Implementing the Accelerator

Rodolfo Azevedo

Institute of Computing, University of Campinas (UNICAMP), Brazil

rodolfo.azevedo@unicamp.br

http://www.ic.unicamp.br/~rodolfo/mo801

Goal of this class

Module 5, Class 2: the int8 MAC array, local BRAM buffer, accumulator, and control FSM in SystemVerilog.

> This is the implementation class. The spec is fixed (M04A04). The goal is a Verilator testbench that passes for all test vectors before we connect anything to the bus.

At the end of this class, you should be able to:

- Implement an int8 MAC array in SystemVerilog and verify it produces correct results with Verilator.
- Design the local BRAM buffer and accumulator for the KWS layer computation.
- Write the control FSM that sequences data fetch, multiply-accumulate, and writeback operations.
- Run the accelerator testbench against all test vectors to confirm bit-identical results before bus integration.

Top-level module interface

```
module kws_accel (
    input  logic        clk, rst,
    // Bus interface (from Project 2)
    input  bus_req_t    req,
    input  logic        en,          // selected by address decoder
    output bus_resp_t   resp,
    // DMA-style access to DMEM (to fetch input activations)
    output logic [31:0] dmem_addr,
    output logic        dmem_re,
    input  logic [31:0] dmem_rdata,
    input  logic        dmem_ready
);
    // Internal signals
    logic        start, done, busy;
    logic [31:0] input_addr, weight_addr, output_addr;
    logic [7:0]  length;
    // ...
endmodule
```

The accelerator has two interfaces:

- **Bus interface** — CPU configures registers and reads STATUS.
- **DMEM interface** — accelerator fetches input activations directly from data memory (DMA-like). Weights are pre-loaded into the local BRAM by the CPU via the bus.

The int8 MAC unit

A single int8 MAC: multiply two 8-bit signed integers and accumulate into a 32-bit register.

```
module mac_unit (
    input  logic        clk, rst,
    input  logic        en,           // compute one MAC per cycle when high
    input  logic        clear,        // reset accumulator to zero
    input  logic signed [7:0]  a, b,  // int8 operands
    output logic signed [31:0] acc    // running sum
);
    logic signed [15:0] product;
    assign product = a * b;           // 8×8 → 16-bit product (no overflow)

    always_ff @(posedge clk) begin
        if (rst || clear) acc <= '0;
        else if (en)      acc <= acc + {{16{product[15]}}, product};
                                       // sign-extend 16→32 before adding
    end
endmodule
```

- `a * b` — the synthesis tool infers a DSP block for this.
- Sign-extension: `product` is 16-bit signed; extending to 32 bits before accumulation prevents overflow for up to 2^{16} accumulations at max value (127 × 127 = 16,129).

4-wide MAC array

Four MAC units operating in parallel on four lanes:

```
module mac_array #(parameter LANES = 4) (
    input  logic                          clk, rst,
    input  logic                          en, clear,
    input  logic signed [7:0]             a [0:LANES-1],
    input  logic signed [7:0]             b [0:LANES-1],
    output logic signed [31:0]            acc [0:LANES-1]
);
    genvar i;
    generate
        for (i = 0; i < LANES; i++) begin : lane
            mac_unit u (.clk(clk), .rst(rst), .en(en), .clear(clear),
                        .a(a[i]), .b(b[i]), .acc(acc[i]));
        end
    endgenerate
endmodule
```

Each lane accumulates one partial result. After `LENGTH/LANES` cycles, each accumulator holds a partial dot product. Sum the four accumulators to get the final result.

For the Conv2D inner loop (length = KH×KW×C_in, 4 lanes): **length/4 MAC cycles** to compute one output value, vs. length cycles for a scalar implementation.

