE-141L-Project/WI22-csel41l-testbench-fixes/basic_proc/ALU.sv
# ** Warning: (vsim-3722) C:/Users/johna/Desktop/Angelica-CSE141L/CSE-141L-Project/WI22-csel41l-testbench-fixes/basic_proc/ALU_tb.sv(26): [TFMPC] - Missing connection for port 'Parity'.
# ** Warning: (vsim-3722) C:/Users/johna/Desktop/Angelica-CSE141L/CSE-141L-Project/WI22-csel41l-testbench-fixes/basic_proc/ALU_tb.sv(26): [TFMPC] - Missing connection for port 'Odd'.
VSIM 4> run -all
#                 1000 YAY!! inputs = 01 01, opcode = 0000, Zero 0   Testing LSL
#                 7000 YAY!! inputs = 01 01, opcode = 0001, Zero 1   LSR
#                13000 YAY!! inputs = 01 01, opcode = 0010, Zero 0   Bitwise AND
#                19000 YAY!! inputs = 01 01, opcode = 0001, Zero 1   LSR with different timing
#                40000 YAY!! inputs = 01 01, opcode = 0010, Zero 0   AND with different timing
#                61000 YAY!! inputs = 01 00, opcode = 0011, Zero 0   Bitwise OR
#                82000 YAY!! inputs = 03 04, opcode = 1000, Zero 1   GEQ Comparison
#               103000 YAY!! inputs = 02 02, opcode = 1001, Zero 0   EQ comparison
#               124000 YAY!! inputs = 01 01, opcode = 1010, Zero 0   NEG comparison
#               145000 YAY!! inputs = 01 01, opcode = 1011, Zero 0   ADD comparison
#               166000 YAY!! inputs = 01 03, opcode = 1101, Zero 0   NEQ comparison
```

All tests are in working order

Program Counter Timing Diagram



Program Counter Transcript

```
add wave -position insertpoint sim:/progctr_tb/dut/*
VSIM 5> run -all
# Check Reset Asserts if reset is successful
# Check if PC jumps to target Asserts if jump is successful
# Check if PC increments by 1 Asserts if increment/step is successful
```

Component 5: Answering the Question

Our ALU will *indirectly* be used for non-arithmetic instructions such as load and store. While we do indeed need to make address pointer calculations, the programmer will be responsible for calculating these addresses via manual shift operations of 3 bit immediates into 8 bit values. As such, the complexity of the design is unaffected.

MILESTONE 3

Date Created: 2/22/2022

Date Last Updated: 2/23/2022

Group Members: Daisuke Chon, Angelica Consengco, Matthew Larkins

PIDs: A15388691, A14113566, A16052530

Component 1: Changelog

1. Changed how jumps are done using the program counter. Information is updated on Milestone 1.
2. Changed design of how data addressing works. Information is updated on Milestone 1.

Component 2: Assembler Input and Output Example

We wrote the assembler in Python and the code is shown below. For debugging purposes, the rest of the instruction memory is filled with "xxxxxxxxx" as implemented in the original sample code and the "unused bits" in the R-Type instruction are replaced with a '0' padding. To test the assembler, we created an input file 'test_assembly.asm' that used all 14 of our ISA's instructions and mixed in all of possible registers that could be used. We also included comments and blank lines to test if the assembler would handle these cases. We created a file with the expected output of the assembler by typing out the encoding for each instruction so that we could compare this against the actual output. The output produced by the assembler was correct. The input, output, and test files mentioned are all shown below.

Code:

```
import re
#!/usr/bin/env python3

# Totally optional, here's a neat thing:
# You can create a reverse-mapping, such that waveform tools will show the
# original instruction.
# The waveform viewing tool needs a mapping of # `$value $display_string`.
# Only want to write each unique machine code once though.
mcodes = set()

# Here's a lookup table for opcodes
ops = {
                'ls': '0000',
                'rs': '0001',
                'and': '0010',
                'or' : '0011',
                'ldi' : '0100',
                'ldr' : '0101',
                'str' : '0110',
                'bnz' : '0111',
                'geq' : '1000',
                'eq' : '1001',
                'neg' : '1010',
                'add' : '1011',
                'addi' : '1100',
                'neq' : '1101'
                }
```

