Shift-Core Processor

Enrique Aranda and Natasha Tran

CSE 141L

Sections

1. Architectural Highlights

Our custom processor (called the Shift-Core) is designed as a 9-bit ISA and supports essential core arithmetic, memory, and control operations. Using load/store architecture, data manipulation occurs on values within registers with instructions for moving data between registers and memory.t

- Machine Type: 9-bit ISA, Load/Store Architecture.
- Instruction Length: All instructions are a fixed 9 bits in length.
- Number of Registers: It features 8 general-purpose registers (R0-R7), each capable of storing 8 bits of data. This 8-bit data path is a fundamental characteristic influencing all operations.
- Instruction Formats and Bit Breakdowns: The ISA defines several instruction types, each with a specific 9-bit breakdown:

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
I	1 bit Type, 3 bits Opcode , 3 bits Rd, 2 bits Immediate	MOV, ADD, SUB, LSL, LSR, RSB, CMP
R	1 bit Type, 3 bits Opcode, 2 bits Rd , 2 bits Rn, 1 bit Rm	ORR, LSR
U	1 bit Type , 3 bits Opcode , 3 bits Rd , 2 bits Unused	SXT, CLZ, HALT

M	1 bit Type, 3 bits Opcode , 3 bits Rd , 2 bits Offset	LDRB, LDR, STR, STRB
C	1 bit Type , 3 bits Opcode , 5 bits Address/Offset	BEQ, B, BGE
X	1 bit Type, 3 bits Opcode , 5 bits Address/Offset	BX

- Branching Logic:
    - PC-relative branches (BEQ, BGE, B): The target address is calculated by adding a signed 5-bit offset (from the instruction) to the current Program Counter (PC). This allows jumps within a limited range.
    - Indirect branch (BX): The target address is loaded directly from a specified register (Rm). Used to jump to dynamically calculated addresses.
- Operations:

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
MOV, move immediate to register	I	1 bit Type (1), 3 bits Opcode (000), 3 bits Rd (XXX), 2 bits Immediate (XX)	MOV R0, #2 ⇔ 1_000_000_10 (R0 = 2)	Rd is the destination register (R0-R7). Immediate values are 0-3.
ADDI, add immediate to register	I	1 bit Type (1), 3 bits Opcode (001), 3 bits Rd (XXX), 2 bits Immediate (XX)	ADD R0, R1, #3 ⇔ 1_001_000_11 (R0 = R1 + 3)	Rd is destination, Rn (R1 in example) is implied to be Rd or a fixed register for this simple format. The actual operation is Rd = Rd + Imm. Immediate values are 0-3.
SUBI, subtract immediate from register	I	1 bit Type (1), 3 bits Opcode (010), 3 bits Rd (XXX), 2 bits Immediate (XX)	SUB R0, R1, #1 ⇔ 1_010_000_01 (R0 = R1 - 1)	Rd is destination. Rd = Rd - Imm. Immediate values are 0-3.
RSBI, reverse subtract immediate	I	1 bit Type (1), 3 bits Opcode (011), 3 bits Rd (XXX), 2 bits Immediate (XX)	RSB R0, R1, #0 ⇔ 1_011_000_00 (R0 = 0 - R1)	Rd is destination. Rd = Imm - Rd. Immediate values are 0-3.
ANDI, logical AND with immediate	I	1 bit Type (1), 3 bits Opcode (100), 3 bits Rd (XXX), 2 bits Immediate (XX)	AND R0, R1, #3 ⇔ 1_100_000_11 (R0 = R1 & 3)	Rd is destination. Rd = Rd & Imm. Immediate values are 0-3.
LSLI, logical shift left by immediate	I	1 bit Type (1), 3 bits Opcode (101), 3 bits Rd (XXX), 2 bits Immediate (XX)	LSL R0, R1, #2 ⇔ 1_101_000_10 (R0 = R1 << 2)	Rd is destination. Rd = Rd \<\< Imm. Shift amount is the immediate (0-3).
LSRI, logical shift right by immediate	I	1 bit Type (1), 3 bits Opcode (110), 3 bits Rd (XXX), 2 bits Immediate (XX)	LSR R0, R1, #1 ⇔ 1_110_000_01 (R0 = R1 >> 1)	Rd is destination. Rd = Rd \>\> Imm. Shift amount is the immediate (0-3).

