CSE 141L Milestone 1

Ada Qi, A16999495

Academic Integrity

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

To uphold academic integrity, students shall:

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

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

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

Ada Qi

0. Team

1 Member
Ada Qi

1. Introduction

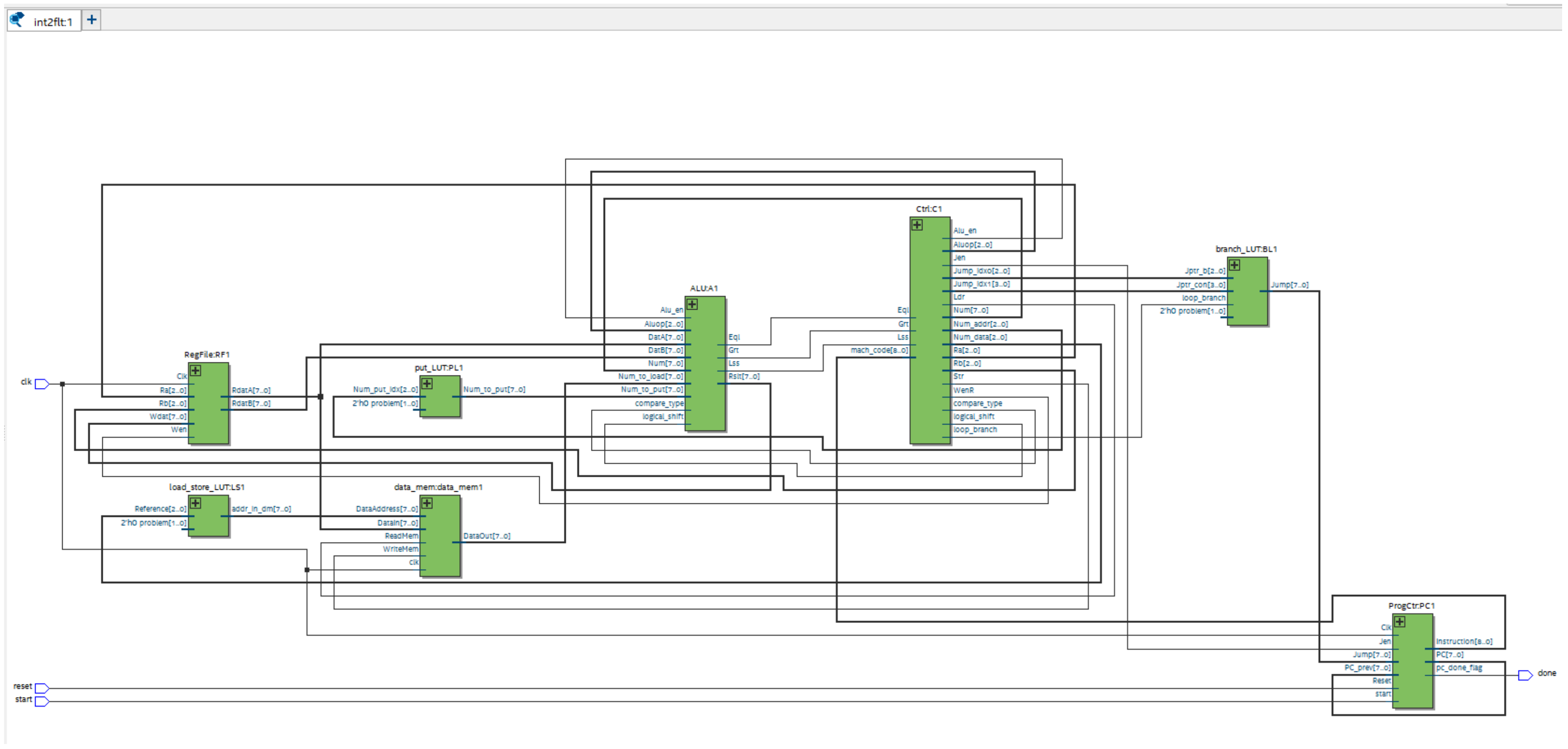
TODO. Name your architecture. What is your overall philosophy? What specific goals did you strive to achieve? Can you classify your machine in any of the standard ways (e.g., stack machine, accumulator, register-register/load-store, register-memory)? If so, which? If not, devise a name for your class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

My processor would be in form of register-register/load-store.

2. Architectural Overview

TODO. This must be in picture form. What are the major building blocks you expect your processor to be made up of? You must have data memory in your architecture. (Example of MIPS: https://www.researchgate.net/figure/The-MIPS-architecture_fig1_251924531)

RTL Viewer Graph


int2flt:1
clk
reset
start
RegFile:RF1
Clk
Ra[2..0]
Rb[2..0]
Wdat[7..0]
Wen
RdatA[7..0]
RdatB[7..0]
put_LUT:PL1
Num_put_idx[2..0]
2'h0 problem[1..0]
Num_to_put[7..0]
ALU:A1
Alu_en
Aluop[2..0]
DatA[7..0]
DatB[7..0]
Num[7..0]
Num_to_load[7..0]
Num_to_put[7..0]
compare_type
logical_shift
Eql
Grt
Lss
Rslt[7..0]
load_store_LUT:LS1
Reference[2..0]
2'h0 problem[1..0]
addr_in_dm[7..0]
data_mem:data_mem1
DataAddress[7..0]
DataIn[7..0]
ReadMem
WriteMem
clk
DataOut[7..0]
Ctrl:C1
Eql
Grt
Lss
mach_code[8..0]
Alu_en
Aluop[2..0]
Jen
Jump_idx0[2..0]
Jump_idx1[3..0]
Ldr
Num[7..0]
Num_addr[2..0]
Num_data[2..0]
Ra[2..0]
Rb[2..0]
Str
WenR
compare_type
logical_shift
loop_branch
branch_LUT:BL1
Jptr_b[2..0]
Jptr_con[3..0]
loop_branch
2'h0 problem[1..0]
Jump[7..0]
ProgCtr:PC1
Clk
Jen
Jump[7..0]
PC_prev[7..0]
Reset
start
Instruction[8..0]
PC[7..0]
pc_done_flag
done

3. Machine Specification

Instruction formats

Two example rows have been filled for you. When you submit, do not include the example types. Add rows as necessary. In your submission, please delete this paragraph.

TYPE	**FORMAT**	**CORRESPONDING INSTRUCTIONS**
R	3 bits opcode, 3 bits operand register A (also destination), 3 bits operand register B	add, sub, etc.
B	2 bits instruction type, 3 bits branching type, 4 bits address	beq, bne, etc.
Special Case FLIP	It’s also an R instruction, but with 6 bits opcode(because the opcodes are limited, and flipping bits in 1 single register takes only 3 bits to represent the index of register)	flip

Operations

An example row has been filled for you. When you submit, do not include the example type. In the name column, be sure to also add the definition of what the example actually does. For example, "lsl = logical shift left" would be an appropriate value to put in the name column. In the bit breakdown column, add in parenthesis what specific values the bits should be in order. X indicates that it will be specified by the programmer's instruction itself (i.e. specifying registers). In the example column, give an example of an "assembly language" instruction in your machine, then translate it into machine code. Add rows as necessary. In your submission, please delete this paragraph.

