Introduction to Program

This is a simple two-pass assembler in C++ for a particular FISC (Four or Five Instruction Set Computer) CPU. A two-pass assembler process the input file twice. On the first pass, it determines the address of the labels, and on the second pass it writes out the machine code.

The FISCAS simulator takes in a simplified text-based object file and simulates the execution of a CPU. The output object file will contain a single hexadecimal number that matches the bit width of the memory cell for each line. Since it will be writing out bytes for this assembler, each line will contain two hex symbols.

This is a FISC Assembler that takes in the assembly file, parses each line from the file, and converts it into machine code then to a hexadecimal number. The .hex file is the output file that contains all the hexadecimal numbers. The .s file must already exist for the assembler to work.

Instructions

Op	Instruction	Format	Action
0	ADD	ADD Rd Rn Rm	Rd ← Rn + Rm
1	AND	AND Rd Rn Rm	Rd ← Rn * Rm
2	NOT	NOT Rd Rn	Rd ← Rn (not)
3	BNZ	BNZ target	PC ← target

Instruction Word Formats

Three Operand Instructions

Op	Rn	Rm	Rd
2	2	2	2

The three operand instructions require the addresses of three registers to be specified in the instruction word. We require two bits to specific the instruction since there are four instructions.

Two Operand Instructions

Op	Rn	-	Rd
2	2	2	2

The two operand instructions require the addresses of two registers to be specified in the instruction word. For this CPU, there is only one such instruction.

One Operand Instructions

Op	Target Address
2	6

There is only a single one operand instruction for this CPU, and it is the branch instruction BNZ.

How to Run in Terminal

Compile (Linux terminal, Ubuntu): g++ [cpp file] -o [executable file]
    EX) g++ fiscas.cpp -o fiscas
Command Line Arguments: ./fiscas <source file> <object file> [-l]
    EX) ./fiscas t1.s t1.hex

**Assembly files are used for testing, can be modified

Output

– FISCAS –
This shows in your terminal when you include -l

*** LABEL LIST ***
start 00
end 06
*** MACHINE PROGRAM ***
01:90 not r0 r1
02:50 and r0 r1 r0
03:81 not r1 r0
04:15 add r1 r1 r1
05:C6 bnz end
06:91 not r1 r1
07:57 and r3 r1 r1
08:C6 bnz end

The hex file will show:
v2.0 raw // for logisim
90
50
81
15
C6
91

57
C6

–FISCSIM–

jtrans@DESKTOP-5DVSH1I:/mnt/c/Users/trans/Documents$ ./fiscsim fibo1.hex 56 -d
Cycle:1 State:PC:01 Z:0 R0: FF R1: 00 R2: 00 R3: 00
Disassembly: not r0 r1

Cycle:2 State:PC:02 Z:1 R0: 00 R1: 00 R2: 00 R3: 00
Disassembly: and r0 r0 r1

Cycle:3 State:PC:03 Z:1 R0: 00 R1: 00 R2: 00 R3: 00
Disassembly: and r1 r0 r0

Cycle:4 State:PC:04 Z:1 R0: 00 R1: 00 R2: 00 R3: 00
Disassembly: and r3 r0 r0

Cycle:5 State:PC:05 Z:0 R0: 00 R1: 00 R2: FF R3: 00
Disassembly: not r2 r0

Cycle:6 State:PC:06 Z:0 R0: 00 R1: 00 R2: FE R3: 00
Disassembly: add r2 r2 r2

Cycle:7 State:PC:07 Z:0 R0: 00 R1: 00 R2: 01 R3: 00
Disassembly: not r2 r2

……. (more till it reaches 56 cycles)

How The FISC Assembler Works

The FISCAS assembler checks the number of arguments on the command line before proceeding to open the file and obtain its contents. The main function calls openFile and one of the variables is referenced and updated in the openFile function. In the openFile function, it checks if the number of arguments the user inputted in the command line is enough. If not, it sends outputs an error message in the terminal and ends the program. It also checks if the file exists and will output an error message if it does not exist in the directory. The next part of the function parses through the .s file and stores each line into a vector of strings which will be important in the next part of the code. It removes all the whitespace and comments, allowing the next parsing step to run smoothly.

