RVV-F Coding Challenge

Challenge #01 RVV-F Assembly

Q1

Problem Statement:

Write an assembly program using the **RISC-V Vector Extension (RVV)** that computes the dot product of two 8-element integer vectors stored in memory and stores the result in a scalar register.

Details:

1. The input vectors are stored in memory as:
   - Vector A: `0x80000000` (8 elements, 32-bit integers)
   - Vector B: `0x80000020` (8 elements, 32-bit integers)
2. The result of the dot product should be stored in the scalar register `x10`.
3. Assume vector registers `v0` and `v1` are used to load the input vectors.

Expectations:

- Correct implementation of vectorized operations.
- Efficient use of RVV instructions (e.g., `vle32.v`, `vadd.vv`, `vmul.vv`, `vredsum.vs`).
- Commented code for clarity.

Example Input:

Vector A: `{1, 2, 3, 4, 5, 6, 7, 8}`
Vector B: `{8, 7, 6, 5, 4, 3, 2, 1}`

Example Output:

Dot Product: `120`

Q2

Problem Statement:

Write an assembly program using the **RISC-V Vector Extension (RVV)** to add two 16-element integer vectors stored in memory and store the result in a separate memory location.

Details:

- The input vectors are stored in memory as:
  - Vector A: `0x80000000` (16 elements, 32-bit integers)
  - Vector B: `0x80000040` (16 elements, 32-bit integers)
- The result should be stored starting at `0x80000080`.
- Use vector registers `v0` and `v1` to load the input vectors.
- Store the result in memory using `vse32.v`.

Expectations:

- Correct use of RVV instructions (`vle32.v`, `vadd.vv`, `vse32.v`).
- Efficient handling of memory and vector operations.
- Include comments to explain each step.

Example Input:
Vector A: `{1, 2, 3, ..., 16}`
Vector B: `{16, 15, 14, ..., 1}`

Example Output:
Result Vector: `{17, 17, 17, ..., 17}`

Q3

Problem Statement:
Write an assembly program using the **RISC-V Vector Extension (RVV)** to find the maximum value in a 32-element vector of integers stored in memory.

Details:

- The input vector is stored in memory at address `0x80001000` (32 elements, 16-bit integers).
- Store the maximum value in the scalar register `x11`.
- Use reduction instructions like `vmax.vs`.

Expectations:

- Correct use of vector load (`vle16.v`) and reduction operations (`vredmax.vs`).
- Efficient computation of the maximum value.
- Commented code to enhance readability.

Example Input:
Vector: `{5, 12, 7, 19, ..., 3}`

Example Output:
Maximum Value: `19`

Q4

Problem Statement:
Write an assembly program using the **RISC-V Vector Extension (RVV)** to multiply all elements of a 20-element vector by a scalar value and store the result in memory.

Details:

- Input vector is stored at `0x80002000` (20 elements, 32-bit integers).
- Scalar value: `5` (stored in `x12`).
- Output vector should be stored starting at `0x80003000`.

Expectations:

- Use `vle32.v` to load the vector, `vmul.vx` for scalar multiplication, and `vse32.v` to store the result.
- Clear and efficient code with comments.

Example Input:
Vector: `{1, 2, 3, ..., 20}`
Scalar: `5`

Example Output:
Result Vector: `{5, 10, 15, ..., 100}`

Q5

Problem Statement:
Write an assembly program using the **RISC-V Vector Extension (RVV)** to compare two 12-element vectors element-wise and generate a mask vector indicating where elements of Vector A are greater than Vector B.

Details:

- Input vectors:
  - Vector A: `0x80004000`
  - Vector B: `0x80004030`
- The mask vector should be stored in `v0`.

Expectations:

- Use `vle32.v` to load vectors and `vmslt.vv` or similar instructions for comparison.
- Include comments for clarity.

Example Input:
Vector A: `{3, 5, 7, 9, ..., 25}`
Vector B: `{4, 3, 8, 7, ..., 20}`

Example Output:
Mask Vector: `{0, 1, 0, 1, ..., 1}`

Q6

Problem Statement:
Write an assembly program using the **RISC-V Vector Extension (RVV)** to compute the dot product of two 10-element vectors but include only elements where the first vector has non-zero values.

Details:

- Input vectors:
    - Vector A: `0x80005000` (mask based on non-zero elements).
    - Vector B: `0x80005028`.
- Result should be stored in scalar register `x10`.

Expectations:

- Correct use of masking instructions (`vmseq.vx`) and reduction (`vredsum.vs`).
- Commented code for understanding.

Example Input:
Vector A: `{1, 0, 2, 0, 3}`
Vector B: `{4, 5, 6, 7, 8}`

Example Output:
Dot Product: `1*4 + 2*6 + 3*8 = 32`

Challenge #02 CHISEL Scala Programming

Q1

Problem Statement:
Design a **Chisel module** that implements a simple Arithmetic Logic Unit (ALU) capable of performing addition, subtraction, AND, and OR operations based on a 2-bit control signal.