NAME	**TYPE**	**BIT BREAKDOWN**	**EXAMPLE**	**NOTES**
and = logical and	R	1 bit type (0) bits opcode (010), 1 bit funct (1), 1 bit operand register (X), 1 bit operand register (X), 2 bit destination register (XX)	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` `and R0, R1, R2 ⇔ 0_010_1_0_1_10` # after and instruction, R2 now holds `0b0001_0000`	This is a completely bogus example, since this implies that there are only 2 possible operand registers and 4 possible destination registers. Mention special things like implied destination register (i.e. stack) or special notes here.
MOV	R	000_aaa_bbb Move a number from register b to register a	Write to Ra	
PUT	R	001_aaa_nnn Move a number to register a	Write to Ra	Up to 7 pre-defined value, stored in LUT
ADD	R	010_aaa_bbb Add the number in register a and b to register a	Write to Ra	
SUB	R	011_aaa_bbb	Write to Ra	

		Sub the number in register b from a and store it in register a		
LSL/ LSR/	R	100_x_aaa_bb If x = 0, operate LSL if x = 1, operate LSR	Write to Ra	Register B restricted to r0, r1, r2, r3
CMP(r)	B	101_0_aa_bbb Compare number in register a and register b. Set Flags, No Write	Write to Ra	Register A restricted to r0, r1, r2, r3
CMP(n)	B	101_1_aaa_nn	Write to Ra (just to simplify the case of comparing with 0 or 1)	Can only compare to number 0, 1, 2, 3
		All branching instructions		The total number of addresses “d” cannot exceed 15, meaning that the total number of branches should not exceed 15, and there’s a LUT for all existing branches
BEQ	B	11_000_dddd	Jump to address dddd	If the “equal” signal is high
BNE	B	11_001_dddd	Jump to address dddd	If the “equal” signal is low
BGE	B	11_010_dddd	THIS INSTRUCTION IS REMOVED!!!	If the “greater” signal or the “equal” signal is high
BGT	B	11_010_dddd	Jump to address dddd	If the “greater” signal is high
BLE	B	11_011_dddd	Jump to address dddd	If the “less” signal or the “equal” signal is high
BLT	B	11_100_dddd	Jump to address dddd	If the “less” signal is high
B	B	11_101_0_ddd	Directly jump to address ddd	Address serial number restricted to 0-7, meaning that the branching number should

				be assigned to 0-7
FLIP	R	11_101_1_ddd	Write to Rd(Ra)	Flipping all bits of register a
LOAD	/	11_110_a_nnn	Load from data memory	need LOAD_LUT
STORE	/	11_111_a_nnn	Store to data memory	need STORE_LUT

Each line of machine code is 9-bit wide.
If the instruction is MOV, then the first 3 bits should be 000, and the middle 3 bits should be register a, the last 3 bits should be the register b.
For example, "MOV r4 r0" should be 000_100_000.

If the instruction is PUT, then the first 3 bits should be 001, and the middle 3 bits should be register a, the last 3 bits should be the number n.
For example, "PUT r2 6" should be 001_010_110.

If the instruction is ADD, then the first 3 bits should be 010, and the middle 3 bits should be register a, the last 3 bits should be the register b.
For example, "ADD r0 r2" should be 010_000_010.

If the instruction is SUB, then the first 3 bits should be 011, and the middle 3 bits should be register a, the last 3 bits should be the register b.
For example, “SUB r1 r3” should be 011_001_011.

If the instruction is LSL, then the first 4 bits should be 100_0, and the middle 3 bits should be register a, the last 2 bits should be the register b.
For example, “LSL r3 r2” should be 100_0_011_10.

If the instruction is LSR, then the first 4 bits should be 100_1, and the middle 3 bits should be register a, the last 2 bits should be the register b.
For example, “LSR r4 r3” should be 100_1_100_11.

If the instruction is CMPR, then the first 4 bits should be 101_0, and the middle 2 bits should be register a, the last 3 bits should be the register b.
For example, “CMPR r0 r2” should be 101_0_00_010.

If the instruction is CMPN, then the first 4 bits should be 101_1, and the middle 3 bits should be register a, the last 2 bits should be the number n.
For example, “CMPN r0 0” should be 101_1_000_00.

If the instruction is BEQ, then the first 5 bits should be 11_000, and the last 4 bits should be a number suggesting the branching serial number.
For example, “BEQ 4” should be 11_000_0100.

If the instruction is BNE, then the first 5 bits should be 11_001, and the last 4 bits should be a number suggesting the branching serial number.
For example, “BNE 6” should be 11_001_0110.

If the instruction is BGT, then the first 5 bits should be 11_010, and the last 4 bits should be a number suggesting the branching serial number.
For example, “BGT 1” should be 11_010_0001.

If the instruction is BLE, then the first 5 bits should be 11_011, and the last 4 bits should be a number suggesting the branching serial number.
For example, “BLE 9” should be 11_011_1001.

If the instruction is BLT, then the first 5 bits should be 11_100, and the last 4 bits should be a number suggesting the branching serial number.
For example, “BLT 5” should be 11_100_0101.

If the instruction is B, then the first 6 bits should be 11_101_0, and the last 3 bits should be a number suggesting the branching serial number.
For example, “B 0” should be 11_101_0_000.

If the instruction is FLIP, then the first 6 bits should be 11_101_1, and the last 3 bits should be the register a.
For example, “FLIP r0” should be 11_101_1_000.

If the instruction is LOAD, then the first 5 bits should be 11_110, and the middle 2 bits should be the register a, the last 2 bits should be the number n.
For example, “LOAD r0 0” should be 11_110_00_00.

If the instruction is STORE, then the first 5 bits should be 11_111, and the middle 2 bits should be the register a, the last 2 bits should be the number n.
For example, “STORE r2 2” should be 11_111_10_10.

Internal Operands

TODO. How many registers are supported? Is there anything special about any of the registers (e.g. constant, accumulator), or all of them general purpose?

The design supports a total of eight registers. Although most registers can be considered general-purpose, certain registers serve specialized roles. For instance, when loading and storing data from and to memory, the operations are always performed through register R0 or R1. Additionally, register R7 typically holds the sign bit of the current number being processed. Apart from these specific uses, the remaining registers can be used flexibly, offering general-purpose functionality.

Control Flow (branches)

TODO. What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported? How do you accommodate large jumps?

The architecture supports both conditional and unconditional branching. Branch target addresses are determined by values stored in a dedicated Branching Lookup Table (LUT), which correlate closely to their positions in the machine code. Since branching is managed through this LUT rather than direct, fixed offsets, there is effectively no inherent limitation on the branch distance. Programmers must, however, update the LUT values if instructions are added or removed, ensuring that the branch targets remain accurately mapped to their intended locations.

Addressing Modes

TODO. What memory addressing modes are supported, e.g. direct, indirect? How are addresses calculated? Give examples.

The system supports indirect addressing. Three separate lookup tables are implemented to handle different problems or use-cases. Each table maps a 3-bit index to an 8-bit memory address, effectively serving as a pointer into data memory. This approach streamlines access to memory by allowing the programmer to reference locations indirectly through compact indices rather than fixed or immediate addresses.

4. Programmer's Model [Lite]

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

All operands should be placed into registers at the beginning of the computation chain, if possible. By filling up the registers early, most of the computation can be done with little access to memory. This keeps the memory read and write signals stable, or better yet, turned off, during the main processing stages. After all the steps in the process are done, the final results can be saved back to memory at the end of the program. This makes things simpler and lowers the chance of time problems or issues with synchronization that can happen when reading or writing memory a lot during an action. To put it another way, a programmer's work should be organized into three steps: (1) Initialization, which loads values from memory into registers; (2) Computation, which mostly works with data already in the registers; and (3) Finalization, which saves the results back to memory. Following this organized model can make debugging easier, make things more predictable overall.

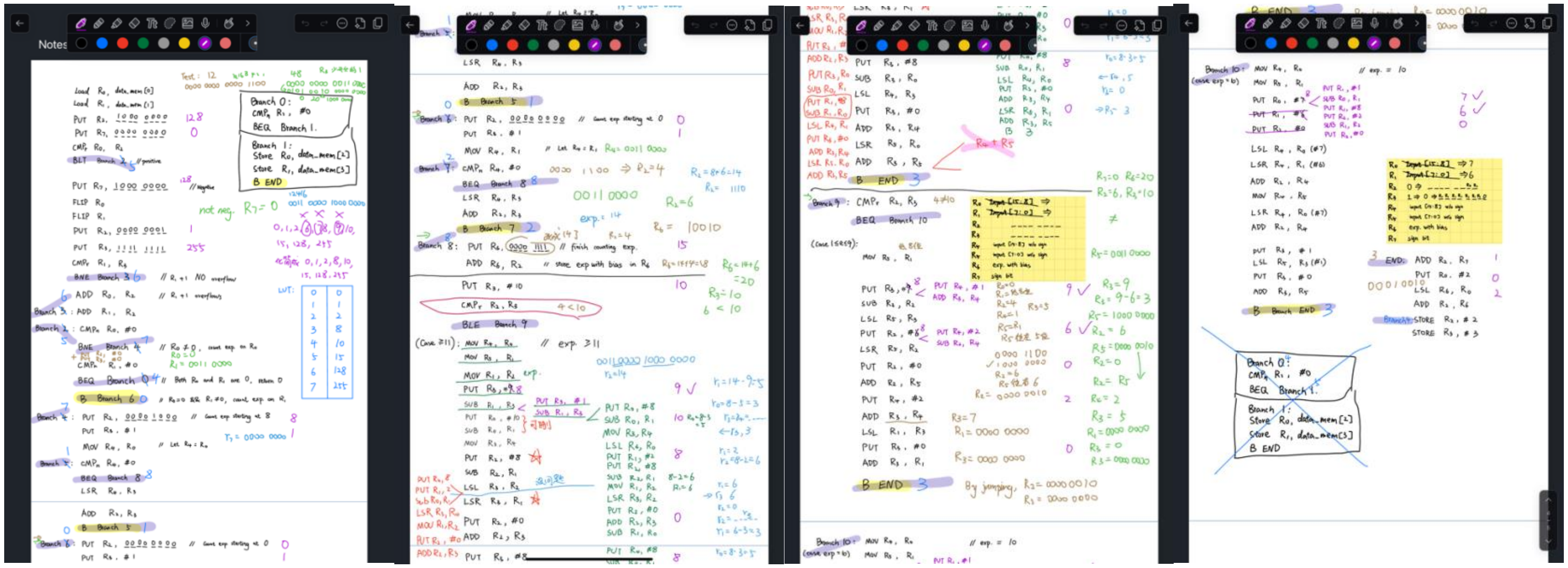
TODO. 4.2 Can we copy the instructions/operation from MIPS or ARM ISA? If no, explain why not? How did you overcome this or how do you deal with this in your current design? Word limit: 100 words.

