Bonus Sections

Assembler.py

An assembler is a crucial part to any RISC style computer as the human understandable machine instructions must be parsed into a form that the computer can understand. An assembler has been written in the language Python to parse an input stream of instructions into a 32 bitstream to be used in conjunction with the MiniSRC computer. This assembler was used during Phase Three of the project to assemble to test program to load it into the MiniSRC's memory to be run.

The assembler reads the contents of the file "code.asm" as input and assembles it into the file "assembled.asm." There are two modes of operation, which can be changed by setting the "bitstring" variable to True or False. The first mode of operation (True) simply assembles the instructions into the bitstring form, with the fields of the instruction (eg opcode_ra_rb_c) separated by underscores for visibility. The second mode operation (False) assembles the instructions into the bitstring form and appends a prefix "ram[x] <= 32'b", where x is the current memory location. This mode of operation is used to directly assemble the instructions into a program to manually load into the ram512.v file of the MiniSRC.

The assembler itself is a very simple script. It uses lookup tables (Python dictionaries) to match human machine instruction to a bitstring value. This makes the assembler very sustainable as it is very easy for a person with no programming experience to add a new instruction. The fields of the instructions are read and parsed in order of appearance. The bitstring is then assembled, temporarily excluding the constant, additionally making sure that each value is placed in the correct field depending on the instruction. Finally, the constant is sign extended from 8 bits to bring the total size of the instruction to 32 bits. The source code of the assembler is available in Appendix A. The execution of the assembler on the test program from Phase Three is also available in Appendix B.

Inferred RAM

A custom RAM module has been written for the MiniSRC instead of using the Altera Megafunction pre-built Oneport RAM. The file ram512.v, is an asynchronous inferred RAM module. Like the Megafunction RAM, the module accepts an address and read/write signals to define a mode of operation. The RAM has been written to be asynchronous to ensure that the memory operations will be complete and ready for input/output by the next clock cycle. The RAM has been tested fully and there have been no hazards detected from the asynchronous operation.

The memory contents of the RAM are defined by "reg [31:0] ram [0:511];" creating an array called ram of size 512x32. Each memory location of ram is therefore to hold a full 32 bits, allowing the MiniSRC to use increments of one for addressing PC. Finally, the RAM is preloaded with the instructions for the program prior to the execution of the test bench. This is done by setting the value of any memory location with a statement such as "ram[0] <= 32'b00001_0011_0000_0000000000001010111;."

Inc Instruction

A new register increment instruction has been added to the MiniSRC. Register increment is a very useful hardware instruction for optimizing code. Incrementing a value is a very common line of code in all programs (eg i++) so having an instruction to optimize it saves high amounts of machine code during program compilation. Not only does the instruction allow lower code generation, but it also is able to execute in four cycles instead of the five cycles of a typical add instruction. Shown below is the documentation for the new instruction:

Increment (11011xxxxxxxxxxxx---------------)	R[Ra] 🡐 R[Ra] + $1	inc Ra

The increment instruction has been added to the alu.v file as it is an arithmetic instruction and its implementation can be seen in the source code. A testbench of the instruction can be seen below in the Testing section.

Memory Stack

A built-in hardware stack module has also been written for the MiniSRC computer. Hardware stacks are a very useful, fast way to store and retrieve values, especially to preserve register contents upon entrance and exit of a subroutine. The memory stack consists of a 32x32 array of memory, initialized similar to that in the "ram512.v" file and an internal Stack Pointer register. The interface to the module is also identical to that of the RAM, excluding the address as that is handled by the internal stack pointer. The memory stack written in "stack32.v" and can be found in the source code.

When the write signal is asserted, the value will be stored in memory at the location of the stack pointer, which always points to the top empty location of the stack. Once the write is complete, the stack pointer will be incremented to again be pointing at the top free location of the stack. When the read signal is asserted, the stack pointer will first be decremented to be pointing at the top value in the stack. Then the value is read and outputted. This is analogous to popping from the stack, simply for efficiency the old value is not deleted and instead marked as free space to be overwritten.