```
# This is a neat trick to catch programming errors
TOTAL_IMEM_SIZE = 2**10

# Don't need to do anything fancy here
with open('test_assembly.asm') as ifile, open('machine_out.hex', 'w') as imem,
open('gtkwave/mcode.fmt', 'w') as wavefmt:
        for lineno, line in enumerate(ifile):
                try:
                        # Skip over blank lines, remove comments
                        line = line.strip()
                        line = line.split('//')[0].strip()
                        if line == '':
                                continue

                        # Special-case this:
                        if line[:4] == 'halt':
                                machine_code = '111111111'
                        # I-Type: Op rs, imm
                        # R-Type: Op rd, rs (plus one zero for padding unused bit)
                        insn = re.split(' |, ', line)
                        op = insn[0]
                        #if instruction is I-Type: Op rs, imm
                        if insn[2].startswith('#'):
                                imm = '{:03b}'.format(int(insn[2].split('#')[1]))
                                rs = '{:02b}'.format(int(insn[1].split('r')[1]))
                                machine_code = ops[op] + rs + imm
                        #if instruction is nop
                        elif (op == 'nop'):
                                machine_code = '111100000'
                        #if instruction is R-type: Op rd, rs (plus one zero for
padding unused bit)
                        else:
                                rd = '{:02b}'.format(int(insn[1].split('r')[1]))
                                rs = '{:02b}'.format(int(insn[2].split('r')[1]))
                                machine_code = ops[op] + rd + rs + '0'
                        # Write the imem entry
                        imem.write(machine_code + '\n')
                        TOTAL_IMEM_SIZE -= 1

                        # Write out our waveform decoder
                        if machine_code not in mcodes:
                                line = line.replace('\t', ' ')
                                wavefmt.write('{} {}\n'.format(machine_code,
line))
                                mcodes.add(machine_code)
                except:
                        print("Error Parsing Line ", lineno)
                        print(">>>{}<<<".format(line))
                        print()
                        raise
```

```
        # This is a neat trick to catch programming errors:
        # Fill the rest of instruction memory with illegal instructions.
        #
        wavefmt.write('xxxxxxxxx ILLEGAL!')
        while TOTAL_IMEM_SIZE:
                imem.write('xxxxxxxxx\n')
                TOTAL_IMEM_SIZE -= 1
```

Input file (test_assembly.asm):

