Integration & Project 2

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 3, Class 4: top-level integration, end-to-end verification, and running KWS for the first time.

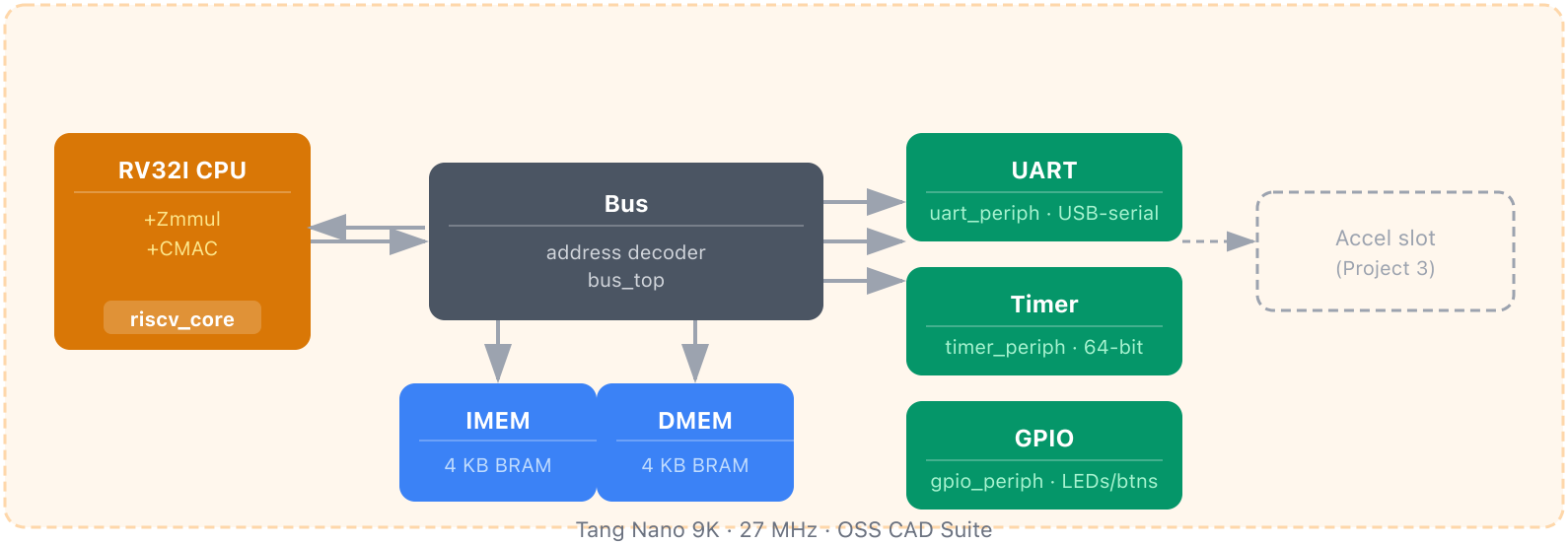
> All the pieces exist. Today we wire them together, verify end-to-end, and run the first version of the KWS inference kernel on hardware you built. It will be slow — that is the point. Measuring *how slow* is what motivates Project 3.

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

- Integrate the processor, bus, and peripherals into a complete SoC top-level SystemVerilog module.
- Verify end-to-end SoC operation with compiled C programs in Verilator simulation.
- Run the KWS inference kernel on the Tang Nano 9K and observe inference results via UART.
- Measure baseline inference latency to establish the optimization target for Project 3.

Top-level integration

The full platform for Project 2:



