Assignment: **RVX10** — Add 10 New Single-Cycle Instructions to the RV32I Core

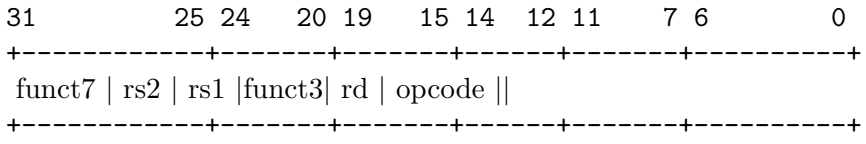
Why a "new" set? The base **RV32I** ISA already defines a fixed set of instructions. To practice ISA/RTL co-design without changing memory or pipeline structure, this assignment introduces a small, RV32I-*compatible* course extension called **RVX10**. It uses the RISC-V *reserved custom opcode* space so your core remains RV32I-compatible while adding 10 practical, single-cycle operations. You will implement decode, ALU logic, and tests end-to-end.

Rules & Success Criterion

- **No architectural state changes** beyond registers (no new CSRs, no flags).
- **Single-cycle** semantics (no stalls, no multi-cycle units).
- **Use only existing datapath blocks** (adder, shifter, comparator, basic logic). You may combine them (e.g., shift+OR for rotate).
- **Pass/Fail Harness:** At end of your test program, store the value **25** to memory address **100** ⇒ the provided testbench prints *"Simulation succeeded"*.

Encoding Overview (CUSTOM-0, R-type)

All RVX10 instructions use the R-type shape with **opcode = 0x0B** (binary `0001011`, "CUSTOM-0"). Unary ops ignore `rs2` (set it to `x0`).



Machine code (unsigned):
`inst = (funct7<<25)` (rs2«20) | (rs1«15) | (funct3«12) | (rd«7) | opcode|

The RVX10 Instruction Set (10 ops)

Name	Semantics (32-bit)	Type	funct7	funct3
ANDN	$rd = rs1 \,\&\, \sim rs2$	R	0000000	000
ORN	$rd = rs1 \mid \sim rs2$	R	0000000	001
XNOR	$rd = \sim (rs1 \oplus rs2)$	R	0000000	010
MIN	$rd = (\text{int32}(rs1) < \text{int32}(rs2))?rs1 : rs2$	R	0000001	000
MAX	$rd = (\text{int32}(rs1) > \text{int32}(rs2))?rs1 : rs2$	R	0000001	001
MINU	$rd = (rs1 < rs2)?rs1 : rs2$ (unsigned)	R	0000001	010
MAXU	$rd = (rs1 > rs2)?rs1 : rs2$ (unsigned)	R	0000001	011
ROL	$rd = (rs1 \ll s) \mid (rs1 \gg (32-s))$; s:=$rs2$[4:0]	R	0000010	000
ROR	$rd = (rs1 \gg s) \mid (rs1 \ll (32-s))$; s:=$rs2$[4:0]	R	0000010	001
ABS	$rd = (\text{int32}(rs1) \geq 0)?rs1 : -rs1$ (rs2=x0)	R	0000011	000

Notes.

- **Unary ops:** For `ABS`, encode as R-type with `rs2 = x0`; hardware ignores `rs2` when `funct7=0000011`.
- **Rotate by 0:** Define `ROL/ROR` with $s = 0$ to return `rs1` unchanged (avoid shifts by 32). Implement with a conditional.
- **ABS overflow (INT_MIN):** Two's complement wrap: `ABS(0x80000000) = 0x80000000`. No traps or flags.
- **x0:** Writes to `x0` must be ignored.

ALU/RTL Integration Guide

Decode

Add a case for `opcode == 7'b0001011`. Within it, select operation by `funct7` and `funct3` as per the table. Ensure register file read enables (both ports) and write-back enable for all RVX10 ops.

