CSEE 4290: Computer Architecture

Top Level SCC – Spring 2025 – Group 5

Dr. Herring

Jordan Williams

Matthew Sherritt

Nathaniel LeBlanc

Ricky Chan

Github

Table of Contents

Group Responsibilities / Primary Roles 3
ISA Map 4
Testbench 5
Multiply Testcase 6
Multiply Algorithm 6
uCode Handler 6
Floating Point Addition Testcase 8
(Class Testcases) Bugs 9
Diagrams 10
Test Plan 10
Schedule 10
Next Steps/Ideas for Improvement 10
Group feedback 10

Group Responsibilities / Primary Roles

- Jordan Williams
  - Self-Checking Testbench (Joined Group after Group 4 Disbanded)

- Ricky Chen
  - Instruction Fetch, Registers, Test Case and MUL Algorithm

- Matthew Sherritt
  - Instruction Decode, Memory, Verification/Debug, uCode Control

- Nathaniel LeBlanc
  - Execute/ALU, uCode Control, Writeback

ISA Map

- MOV: Bubble, 1, 2, 3, 5
- MOVT: 2, 5, 6
- MOVF: 2
- LOAD: Bubble, 1, 2, 5
- STOR: Bubble, 1, 2, 3, 5, 6, 7
- MUL: Multiply (uCode Inclass Check), 1, 5, 6
- MULS: Multiply (uCode Inclass Check), 1, 6
- ADD: Bubble, 1, 2, 5, 6
- ADDS: 3, 5
- SUB: Bubble, 1, 2, 3, 5
- SUBS: Bubble, 2, 3, 5, 6
- AND: Multiply (Test Case), 2, 3, 5, 6
- ANDS: Multiply (Test Case), 5, 6
- OR: 3, 5, 6
- ORS: 3, 5
- XOR: 2, 3, 6
- XORS: 3
- NOT: Multiply (Test Case), 5
- B: Bubble Sort: 1, 3, 5, 6
- B.cond: Bubble Sort: 1, 2, 3, 5, 6
- BR: 2, 6
- HALT: ALL
- LSL: Multiply (Test Case), 1, 2, 5, 6
- LSR: Multiply (Test Case), 1, 2, 5, 6
- CLR: Bubble Sort, 6
- SET: 2, 6
- NOP: 1

Group #	Checkpoint 4 Additional Program	Checkpoint 5 Testcase
1	nl, fibbonacci, base^exponent, LUT	sine function LUT
2	fibbonacci	bit-logic checksum
3	greatest common divisor	Encryption
4		
5	Shift Multiplier	Floating point arithmetic
6	shift counter (count 1s and 0s)	shift counter (count 1s and 0s)
7	fibbonacci	Kadane's maximum subarray ALG
Bubble (Special)		

Load/Store	
Instr	Tested By
LOAD	Bubble, 2, 1, 7
STOR	Bubble, 2, 6, 3, 7

Data Immediate	
Instr	Tested By
MOV	Bubble, 1, 2, 5, 3
MOVT	6,2,5
MOVF	2
SAVF	2, 5
MUL	1, 6
MULS	3
ADD	Bubble, 6,2,5,7
ADDS	3,7
SUB	Bubble,2,5,7
SUBS	6,2,5, 3,7
AND	3
ANDS	6
OR	3
ORS	3
XOR	3
XORS	3
LSL	6,2,1,5
LSR	1,6,2,5
CLR	6, 7
SET	6, 2

Data Register	
Instr	Test By
MUL	1, 6
MULS	5
ADD	1,7
ADDS	5,7
SUB	Bubble, 1, 3
SUBS	Bubble,2, 3,7
AND	2, 5
ANDS	5
OR	6, 5
ORS	5
XOR	6, 2
XORS	3
NOT	5

System/Branch	
Instr	Tested By
B	1, Bubble, 5, 3, 6, 7
B.cond	1, Bubble, 5, 6, 7
BR	6, 2
NOP	1
HALT	All