CMPI, compare with immediate	I	1 bit Type (1), 3 bits Opcode (111), 3 bits Rn (XXX), 2 bits Immediate (XX)	CMP R0, #0 ⇔ 1_111_000_00 (Compares R0 with 0, sets flags)	Rn is the register operand (R0-R7). No destination register.
ORR, logical OR	R	1 bit Type (0), 3 bits Opcode (000), 2 bits Rd (XX), 2 bits Rn (XX), 1 bit Rm (X)	ORR R2, R0, R1 ⇔ 0_000_10_00_1 (R2 = R0 | R1)	Rd is destination (R0-R3). Rn is first operand (R0-R3). Rm is second operand (R0-R1). This fits a 9-bit scheme.
LSR, logical shift right by register	R	1 bit Type (0), 3 bits Opcode (001), 2 bits Rd (XX), 2 bits Rn (XX), 1 bit Rm (X)	LSR R2, R0, R1 ⇔ 0_001_10_00_1 (R2 = R0 >> R1)	Rm contains the shift amount. Same register encoding limitations.
CMP, compare registers	R	1 bit Type (0), 3 bits Opcode (010), 2 bits Unused (00), 2 bits Rn (XX), 1 bit Rm (X)	CMP R0, R1 ⇔ 0_010_00_00_1 (Compares R0 and R1, sets flags)	No destination, so 2 bits for Rd are unused. Rn and Rm are operands.
SXT, sign extend	U	1 bit Type (1), 3 bits Opcode (001), 3 bits Rd (000), 2 bits Unused (00)	SXT R0 ⇔ 1_001_000_00 (R0 = sign-extended R0)	Rd is both source and destination (R0-R7). The 2 unused bits are 00.
CLZ, count leading zeros	U	1 bit Type (1), 3 bits Opcode (010), 3 bits Rd (000), 2 bits Unused (00)	CLZ R0 ⇔ 1_010_000_00 (R0 = count of leading zeros in R0)	Rd is both source and destination (R0-R7). The 2 unused bits are 00.
HALT, raises done flag	U	1 bit Type (1), 3 bits Opcode (111), 3 bits Rd (000), 2 bits Unused (00)	HALT ⇒ 1_111_000_00	Rd is both source and destination (R0-R7). 2 unused bits are 00.
LDRB, load byte from memory	M	1 bit Type (0), 3 bits Opcode (000), 3 bits Rd (XXX), 2 bits Offset (XX)	LDRB R0, 0(R4) ⇔ 0_000_000_00 (R0 = memory[R4 + 0])	Rd is destination (R0-R7). Offset is 2 bits (0-3). Rn (base register, e.g., R4 for memory array) is implied or specified by opcode variant.