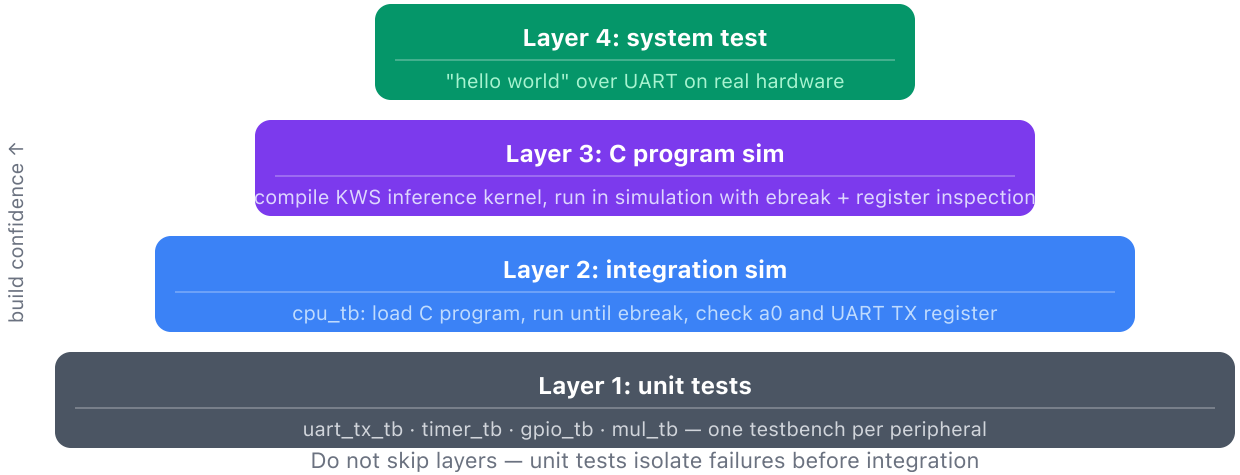
The top-level SV module instantiates: `riscv_core`, `bus_top`, `uart_periph`, `timer_periph`, `gpio_periph`, instruction BRAM, data BRAM.

Top-level SystemVerilog

```
module top (
    input  logic       clk,      // 27 MHz
    input  logic       rst_n,    // active-low (button)
    input  logic [1:0] btn,
    output logic [5:0] led,
    output logic       uart_tx,
    input  logic       uart_rx
);
    logic rst;
    assign rst = ~rst_n;

    bus_req_t  cpu_req;
    bus_resp_t cpu_resp;

    riscv_core cpu (
        .clk(clk), .rst(rst),
        .bus_req(cpu_req),
        .bus_resp(cpu_resp)
    );

    bus_top bus (
        .clk(clk), .rst(rst),
        .cpu_req(cpu_req),
        .cpu_resp(cpu_resp),
        .uart_tx(uart_tx), .uart_rx(uart_rx),
        .gpio_in({4'b0, btn}),
        .gpio_out(led)
    );
endmodule
```

Verification strategy: bottom-up

Build confidence layer by layer before integration:



Do not skip layers. If Layer 2 fails and you have not done Layer 1, you have no idea which peripheral is broken. Unit tests isolate the failure.

Unit testbench pattern: UART TX

```
module uart_tx_tb;
    logic clk = 0, rst, tx_valid;
    logic [7:0] tx_data;
    logic tx_ready, tx_out;

    uart_tx #(.CLK_FREQ(27_000_000), .BAUD_RATE(115_200)) dut (
        .clk(clk), .rst(rst),
        .tx_data(tx_data), .tx_valid(tx_valid),
        .tx_ready(tx_ready), .tx_out(tx_out)
    );

    always #18 clk = ~clk;   // ~27 MHz (18 ns half-period)

    task send_byte(input [7:0] b);
        @(posedge clk); tx_data = b; tx_valid = 1;
        @(posedge clk); tx_valid = 0;
        // Wait for transmission to complete
        wait(tx_ready);
    endtask

    initial begin
        $dumpfile("uart_tx.vcd"); $dumpvars(0, uart_tx_tb);
        rst = 1; repeat(4) @(posedge clk); rst = 0;
        send_byte(8'h41);   // 'A'
        send_byte(8'h0A);   // newline
        #10000; $finish;
    end
endmodule
```

Open `uart_tx.vcd` in GTKWave and verify: start bit = 0, 8 data bits LSB-first, stop bit = 1, each lasting 234 cycles.