The stack is accessed through the use of two new instructions "ststk" and "ldstk" for store and load on the stack respectively. Shown below is the documentation for the new instructions:

Load from Stack (11110xxxxxxxxxxxx---------------)	R[Ra] 🡐 S[SP]	ldstk Ra
Store on Stack (11111xxxxxxxxxxxx---------------)	S[SP] 🡐 R[Ra]	ststk Ra

The MDR is used for both the memory stack and the RAM modules so slight modifications needed to be made. Since both modules will be outputting to the same source, blocking needed to be added to both the memory stack and RAM so they do not assert any signal when not in use. The memory stack and RAM were altered to output a bitstring of 32 '0's when not reading a value. This allows the two outputs to be ORed together and then the result of that or be the input to the MDR. This makes the architecture and design of the memory stack much more efficient as it can use the same interface as the RAM and mostly the same control signals.

The memory stack and new instructions are tested in the Testing section below.

Testing

The bonus sections have been tested in tandem with the below test program, which utilizes the assembler to assemble it, the RAM to store the instructions and executes instructions with increment and the memory stack.

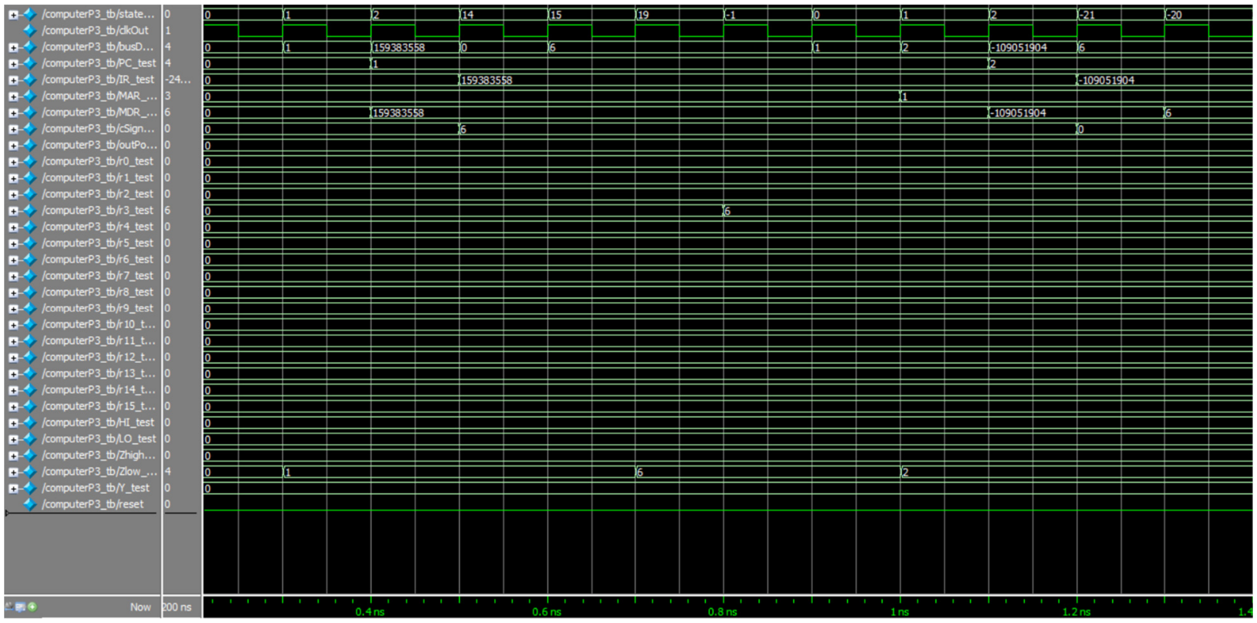
code.asm

```
ldi R3, 6
ststk R3
inc R3
ldstk R3
halt
```