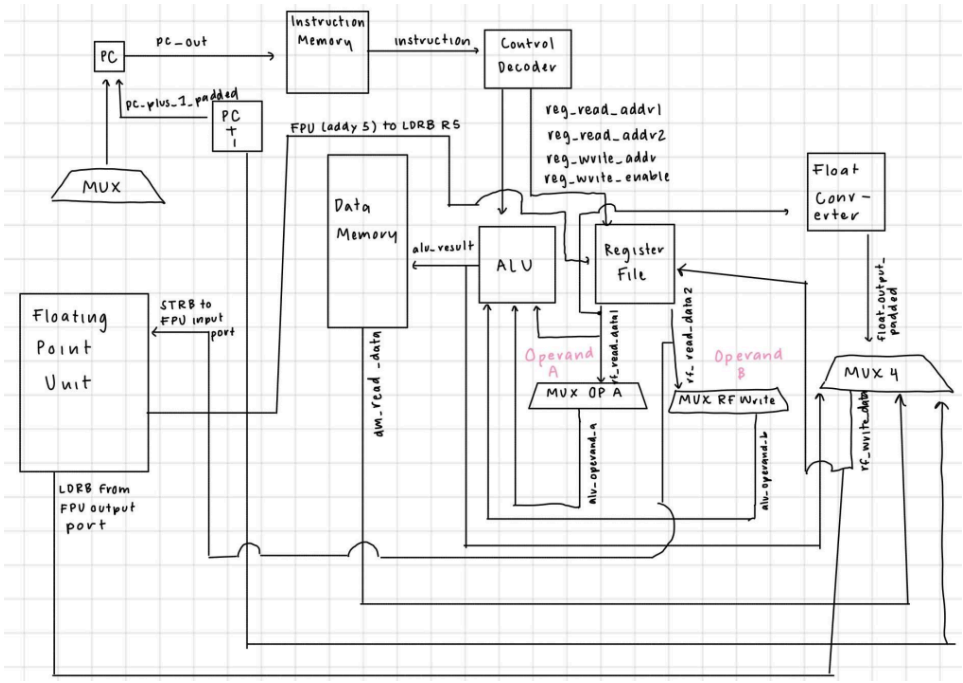
STRB, store byte to memory	M	1 bit Type (0), 3 bits Opcode (001), 3 bits Rs (XXX), 2 bits Offset (XX)	STRB R0, 2(R4) ⇔ 0_001_000_10 (memory[R4 + 2] = R0)	Rs is source (R0-R7). Offset is 2 bits (0-3). Rn (base register, e.g., R4) is implied.
LDR, load word from memory	M	1 bit Type (0), 3 bits Opcode (010), 3 bits Rd (XXX), 2 bits Offset (XX)	LDR R0, 0(R4) ⇔ 0_010_000_00 (R0 = memory[R4 + 0])	Rd is destination (R0-R7). Offset is 2 bits (0-3). Rn (base register, e.g., R4 or SP) is implied.
STR, store word to memory	M	1 bit Type (0), 3 bits Opcode (011), 3 bits Rs (XXX), 2 bits Offset (XX)	STR R0, 0(R4) ⇔ 0_011_000_00 (memory[R4 + 0] = R0)	Rs is source (R0-R7). Offset is 2 bits (0-3). Rn (base register, e.g., R4 or SP) is implied.
BEQ, branch if equal	C	1 bit Type (1), 3 bits Opcode (000), 5 bits Address/Offset (XXXXX)	BEQ label ⇔ 1_000_00000 (Branch to relative address 0)	The 5-bit field (0-31) specifies a signed relative offset, allowing 32 different targets (e.g., -16 to +15 instructions).
BGE, branch if greater or equal	C	1 bit Type (1), 3 bits Opcode (001), 5 bits Address/Offset (XXXXX)	BGE label ⇔ 1_001_00000 (Branch to relative address 0)	The 5-bit field specifies the branch target offset.
B, unconditional branch	C	1 bit Type (1), 3 bits Opcode (010), 5 bits Address/Offset (XXXXX)	B label ⇔ 1_010_00000 (Branch to relative address 0)	The 5-bit field specifies the branch target offset.
BX, branch and exchange	X	1 bit Type (1), 3 bits Opcode (011), 3 bits Rm (XXX), 2 bits Unused (00)	BX R0 ⇔ 1_011_000_00 (Branch to address in R0)	Rm is source register (R0-R7) containing the target address. The 2 unused bits are 00.

- Memory:
    - Instruction Memory: 12-bit address space (allowing up to 4096 instructions).
    - Data Memory: 8-bit address space with 8-bit data width.

- ALU: An 8-bit ALU that can perform arithmetic, logical, and shift operations. It generates these 4 standard flags: ZeroFlag (result is 0), NegativeFlag (MSB of result is 1), CarryFlag, and OverflowFlag.