Condition Flags (Branching)			
Name	Meaning	Flags	Tested By
EQ	Equal	Z==1	Bubble, 6, 5, 3,7
NE	Not Equal	Z==0	2 , 5 , 7
HS (CS)	Unsigned Higher or Same (Carry Set)	C==1	1
LO (CC)	Unsigned Lower (Carry Clear)	C==0	5
MI	Minus (Negative)	N==1	Bubble, 3, 5
PL	Plus (Positive or Zero)	N==0	5
VS	Overflow Set	V==1	5
VC	Overflow Clear	V==0	5
HI	Unsigned Higher	C==1 && Z==0	5
LS	Unsigned Lower or Same	!(C==1 && Z==0)	5
GE	Signed Greater Than or Equal	N==V	5
LT	Signed Less Than	N!=V	3,7
GT	Signed Greater Than	Z==0 && N==V	3
LE	Signed Less Than or Equal	!(Z==0 && N==V)	5,7

Testbench

Our testbench instantiates your single-cycle computer core and drives it with a free-running, simulation clock while holding reset high for the first three rising edges to ensure a clean start for the simulation. Clock enable is tied active, and DUT outputs halt_f, which is used functionally in the testbench, and visibility taps for instructions, error bits and data paths are exposed for observation. For debugging, the bench writes a dump.vcd file so you can inspect internal activity in a viewer. The self checking function of the testbench is the file-based checker that blocks until the processor asserts halt_f, then waits one more clock to let the DUT finish writing its memory dump. This seflchecker opens two files: the expected output of the SCC given the instructions ran on the class emulator (dataoutput.csv) and the memory output from the SCC DUT running the same instructions (scc_out.txt). The first line in each file is skipped as a header, and lines after this are parsed into <address, value> pairs. The checker checks the emulator output file line by line and, when necessary, goes through the SCC DUT dump to synchronize on matching addresses. For every matched address, it performs bounds checking against the declared 64 KB memory space (MEM_BYTES = 1<<16) to prevent out of range access and then compares the SCC DUT value with the expected value from the CSV. On any discrepancy, it immediately prints a FAIL message with the line number, address, observed value, and expected value, and ends the simulation. If a line matches the expected output, the testbench displays "Pass @ line ..." for each match. If the testbench reaches the end without a mismatch, it displays a PASS and ends the simulation.

Multiply Testcase

One of the testcases used was our multiply instruction but ran without microcode to ensure it worked normally. The algorithm is explained in more detail in the Multiply section.

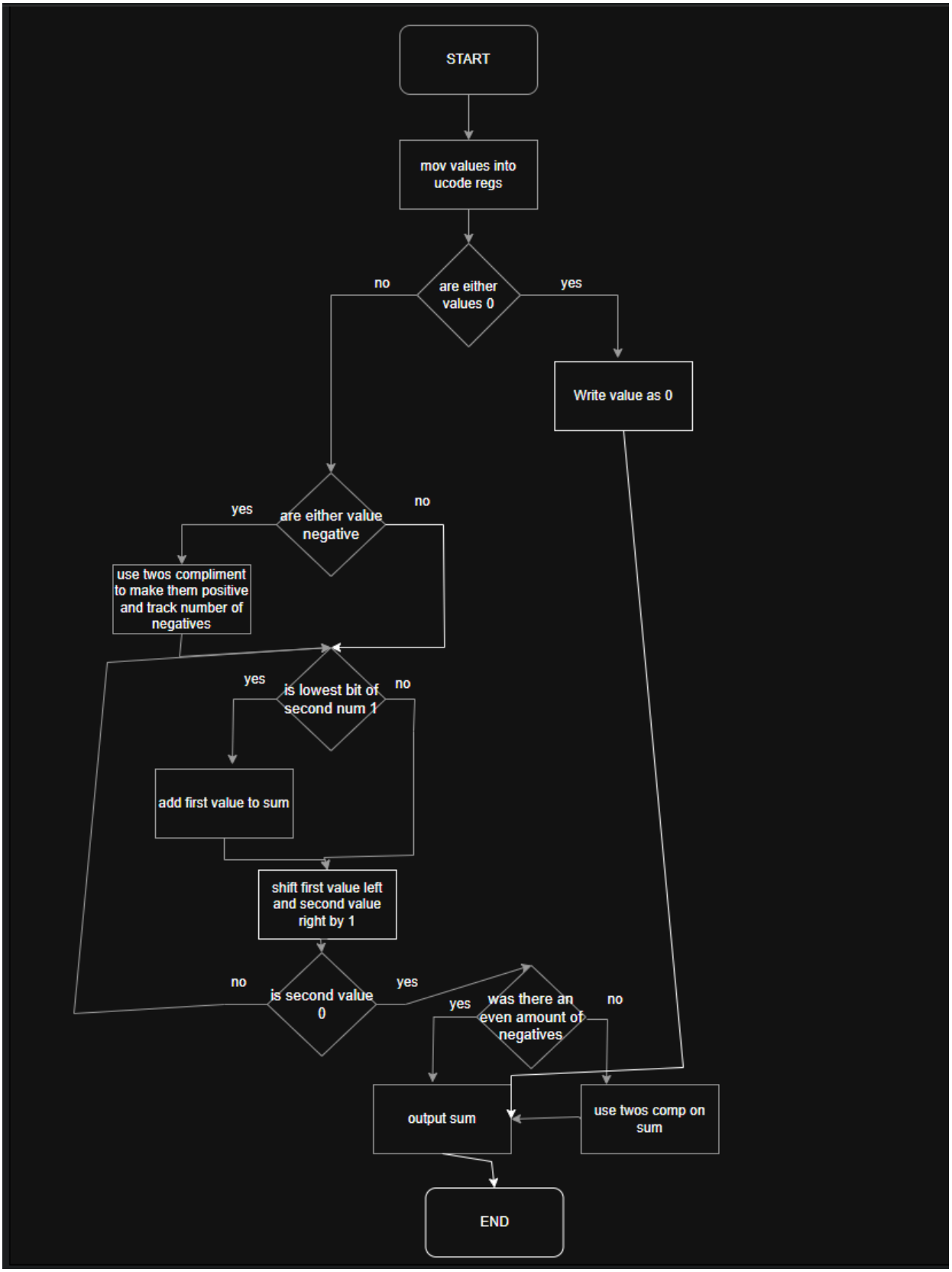
Instructions used: ADD, SUBS, B.cond, ANDS, LSL, LSR, B, NOT, HALT