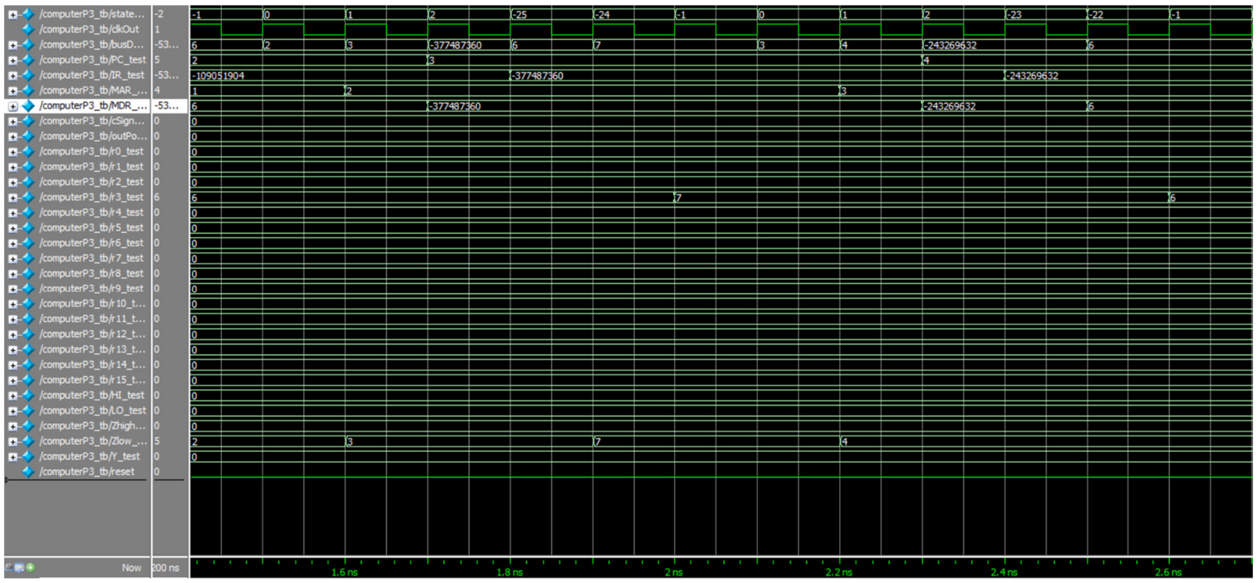
assembled.asm

```
00001_0011_00000000000000000000110
11111_0011_00000000000000000000000
11101_0011_00000000000000000000000
11110_0011_00000000000000000000000
11100_000000000000000000000000000
```

ldi R3, 6 + ststk R3



inc R3 + ldstk R3



Shown above is the execution of the simple program to test the bonus elements. In the first picture, the ldi instruction loads the value 6 into register three at the end of state 19. Then the ststk instruction executes. The value of 6 from register three can be seen in the MDR in state -20, which indicates that it is being sent out to the stack memory. Later the same value will be retrieved from the stack.

In the second picture the inc instruction is used on register three. The value 6 of register three can be seen on the bus going into the ALU in state -25 and is then incremented by one, ending up in Zlow at the start of cycle -24. Then register three's value is set to 7 at the end of state -24. Next is ldstk instruction, to restore register three to its original stored value. The value of the MDR can be seen to be 6 as the value is retrieved from the stack in sate -22, then the value of register three is set back to 6 at the end of state -22. This test indicates that all new instructions and functionality is working as intended.