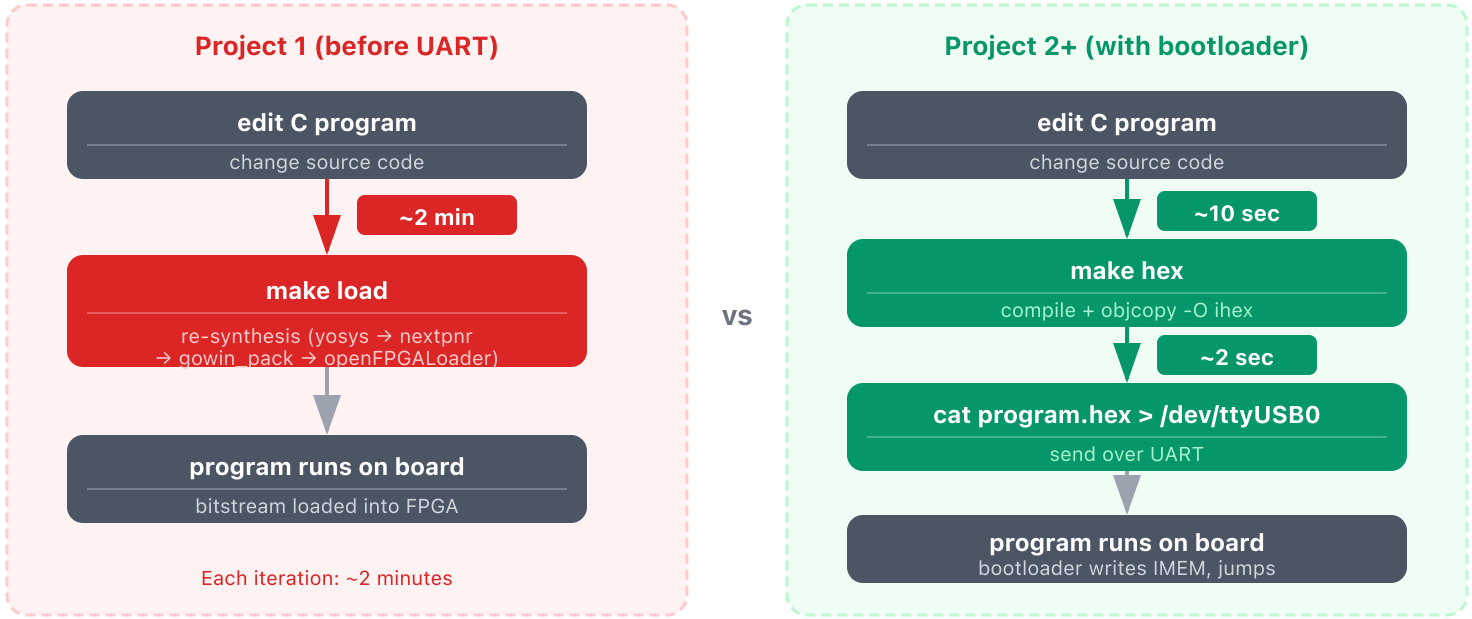
Integration testbench: C program on the CPU

```
module integration_tb;
    logic clk = 0, rst;
    // ... DUT instantiation (top.sv with UART looped back)

    always #18 clk = ~clk;

    initial begin
        $readmemh("hello.hex", top.bus.imem.mem);
        rst = 1; repeat(4) @(posedge clk); rst = 0;

        // Wait for ebreak or timeout
        fork
            begin wait(top.cpu.ebreak_detected); end
            begin #10_000_000; $display("TIMEOUT"); $finish; end
        join_any

        $display("x10 (a0) = %0d", top.cpu.regfile[10]);
        $finish;
    end

    // Monitor UART TX register writes for expected output
    always @(posedge clk) begin
        if (top.bus.sel_uart && top.cpu_req.we
            && top.cpu_req.addr[3:2] == 2'b00)
            $write("%c", top.cpu_req.wdata[7:0]);
    end
endmodule
```

The `$write` trick lets you see UART output in the simulation log without needing a full UART model.

The new loading workflow

Once the bootloader is in BRAM and the UART works on hardware, the development loop changes completely:



The bitstream (hardware) stays the same — only the software changes. Re-synthesis is only needed when you modify the SystemVerilog.

Running KWS for the first time

Once the platform works for "hello world", compile the KWS inference kernel:

```
riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 -nostdlib \
    -Wl,-Ttext=0x0 -O2 \
    kws_inference.c uart_driver.c timer_driver.c \
    -o kws.elf
riscv64-unknown-elf-objcopy -O binary kws.elf kws.bin
# convert to hex for $readmemh
```

The kernel:

1. Reads MFCC features from a lookup table in DMEM.
2. Runs the Conv2D inference layers.
3. Outputs the predicted class over UART.
4. Writes cycle counts to a buffer, then `ebreak`.