The main function calls createTable, which executes pass 1 and pass 2 in order to obtain the hex numbers needed for the output. The first for loop in the createTable() checks if the first word is a label or instruction. In the for loop, it removes unnecessary punctuation and stores the label name in a container to keep track of the information being used. This allows there to be a way to check for duplicate labels. In the next for loop, stringstream is used to parse through each line in each element of the vector. The while loop grabs each word and confirms if it is an instruction by finding if the word exists in the map of instructions. If it does, it compares the word to each instruction in order to determine how to approach the problem and get the correct byte string for each line in the .s file. When the word is not “not”, “and”, or “and”, this means that the instruction would be bnz. In this part of the else statement, it checks if the target or label name exists in the container of label names. If target was not a label in the stl container, then the target value can not be equal to six but to zero. For each line, this process continues and updates the hex numbers using the getHexNums function until it reaches the end of the vector of lines. getHexNums() essentially converts the byte string by changing it to a bitset of 8 bits and changes it to a hexadecimal number. The hexadecimal number is added to the vector of strings which will be used to determine the output in the terminal and .hex file.

The assembler was tested using the ubuntu terminal. The .s files were modified to determine whether or not the program was functioning correctly such as displaying the correct error messages and knowing how to deal with certain test cases.

How The FISC Simulator Works

The FISC simulator runs after the user inputs the desired command line arguments to figure out which hex file to open, the total number of cycles, and whether or not to include the disassembly output. The openFile function determines if the user inputted the correct number of arguments in the command line, allowing the program to proceed to the next step which is getting each hex number in the form of a string and append it to a vector to store the information.

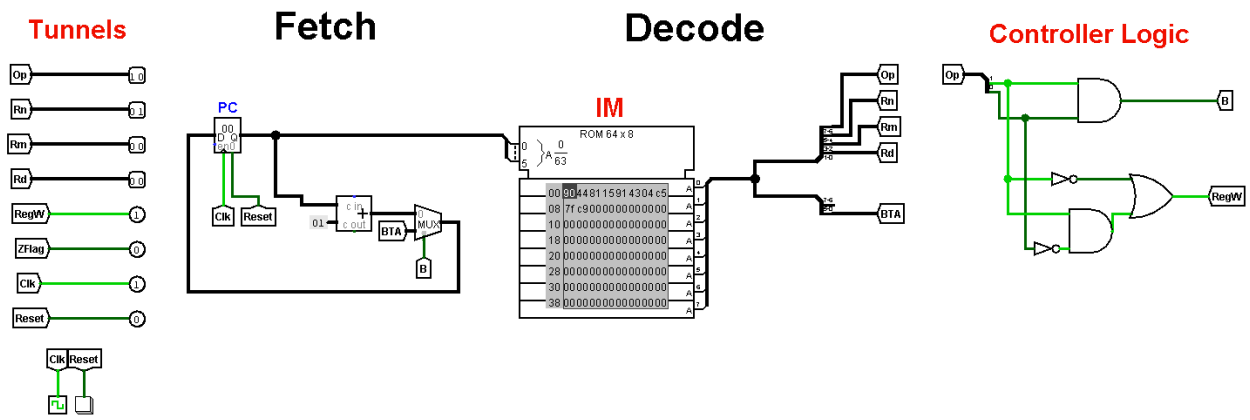
The next function in the main function is decodeHex(), and this is where the decoding step occurs. In decodeHex(), it declares map containers with opcodes of operands and instructions. The third map container stores the registers opcode and is continuously updated throughout the program. Each cycle happens in the for loop that increments by 1 until it reaches the total number of cycles. In the for loop, it converts each element of the vector that contains hex numbers from the hex file into a binary number. The binary number to converted to a string known as the byte string. There is a nested for loop and its purpose is to go through the byte string and decode it. The op_code string takes in the first two characters in byte string and checks which instruction it is in the map of instructions. After the for loop, it starts to create the line of assembly code after decoding the opcodes. Depending on what the instruction is, it carries out different executions in order to get the correct information to be stored in the registers. For each cycle, it also outputs the message in the terminal. If the user included -d in the command line argument, then it will output the disassembly message below.