```
//left shift: r0 = r0 << 2
ls r0, #2

//right shift: r0 = r0 << 3
rs r0, #3

//bitwise AND: r0 = r0 & r1
and r0, r1

//bitwise OR: r0 = r0 || r1
or r0, r1

//load immediate: r0 = 7
ldi r0, #7

//load register: r2 = MEM[r0]
ldr r2, r0

//store register: MEM[r0] = r2
str r2, r0

//equals: r1 = (r1 == r2)
eq r1, r2

//branch if not equal to zero: if r1 == 1, go to address in R3
bnz r3, r1

//greater than or equal to: r0 = (r0 >= r1)
geq r0, r1

//2's complement: r1 = (~r1 + 1)
neg r1, r1

//add: r0 = r0 + r1
add r0, r1

//add immediate: r0 = r0 + 3
addi r0, #3
```

```
//not equal to: r0 = (r0 != r1)
neq r0, r1
```

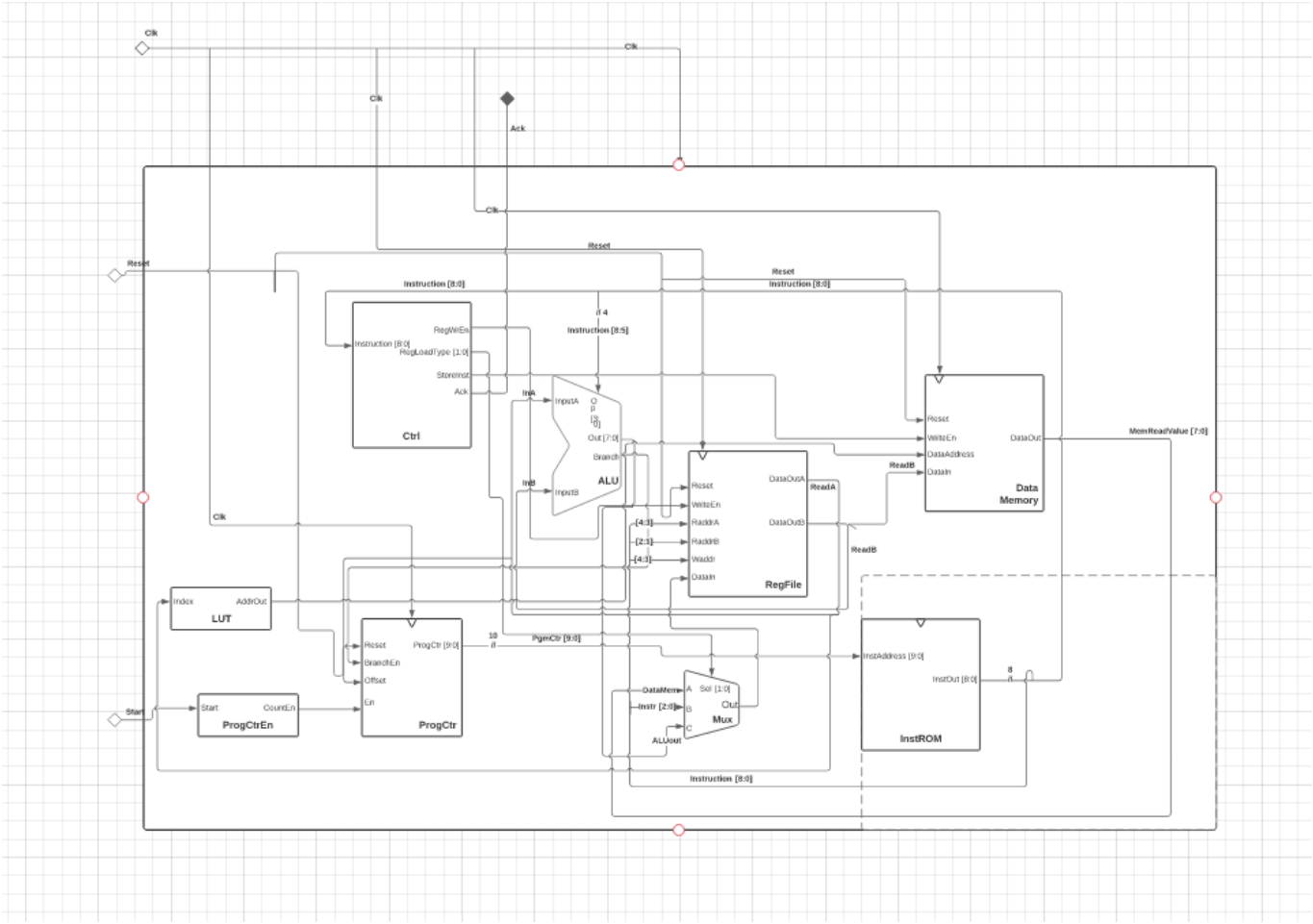
Expected output file (expected_machine_code.txt):

```
000000010 //ls r0, #2
000100011 //rs r0, #3
001000010 //and r0, r1
001100010 //or r0, r1
010000111 //ldi r0, #7
010110000 // ldr r2, r0
011010000 //str r2, r0
100101100 //eq R1, R2
011111010 //bnz R3, R1
100000010 //geq r0, r1
101001010 //neg r1, r1
101100010 //add r0, r1
110000011 //addi r0, #3
110100010 //neq r0, r1
```

Actual output file (machine_out.hex), with the 'xxxxxxxxx' lines truncated for space:

```
000000010
000100011
001000010
001100010
010000111
010110000
011010000
100101100
011111010
100000010
101001010
101100010
110000011
110100010
```

Component 3: Architectural Overview figure


Clk
Ack
Reset
Instruction [8:0]
Instruction [8:5]
Ctrl
RegWrEn
RegLoadType [1:0]
StoreInst
Ack
ALU
InputA
InputB
Out [7:0]
Branch
RegFile
DataOutA
DataOutB
ReadA
ReadB
Data Memory
MemReadValue [7:0]
LUT
Index
AddrOut
ProgCtrEn
Start
CountEn
ProgCtr
BranchEn
Offset
En
ProgCtr [9:0]
PgmCtr [9:0]
Mux
DataMem
Instr [2:0]
ALUout
Sel [1:0]
Out
InstROM
InstAddress [9:0]
InstOut [8:0]