Details:

- The ALU takes two 8-bit inputs (`a` and `b`) and a 2-bit control signal (`ctrl`).
- Based on the value of `ctrl`, the ALU performs:
  - `00` -> Addition (`a + b`)
  - `01` -> Subtraction (`a - b`)
  - `10` -> AND (`a & b`)
  - `11` -> OR (`a | b`)
- The result should be output as an 8-bit value (`result`).

Expectations:

- Design a clean and parameterized ALU module.
- Include a testbench that tests all operations with multiple inputs.

Example Input:

- `a = 8'b00001111`, `b = 8'b00000011`, `ctrl = 2'b00`
- `a = 8'b10101010`, `b = 8'b01010101`, `ctrl = 2'b10`

Example Output:

- `result = 8'b00010010` (Addition)
- `result = 8'b00000000` (AND)

Q2

Problem Statement:
Design a **4-to-1 multiplexer** in Chisel that selects one of four 16-bit input values based on a 2-bit selection signal.

Details:

- Inputs:
  - `in0`, `in1`, `in2`, `in3` (each 16 bits).
  - `sel` (2 bits).
- Output:
  - `out` (16 bits).
- The selected input (`in[sel]`) should be routed to `out`.

Expectations:

- Implement the multiplexer using conditional expressions (`when`, `elsewhen`).
- Provide a testbench to verify the correct operation for all selection values.

Example Input:

- `in0 = 16'h000F`, `in1 = 16'h00F0`, `in2 = 16'h0F00`, `in3 = 16'hF000`, `sel = 2'b10`

Example Output:

- `out = 16'h0F00`

Q3

Problem Statement:
Design a **parameterized counter** in Chisel that increments on every clock cycle and resets when it reaches a maximum value.

Details:

- Parameter: `n` (bit-width of the counter).
- Inputs:
  - `reset` (active high).
  - `enable` (enable signal for counting).
- Output:
  - `count` (current count value).
- The counter should wrap around to `0` after reaching its maximum value (`2^n - 1`).

Expectations:

- Implement a parameterized counter using `RegInit` and `when` conditions.
- Test the counter with different values of `n`.

Example Input:

- `n = 4`, `reset = 1`, `enable = 1`

Example Output:

- Count values: `0, 1, 2, ..., 15, 0, 1, ...`

Q4

Problem Statement:
Design a **priority encoder** in Chisel that takes an 8-bit input and outputs the position of the highest-priority bit that is set to `1`.

Details:

- Input:
  - `in` (8 bits).
- Output:
  - `pos` (3 bits, position of the highest-priority `1`).
  - If no bits are set, output should be `pos = 3'b000`.
- Priority: The most significant bit (MSB) has the highest priority.

Expectations:

- Use a combination of bitwise operations and conditionals to determine the position.
- Test the design with all possible inputs.

Example Input:

- `in = 8'b00101000`
- `in = 8'b10000000`

Example Output:

- `pos = 3'b101`
- `pos = 3'b111`

Q5

Problem Statement:
Design a **4-bit shift register** in Chisel that supports both left and right shifts based on a control signal.

Details:

- Inputs:
  - `in` (1-bit data input).
  - `load` (load new data).
  - `shift` (1 for shift-right, 0 for shift-left).
- Outputs:
  - `out` (4-bit output).
- If `load` is active, the value of `in` should be loaded into the least significant bit (LSB).

Expectations:

- Implement the shift register using `Reg` and conditionals (`when`).
- Verify operation through a testbench with a sequence of shifts and loads.

Example Input:

- Sequence: `load=1, in=1 -> shift=0 -> shift=0 -> shift=1`

Example Output:

- Output Values: `0001 -> 0010 -> 0100 -> 0010`

Q6

Problem Statement:
Design a **Barrel Shifter** module in Chisel that performs left or right shifts by a specified number of bits in a single clock cycle.

Details:

- Inputs:
  - `data_in` (N-bit input data).
  - `shift` (log2(N)-bit input specifying the number of positions to shift).
  - `direction` (1-bit input: `0` for left shift, `1` for right shift).
- Output:
  - `data_out` (N-bit output data).
- Behavior:
  - For `direction = 0`, the input is shifted left by the specified number of positions, with `0`s filling the least significant bits.
  - For `direction = 1`, the input is shifted right, with `0`s filling the most significant bits.

Expectations:

- Implement the design using combinational logic (e.g., bit slicing or conditional expressions).
- Make the module parameterized for `N`.
- Testbench should verify shifts for various input sizes and directions.

Example Input:

- `data_in = 8'b10110011`, `shift = 3`, `direction = 0` (left shift).
- `data_in = 8'b10110011`, `shift = 2`, `direction = 1` (right shift).

Example Output:

- `data_out = 8'b10011000` (left shift).
- `data_out = 8'b00101100` (right shift).