Expected result at this stage: it works, but it is slow — on the order of tens of millions of cycles (seconds at 27 MHz). Record the exact number. This is your **baseline** for Project 3.

Synthesis and resource report

After simulation passes, synthesize the full platform:

```
make load   # yosys → nextpnr → gowin_pack → openFPGALoader
```

Read the nextpnr output carefully:

```
Info: Device utilisation:
   SLICE:     1247/4608  (27%)
   LUT4:      2034/9216  (22%)
   DFF:        512/6480  ( 8%)
   BRAM:         4/26   (15%)
   DSP:          2/20   (10%)   ← Zmmul + CMAC
```

Key observations:

- Project 2 uses ~27% of slices — plenty of room for Project 3's accelerator.
- Two DSP blocks used for Zmmul and CMAC — DSPs are much faster than LUT-based multipliers.
- BRAM: 4 blocks for IMEM + DMEM. 22 blocks remain for the accelerator's local buffers.

Project 2 checklist

Before submitting, verify:

- [ ] IMEM é SRAM gravável (não ROM) — bootloader pode escrever nela.
- [ ] Bootloader recebe um Intel HEX via UART, escreve em IMEM e salta para `0x0100`.
- [ ] Zmmul (MUL, MULH, MULHU, MULHSU) passes all directed tests in simulation.
- [ ] CMAC instruction accumulates correctly; `csrr` / `csrw` on `macc` CSR works.
- [ ] Bus routes transactions to all peripherals correctly (assert `sel_*` signals).
- [ ] UART TX sends a known byte in simulation (GTKWave: verify start/data/stop bits).
- [ ] Timer counter increments correctly (read at t_0, wait N cycles, read at t_1, check $t_1 - t_0 = N$).
- [ ] GPIO writes turn LEDs on/off in hardware.
- [ ] C program "hello world" appears on UART terminal on the board.
- [ ] KWS inference kernel runs and produces a correct classification in simulation.
- [ ] Synthesis report: no timing violations at 27 MHz; LUT/FF/BRAM utilization recorded.
- [ ] **Security**: identify one security vulnerability in your design and describe how you would mitigate it.

> **The slides that follow cover security, energy, and reproducibility.** The security corners are required reading — one of them is directly referenced in the Project 2 checklist. The energy corners are optional enrichment.

Security corner: the UART bootloader attack surface

Your bootloader accepts Intel HEX data over UART and writes it directly into IMEM. There is **no authentication, no encryption, and no integrity check**.

Anyone with physical access to the serial port can:

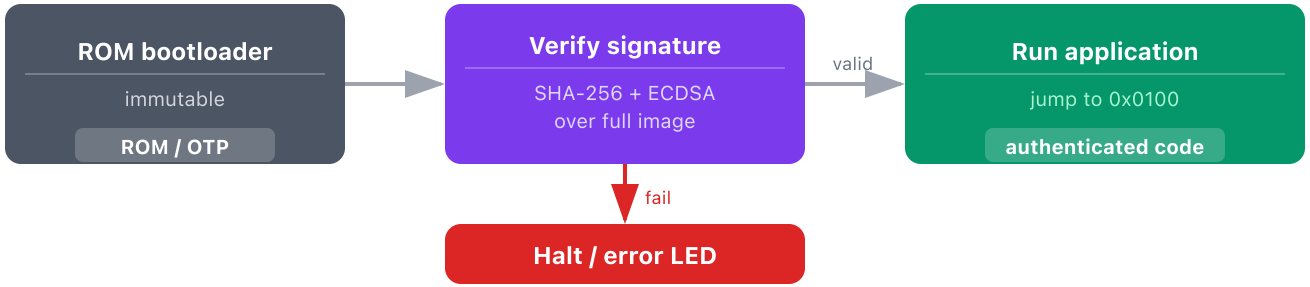
1. **Inject arbitrary code** — send a crafted `.hex` file that overwrites the entire program.
2. **Exfiltrate data** — load a program that reads DMEM and sends it back over UART.
3. **Brick the device** — overwrite the bootloader itself (if IMEM includes the bootloader code).