Appendix A: Assembler.py

```python
import shlex, sys

# Table to reference command name to opcode
opcode_table = {
    "ld" : "00000",
    "ldi" : "00001",
    "st" : "00010",
    "ldr" : "00011",
    "str" : "00100",
    "add" : "00101",
    "sub" : "00110",
    "and" : "00111",
    "or" : "01000",
    "shr" : "01001",
    "shl" : "01010",
    "ror" : "01011",
    "rol" : "01100",
    "addi" : "01101",
    "andi" : "01110",
    "ori" : "01111",
    "mul" : "10000",
    "div" : "10001",
    "neg" : "10010",
    "not" : "10011",
    "brzr" : "10100",
    "brnz" : "10100",
    "brmi" : "10100",
    "brpl" : "10100",
    "jr" : "10101",
    "jal" : "10110",
    "in" : "10111",
    "out" : "11000",
    "mfhi" : "11001",
    "mflo" : "11010",
    "nop" : "11011",
    "halt" : "11100",
    "inc" : "11101",
    "ldstk" : "11110",
    "ststk" : "11111"
}

# IR[1:0] for branch instructions with sign extend bit
branch = {
    "brzr" : "000",
    "brnz" : "001",
    "brpl" : "010",
    "brmi" : "011"
}

# Table to reference register name to ID bitstring
reg_table = {
    "r0" : "0000",
    "r1" : "0001",
    "r2" : "0010",
    "r3" : "0011",
    "r4" : "0100",
    "r5" : "0101",
    "r6" : "0110",
    "r7" : "0111",
    "r8" : "1000",
    "r9" : "1001",
    "r10" : "1010",
    "r11" : "1011",
    "r12" : "1100",
    "r13" : "1101",
    "r14" : "1110",
    "r15" : "1111"
}


def parse_instream(txt):
    """Parse the current instruction by stripping unecessary
    characters and separating C + RX notation into two elements"""
    seq = []  # Container to hold parsed instruction
    for i in range(0, len(txt)):
        s = txt.pop(0).replace(",", "").lower()  # Strip commas and make
lowercase for detection
        if s[-1] == ")":  # Strip constant(reg) notation into two parts
            start = s.find("(")
            end = s.find(")")
            a = s[:start]
            b = s[start + 1:end]
            seq.append(a)
            seq.append(b)
        else:
            seq.append(s)
    return seq


def twos_comp(val, bits=8):
    """Compute the 2's compliment of int value val in 8 bit form"""
    if (val & (1 << (bits - 1))) != 0:  # If sign bit is set
        val -= (1 << bits)  # Compute negative value
    return val


def binary(x):
    """Convert int x into a bitstring"""
    if x[0] == "-":  # Negative number
        num = '{0:08b}'.format(int(x[1:]))  # Convert to binary and strip '-'
sign
        num = bin(int(''.join('1' if x == '0' else '0' for x in num), 2) +
1)[2:]  # Do a 2's compliment
    else:
        num = '{0:08b}'.format(int(x))  # Positive number; Convert to binary
    return num


def c_sign_extend(ir, c):
    """Sign extend constant c and add to instruction ir so far
    giving a 32 bit instruction"""
    size = len(ir) - ir.count("_")  # Total bits used so far
    extend = 32 - size - len(c)  # How many times to extend
    c_sign = extend * c[0] + c
    code = ir + c_sign
    return code


def parse_instruction(txt):
    """Accept an instruction, txt in list form and parse
    it into a 32 bit string IR code"""
    ir = ""  # String to hold the instruction
    c_immediate = "0"  # Default constant of 0 if no replacement is found

    opcode = txt.pop(0)  # Remove opcode from list to parse
    if opcode in opcode_table:
        if opcode == "jal":  # jal has a custom r14 for rb
            element = txt.pop(0)
            ir = opcode_table[opcode] + "_" + reg_table[element] + "_" +
"1110_000000000000000100"
        else:  # General case
            if opcode in branch:  # Special case immediate for branch
                c_immediate = branch[opcode]  # Set branch immediate from
branch type
            ir += opcode_table[opcode] + "_"  # Add opcode to the instruction
            for i in range(0, len(txt)):  # Parse remaining elements
                element = txt.pop(0)
                if element[0] == "r":  # Check for register (rX)
                    ir += reg_table[element] + "_"  # Add register to the
instruction
                elif element[0].isdigit() or element[0] == "-":  # Check for
positive or negative immediate
                    c_immediate = binary(element)
            if opcode == "st" and len(ir) > 11:  # Store has rb first in
instruction so swap when using both reg
                rb = ir[6:10]
                ra = ir[11:15]
                ir = ir[:6] + ra + ir[10] + rb + ir[15:]
            ir = c_sign_extend(ir, c_immediate)  # Add immediate when all
registers are parsed
    else:
        sys.exit("ERROR: Unknown instruction found; Aborting")
    return ir


## MAINLINE ##

bitstring = True  # True for bitstring output; False for verilog output

print("Parsing Begin\n...")
instructions = []
with open("code.asm", "r") as f:
    for line in f:
        txt = parse_instream(shlex.split(str.strip(line)))
        instructions.append(parse_instruction(txt))
print("Parsing Complete\n...")

with open("assembled.asm", "w") as f:
    i = 0
    for code in instructions:
        if bitstring:
            output = code + "\n"
        else:
            output = "ram[" + str(i) + "] <= 32'b" + code + ";\n"
            i += 1
        f.write(output)
print("Output Available in 'assembled.asm'")
```

