Digital Building Blocks

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 1, Class 5: ALU, register file, memory arrays — and parameterized/generated modules.

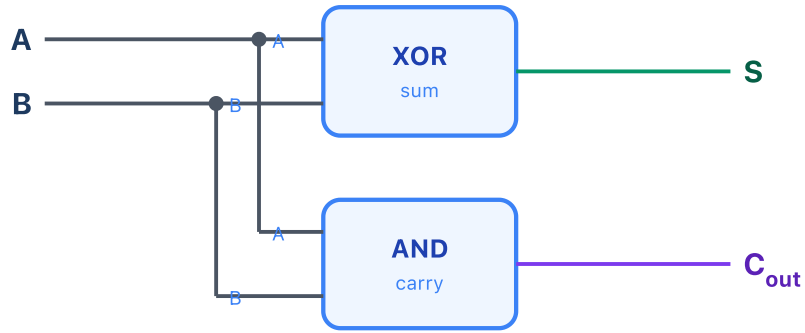
> Today we assemble the components of the RV32I datapath before we formally introduce RISC-V. By the end of this class, you will have seen every hardware module that Project 1 requires — what remains is wiring them together and adding the control FSM.

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

- Implement adders, an ALU with shift and comparison operations, and a barrel shifter in SystemVerilog.
- Design a register file with simultaneous read ports and a write port.
- Use parameterized modules and `generate` constructs to create scalable hardware structures.
- Describe synchronous memory arrays in SystemVerilog and explain how BRAM inference works.

Arithmetic: half adder

A **half adder** adds two 1-bit values:



Half adder: S = A ⊕ B, Cout = A · B

$$S = A \oplus B \qquad C_{out} = A \cdot B$$

```
module half_adder (input logic a, b, output logic s, cout);
    assign s    = a ^ b;
    assign cout = a & b;
endmodule
```

Full adder

A **full adder** also accepts a carry-in:

$$S = A \oplus B \oplus C_{in} \qquad C_{out} = AB + C_{in}(A \oplus B)$$

```
module full_adder (input logic a, b, cin, output logic s, cout);
    assign s    = a ^ b ^ cin;
    assign cout = (a & b) | (cin & (a ^ b));
endmodule
```

Ripple-carry adder

Chain N full adders to add two N-bit numbers:

```
module adder_rca #(parameter WIDTH = 32) (
    input  logic [WIDTH-1:0] a, b,
    input  logic             cin,
    output logic [WIDTH-1:0] sum,
    output logic             cout
);
    logic [WIDTH:0] carries;
    assign carries[0] = cin;
    assign cout = carries[WIDTH];

    genvar i;
    generate
        for (i = 0; i < WIDTH; i++) begin : fa_chain
            full_adder fa (
                .a(a[i]), .b(b[i]), .cin(carries[i]),
                .s(sum[i]), .cout(carries[i+1])
            );
        end
    endgenerate
endmodule
```

The critical path runs through all N carry chains — $O(N)$ delay. For 32 bits at 27 MHz this is fine; at GHz clock rates you'd need a **carry-lookahead adder** ($O(\log N)$ delay).

`parameter` and `localparam`

```
module adder #(
    parameter  WIDTH = 32,          // overridable from outside
    localparam HALF  = WIDTH / 2    // internal constant, not overridable
) (
    input  logic [WIDTH-1:0] a, b,
    output logic [WIDTH-1:0] sum,
    output logic             cout
);
    assign {cout, sum} = a + b;    // let the tool infer the adder
endmodule
```

- `parameter` — can be overridden at instantiation: `adder #(.WIDTH(8)) u0 (...)`.
- `localparam` — a constant derived from parameters; cannot be overridden externally.
- Parameterized modules are the standard way to write reusable, width-independent components. `parameter WIDTH = 32` is in almost every datapath module you will write.