Real-world parallel

This is exactly how IoT devices get compromised: a UART console left exposed on a production PCB (often labeled `TX` / `RX` on the board) gives an attacker full control. In 2020, researchers compromised a popular home router by soldering wires to its UART pads and loading a modified firmware.

Security corner: secure boot — what a real system does

A secure boot chain prevents unauthorized code from running:



Component	Your Project 2	Production system
Boot code location	Writable SRAM	Immutable ROM
Code authentication	None	ECDSA / RSA signature

What you could add (conceptually)

- **HMAC verification**: compute HMAC-SHA256 of the received binary using a key stored in BRAM. If the HMAC does not match the last 32 bytes of the payload, do not jump to `0x0100`.
- **Lock IMEM after boot**: a one-way latch register — once the bootloader sets `imem_locked = 1`, no further writes to IMEM are accepted until the next reset.

Security corner: memory-mapped I/O without access control

Every peripheral in your memory map is accessible from any code:

- A bug in application code could accidentally write to the UART TX register, sending garbage to the serial port.
- A compromised application could reprogram the timer to fire interrupts at maximum rate, causing a denial-of-service.

The defense: bus firewalls

In production SoCs, a **bus firewall** (or TrustZone on Arm) restricts which masters can access which address ranges. RISC-V's PMP can enforce this at the CPU level:

- Only machine-mode code can access I/O registers.
- User-mode code gets an access fault if it tries to write to `0x0002_xxxx`.

> **Project 2 security question**: *"Your bootloader has no authentication. Describe a practical attack that exploits this and one mechanism that would prevent it."*

Energy corner: peripheral power and idle management (Optional)

Every peripheral you added in Project 2 consumes power — even when idle:

Peripheral	Idle behavior	Power impact
UART TX	Shift register holds `1` (mark)	Minimal — no toggles

The real cost: the bus

The bus routes every load/store through the address decoder, even when the target is DMEM (the most common case). Every bus transaction toggles the mux/decoder logic for **all** peripherals, not just the target.

In production SoCs:

- **Clock gating per peripheral**: disabled peripherals consume zero dynamic power.
- **Power domains**: entire peripheral blocks can be powered off (not just clock-gated).
- **Bus isolation**: inactive peripherals are disconnected from the bus to reduce capacitive load.

Measuring power on the Tang Nano 9K

You can measure your design's power with a USB power meter:

- Baseline (FPGA configured, no design): ~80 mA at 5V
- Your Project 2 system: measure it! Compare with Project 1 — the difference is the cost of your peripherals.

Energy corner: UART baud rate and energy (Optional)

A faster baud rate sends the same data in less time — but does it save energy?

- **Transmit energy per byte** ≈ constant (same number of bit transitions regardless of speed).
- **System energy during transfer** decreases at higher baud rates: the CPU waits fewer cycles polling the UART status register → fewer clock cycles → less total energy.
- At 115200 baud, sending 1 KB takes ~89 ms. At 9600 baud, it takes ~1.07 s. The CPU runs for 12× longer at the slower rate.

> For the bootloader, higher baud rate is strictly better: same data, less waiting, less energy.

Reproducibility note: memory map as a contract

Your memory map (M03A01) is a **hardware/software contract**:

```
0x0000_0000 – 0x0000_0FFF   IMEM (4 KB)
0x0001_0000 – 0x0001_0FFF   DMEM (4 KB)
0x0002_0000                  UART
0x0002_0100                  Timer
0x0002_0200                  GPIO
0x0002_0300                  Accelerator (reserved)
```

- **Document it in your README** — the C driver code depends on these addresses.
- **Use `#define` or a header file** for all addresses — never hardcode `0x00020000` in driver code.
- **Version it**: if the memory map changes between Project 2 and Project 3, the old C code must not compile silently with wrong addresses.

> A mismatch between the memory map in hardware and in software is one of the hardest bugs to find — everything synthesizes, everything compiles, but loads/stores go to the wrong peripheral.

Next module

Module 4 — TinyML on the Edge: the KWS model in depth, the inference kernel in C, and profiling — finding exactly which operation is responsible for the baseline cycle count we just measured.