It is not feasible to directly use MIPS or ARM instructions because their formats are typically more complex, often including multiple source and destination registers in a single instruction. In contrast, this design streamlines each instruction into fewer fields—generally an opcode plus one or two operand registers or immediate values. By reducing instruction complexity, the machine trades off some functionality and flexibility for simplicity and easier implementation. This approach avoids the overhead of decoding multi-operand instructions, making it simpler to implement while still supporting the essential operations needed for the intended tasks.

5. Program Implementation

An example Pseudocode and Assembly Code has been filled out for you. When you submit, please delete the example along with this paragraph.

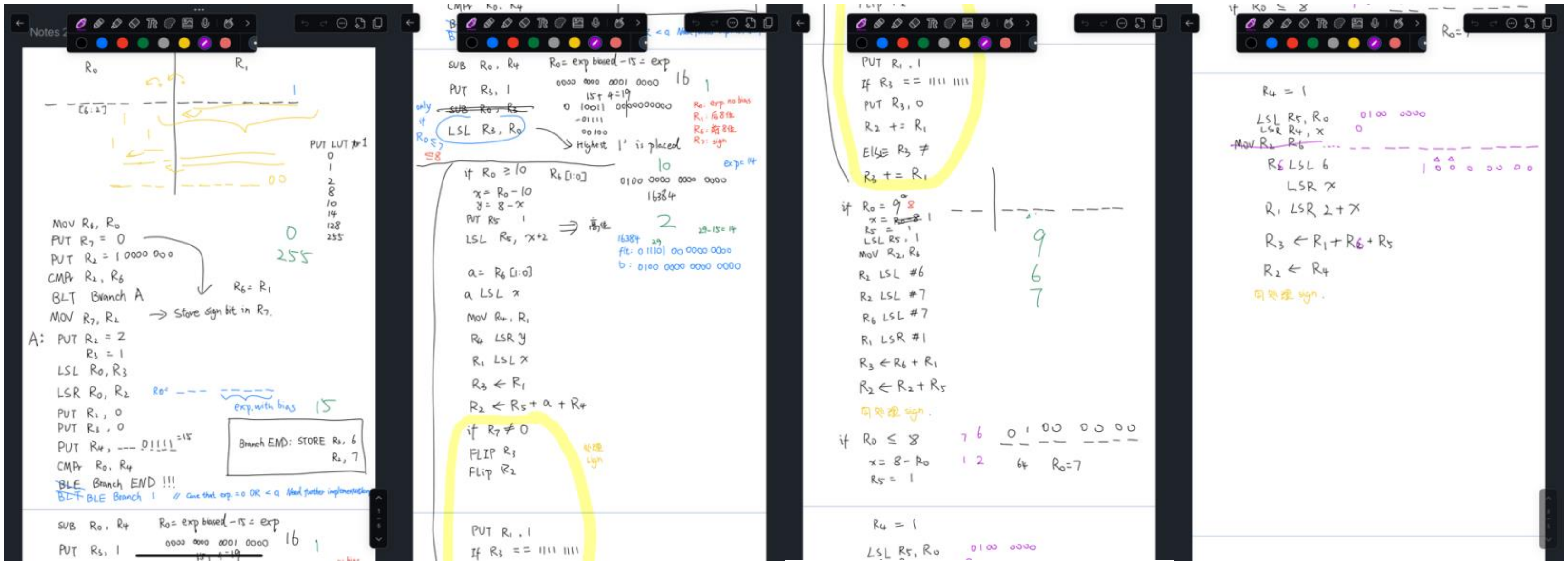
Program 1 Pseudocode



Program 1 Assembly Code

https://github.com/z3qi/CSE141L/blob/main/Assembly1_revised%20copy.s

Program 2 Pseudocode

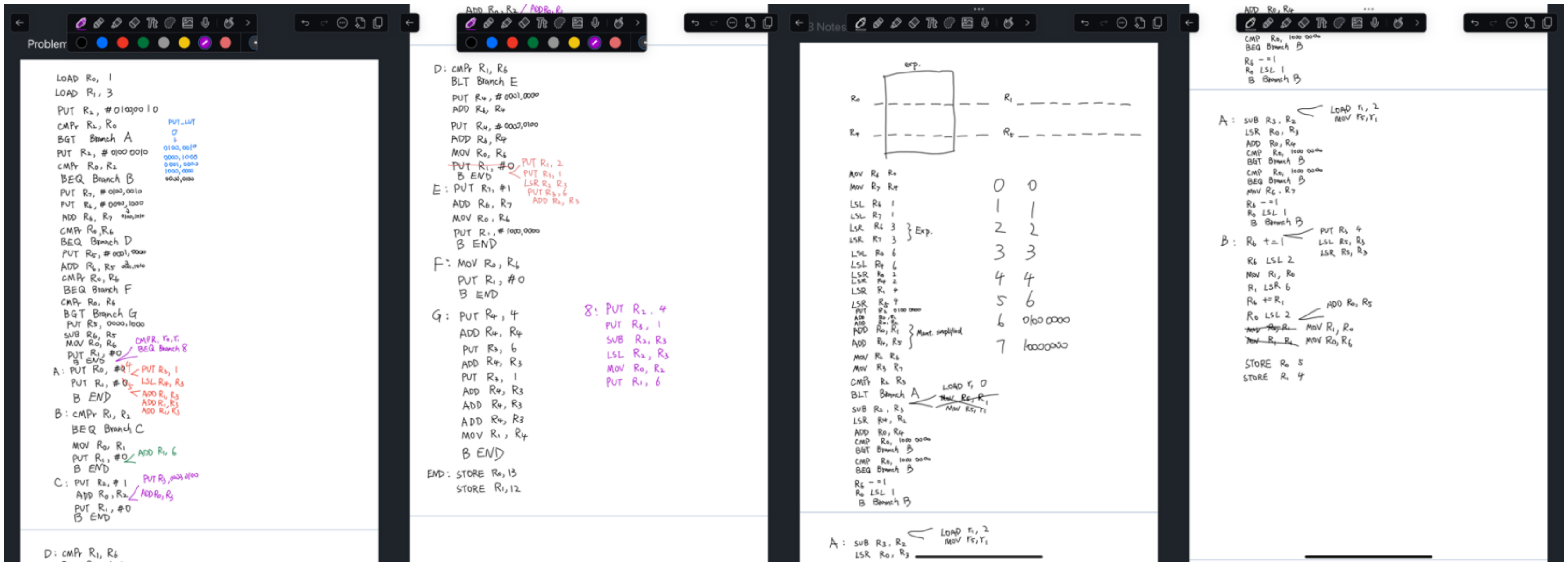


```
MOV R6, R0
PUT R7 = 0
PUT R2 = 1 0000 000
CMPr R2, R6
BLT Branch A
MOV R7, R2          -> Store sign bit in R7.
A: PUT R2 = 2
       R3 = 1
   LSL R0, R3
   LSR R0, R2
   PUT R1, 0
   PUT R3, 0
   PUT R4, --- 01111          =15
   CMPr R0, R4
   ~~BLE~~ Branch END !!!
   ~~BLT~~ BLE Branch 1   // Case that exp. = 0 OR < a  Need further implementation

Branch END: STORE R3, 6
                  R2, 7

SUB R0, R4          R0 = exp biased - 15 = exp
PUT R5, 1
~~SUB R0, R2~~
LSL R3, R0          -> Highest '1' is placed
    (only if R0 ≤ 7)  ≤8

if R0 ≥ 10          R6 [1:0]
    x = R0 - 10
    y = 8 - x
    PUT R5 1
    LSL R5, x+2     => 高位
    a = R6 [1:0]
    a LSL x
    MOV R4, R1
    R4 LSR y
    R1 LSL x
    R3 <- R1
    R2 <- R5 + a + R4
    if R7 ≠ 0
        FLIP R3
        FLip R2
        PUT R1, 1
        If R3 == 1111 1111
            PUT R3, 0
            R2 += R1
        ElsE R3 ≠
            R3 += R1

if R0 = 9  8
    x = ~~R0-8~~ 1
    R5 = 1
    LSL R5, 1
    MOV R2, R6
    R2 LSL #6
    R2 LSL #7
    R6 LSL #7
    R1 LSR #1
    R3 <- R6 + R1
    R2 <- R2 + R5
    同处理 sign.

if R0 ≤ 8
    x = 8 - R0
    R5 = 1
    R4 = 1
    LSL R5, R0
    LSR R4, x
    ~~MOV R2, R6~~
    R6 LSL 6
       LSR x
    R1 LSR 2 + x
    R3 <- R1 + R6 + R5
    R2 <- R4
    同处理 sign.
```