`generate` — repetitive instantiation

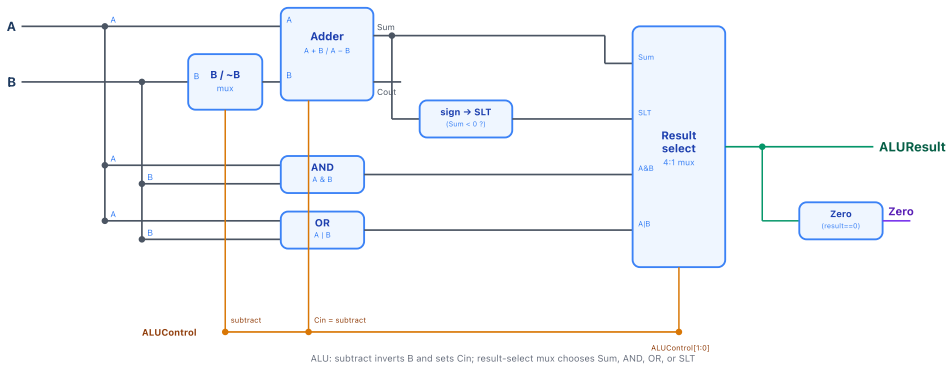
`generate for` instantiates or assigns hardware in a loop that is **unrolled at compile time**:

```
genvar i;
generate
    for (i = 0; i < 4; i++) begin : mux_array
        mux2 #(8) m (
            .d0(data_a[i]),
            .d1(data_b[i]),
            .sel(sel[i]),
            .y(out[i])
        );
    end
endgenerate
```

Rules:

- `genvar` is only visible inside the `generate` block.
- The loop index `i` must be a constant expression — it is elaborated (evaluated) at compile time, not at runtime.
- Each iteration creates a separate hardware instance, named `mux_array[0].m`, `mux_array[1].m`, etc.
- Use `generate` when you find yourself copy-pasting the same instantiation with different indices.

The ALU



ALU: subtract inverts B and sets Cin; result-select mux chooses Sum, AND, OR, or SLT

The RV32I ALU performs: ADD, SUB, AND, OR, XOR, SLT (set-less-than), SLL/SRL/SRA (shifts):

ALU code

```
module alu #(parameter WIDTH = 32) (
    input  logic [WIDTH-1:0] a, b,
    input  logic [3:0]       op,       // ALU operation select
    output logic [WIDTH-1:0] result,
    output logic             zero      // 1 if result == 0 (used for BEQ)
);
    always_comb begin
        case (op)
            4'b0000: result = a + b;
            4'b0001: result = a - b;
            4'b0010: result = a & b;
            4'b0011: result = a | b;
            4'b0100: result = a ^ b;
            4'b0101: result = ($signed(a) < $signed(b)) ? 1 : 0; // SLT
            4'b0110: result = a << b[4:0];                       // SLL
            4'b0111: result = a >> b[4:0];                       // SRL
            4'b1000: result = $signed(a) >>> b[4:0];             // SRA
            default: result = '0;
        endcase
    end
    assign zero = (result == '0);
endmodule
```

`$signed(a) >>> b` is the **arithmetic right shift** — fills with the sign bit rather than zeros. SRA is the only arithmetic shift; the others are logical.

Barrel shifter

A **barrel shifter** shifts by any amount in a single cycle — required by RV32I's `SLL`, `SRL`, and `SRA` instructions:

```
module barrel_shift #(parameter WIDTH = 32) (
    input  logic [WIDTH-1:0]          a,      // value to shift
    input  logic [$clog2(WIDTH)-1:0]  shamt,  // shift amount (0–31 for WIDTH=32)
    input  logic [1:0]                mode,   // 2'b00=SLL, 2'b01=SRL, 2'b10=SRA
    output logic [WIDTH-1:0]          y
);
    always_comb begin
        case (mode)
            2'b00: y = a << shamt;                // SLL: logical left (zero-fill right)
            2'b01: y = a >> shamt;                // SRL: logical right (zero-fill left)
            2'b10: y = $signed(a) >>> shamt;      // SRA: arithmetic right (sign-fill)
            default: y = a;
        endcase
    end
endmodule
```

- `$clog2(32) = 5`, so `shamt` is a 5-bit field — matching `instr[24:20]` in RV32I I-type exactly.
- `<<` and `>>` are **logical** — vacated bits become 0.
- `>>>` on a `$signed` operand is **arithmetic** — vacated bits copy the sign bit.