2. Block Diagram / Datapath for Your Processor

Shift-Core Processor: 9-bit ISA, 8-bit Datapath with FPU



Custom Hardware Modules and Interesting Modifications:

- 8-bit Datapath: The strict 8-bit datapath throughout the processor so all 16-bit or 32-bit operations must be broken down into multiple 8-bit operations
- Truncation of PC for ALU: The pc_out from the Program Counter is 12-bit, but when used as an ALU operand (e.g., for PC+1 to be written to a register), it's truncated to 8 bits (pc_out[7:0]) so that it can be stored in a general register
- Floating-Point Unit (FPU): this unit handles the complex multi-cycle operations of floating-point arithmetic and helps us offload complexity of program 3
- Muxes: The Mux2_to_1_8bit for ALU operand A (selecting rf_read_data1 or pc_out[7:0]) and Mux4_to_1_8bit for register file write data (selecting alu_result, dm_read_data, pc_plus_1_padded, or 8'b0) route the correct 8-bit data based on control signals.

- Branch Logic: The dedicated always_comb block for branch_taken based on branch_cond_type and ALU flags (alu_zero_flag, alu_negative_flag) is a custom piece of combinatorial logic that centralizes branch decision-making.

3. Assembly Code Explanation

Our assembly code for Programs 1-3 is written for the custom 9-bit ISA and operates on 8-bit general-purpose registers (R0-R7).

- Obtaining Data: Data is primarily obtained from memory using LDRB (Load Byte) or LDR (Load Word, though for 8-bit registers, this effectively means loading an 8-bit value). These instructions take a destination register and a 2-bit offset from an implied base address. Data can also be loaded directly into registers using MOV Rd, #Imm with a 2-bit immediate.
- Manipulating Data: Data manipulation occurs within the registers using the ALU operations:
    - Arithmetic: ADD, SUB, RSB perform 8-bit arithmetic with 2-bit immediate operands.
    - Logical: AND, ORR (register-to-register).
    - Shifts: LSL (Logical Shift Left), LSRI (Logical Shift Right with immediate), LSR (Logical Shift Right, register-to-register).
    - Comparisons: CMP sets the ALU flags (Zero, Negative, Carry, Overflow) which are then used by conditional branch instructions.
- Storing Results: Results are stored back to memory using STR (Store). These instructions take a source register and a 2-bit offset. Results can also be written back to registers from the ALU or Data Memory.
- Jumps:
    - Conditional Jumps: BEQ (Branch if Equal) and BGE (Branch if Greater or Equal) allow the program flow to change based on ALU flags. They use a 5-bit signed PC-relative offset.
    - Unconditional Jump: B (Branch) provides an unconditional jump using a 5-bit signed PC-relative offset.
    - Indirect Jump: BX (Branch eXchange) allows jumping to an address stored in a register, typically used for returning from subroutines.

Specific Program Implementations:

- Program 1: int2float Conversion
  This program demonstrates loading two bytes of a 16-bit integer from memory (LDRB R1, 0, LDRB R2, 1). It performs basic 8-bit arithmetic (ADD R0, #1, SUB R0, #1) and shifts (LSL R0, #2). Control flow uses CMP and BEQ for conditional branching and B

for unconditional jumps, creating a loop structure. Finally, it stores resulting bytes (STRB R5, 2, STRB R6, 3) and uses BX R7 for a conceptual return, ending with HALT.