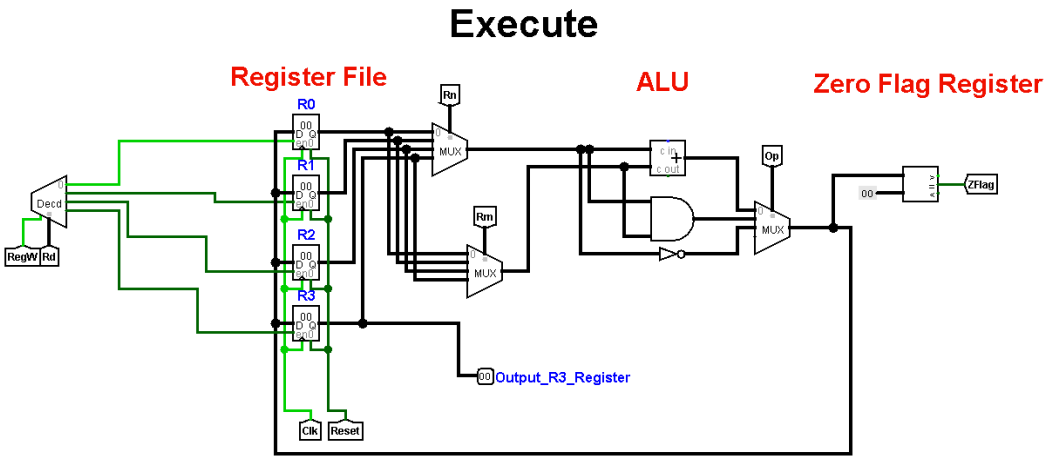
There are three other functions such as the addFunc(), andFunc(), and notFunc() that run if the instruction equals one of the following: “add”, “and”, or “not”. They use bitwise operators, making it more efficient to change the binary number and get the proper answers for each register for every cycle. In each of these functions, it converts the strings into an unsigned integer, allowing bitwise operators to work. Once that is done, it is converted back to a string and updated in the map of registers.

In the Ubuntu terminal, the FISC simulator is compiled to create an executable. It is ran using the format “./fiscsim [hex file]” and other arguments such as the number of cycles and -d is optional. On my personal laptop, I used visual studio code to debug the program by commenting out argv and argc and opening certain hex files by using the code “std::ifstream inFile(“t1.hex”);”. This allowed me to observe whether it was decoding properly and if the byte string had the opcodes in correct places in the string. In order to understand how to get the desired output, I had to do hand assembly. This helped me figure out how to use the z-flag and update the registers r0, r1, r2, and r3 to the right hex numbers.

FISC Simulator

FISCSIM



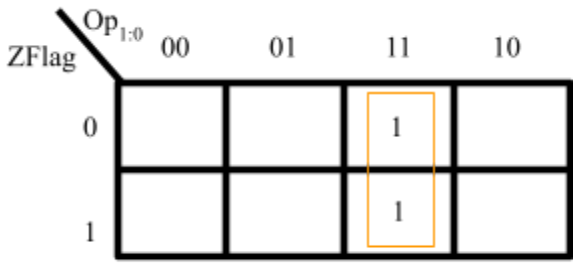


Truth Table: Build Controller Logic

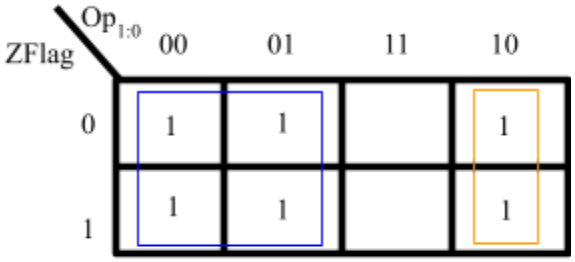
Inputs			Outputs	
Op_1	Op_0	ZFlag	B	RegW
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	1	0