Appendix B: Assembler Results

Input File: code.asm

```
ldi R3, 87
ldi R3, 1(R3)
ld R2, 66
ldi R2, -1(R2)
ldr R7, 61
str 60, R7
ld R1, 0(R2)
ldi R0, 1
nop
add R3, R2, R3
addi R7, R7, 2
neg R7, R7
not R7, R7
andi R7, R7, 15
ori R7, R1, 3
shr R2, R3, R0
st 56, R2
ror R1, R1, R0
rol R2, R2, R0
or R2, R3, R0
and R1, R2, R1
st 76(R1), R3
sub R3, R2, R3
shl R1, R2, R0
ldi R4, 5
ldi R5, 31
mul R5, R4
mfhi R7
mflo R6
div R5, R4
ldi R10, 0(R4)
ldi R11, 0(R5)
ldi R12, 0(R6)
ldi R13, 0(R7)
jal R12
ststk R4
ststk R8
ldstk R8
ldstk R4
halt
```

Output File: assembled.asm

```
00001_0011_00000000000000001010111
00001_0011_0011_0000000000000000001
```

```
00000_0010_00000000000000001000010
00001_0010_0010_1111111111111111111
00011_0111_00000000000000000111101
00100_0111_00000000000000000111100
00000_0001_0010_0000000000000000000
00001_0000_00000000000000000000001
11011_000000000000000000000000000
00101_0011_0010_0011_000000000000000
01101_0111_0111_0000000000000000010
10010_0111_0111_0000000000000000000
10011_0111_0111_0000000000000000000
01110_0111_0111_0000000000000001111
01111_0111_0001_0000000000000000011
01001_0010_0011_0000_000000000000000
00010_0010_00000000000000000111000
01011_0001_0001_0000_000000000000000
01100_0010_0010_0000_000000000000000
01000_0010_0011_0000_000000000000000
00111_0001_0010_0001_000000000000000
00010_0011_0001_0000000000001001100
00110_0011_0010_0011_000000000000000
01010_0001_0010_0000_000000000000000
00001_0100_00000000000000000000101
00001_0101_00000000000000000011111
10000_0101_0100_0000000000000000000
11001_0111_00000000000000000000000
11010_0110_00000000000000000000000
10001_0101_0100_0000000000000000000
00001_1010_0100_0000000000000000000
00001_1011_0101_0000000000000000000
00001_1100_0110_0000000000000000000
00001_1101_0111_0000000000000000000
10110_1100_1110_0000000000000000100
11111_0100_00000000000000000000000
11111_1000_00000000000000000000000
11110_1000_00000000000000000000000
11110_0100_00000000000000000000000
11100_000000000000000000000000000
```