Program 2 Assembly Code

https://github.com/z3qi/CSE141L/blob/main/Assembly_2%20copy.s

Program 3 Pseudocode



```
LOAD R0, 1
LOAD R1, 3
PUT R2, #0100,0010
CMPr R2, R0
BGT Branch A
PUT R2, #0100 0010
CMPr R0, R2
BEQ Branch B
PUT R7, #0100,0010
PUT R6, #0000,1000
ADD R6, R7   0100,1010
CMPr R0,R6
BEQ Branch D
PUT R5, #0001,0000
ADD R6, R5   0001,1010
CMPr R0, R6
BEQ Branch F
CMPr R0, R6
BGT Branch G
PUT R5, 0000,1000
SUB R6, R5
MOV R0, R6
PUT R1, #0
B END
A: PUT R0, #0
   PUT R1, #0
   B END
B: CMPr R1, R2
   BEQ Branch C
   MOV R0, R1
   PUT R1, #0
   B END
C: PUT R2, #1
   ADD R0, R2
   PUT R1, #0
   B END
```

```
PUT_LUT
0
1
0100,0010
0000,1000
0001,0000
1000,0000
0000,0100
```

```
CMPR, r0, r1
BEQ Branch 8
PUT R3, 1
LSL R4, R3
ADD R4, R3
ADD R1, R3
ADD R1, R3
ADD R1, 6
PUT R3, 0000,0100
ADD R0, R3
```

```
D: CMPr R1, R6
   BLT Branch E
   PUT R4, #0001,0000
   ADD R6, R4
   PUT R4, #0000,0100
   ADD R6, R4
   MOV R0, R6
   ~~PUT R1, #0~~
   B END
E: PUT R7, #1
   ADD R6, R7
   MOV R0, R6
   PUT R1, #1000,0000
   B END
F: MOV R0, R6
   PUT R1, #0
   B END
G: PUT R4, 4
   ADD R4, R4
   PUT R3, 6
   ADD R4, R3
   PUT R3, 1
   ADD R4, R3
   ADD R4, R3
   ADD R4, R3
   MOV R1, R4
   B END
END: STORE R0, 13
     STORE R1, 12
```

```
PUT R1, 2
PUT R3, 1
LSR R2, R3
PUT R3, 6
ADD R1, R3
```

```
8: PUT R2, 4
   PUT R3, 1
   SUB R2, R3
   LSL R2, R3
   MOV R0, R2
   PUT R1, 6
```

```
exp.
R0 - - - - - - - -   R1 - - - - - - - -
R4 - - - - - - - -   R5 - - - - - - - -
```

```
MOV R6 R0
MOV R7 R4
LSL R6 1
LSL R7 1
LSR R6 3  } Exp.
LSR R7 3
LSL R0 6
LSL R4 6
LSR R0 2
LSR R4 2
LSR R1 4
LSR R5 4
PUT R2 0100 0000
ADD R0, R1
ADD R4, R5
ADD R0, R1  } Mant. simplified
ADD R4, R5
MOV R2 R6
MOV R3 R7
CMPr R2 R3
BLT Branch A
SUB R2, R3
LSR R4, R2
ADD R0, R4
CMP R0, 1000 0000
BGT Branch B
CMP R0, 1000 0000
BEQ Branch B
R6 -=1
R0 LSL 1
B Branch B
```

```
LOAD r1, 0
~~MOV R5, R1~~
MOV R5, r1
```

```
0  0
1  1
2  2
3  3
4  4
5  6
6  0100 0000
7  1000 0000
```

```
A: SUB R3, R2
   LSR R0, R3
   ADD R0, R4
   CMP R0, 1000 0000
   BGT Branch B
   CMP R0, 1000 0000
   BEQ Branch B
   MOV R6, R7
   R6 -=1
   R0 LSL 1
   B Branch B
B: R6 +=1
   R6 LSL 2
   MOV R1, R0
   R1 LSR 6
   R6 += R1
   R0 LSL 2
   ~~MOV R0, R1~~
   ~~MOV R1, R6~~
   STORE R0 5
   STORE R1 4
```

```
LOAD r1, 2
MOV r5, r1
```

```
PUT R5 4
LSL R5, R3
LSR R5, R3
```

```
ADD R0, R5
MOV R1, R0
MOV R0, R6
```

Program 3 Assembly Code

https://github.com/z3qi/CSE141L/blob/main/Assembly_3.s

Individual Component Specification

Top Level

Int2flt, flt2int, fltflt

Functionality Description

The Top module is a digital system that coordinates operations between multiple components: a program counter, control unit, register file, ALU, and memory interface. It uses a state machine to manage its control flow, moving through different stages like LOAD, STORE, and DONE. The module starts by loading data from memory, performs computations with the ALU, and eventually stores results back to memory. The state machine ensures synchronization among these components while handling the start and reset inputs.

Schematic

Key components of the Top module include a program counter (PC1) to manage instruction flow, a control unit (C1) for signal generation, a register file (RF1) for data storage, and an ALU (A1) for arithmetic and logic operations. The control unit coordinates interactions between these blocks based on the machine code and feedback from the ALU. The memory interface handles read and write operations, with addresses generated by the ALU and data coming from the register file. The module operates until reaching the DONE state, signaling the successful completion of all tasks.

Program Counter

Module file name: ProgCtr.sv
Module testbench file name: ProgCtr_tb.sv

Functionality Description

The ProgCtr module is a program counter that manages the current instruction address in the execution flow of a digital system. It takes in inputs like clock (Clk), reset (Reset), and a jump enable signal (Jen) along with a jump address (Jump). The program counter

(PC) is a 6-bit register that outputs the current address, which is used to fetch the corresponding machine code (Instruction) from an instruction ROM (InstROM).