Local weight BRAM buffer

The weight buffer holds one row of the weight matrix (64 int8 values = 64 bytes):

```
module weight_bram #(
    parameter DEPTH = 64,
    parameter WIDTH = 8
) (
    input  logic                         clk,
    // Write port: CPU fills via bus
    input  logic [$clog2(DEPTH)-1:0] waddr,
    input  logic [WIDTH-1:0]          wdata,
    input  logic                         we,
    // Read port: accelerator reads 4 values per cycle (4×8 = 32 bits)
    input  logic [$clog2(DEPTH)-3:0] raddr,   // address in units of 4 bytes
    output logic [WIDTH*4-1:0]        rdata    // 4 int8 values
);
    logic [WIDTH-1:0] mem [0:DEPTH-1];

    always_ff @(posedge clk) begin
        if (we) mem[waddr] <= wdata;
        rdata <= {mem[{raddr,2'b11}], mem[{raddr,2'b10}],
                  mem[{raddr,2'b01}], mem[{raddr,2'b00}]};
    end
endmodule
```

The read port delivers 4 int8 values in one cycle — matched to the 4-wide MAC array. One full weight row (64 int8) takes 16 read cycles to deliver to all 4 lanes.

Requantization in hardware

The int32 accumulator must be requantized to int8 before writing the output:

```
module requant (
    input  logic signed [31:0] acc,
    input  logic [31:0]        bias,
    input  logic [15:0]        multiplier,  // fixed-point scale factor
    input  logic [4:0]         shift,       // right-shift amount
    input  logic signed [7:0]  zero_point,
    output logic signed [7:0]  result
);
    logic signed [63:0] biased;
    logic signed [63:0] scaled;
    logic signed [31:0] rounded;

    assign biased  = acc + {{32{bias[31]}}, bias};
    assign scaled  = biased * multiplier;
    assign rounded = scaled >>> shift;         // arithmetic right shift

    // Saturate to int8 and add zero point
    logic signed [31:0] shifted_zp;
    assign shifted_zp = rounded + zero_point;
    assign result = (shifted_zp > 127)  ? 8'sd127  :
                    (shifted_zp < -128) ? -8'sd128 :
                    shifted_zp[7:0];
endmodule
```

The multiplier+shift trick avoids floating-point: `scale ≈ multiplier × 2^{-shift}`, precomputed by the training

Control FSM: IDLE → LOAD → COMPUTE → DONE

```
typedef enum logic [1:0] {IDLE, LOAD, COMPUTE, DONE} accel_state_t;
accel_state_t state;
logic [6:0] cycle_cnt;    // counts up to 64 (for length-64 vectors)
logic [6:0] lane_cnt;     // which group of 4 elements we're on

always_ff @(posedge clk) begin
    if (rst) state <= IDLE;
    else case (state)
        IDLE:    if (start)        begin state <= LOAD;    cycle_cnt <= 0; end
        LOAD:    if (dmem_ready)   begin                   // fetch next activation word
                     if (cycle_cnt == (length>>2)-1)
                                   begin state <= COMPUTE; cycle_cnt <= 0; end
                     else          cycle_cnt <= cycle_cnt + 1;
                 end
        COMPUTE: begin mac_en <= 1;
                     if (cycle_cnt == (length>>2)-1)
                                   begin state <= DONE;    mac_clear <= 1; end
                     else          cycle_cnt <= cycle_cnt + 1;
                 end
        DONE:    begin done <= 1;  if (~status_read) state <= IDLE; end
    endcase
end
```

LOAD and COMPUTE can be **overlapped** (load next activation while current MACs are computing) for a 2× throughput gain — a pipeline optimization for later.

Timing analysis: cycles per inference call

For one Conv2D output value (one output channel, one spatial position), with `LENGTH = KH×KW×C_in`, `LANES = 4`:

Phase	Duration	Notes
LOAD (activations from DMEM)	`LENGTH/4` cycles	4 int8 values fetched per cycle; may stall if DMEM busy
COMPUTE (MAC array)	`LENGTH/4` cycles	Runs in lock-step with weight BRAM reads
WRITEBACK (requantize + write)	1–2 cycles	Combinational requant + one DMEM write

For the full Conv2D layer with `H_out × W_out × C_out` output values:

$$\text{Total cycles} \approx H_{out} \times W_{out} \times C_{out} \times \frac{K_H \times K_W \times C_{in}}{2}$$

Compare to the software baseline:

$$\text{SW cycles} \approx H_{out} \times W_{out} \times C_{out} \times K_H \times K_W \times C_{in} \times 4 \text{ (cycles/MAC)}$$

Predicted speedup = SW / HW $\approx 4 \times K_H \times K_W \times C_{in} / (K_H \times K_W \times C_{in}/2)$ = **8×** — but Amdahl's Law reduces system speedup to ~4.5× for $f = 0.92$.