K-maps for B and RegW

K-map for B



K-map for RegW



Explanation

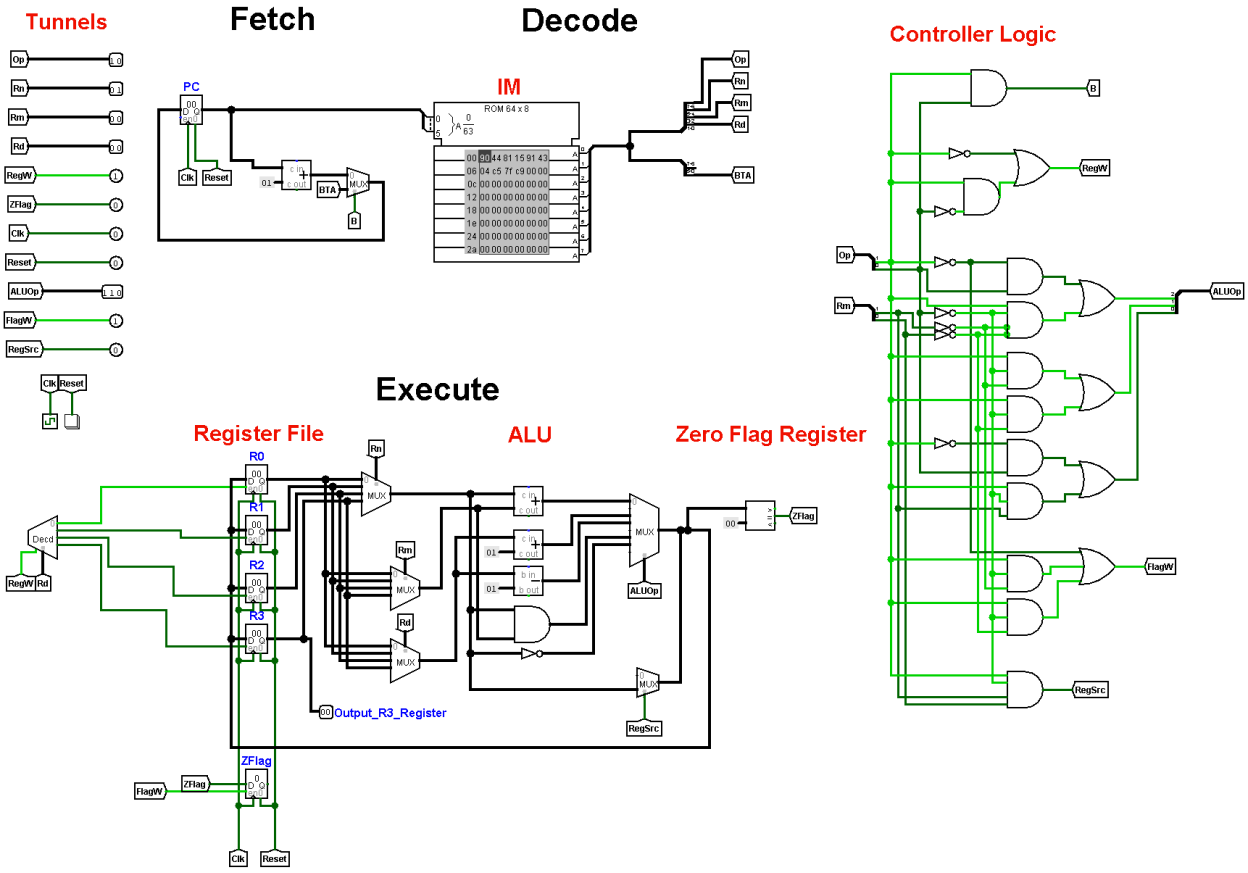
The single-cycle CPU goes through the instruction cycle which consists of the fetch, decode, and execute sections. In the FISCSIM simulator, the fetch section has a register that keeps track of the program counter (PC) and increments by one as the clock works. The reset button and clock are connected to the PC, helping it run and making it easier to debug. The adder is connected to a 2:1 MUX (6-bit) and the selector is attached to the branch (B) tunnel. If B is true then the PC will be set to the branch target address (BTA) which allows it to jump addresses, and if B is not true then the PC will continue to increment. The PC is the input to the instruction memory (IM) and it uses a 64x8 ROM, containing the HEX numbers from the hex files.