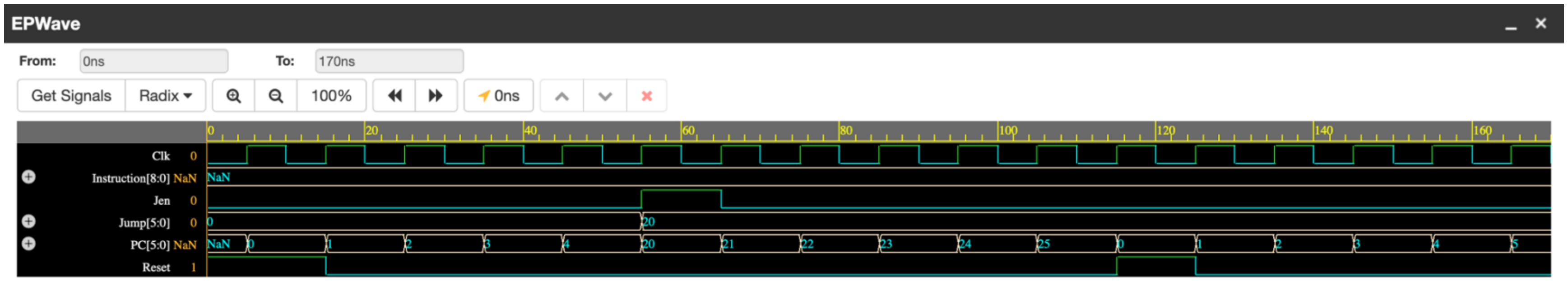
(Optional) Testbench Description

The ProgCtr_tb testbench thoroughly tests the functionality of the ProgCtr module through a series of specific test cases. First, it applies a reset (Reset signal high) to ensure that the program counter (PC) initializes to zero. After holding the reset signal for 15 ns, the testbench de-asserts Reset to allow the PC to increment sequentially with each clock cycle, verifying basic counting behavior. Next, the testbench applies a jump operation by asserting the jump enable (Jen) and setting the jump address (Jump = 20). This verifies that the PC correctly takes the given address (20) instead of continuing sequentially. Once the jump is performed, Jen is de-asserted to allow PC to resume sequential incrementing from the jump address (20 onwards).
Lastly, the reset signal is asserted again to ensure that the program counter can properly reset to zero even after a jump. The testbench ends after verifying all these behaviors. The $monitor statement is used throughout to observe changes in key signals (Reset, Jen, Jump, PC) during the simulation, while $finish ends the simulation after all tests are complete. These specific test cases ensure the module's ability to handle sequential operations, jumps, and proper resets.

Schematic

On each clock cycle, the ProgCtr module updates the program counter. If the Reset signal is active, PC is set to zero. If the jump enable (Jen) is asserted, PC is updated with the jump address (Jump). Otherwise, the program counter increments sequentially to point to the next instruction. This module ensures that the system can either proceed sequentially or jump to a different address based on control signals, allowing for efficient control flow in the overall system.

(Optional) Timing Diagram


From: 0ns To: 170ns

Instruction Memory

Module file name: InstROM.sv

Functionality Description

The InstROM module is an instruction memory that stores and provides machine code instructions for a digital system. It uses a 6-bit program counter (PC) as the address to fetch instructions from a 64-entry memory array (Core). Each instruction is represented by a 9-bit value (mach_code). During initialization, the instruction memory is loaded from an external file (mach_code.txt) using $readmemb, allowing easy modification of the instruction set. The module continuously outputs the instruction corresponding to the current PC value, ensuring the correct machine code is available for execution.

Schematic

Input (PC): The 6-bit program counter (PC) acts as an address that selects which instruction to fetch from memory.
Output (mach_code): The InstROM module outputs the 9-bit machine code based on the value of the PC.

Control Decoder

Module file name: Ctrl.sv

Functionality Description

The Ctrl module is a control unit that interprets machine code instructions (mach_code) and generates the necessary control signals for the other components of the system, such as the ALU, register file, and memory. It decodes the 9-bit machine code and determines the operation to be performed by evaluating the opcode, condition codes, and flags like Zero and Neg from the ALU. The module controls operations such as register writes, memory reads/writes, arithmetic and logic operations, and branching.

Schematic

The input machine code (mach_code) is divided into fields: the opcode (bits [8:6]), and register or immediate values. Depending on the opcode, the module sets control signals such as WenR (register write enable), WenD (memory write enable), Ldr (load enable), Str (store enable), and Jen (jump enable). The control unit also manages conditional jumps based on the status flags (Zero, Neg) and branch condition codes. This module ensures the system executes instructions correctly by providing the required signals at each step.

Register File

Module file name: RegFile.sv

Functionality Description

The RegFile module is an 8-entry register file that serves as a storage mechanism for temporary data used by the system. It allows simultaneous reading from two registers and writing to a third one, based on the given control signals. The module uses an array (Core) of eight 8-bit registers to store data, providing fast access for reading and writing operations, which are crucial for ALU operations and data handling.

Schematic

The RegFile has three main inputs: Ra, Rb, and Wd, which are 3-bit addresses for reading from register A, reading from register B, and writing to a specific register, respectively. The Wdat input holds the data to be written when the Wen (write enable) signal is asserted. On the positive edge of the clock, if Wen is high, the value in Wdat is written to the register addressed by Wd. The outputs RdatA and RdatB provide the values of the registers addressed by Ra and Rb, allowing parallel data read operations.

ALU (Arithmetic Logic Unit)

Module file name: ALU.sv
Module testbench file name: ALU_tb.sv

Functionality Description

The ALU (Arithmetic Logic Unit) module performs arithmetic and logical operations on two 8-bit inputs (DatA and DatB) based on a 3-bit operation code (Aluop). The module supports operations such as moving data, addition, subtraction, bitwise AND/OR, and left or right shifts. The result of the operation (Rslt) is also accompanied by status flags (Zero and Neg) that provide information about the output value, which can be used for decision-making in control flow.

(Optional) Testbench Description

The `ALU_tb` testbench validates the functionality of the `ALU` module by providing a sequence of operations to verify each supported ALU function. It generates a set of inputs (`Aluop`, `DatA`, `DatB`, `LSL_sel`, and `ORR_sel`) and monitors the output (`Rslt`) along with status flags (`Zero` and `Neg`). The testbench covers all the ALU operations: `MOV`, `CMP`, `ADD`, `SUB`, `NEG`, logical shifts (`LSL`, `LSR`), and bitwise operations (`AND`, `ORR`). Each operation is tested with different values for `DatA` and `DatB` to ensure the correctness of the results and the proper setting of flags (`Zero` and `Neg`). The `$monitor` statement is used to observe changes in the signals during the simulation, and `$finish` ends the simulation after all tests are complete.

ALU Operations

1. MOV (Move Data)
    - Operation Code (Aluop = 3'b000): The MOV operation copies the value of DatB to the result (Rslt).
    - Relevant Instruction: MOV is used for moving data from one register to another or from memory to a register.
2. PUT (PUT an assigned number to a register)
    - Operation Code (Aluop = 3'b001): The PUT operation put number with index n to register a.
3. ADD (Addition)
    - Operation Code (Aluop = 3'b010): The ADD operation adds DatA and DatB to produce the result (Rslt).
    - Relevant Instruction: ADD is used for performing arithmetic addition between two register values or constants.
4. SUB (Subtraction)
    - Operation Code (Aluop = 3'b011): The SUB operation subtracts DatB from DatA to produce the result (Rslt).

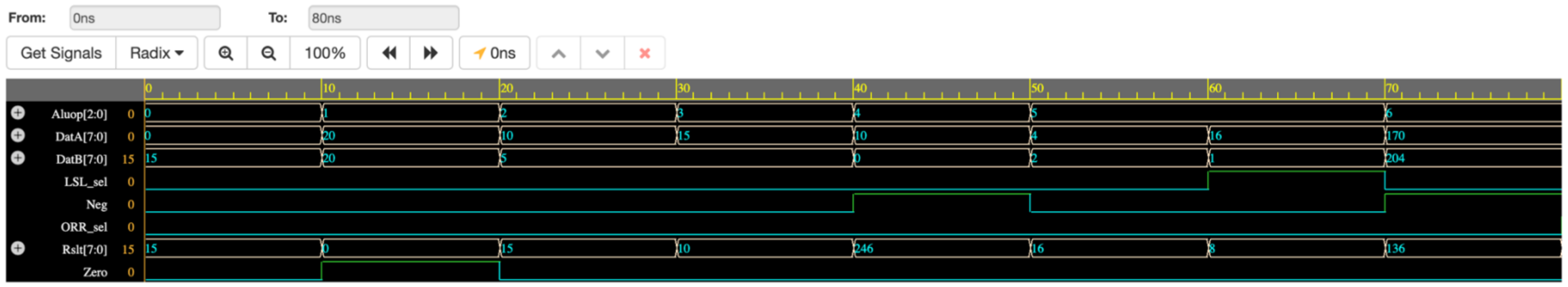
- Relevant Instruction: SUB is used for arithmetic subtraction, often to decrement counters or calculate differences.