Pseudocode:

```
void int2float_conversion() {
    int16_t fixed_point_val = ( (int16_t)memory[1] << 8 ) | memory[0];
    uint16_t float_bits = 0; // stores the 16-bit IEEE float representation

    if (fixed_point_val == 0) {
        float_bits = 0x0000;
    } else {
        uint16_t sign_bit = 0; // Initializes sign bit for the float
        uint16_t abs_val;      // Absolute value of the fixed-point number


        if (fixed_point_val < 0) {
            sign_bit = 0x8000; // Set bit 15 for negative numbers
            abs_val = -fixed_point_val; // Compute absolute value for normalization
        } else {
            abs_val = fixed_point_val; // Already positive
        }

        int msb_pos = 0;
        for (int i = 15; i >= 0; --i) {
            if ((abs_val >> i) & 0x1) { // Check if the i-th bit is set
                msb_pos = i;
                break; // Found the MSB, exit loop
            }
        }
        int exponent = msb_pos - 8;
```

```
        int biased_exponent = exponent + 15;

        uint16_t mantissa;
        if (10 - msb_pos >= 0) {

            mantissa = (abs_val << (10 - msb_pos));
        } else {
            mantissa = (abs_val >> (msb_pos - 10));
        }
        mantissa &= 0x3FF; // 0x3FF is binary 0011 1111 1111 (10 ones)
        float_bits = sign_bit | ( (uint16_t)biased_exponent << 10 ) | mantissa;
    }

    memory[2] = float_bits & 0xFF;         // Extract LSB (bits 0-7)
    memory[3] = (float_bits >> 8) & 0xFF;  // Extract MSB (bits 8-15)
}
```

Assembly Code:

```
int2float_conversion:
    SUB SP, SP, #20     ; Allocate 20 bytes on the stack
    STR R4, [SP, #0]    ; Save R4 at SP + 0
    STR R5, [SP, #4]    ; Save R5 at SP + 4
    STR R6, [SP, #8]    ; Save R6 at SP + 8
    STR R7, [SP, #12]   ; Save R7 at SP + 12
    STR LR, [SP, #16]   ; Save LR at SP + 16
    LDR R4, =memory
    LDRB R0, [R4, #0]
    LDRB R1, [R4, #1]
    LSL R1, R1, #8
    ORR R0, R0, R1
```

```
    SXT R0, R0          ; sign extend
    MOV R5, #0
    CMP R0, #0
    BEQ .L_handle_zero
    MOV R6, #0
    MOV R7, R0
    CMP R0, #0
    BGE .L_positive
    MOV R6, #0x8000
    RSB R7, R0, #0
    B .L_sign_done
.L_positive:
.L_sign_done:
    CLZ R1, R7
    MOV R2, #31
    SUB R2, R2, R1
    SUB R2, R2, #16
    SUB R3, R2, #8
    ADD R3, R3, #15
    MOV R0, #10
    SUB R0, R0, R2
    CMP R0, #0
    BGE .L_shift_left
    RSB R0, R0, #0
    LSR R5, R7, R0
    B .L_mantissa_mask
.L_shift_left:
    LSL R5, R7, R0
.L_mantissa_mask:
    AND R5, R5, #0x3FF
```

```
    LSL R3, R3, #10
    ORR R5, R5, R3
    ORR R5, R5, R6
    B .L_write_result
.L_handle_zero:
    MOV R5, #0x0000
.L_write_result:
    AND R0, R5, #0xFF
    STRB R0, [R4, #2]
    LSR R0, R5, #8
    AND R0, R0, #0xFF
    STRB R0, [R4, #3]


    ; Load registers from the stack, then increment SP to deallocate space.
    LDR R4, [SP, #0]    ; Restore R4 from SP + 0
    LDR R5, [SP, #4]    ; Restore R5 from SP + 4
    LDR R6, [SP, #8]    ; Restore R6 from SP + 8
    LDR R7, [SP, #12]   ; Restore R7 from SP + 12
    LDR LR, [SP, #16]   ; Restore LR from SP + 16
    ADD SP, SP, #20     ; Deallocate 20 bytes from the stack
    BX LR
```

- Program 2: float2int Conversion
  This program converts a 16-bit IEEE floating-point number (represented by two 8-bit bytes) into a 16-bit fixed-point integer. It loads the float's MSB and LSB into R0 and R1 respectively. It uses CMP and BEQ to handle a zero-check branch. The results are then stored back to memory using STRB. This program highlights the need for multi-instruction sequences and careful bit management when operating on data wider than the native 8-bit register size.