The output bits from the IM are divided into separate bits by using splitters. This is where the decode section begins and retrieves the op-code extracted from the instruction word, address of the destination register, source register, and the target branch. The decode (controller) section takes the op-code and goes through logic gates to determine whether B or RegW is true. A truth table and k-maps were utilized to figure out which gates to use for the solution.

The execute section consists of a decoder, register file (RF), and ALU. The 1:4 decoder takes in Rd, the destination register, as its selector and RegW is connected to the enable. In the CPU created in Logisim above, it shows that if RegW is true then it writes to the register and false otherwise. Rd determines which 8-bit register to modify out of four registers. Two multiplexers (MUX) are used to read the data from the registers in order to go through the ALU and determine whether to update the registers or zero flag (ZFlag). The first MUX has a select bit of 2-bit and it takes in Rn, one of the source registers, to grab the value from the correct register. The other MUX is the same, but instead, takes Rm as the selector. The outputs of these two multiplexers go through an arithmetic logic unit (ALU) and perform arithmetic operations between the two inputs. They are connected to a MUX that takes in Op, specifying which instruction was used, and outputs the value needed to write to the destination register. This is also where the ZFlag is determined by comparing the output value to 0. If the output equals zero, then the ZFlag is set to true and false otherwise. The last step in the FISC simulator is outputting R3 register which is attached to a 2-bit output pin in order to see the answer after decoding the hex number, turning it to machine language, and performing arithmetic operations to receive the desired output. The FISC CPU has many tunnels to make it easier to connect circuits. There is the Op, Rn, Rm, Rd, RegW, ZFlag, Clk, Reset, ALUOp, FlagW, and RegSrc.

SISCIM Simulator

SISCSIM

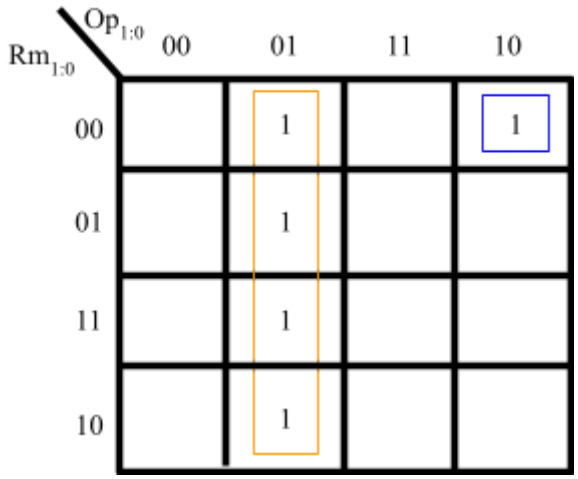


Truth Table: Generate 3-bit ALUOp in Controller

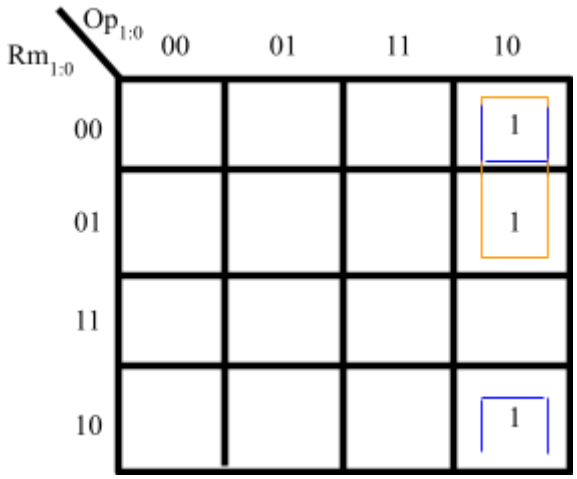
Inputs				Outputs				
Op_1	Op_0	Rm_1	Rm_0	$ALUOp_2$	$ALUOp_1$	$ALUOp_0$	FlagW	RegSrc
0	0	X	X	0	0	0	1	0
0	1	X	X	1	0	1	1	0
1	0	0	0	1	1	0	1	0
1	0	0	1	0	1	0	1	0
1	0	1	0	0	1	1	1	0
1	0	1	1	X	X	X	0	1
1	1	X	X	X	X	X	0	0