5. FLIP (Negate) // This one is to be eliminated to reduce the number of instructions -> SUB 0 Ra
   - Operation Code (Aluop = 3'b100): The NEG operation negates DatA by computing its two's complement, effectively producing -DatA.
   - Relevant Instruction: NEG is used to negate a value, often as part of subtraction or other arithmetic operations involving negative values.
6. LSL/LSR (Logical Shift Left/Right)
   - Operation Code (Aluop = 3'b101): The LSL_sel signal determines whether DatA is shifted left (<<) or right (>>) by a number of positions defined by the lower 3 bits of DatB.
   - Relevant Instructions: LSL (Logical Shift Left) and LSR (Logical Shift Right) are used for bit manipulation, multiplication/division by powers of 2, or adjusting bit positions.
7. AND/ORR (Bitwise AND/OR)
   - Operation Code (Aluop = 3'b110): The ORR_sel signal determines whether DatA is combined with DatB using bitwise AND (&) or OR (|).
   - Relevant Instructions: AND is used for masking bits, while ORR is used for setting specific bits in a value.
8. Branching instructions

Schematic

The Aluop input selects the operation to be performed by the ALU. The supported operations include MOV (move data from DatB), CMP (compare by subtracting DatB from DatA), ADD, SUB, NEG (negate DatA), and LSL/LSR (logical shift left/right) based on the LSL_sel signal. Similarly, the module performs bitwise AND or ORR operations depending on the ORR_sel signal. The result (Rslt) is computed based on the selected operation, and the flags (Zero and Neg) are set accordingly to indicate whether the result is zero or negative.

(Optional) Timing Diagram



Data Memory

Module file name: data_mem.sv

Functionality Description

The data_mem module is an 8-bit wide, parameterizable-depth data memory used for storing and retrieving data in the system. It is designed to support both read and write operations based on control signals (ReadMem and WriteMem). The memory depth is determined by the parameter AW, which defines the number of address bits, allowing for flexibility in the size of the memory. The module includes a memory core (mem_core) consisting of 2**AW 8-bit elements, with a default depth of 256.
The data_mem module reads and writes data based on the clock signal (clk) and the respective control signals. Reads (ReadMem) are combinational, meaning the output (DataOut) reflects the value at the given address (DataAddress) immediately when ReadMem is enabled. Writes (WriteMem) are clocked operations that occur on the rising edge of the clock, storing the input data (DataIn) at the specified address. The module provides a high-impedance output (DataOut) when not reading, ensuring it does not drive any unintended values. This memory module can optionally be initialized with predefined values, making it adaptable to different project requirements.

Schematic

- Memory Core (mem_core): Contains 2**AW elements of 8 bits each, used for data storage.
- Read Logic: Controlled by ReadMem, providing data from the address pointed to by DataAddress.
- Write Logic: Controlled by WriteMem, writing data (DataIn) to the address (DataAddress) on the rising edge of clk.

- Control Signals: ReadMem and WriteMem determine whether the module is performing a read or write operation, respectively.

Lookup Tables

Module file name: branching_LUT, put_LUT, load_store_LUT

Functionality Description

These modules are designed to reduce the bit width it takes to represent a complicated number/address.

7. Software – provide as attachments

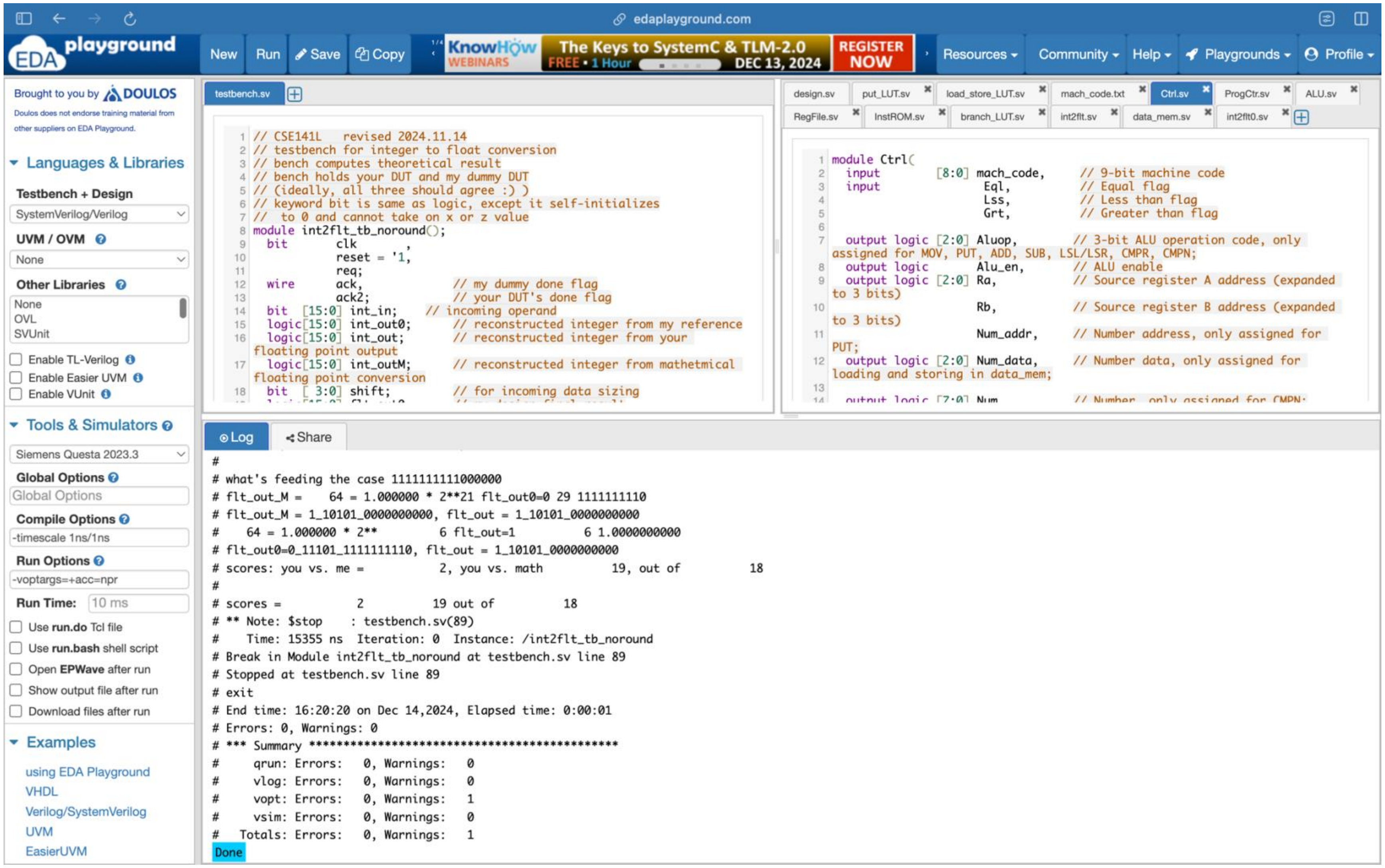
https://github.com/z3qi/CSE141L/tree/main

8. Changelog

- Milestone 3
    - Introduction
        - Adjusted hardware and architectural design to meet additional software requirements, including the integration of branching and memory operations.
        - Documented a catalog of implemented programs (Assembly1, Assembly_2, and Assembly_3) in the GitHub repository.
        - Enhanced instruction set encoding to improve compatibility with assembly programs.
    - Machine Specification
        - Added support for indirect addressing through memory lookup tables (e.g., branching_LUT, load_store_LUT).
        - Improved branching mechanism with LUT optimization for faster lookups.
        - Refined the handling of jump instructions, including removing unused instructions like BGE.
    - ALU (Arithmetic Logic Unit)
        - Completed testbench development for ALU operations (ALU_tb) with extensive validation for edge cases.

        - Verified flag-setting logic for Zero and Negative conditions.
    - Instruction Memory and Data Memory
        - Finalized the implementation of instruction and data memory modules with dynamic loading from external files.
        - Validated read/write functionality through simulation and test cases.
- Milestone 2
    - Introduction
        - Updated the architectural classification from general-purpose to register-register/load-store hybrid for clarity and efficiency.
        - Defined the goal of streamlining instruction formats while maintaining flexibility for essential operations.
    - Architectural Overview
        - Added a detailed schematic highlighting component interactions (e.g., between the Program Counter, Control Unit, and ALU).
        - Clarified data flow for branching and memory access in the overview.
    - Machine Specification
        - Refined instruction formats and provided bit-level breakdowns for operations such as LSL, LSR, CMP, and branching instructions.
        - Introduced a Branching LUT to handle conditional and unconditional branches efficiently.
        - Eliminated complex instructions (e.g., BGE) to simplify control unit design.
    - Register File
        - Specified the roles of all 8 registers, highlighting special-purpose registers (R0, R1, R7) for memory and sign handling.
- Milestone 1
    - Initial Version
        - Defined the foundational architecture as register-register/load-store.
        - Proposed a 9-bit machine code format for instructions.
        - Outlined basic instruction formats (MOV, ADD, SUB, etc.) and placeholder operations.
        - Established preliminary branching logic and memory addressing mechanisms.