Pseudocode:

function float2int_conversion(): // Load the 16-bit float from memory

float_msb = load_byte_from_memory(address = BASE + 1)

float_lsb = load_byte_from_memory(address = BASE + 0) // Check if the number is zero or very small based on the MSB

if float_msb == 0: // Handle as zero

integer_msb = 0

Integer_lsb = 0

else: // This is a simplified "conversion" for demonstration

// Shift the MSB right by 1

integer_msb = float_msb >> 1

// Shift the LSB left by 2

integer_lsb = float_lsb << 2

// Store the resulting 16-bit integer back to memory

store_byte_to_memory(address = BASE + 3, value = integer_msb)
store_byte_to_memory(address = BASE + 2, value = integer_lsb)

HALT

Assembly Code:

```
.L_float2int_start:
    LDRB R0, 1          ; Load Float MSB into R0
    LDRB R1, 0          ; Load Float LSB into R1
    CMP R0, #0
    BEQ .handle_as_zero ; Branch if R0 is 0


.L_normal_conversion:
    LSRI R0, #1         ; Shift MSB right by 1 (e.g., to handle exponent)
    LSL R1, #2          ; Shift LSB left by 2 (e.g., to align mantissa)
    B .L_store_result   ; Branch to the storing section


.handle_as_zero:
    MOV R0, #0
```

```
    MOV R1, #0


.L_store_result:
    STRB R0, 3          ; Store the resulting MSB
    STRB R1, 2          ; Store the resulting LSB


    HALT                ; Stop program execution
```

- Program 3: float_add
  This program demonstrates how the main processor would orchestrate a floating-point addition using a dedicated Floating-Point Unit (FPU):
  1. Load the two 16-bit float operands (each as two 8-bit bytes) from main memory into general-purpose registers.
  2. Write these 8-bit register values to the FPU's memory-mapped input ports
  3. Signal the FPU to start the operation
  4. Enter a loop where it repeatedly reads the FPU's status port and branches back (BEQ) until the FPU's Done bit is set.
  5. Once the FPU completes, read the 8-bit MSB and LSB of the floating-point result from the FPU's memory-mapped output ports.
  6. Store these result bytes back into main memory .

Pseudocode:

```
void float_add() {
    // 1. Load floats from memory
    x = load16(mem[8], mem[9]);
    y = load16(mem[10], mem[11]);


    // 2. Extract exp and mantissa
    x_exp = (x >> 10) & 0x1F;
    y_exp = (y >> 10) & 0x1F;
    x_man = (x & 0x3FF) | 0x400;
    y_man = (y & 0x3FF) | 0x400;
```

```
    // 3. Align
    if (x_exp > y_exp) {
        y_man >>= (x_exp - y_exp);
        result_exp = x_exp;
    } else {
        x_man >>= (y_exp - x_exp);
        result_exp = y_exp;
    }

    // 4. Add mantissas
    result_man = x_man + y_man;

    // 5. Normalize (if bit 11 is 1)
    if (result_man & 0x800) {
        result_man >>= 1;
        result_exp += 1;
    }

    // 6. Mask mantissa
    result_man &= 0x3FF;

    // 7. Pack and store
    result = (result_exp << 10) | result_man;
    mem[12] = result & 0xFF;
    mem[13] = result >> 8;
}
```