K-maps for $ALUOp_{2:0}$ and FlagW

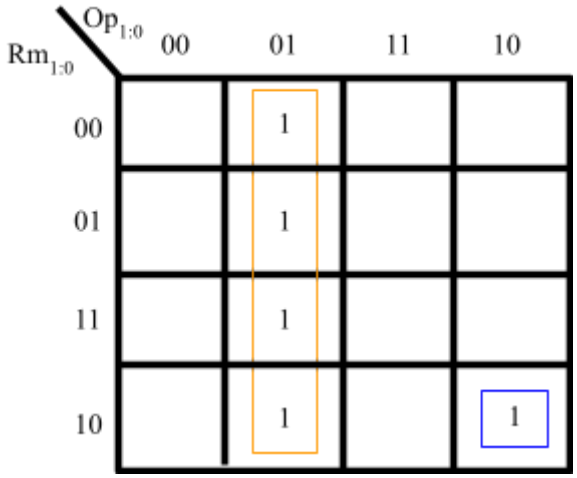
K-map for $ALUOp_2$



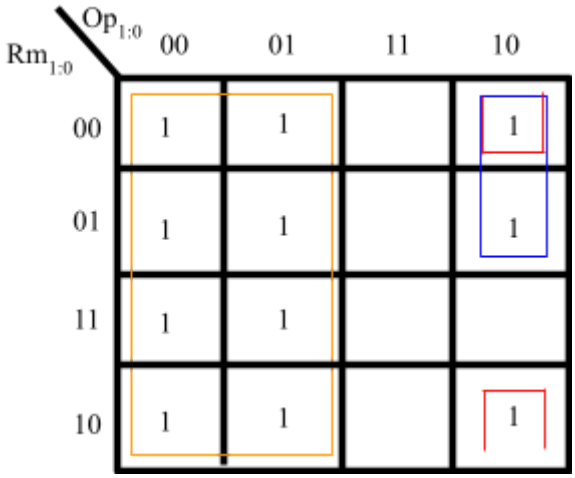
K-map for $ALUOp_1$



K-map for $ALUOp_0$



K-map for FlagW



How the SISCSIM Works

The single-cycle CPU goes through the instruction cycle which consists of the fetch, decode, and execute sections. The SISCSIM simulator is similar to the FISCSIM simulator; however, the only difference is that it has the ability to take in more instructions and the ALU is more complex. A SISC CPU has six or seven instructions while the FISC CPU has four or five instructions. The fetch process is the same as the FISC simulator. The controller logic now determines whether FlagW and RegSrc are true or false, it takes in Op and Rm as the inputs to get the 3-bit ALUOp. The truth table to find the schematic is above as well as the k-maps used to figure out the logic gates to use.

In the execute section, a zero flag register has been added, the ZFlag tunnel is connected to the data input, and the FlagW tunnel is connected to enable. If FlagW is true, then the zflag register gets updated with the zflag value. If FlagW is false, then the zflag register does not get updated at all. The logic gates in the ALU have been modified. There is now an 8:1 8-bit MUX that takes in the ALUOp as the selector. An increment and decrement instruction was added, so there is an adder that adds Rd and a constant number 1 as well as a subtractor that subtracts Rd and a constant number 1. The source register, Rn, is also connected to another 2:1 MUX with RegSrc as the selector which essentially writes the contents of Rn to the destination register if the RegSrc is true (R=1) and false otherwise.