ALU

Add enumerations (e.g., `ALU_ANDN, ALU_MIN, ...`) and implement in the combinational block. Sketches:

```
// assume rs1_val, rs2_val are 32b
wire signed [31:0] s1 = rs1_val;
wire signed [31:0] s2 = rs2_val;
reg [31:0] alu_y;

always @* begin
  case (alu_op)
    ALU_ANDN: alu_y = rs1_val & ~rs2_val;
    ALU_ORN : alu_y = rs1_val | ~rs2_val;
    ALU_XNOR: alu_y = ~(rs1_val ^ rs2_val);

    ALU_MIN : alu_y = (s1 < s2) ? rs1_val : rs2_val;
    ALU_MAX : alu_y = (s1 > s2) ? rs1_val : rs2_val;
    ALU_MINU: alu_y = (rs1_val < rs2_val) ? rs1_val : rs2_val;
    ALU_MAXU: alu_y = (rs1_val > rs2_val) ? rs1_val : rs2_val;

    ALU_ROL : begin
      logic [4:0] sh = rs2_val[4:0];
      alu_y = (sh == 0) ? rs1_val : ((rs1_val << sh) | (rs1_val >> (32 - sh)));
    end
    ALU_ROR : begin
      logic [4:0] sh = rs2_val[4:0];
      alu_y = (sh == 0) ? rs1_val : ((rs1_val >> sh) | (rs1_val << (32 - sh)));
    end

    ALU_ABS : alu_y = (s1 >= 0) ? rs1_val : (0 - rs1_val);
    default : alu_y = 32'b0;
  endcase
end
```

Control/Datapath

All RVX10 ops are ALU-to-RD; no memory or PC changes. Ensure write-back mux selects ALU result. Keep branch/memory controls deasserted.

Worked Examples (Semantics)

- **ANDN** with `rs1=0xF0F0_A5A5`, `rs2=0x0F0F_FFFF` $\Rightarrow$
  $\sim rs2 = 0xF0F0\_0000$; result $= 0xF0F0\_A5A5 \,\&\, 0xF0F0\_0000 =$ **0xF0F0_0000**.
- **MINU** with `rs1=0xFFFF_FFFE` and `rs2=0x0000_0001`:
  Unsigned compare: $0xFFFF\_FFFE > 0x1$ so result = **0x0000_0001**.
- **ROL** by `rs2=3` and `rs1=0x8000_0001`:
  $(x << 3) = 0x0000\_0008$; $(x >> 29) = 0x0000\_0003$; OR $\Rightarrow$ **0x0000_000B**.
- **ABS** with `rs1=0xFFFF_FF80` (-128): result = **0x0000_0080**.

Encoding Table (Concrete)

Instr	opcode (hex)	funct7 (bin)	funct3 (bin)	rs2 usage
ANDN	0x0B	0000000	000	rs2
ORN	0x0B	0000000	001	rs2
XNOR	0x0B	0000000	010	rs2
MIN	0x0B	0000001	000	rs2
MAX	0x0B	0000001	001	rs2
MINU	0x0B	0000001	010	rs2
MAXU	0x0B	0000001	011	rs2
ROL	0x0B	0000010	000	rs2[4:0] for shamt
ROR	0x0B	0000010	001	rs2[4:0] for shamt
ABS	0x0B	0000011	000	ignored (set rs2=x0)

Manual encoding example (ANDN x5,x6,x7).

$$
\begin{aligned}
\text{funct7} &= 0b0000000 \Rightarrow 0x00 \ (\ll 25) \\
rs2 &= x7 = 7 \ (\ll 20) \\
rs1 &= x6 = 6 \ (\ll 15) \\
funct3 &= 0b000 \ (\ll 12) \\
rd &= x5 = 5 \ (\ll 7) \\
opcode &= 0x0B \\
\text{inst} &= (0x00 \ll 25) | (7 \ll 20) | (6 \ll 15) | (0 \ll 12) | (5 \ll 7) | 0x0B \\
&= \text{0x00E} \ldots \text{(compute full 32-bit hex).}
\end{aligned}
$$

(Provide such worked encodings in your submission.)

Test Plan & Deliverables

Self-checking strategy

For each op, compute a deterministic check value; accumulate into a checksum register (e.g., `x28`). End by storing **25** to address **100**. Store intermediate diagnostics to `[96 + 4*i]` if helpful.

Files to submit

- `src/riscvsingle.sv` (modified with RVX10 decode/ALU)
- `docs/ENCODINGS.md` (your exact bitfields + worked encodings)

- `docs/TESTPLAN.md` (per-op inputs, expected results)
- `tests/rvx10.hex` (the `$readmemh` image you run)
- `README.md` (how to build/run; any notes)

Checklist Before Submission

- Rotate by 0 returns `rs1`; no shift-by-32 in RTL.
- `ABS(INT_MIN)` returns `0x80000000`.
- `x0` writes are ignored.
- Final store writes **25** to address **100**.