Outputs:

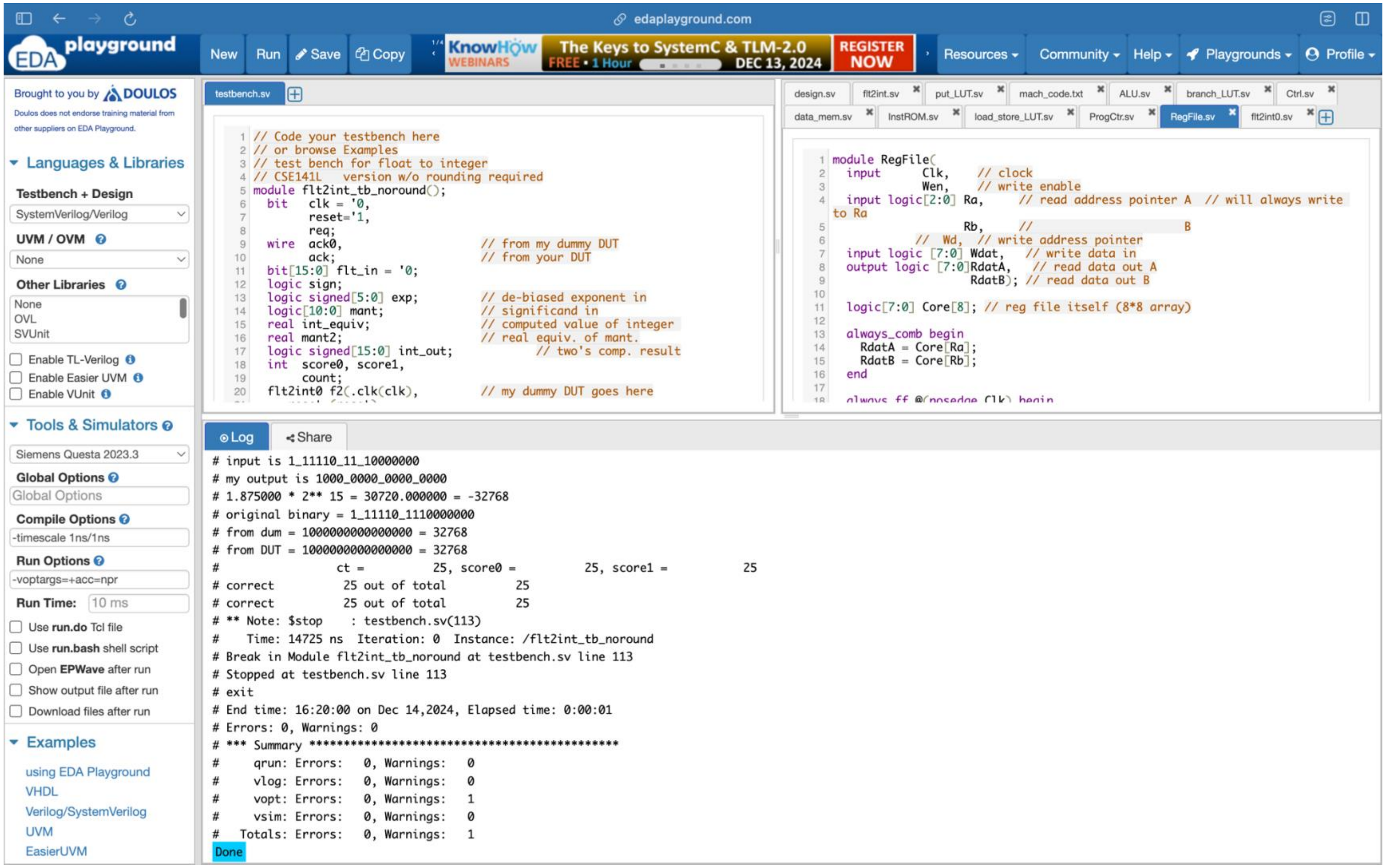


Testbench + Design: SystemVerilog/Verilog
UVM / OVM: None
Other Libraries: None, OVL, SVUnit
Enable TL-Verilog
Enable Easier UVM
Enable VUnit
Siemens Questa 2023.3
Global Options: Global Options
Compile Options: -timescale 1ns/1ns
Run Options: -voptargs=+acc=npr
Run Time: 10 ms
Use **run.do** Tcl file
Use **run.bash** shell script
Open **EPWave** after run
Show output file after run
Download files after run

```
// CSE141L   revised 2024.11.14
// testbench for integer to float conversion
// bench computes theoretical result
// bench holds your DUT and my dummy DUT
// (ideally, all three should agree :) )
// keyword bit is same as logic, except it self-initializes
//  to 0 and cannot take on x or z value
module int2flt_tb_noround();
  bit        clk        ,
             reset = '1,
             req;
  wire       ack,              // my dummy done flag
             ack2;             // your DUT's done flag
  bit  [15:0] int_in;    // incoming operand
  logic[15:0] int_out0;        // reconstructed integer from my reference
  logic[15:0] int_out;         // reconstructed integer from your
floating point output
  logic[15:0] int_outM;        // reconstructed integer from mathetmical
floating point conversion
  bit  [ 3:0] shift;           // for incoming data sizing
```

```
module Ctrl(
  input          [8:0] mach_code,     // 9-bit machine code
  input                 Eql,          // Equal flag
                        Lss,          // Less than flag
                        Grt,          // Greater than flag

  output logic [2:0] Aluop,          // 3-bit ALU operation code, only
assigned for MOV, PUT, ADD, SUB, LSL/LSR, CMPR, CMPN;
  output logic       Alu_en,         // ALU enable
  output logic [2:0] Ra,             // Source register A address (expanded
to 3 bits)
                     Rb,             // Source register B address (expanded
to 3 bits)
                     Num_addr,       // Number address, only assigned for
PUT;
  output logic [2:0] Num_data,       // Number data, only assigned for
loading and storing in data_mem;

  output logic [7:0] Num             // Number, only assigned for CMPN;
```

```
#
# what's feeding the case 1111111111000000
# flt_out_M =    64 = 1.000000 * 2**21 flt_out0=0 29 1111111110
# flt_out_M = 1_10101_0000000000, flt_out = 1_10101_0000000000
#    64 = 1.000000 * 2**          6 flt_out=1          6 1.0000000000
# flt_out0=0_11101_1111111110, flt_out = 1_10101_0000000000
# scores: you vs. me =          2, you vs. math         19, out of          18
#
# scores =          2          19 out of          18
# ** Note: $stop    : testbench.sv(89)
#    Time: 15355 ns  Iteration: 0  Instance: /int2flt_tb_noround
# Break in Module int2flt_tb_noround at testbench.sv line 89
# Stopped at testbench.sv line 89
# exit
# End time: 16:20:20 on Dec 14,2024, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
# *** Summary *********************************************
#     qrun: Errors:   0, Warnings:   0
#     vlog: Errors:   0, Warnings:   0
#     vopt: Errors:   0, Warnings:   1
#     vsim: Errors:   0, Warnings:   0
#   Totals: Errors:   0, Warnings:   1
Done
```