Yosys synthesises this as a cascade of 5 conditional 2:1 MUXes (one per shift-amount bit). Depth is $O(\log_2 N)$ — a 32-bit shift costs only 5 MUX levels, not 32.

Register file

The RV32I register file: 32 registers × 32 bits, two read ports (combinational), one write port (clocked):

```
module regfile #(parameter REGS = 32, WIDTH = 32) (
    input  logic                          clk,
    input  logic [$clog2(REGS)-1:0]  ra1, ra2, wa,  // read/write addresses
    input  logic                          we,           // write enable
    input  logic [WIDTH-1:0]              wd,           // write data
    output logic [WIDTH-1:0]              rd1, rd2      // read data
);
    logic [WIDTH-1:0] regs [0:REGS-1];

    // Reads are combinational (asynchronous)
    assign rd1 = (ra1 == '0) ? '0 : regs[ra1];
    assign rd2 = (ra2 == '0) ? '0 : regs[ra2];

    // Write is synchronous
    always_ff @(posedge clk) begin
        if (we && wa != '0)
            regs[wa] <= wd;
    end
endmodule
```

- `$clog2(REGS)` — ceiling of $\log_2(32) = 5$ — the number of bits needed to address 32 registers.
- Register `x0` is **hardwired to zero** — read always returns 0, writes are ignored. Implemented by the `(ra == '0) ? '0 :` guard on reads and `wa != '0` guard on writes.

Memory arrays: ROM and SRAM

```
// Synchronous-read SRAM (maps to BRAM on GW1NR-9C)
module sram #(parameter DEPTH = 1024, WIDTH = 32) (
    input  logic                     clk,
    input  logic [$clog2(DEPTH)-1:0] addr,
    input  logic [WIDTH-1:0]         wdata,
    input  logic                     we,
    output logic [WIDTH-1:0]         rdata
);
    logic [WIDTH-1:0] mem [0:DEPTH-1];

    always_ff @(posedge clk) begin
        if (we) mem[addr] <= wdata;
        rdata <= mem[addr];    // read registered: output valid on next cycle
    end
endmodule
```

- **Asynchronous read** (combinational): `assign rdata = mem[addr]` — uses LUTs on the FPGA (distributed RAM). Limited size.
- **Synchronous read** (registered): `rdata <= mem[addr]` inside `always_ff` — maps to **BRAM** on the GW1NR-9C. The Tang Nano 9K has ~468 Kbit of BRAM — far more efficient for large memories.
- For the instruction memory in Project 1, use synchronous read BRAM to save LUTs for logic.

BRAM inference pitfalls

Yosys maps a synchronous-read array to dedicated BSRAM blocks. Two coding patterns silently break this and cause Yosys to fall back to thousands of LUTs:

Pitfall 1: async reset in the same process as array writes

```
// WRONG — async reset in write process → Yosys cannot infer BRAM
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) mem <= '{default: '0};   // reset collapses to LUT array
  else if (we) mem[addr] <= wdata;

// CORRECT — separate processes; memory never needs reset
always_ff @(posedge clk)
  if (we) mem[addr] <= wdata;
```

BRAM inference pitfalls

Pitfall 2: writing multiple memories in a single if/else chain

```
// WRONG — Yosys may not split this into separate BRAMs
always_ff @(posedge clk)
  if (sel_a) mem_a[addr] <= wdata;
  else if (sel_b) mem_b[addr] <= wdata;
  else mem_c[addr] <= wdata;

// CORRECT — one process per memory
always_ff @(posedge clk) if (sel_a) mem_a[addr] <= wdata;
always_ff @(posedge clk) if (sel_b) mem_b[addr] <= wdata;
always_ff @(posedge clk) if (sel_c) mem_c[addr] <= wdata;
```

- Diagnostic: run `yosys -p "synth_gowin ... stat"` on the module alone. If a memory that fits in 1–2 BRAMs shows 1000+ LUTs, a pitfall was triggered.