Multiply Algorithm

- First checks if the multiplicand is 0 or a negative with a SUBS #0. If the value is 0, then it branches to the end. If the value is negative, then it branches to make a 2's complement of the multiplicand and adds 1 to a counter that keeps track of how many negative operators the MULT had (R5). If the value is neither a 0 nor negative, then it moves on to check the multiplier with the exact same process.
- Once both operators are positive, the algorithm then checks the LSB of the multiplier to see if it is a 1 by using the ANDS instruction. If it is, then the value of the multiplicand is added to the product register (R2).
- The multiplicand is then shifted left by 1 and the multiplier right by 1, and the process above is repeated until the multiplier is equal to 0.
- The algorithm then used another ANDS on R5 to check how many negative operands the MULT had and determines whether the output should be negative or not. If the output should be negative, then it converts R2 to its 2's complement and goes on to set the proper flags. If the output is positive, then it branches to set the flags.
- The flags are always returned to their original state (Using a series of ANDS to check against the flags in R4 which was set by MOVF). The flags will then be newly set by the final value of the multiply depending on if it was a MUL or MULS instruction.

uCode Handler

Our uCode handler has its own IF stage and set of registers. Once a multiply instruction is detected, the main program's IF stage will halt and the uCode will take over. The uCode's IF stage acts exactly like the main program's except it will inject/change some ROM instructions to allow the uCode to import values from the main registers into the ROM registers, to export the output value from the ROM register to the main destination register (And set flags if MUL/I S was used), and to set the immediate values as an operand if a MULI/S was used. Both the uCode handler and the main program share the same ID, EXE, and WB stages. The uCode knows the multiply is over once it sees a HALT instruction in the ROM.


START
mov values into ucode regs
are either values 0
no
yes
Write value as 0
are either value negative
yes
no
use twos compliment to make them positive and track number of negatives
is lowest bit of second num 1
yes
no
add first value to sum
shift first value left and second value right by 1
is second value 0
no
yes
was there an even amount of negatives
yes
no
output sum
use twos comp on sum
END