Testbench + Design: SystemVerilog/Verilog

UVM / OVM: None

Other Libraries: None, OVL, SVUnit

- [ ] Enable TL-Verilog
- [ ] Enable Easier UVM
- [ ] Enable VUnit

Siemens Questa 2023.3

Global Options: Global Options

Compile Options: -timescale 1ns/1ns

Run Options: -voptargs=+acc=npr

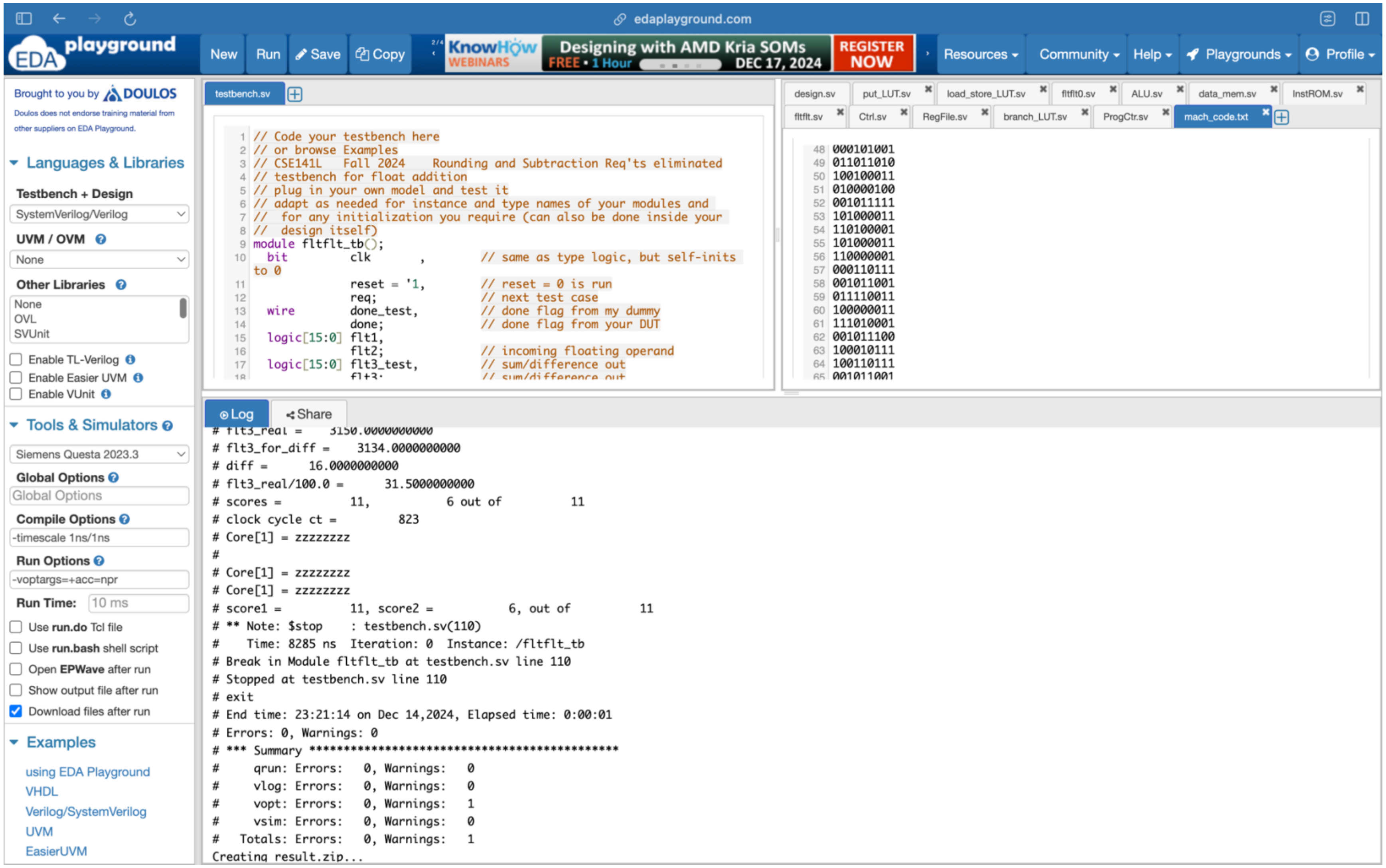
Run Time: 10 ms

- [ ] Use **run.do** Tcl file
- [ ] Use **run.bash** shell script
- [ ] Open **EPWave** after run
- [ ] Show output file after run
- [ ] Download files after run

```
// Code your testbench here
// or browse Examples
// test bench for float to integer
// CSE141L   version w/o rounding required
module flt2int_tb_noround();
  bit   clk = '0,
        reset='1,
        req;
  wire  ack0,                        // from my dummy DUT
        ack;                         // from your DUT
  bit[15:0] flt_in = '0;
  logic sign;
  logic signed[5:0] exp;             // de-biased exponent in
  logic[10:0] mant;                  // significand in
  real int_equiv;                    // computed value of integer
  real mant2;                        // real equiv. of mant.
  logic signed[15:0] int_out;                // two's comp. result
  int  score0, score1,
       count;
  flt2int0 f2(.clk(clk),             // my dummy DUT goes here
```

```
module RegFile(
  input        Clk,     // clock
               Wen,     // write enable
  input logic[2:0] Ra,      // read address pointer A  // will always write
to Ra
                   Rb,      //                          B
           //  Wd,  // write address pointer
  input logic [7:0] Wdat,   // write data in
  output logic [7:0]RdatA,   // read data out A
                    RdatB); // read data out B

  logic[7:0] Core[8]; // reg file itself (8*8 array)

  always_comb begin
    RdatA = Core[Ra];
    RdatB = Core[Rb];
  end

  always_ff @(posedge Clk) begin
```

```
# input is 1_11110_11_10000000
# my output is 1000_0000_0000_0000
# 1.875000 * 2** 15 = 30720.000000 = -32768
# original binary = 1_11110_1110000000
# from dum = 1000000000000000 = 32768
# from DUT = 1000000000000000 = 32768
#                  ct =          25, score0 =          25, score1 =          25
# correct          25 out of total          25
# correct          25 out of total          25
# ** Note: $stop    : testbench.sv(113)
#    Time: 14725 ns  Iteration: 0  Instance: /flt2int_tb_noround
# Break in Module flt2int_tb_noround at testbench.sv line 113
# Stopped at testbench.sv line 113
# exit
# End time: 16:20:00 on Dec 14,2024, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
# *** Summary *********************************************
#     qrun: Errors:   0, Warnings:   0
#     vlog: Errors:   0, Warnings:   0
#     vopt: Errors:   0, Warnings:   1
#     vsim: Errors:   0, Warnings:   0
#   Totals: Errors:   0, Warnings:   1
Done
```



Languages & Libraries

Testbench + Design: SystemVerilog/Verilog

UVM / OVM: None

Other Libraries: None, OVL, SVUnit

☐ Enable TL-Verilog
☐ Enable Easier UVM
☐ Enable VUnit

Tools & Simulators: Siemens Questa 2023.3

Global Options: Global Options

Compile Options: -timescale 1ns/1ns

Run Options: -voptargs=+acc=npr

Run Time: 10 ms

☐ Use run.do Tcl file
☐ Use run.bash shell script
☐ Open EPWave after run
☐ Show output file after run
☑ Download files after run

testbench.sv

```
// Code your testbench here
// or browse Examples
// CSE141L   Fall 2024    Rounding and Subtraction Req'ts eliminated
// testbench for float addition
// plug in your own model and test it
// adapt as needed for instance and type names of your modules and
//  for any initialization you require (can also be done inside your
//  design itself)
module fltflt_tb();
  bit           clk        ,         // same as type logic, but self-inits
to 0
                reset = '1,          // reset = 0 is run
                req;                 // next test case
  wire          done_test,           // done flag from my dummy
                done;                // done flag from your DUT
  logic[15:0] flt1,
                flt2;                // incoming floating operand
  logic[15:0] flt3_test,             // sum/difference out
                flt3;                // sum/difference out
```

mach_code.txt

```
000101001
011011010
100100011
010000100
001011111
101000011
110100001
101000011
110000001
000110111
001011001
011110011
100000011
111010001
001011100
100010111
100110111
001011001
```

Log

```
# flt3_real =    3150.0000000000
# flt3_for_diff =    3134.0000000000
# diff =      16.0000000000
# flt3_real/100.0 =      31.5000000000
# scores =          11,            6 out of          11
# clock cycle ct =         823
# Core[1] = zzzzzzzz
#
# Core[1] = zzzzzzzz
# Core[1] = zzzzzzzz
# score1 =          11, score2 =           6, out of          11
# ** Note: $stop    : testbench.sv(110)
#    Time: 8285 ns  Iteration: 0  Instance: /fltflt_tb
# Break in Module fltflt_tb at testbench.sv line 110
# Stopped at testbench.sv line 110
# exit
# End time: 23:21:14 on Dec 14,2024, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
# *** Summary *********************************************
#     qrun: Errors:   0, Warnings:   0
#     vlog: Errors:   0, Warnings:   0
#     vopt: Errors:   0, Warnings:   1
#     vsim: Errors:   0, Warnings:   0
#   Totals: Errors:   0, Warnings:   1
Creating result.zip...
```