$signed() pitfall: zero-extend vs. sign-extend

`$signed()` reinterprets a bit vector as a signed (two's complement) value. A subtle bug arises when prepending a bit:

```
logic [7:0]  byte_val;   // raw bits, e.g., 8'hFF = 255 unsigned
logic [31:0] extended;

// BUG: prepending 1'b0 makes a 9-bit UNSIGNED value first,
//      then $signed() sees bit 8 = 0 → zero-extends to 32 bits
assign extended = $signed({1'b0, byte_val});   // always positive!

// CORRECT: apply $signed() directly to the 8-bit value
//          bit 7 is the sign → correctly sign-extends to 32 bits
assign extended = $signed(byte_val);           // 8'hFF → -1
```

- Rule: apply `$signed()` to the original narrower signal, not after concatenation.
- This same pitfall appears in C: `(int32_t)(uint8_t)x` zero-extends; `(int32_t)(int8_t)x` sign-extends.

Resource implications on the Tang Nano 9K

Component	Typical resource	Tang Nano 9K budget
32×32 register file (async read)	~64 LUTs	8,640 LUTs total
32×32 register file (sync read)	2 BRAM blocks	26 BRAM blocks total
32-bit ALU	~50 LUTs	(varies with operations)
4 KB instruction memory (BRAM)	1 BRAM block	26 BRAM blocks total
RV32I control FSM	~20–40 LUTs	(state + combinational)
Full minimal RV32I core	**~300–500 LUTs**	**well within budget**

The register file and memories are the largest pieces. Use BRAM for memories; use distributed RAM or LUTs for the register file (it is small enough).

In-class exercise — Yosys resource estimation

Before synthesizing, **estimate** the LUT cost of each module, then run Yosys and compare:

Module	Estimated LUTs	Yosys `LUT4:` count	Ratio
`mux2 #(32)` — 32-bit 2:1 MUX			
`barrel_shift #(32)` — all three modes			
`alu #(32)` — ADD/SUB/AND/OR/XOR/SLT/SLL/SRL/SRA			
`regfile` — 32×32, async read			

```
# Synthesize one module at a time:
yosys -p "read_verilog -sv barrel_shift.sv; \
          synth_gowin -top barrel_shift -json /dev/null" 2>&1 | grep "LUT4:"
```

Discussion questions:

1. Which module uses the most LUTs? Does that match your intuition?
2. The full RV32I core uses roughly 300–500 LUTs. How much headroom is left on the Tang Nano 9K (8,640 LUTs total) for the accelerator in Project 3?
3. Run with `synth_gowin -retime` — does the LUT count change? Why or why not?

Preview: these are Project 1's components

You now have all the hardware you need for the RV32I multicycle core:

Module	Purpose	Covered in
`adder` / `alu`	Arithmetic and logic operations	This class
`regfile`	32 × 32-bit register storage	This class
`sram`	Instruction and data memory	This class
`mux2` / `mux4`	Datapath input selection	M01A02
`register`	IR, PC, MDR, ALUOUT pipeline registers	M01A03
Control FSM	Generate all control signals	M01A04

Module 2 will introduce the RV32I ISA and show you exactly how to connect these components. Project 1 is the wiring and the control FSM — not new hardware primitives.

Next class

Verification & Toolchain: writing Verilator testbenches, reading VCD waveforms in GTKWave, and the full OSS CAD Suite flow from SystemVerilog source to bitstream on the Tang Nano 9K — everything you need to test and deploy Project 1.