Floating Point Addition Testcase

- The main algorithm for the operation is to first check the two operands for the floating-point special cases of 0, +/- inf, and NaN by using ANDS (Check 0 flag) and CMP (Check 0 flag) against a bit mask of 0x7F800000 (Checks for inf).
- If the first operand is an inf, the program will then check the second operand to see if it is the opposite inf. If so, then the result is 0. If not, then the result will be the inf or NaN. If an operand is 0, then the result will be the other operand.
- The program then uses 2 masks and AND operations to store the exponent and mantissa of each operand into 2 registers (4 in total). An OR operation is used to add the implicit MSB of 1 back into the mantissas for future calculations. An LSR is also used to shift the exponents by 23 bits so that the values can be easily added and subbed from.
- The program then does a compare between the exponents of the two operands to see which one is larger. The goal is to shift the smaller of the two operands so that the exponent matches that of the larger exponent. A branch conditional is used to move to the area of the program that shifts either operand 1 or 2. The smaller of the two operands has their mantissa LSR by 1 and their exponent ADD by 1. Another compare is then performed to check if the exponents are equal. If the exponents are not, then a branch is used to loop back. If the exponents are equal, then the program branches to check for negatives.
- A CMP against 0 is used on the original operands to check if the sign is negative or not for both operands. If an N flag is detected, a branch is taken and the 2's complement is performed on the corresponding mantissa before branching back to continue the program.
- The two mantissas are then added together with an ADDS and the N flag is again checked to see if the resulting mantissa is negative. If so, then the 2's complement is taken for the resulting mantissa and a mask is placed into a register with a signed N bit for later.
- The program then checks if the resulting floating-point value is normalized. First, to check if a right shift is needed for normalization, a CMP is done between the bit mask for the mantissa (with the implicit 1 included) and the resulting mantissa. If the resulting mantissa is larger than the mask, then the program LSR the mantissa by 1 and ADD the exponent by 1 before looping back to compare again. Else, the program moves on to check if a left shift is required.
- To check if a left shift is required for normalization, a CMP is done between the resulting mantissa and a mask of just the implicit 1 of the mantissa (where the mantissa consists of all 0s except for the implicit MSB of 1). If the resulting mantissa is smaller than the mask (N flag == 1), then the resulting mantissa is LSL by 1 and the exponent is SUB by 1 before branching back to check for the shift again. Else, the program branches to recompile the results into a single floating-point value.

- To recompile, a mask is first used again to remove the implicit 1 from the mantissa and the exponent is LSR 23 bits back into its original position. 2 OR instructions are then used to combine the exponent, mantissa, and signed bit into the results register.
- In our final test case, instead of nonconditional branches, the SAVF instruction is used alongside others to create the necessary flags to test nearly every conditional branch. The program is also looped 6 times to test 6 different operands for adds.

(Class Testcases) Bugs

- Scc_output.txt was in the incorrect format so other testbenches could not read (fixed formatting)
- $fclose() was never used in memory file when writing the output so the testbenches could not read (fixed by including a close)
- $flcose() was also forgotten in the testbench and crashed (fixed by including a close)
- SET was only setting 1 bit to 1 instead of 32 bits (fixed by setting 32 bits)
- 2 MULs could not be used back-to-back because the flag for preventing catch was not updated in time (fixed by updating the value when the program counter ticks)
- BR did not work because we forgot to add it (fixed by supporting the instruction)
- LOAD was broken because we added a random IF statement in preparation for BR and forgot about it (fixed by removing the statement)
- LOAD was also broken because somehow branch conditionals were added to be checked for a memory operation in EXE (fixed by removing the branch statement)
- Our test case was broken because we accidentally reverted to part of a previous iteration where we used MOV register to register instead of an ADD instruction (fixed by using the ADD)
- Our test case couldn't compile because we accidentally deleted a ; before a comment (fixed by adding a ;)
- Failed Group 2's test because the assembler has a bug where it will skip over a memory address if the value is set to 0x00000000 and the LOAD instruction ends up grabbing XXXXXXXX instead of 0 (fixed by manually adding in the missing memory address value at the location)

Diagrams

https://github.com/Herring-UGACSEE-4290-F25/Group-5/tree/main/Proj2(SCC)/Final/charts_and_docs

Test Plan

- Check writability to R15
- Check every branch condition
- Check ORS and OR
- Check MUL (Data Register)
- Doublecheck MOVT and MOVF in regular program, not just uCode

Schedule

- ID/IF: 10-03
- Execute 3 Instructions/Branch Imm: 10-10
- Execute Complete: 10-20
- Testcases: 10-27
- SCC Complete: 11-03

Next Steps/Ideas for Improvement

- Reduce possibly redundant control flags in our SCC
- Much more separation between data and control lines
    - Such as including a separate module for muxes
- Allow for uC to read from main registers without having to inject it into the instruction
- Make the uC control more efficient by sharing more of IF to save space
- Make it synthesizable

Group feedback

- Add MOV register to register as a pseudo instruction to avoid ADD #0
- Add a floating-point compute unit