> This predicted speedup assumes no stalls and no bus overhead. Your measured number will be lower. The gap between predicted and measured is where debugging happens.

Verification: directed test against the C kernel

```
// kws_accel_tb.sv
initial begin
    // 1. Load weight row 0 from test_vectors.h into weight BRAM
    for (int i = 0; i < 64; i++) begin
        write_reg(WEIGHT_DATA, weights_row0[i]);
        write_reg(WEIGHT_ADDR_REG, i);
        write_reg(WEIGHT_WE, 1);
        @(posedge clk);
    end

    // 2. Set up the accelerator
    write_reg(INPUT_ADDR,  32'h00011000);   // input at DMEM 0x1000
    write_reg(LENGTH,      64);
    write_reg(CONTROL,     1);              // start

    // 3. Wait for done
    do @(posedge clk); while (!read_reg(STATUS)[0]);

    // 4. Compare
    assert (read_reg(RESULT) == expected_output[0])
        else $error("Mismatch: got %0d, expected %0d",
                    read_reg(RESULT), expected_output[0]);

    $display("All checks passed."); $finish;
end
```

Run with `make sim`. Only after all assertions pass: connect to the bus.

In-class exercise — FSM trace

Trace the control FSM for a **length-8 dot product** (`LENGTH = 8`, `LANES = 4`):

Fill in the table cycle by cycle (start from `IDLE`, then `start` asserts):

Cycle	State	`dmem_addr`	`mac_en`	`mac_clear`	`done`	Notes
0	IDLE	—	0	0	0	waiting
1	LOAD					start asserted
2	LOAD					
3	COMPUTE					length/4=2 load cycles done

Questions:

1. In cycle 1, what address does the accelerator put on `dmem_addr`? (Hint: `INPUT_ADDR` register value)
2. In cycle 3, what happens to the weight BRAM address? Does it reset or continue from where LOAD left off?
3. If `dmem_ready` is deasserted for one cycle during LOAD, how does the total cycle count change?

> Expected: 2 LOAD cycles (8 elements / 4 lanes), 2 COMPUTE cycles, 1 DONE cycle = 5 active cycles + overhead. A `dmem_ready` stall in LOAD extends that phase by 1 cycle.

Pipelining LOAD and COMPUTE

The basic FSM does LOAD then COMPUTE sequentially. With double-buffering, they can overlap:

```
// Dual-buffer approach: while computing on buffer A, load into buffer B
typedef enum {IDLE, FILL_A, FILL_B_COMPUTE_A, FILL_A_COMPUTE_B, DONE} pipe_state_t;

// When in FILL_B_COMPUTE_A:
//   - weight_bram read address increments (for MAC array from buffer A)
//   - activation_bram write address increments (loading into buffer B)
//   - mac_en = 1, dmem_re = 1 simultaneously
```

Ideal throughput with overlap:

- Without overlap: `2 × (LENGTH/4)` cycles per output value.
- With overlap: `LENGTH/4 + startup` cycles — the LOAD and COMPUTE phases run in parallel.
- For `LENGTH = 64`: 16 cycles (overlapped) vs. 32 cycles (sequential) — **2× throughput improvement**.

Implementation cost: one extra BRAM for the second activation buffer, and a more complex FSM. This is the "pipeline optimization for later" mentioned in the base FSM slide.

> This is the same principle as the multicycle vs. pipelined processor from M02: separate "stages" that can run simultaneously. The accelerator FSM is a 2-stage pipeline: fetch activations (stage 1) and compute MACs (stage 2).

Next class

Integration & End-to-End Measurement: wiring the accelerator as a memory-mapped peripheral, the C software driver, replacing the software PW conv with `accel_run()`, and measuring cycle counts before and after.