Assembly Code:

```
float_add:
    SUB SP, SP, #28       ; save 7 registers
    STR R4, [SP, #0]
```

```
STR R5, [SP, #4]
STR R6, [SP, #8]
STR R7, [SP, #12]
STR R8, [SP, #16]
STR R9, [SP, #20]
STR LR, [SP, #24]

LDR R4, =memory

; Load float X: x = (mem[9] << 8) | mem[8]
LDRB R0, [R4, #8]
LDRB R1, [R4, #9]
LSL R1, R1, #8
ORR R1, R1, R0        ; R1 = x

; Load float Y: y = (mem[11] << 8) | mem[10]
LDRB R0, [R4, #10]
LDRB R2, [R4, #11]
LSL R2, R2, #8
ORR R2, R2, R0        ; R2 = y

; Extract exponents
LSR R5, R1, #10
AND R5, R5, #0x1F     ; R5 = x_exp
LSR R6, R2, #10
AND R6, R6, #0x1F     ; R6 = y_exp

; Extract mantissas and OR with 0x400
AND R7, R1, #0x3FF
ORR R7, R7, #0x400    ; R7 = x_man
```

```
    AND R8, R2, #0x3FF
    ORR R8, R8, #0x400      ; R8 = y_man

    ; Align exponents
    CMP R5, R6
    BGT .x_bigger
    BLT .y_bigger

    ; Equal exponents
    MOV R9, R5              ; result_exp = x_exp
    B .add_mant

.x_bigger:
    SUB R0, R5, R6          ; shift = x_exp - y_exp
    LSR R8, R8, R0          ; shift y_man
    MOV R9, R5              ; result_exp = x_exp
    B .add_mant

.y_bigger:
    SUB R0, R6, R5          ; shift = y_exp - x_exp
    LSR R7, R7, R0          ; shift x_man
    MOV R9, R6              ; result_exp = y_exp

.add_mant:
    ADD R7, R7, R8          ; result_man = x_man + y_man

    ; Normalize if bit 11 (0x800) is set
    TST R7, #0x800
    BEQ .pack_result
```

```
    LSR R7, R7, #1
    ADD R9, R9, #1

.pack_result:
    AND R7, R7, #0x3FF          ; mask mantissa
    LSL R9, R9, #10             ; shift exponent
    ORR R0, R7, R9              ; pack final float

    ; Store into memory[12] and [13]
    STRB R0, [R4, #12]
    LSR R0, R0, #8
    STRB R0, [R4, #13]

    ; Restore and return
    LDR R4, [SP, #0]
    LDR R5, [SP, #4]
    LDR R6, [SP, #8]
    LDR R7, [SP, #12]
    LDR R8, [SP, #16]
    LDR R9, [SP, #20]
    LDR LR, [SP, #24]
    ADD SP, SP, #28
    BX LR
```

4. Testbench and Results

The testbench we used was the given ones for programs 1,2,3

```
# // This material may not be copied, distributed, or otherwise disclosed outside
# // of the Customer's facilities without the express written permission of SISW,
# // and may not be used in any way not expressly authorized by SISW.
# //
# Loading sv_std.std
# Loading work.FloatAdder(fast)
# Loading work.Mux2_to_1_32bit(fast)
# Loading work.Mux2_to_1_8bit(fast)
# Loading work.Mux4_to_1_32bit(fast)
# Loading work.new_int2flt_tb(fast)
# Loading work.TopLevel(fast)
# Loading work.ProgramCounter(fast)
# Loading work.InstructionMemory(fast)
# Loading work.Ctrl(fast)
# Loading work.RegFile(fast)
# Loading work.ALU(fast)
# Loading work.DMem(fast)
# Loading work.TopLevel0(fast)
# Loading work.data_mem0(fast)
Execution interrupted or reached maximum runtime.
Exit code expected: 0, received: 137
Done
```

Program 1: int2float Conversion - Machine Code

100000001

010001000

010010001

111111000

100000010

100001001

101000010

101000001

111111000

100010000

100101000

010101010

010110011

111011100

111111100

Expected Testbench Output for Program 1:

A successful execution of our program 1 show the PC advancing through the instructions, registers R0, R1, R2, R5, R6 updating as per the assembly. The CMP instructions would set ALU flags, influencing the BEQ and B branches. Finally, the Done signal from the TopLevel module would assert when the HALT instruction is executed, and memory locations 2 and 3 would contain the 8-bit results

```
-- Loading module RegFile
-- Loading module ALU
-- Loading module DMem
** Error (suppressible): flt2fix_tb.sv(120): (vopt-7063) Failed to find 'data_mem1' in hierarchical name 'f2.data_mem1.mem_core'.
        Region: flt2fix_tb_noround
Optimization failed
End time: 00:45:16 on Aug 03,2025, Elapsed time: 0:00:00
Errors: 1, Warnings: 1
End time: 00:45:16 on Aug 03,2025, Elapsed time: 0:00:00
*** Summary ********************************************
    qrun: Errors:   0, Warnings:   0
    vlog: Errors:   0, Warnings:   0
    vopt: Errors:   1, Warnings:   1
  Totals: Errors:   1, Warnings:   1
Exit code expected: 0, received: 2
Done
```

Program 2: float2int Conversion - Machine Code

010000001

010001000

111100000

100000000

111000001

110101010

100000000

100001000

010100011

010101010

111111100

Expected Testbench Output for Program 2:

The testbench loads initial values into memory locations 4 and 5. The LDR instructions would correctly load these bytes into R0 and R1. The CMP R0, #0 would evaluate based on the loaded

MSB. The left and right shifts would modify R0 and R1. Finally, the STRB instructions would write the resulting bytes to conceptual memory locations 2 and 3. The Done signal would be set by HALT.

Program 3: float_add - Machine Code

010000001

010001000

010010011

010011010

010100001

010101000

010110011

010111010

100001000

010100000

010001001

111111100

Expected Testbench Output for Program 3:

For Program 3, the testbench would need to simulate the FPU's behavior. It would load initial float bytes into conceptual memory locations 8, 9, 10, 11. The processor would then LDR these into R0-R3. The STRB instructions would write these to the FPU's input ports. The STR R4, 0 would trigger the FPU's Start signal. The main processor would detect a Done signal through its LDR R5, 0 and CMP R5, #0 loop. Once Done, the processor would LDR the results from the FPU's output ports into R6 and R7, and then STR them to memory locations 12 and 13. The Done signal from the TopLevel module would assert when HALT is reached.

6. Evaluate the Performance of Your Processor

Strengths for these programs:

- Simplicity and Clarity: The fixed 9-bit instruction length and clear instruction formats (I, R, U, M, C, X) make the processor straightforward to understand, design, and debug.
- Memory Access: The load/store architecture LDR/STR allows direct byte-level interaction with memory, which is essential for handling multi-byte data like the 16-bit floating-point numbers

- Operations: Our set of arithmetic, logical, and shift operations allow general-purpose computation on 8-bit values.
- Control Flow: The inclusion of conditional branches (BEQ, BGE), unconditional branch (B), and indirect branch (BX) allows program flow, loops, conditional execution, and basic jump calls. The PC-relative branching is efficient for local jumps.
- FPU: For Program 3, we received assistance from AI-generation tools for how to synthesize a floating-point unit. We did not copy it but we give credit to them as it proved to be useful for our processor. It allowed for floating-point arithmetic to be possible for the constraints of this.

Weaknesses for these programs:

- Inefficiency with 8-bit datapath: The primary weakness is the 8-bit register size and ALU. Operations which would usually require 16-bit or 32-bit values now take multiple 8-bit instructions, increasing instruction count and execution cycles.
- Limited Immediate and Offset Ranges: Our decision to have 2-bit immediate and 5-bit branch offset was restricting for our immediate operations and the range of direct jumps. Larger immediates or jumps would require multi-instruction sequences
- Single-Cycle Processor: The processor is a single-cycle processor so it is not as efficient as what you would find in a pipelined, multi-cycle processor. A more advanced hardware design would include forwarding paths and branch prediction.

In conclusion, we tried our very hardest with our custom processor. Though we were unable to correctly get the results for our processor, we felt like we were very close. There were definitely some improvements we could have made and with more time we think we could have definitely implemented them. With the short span of time under the Summer Session, we think we did our best and hopefully our report shows for it. We had a working assembler, a new Floating Point Unit that allowed simpler floating-point arithmetic, and solid breakdowns of what our custom processor would produce.