VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
Faculty of Electrical and Electronic Engineering

MILESTONE 2 REPORT
COMPUTER ARCHITECTURE

TOPIC
DESIGN OF A SINGLE CYCLE RISC-V PROCESSOR

Instructional lecturer: **TRAN HOANG LINH**
Class: **L01**
Group: **23**
Academic year: **2025–2026**

Ho Chi Minh City, November 2025

HO CHI MINH CITY INTERNATIONAL UNIVERSITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
Faculty of Electrical and Electronic Engineering

MILESTONE 2 REPORT
COMPUTER ARCHITECTURE

TOPIC
DESIGN OF A SINGLE CYCLE RISC-V PROCESSOR

Group's members table

	Name	Student ID
1	Nguyen Thanh Phong	2312626
2	Vu Tien Tuan	2313774
3	Nguyen Hoang Tuan	2313746

Ho Chi Minh City, November 2025

Contents

1 Introduction

This project implements a **single-cycle** RV32I processor that can execute a compliance-style demo and drive on-board peripherals on the DE2 (Cyclone-II) board. The goals are:

- Correct RV32I semantics for all R/I/S/B/U/J instruction groups.
- Follow course constraints (e.g., avoid direct `+`/`-` and logic shifts on the datapath; build them structurally).
- Provide *memory-mapped I/O* (MMIO) for LEDs, seven-segment displays, and LCD.
- Keep RTL modular and verification-friendly.

Supported groups: R-type (ADD/SUB/AND/OR/XOR/SLL/SRL/SRA/SLT/SLTU), I-type ALU (ADDI/ANDI/ORI/XORI/SLLI/SRLI/SRAI/SLTI/SLTIU), LOAD/STORE (LB/LH/LW/LBU/LHU, SB/SH/SW), BRANCH (BEQ/BNE/BLT/BGE/BLTU/BGEU), JAL/JALR, LUI, AUIPC.

2 System Overview

2.1 Block Diagram and Datapath

The `single_cycle` RV32I core is implemented as a textbook single-cycle datapath with separate instruction memory (IMEM), register file, ALU/branch unit, control logic, and a combined memory/I/O block on the data side. For readability, the data memory (DMEM), load/store datapath, MMIO decoder, and output-peripheral registers (HEX/LEDR/LEDG/LCD, SW/KEY) are grouped into a single *LSU/MMIO* block in the diagram. The core still executes one instruction per clock cycle and exposes a debug PC plus MMIO registers for LEDs, seven-segment displays, switches, keys, and LCD.

To make the high-level structure clear, the top-level datapath is summarized in Figure 2.1. The `single_cycle` core reads instructions from a synchronous ROM, fetches operands from the register file, drives the ALU and BRU, issues load/store requests into the LSU/MMIO block, and receives either DMEM data or MMIO data depending on the address region.

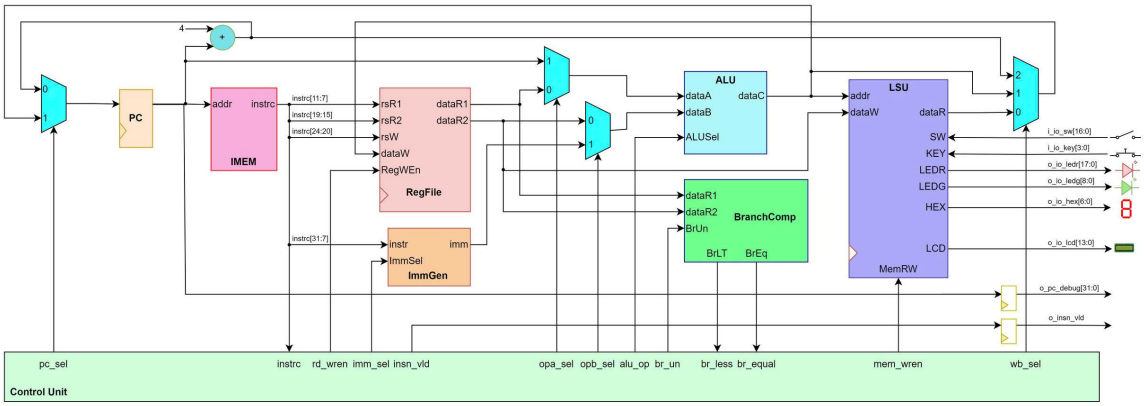

Figure 2.1: *Milestone-2 single-cycle core with IMEM and a combined LSU/DMEM/MMIO block.*

At the board level, the `singlecycle_wrapper` module bridges the DE2 pins (50 MHz clock, keys, switches, LEDs, seven-seg displays, LCD) to the core's abstract I/O. It divides the clock down to 25 MHz, synchronizes reset from `KEY0`, and wires the MMIO LED/HEX/LCD buses to the physical pins, as summarized later in Table 3.8.

2.2 PC and Control Flow

The program counter (PC) is stored in a 32-bit register and updated every cycle based on jump and branch decisions. The instruction ROM is indexed by `pc[31:2]` to enforce word alignment.

2.2.1 PC Update Equation

The next PC is computed according to the RV32I semantics:

$$\text{pc\_next} = \begin{cases} (\text{rs1} + \text{imm}_I) \wedge \sim 1, & \text{JALR} \\ \text{pc} + \text{imm}_J, & \text{JAL} \\ \text{pc} + \text{imm}_B, & \text{branch taken} \\ \text{pc} + 4, & \text{otherwise.} \end{cases}$$

The implemented priority is:

$$\textbf{JALR} > \textbf{JAL} > \textbf{branch-taken} > \textbf{PC+4},$$

with JALR explicitly clearing the least significant bit to keep the PC aligned.

2.2.2 Branch Decision

Branch instructions use the dedicated branch comparison unit (`brc`) and `funct3` to decide whether to take the branch. The control logic decodes BEQ/BNE/BLT/BGE/BLTU/BGEU, sets `o_br_un` for signed/unsigned domain, and the BRU returns separate `less` and `equal` flags. The PC selection logic then combines `w_is_branch` and `w_take_branch` with the JAL/JALR flags to drive `pc_next`.

3 Module Specifications

3.1 Arithmetic Logic Unit (ALU)

3.1.1 Functionality

The 32-bit ALU implements the full RV32I arithmetic and logical operation set required for Milestone 2. It uses a ripple-carry adder/subtractor built from 1-bit full adders; logical operations (AND/OR/XOR) are performed in a separate `logic_ops` block, and shifts are implemented with a 5-stage barrel shifter (1/2/4/8/16) without using built-in shift operators.

Subtraction is realized as addition with inverted B and `cin=1`. Signed and unsigned comparisons (SLT/SLTU) reuse the subtractor flags via a small `compare` module. The shift amount is always taken from `i_op_b[4:0]`.

All ALU operations are selected by a compact 4-bit opcode `i_alu_op = {funct7[5], funct3}`, following the RV32I encoding style.

Figure 3.1 summarizes the internal datapath of the ALU. The operands `operand_a` and `operand_b` are broadcast to three sub-blocks:

- **add_sub_less**: ripple-carry adder/subtractor that also produces the "less" flag for SLT/SLTU.
- **shifter**: 5-stage structural shifter that performs SLL/SRL/SRA using the lower 5 bits of `operand_b`.
- **logic-ops**: bitwise AND/OR/XOR implemented purely from gates.

A final multiplexer selects one of these results based on `i_alu_op` to drive `o_alu_data`.

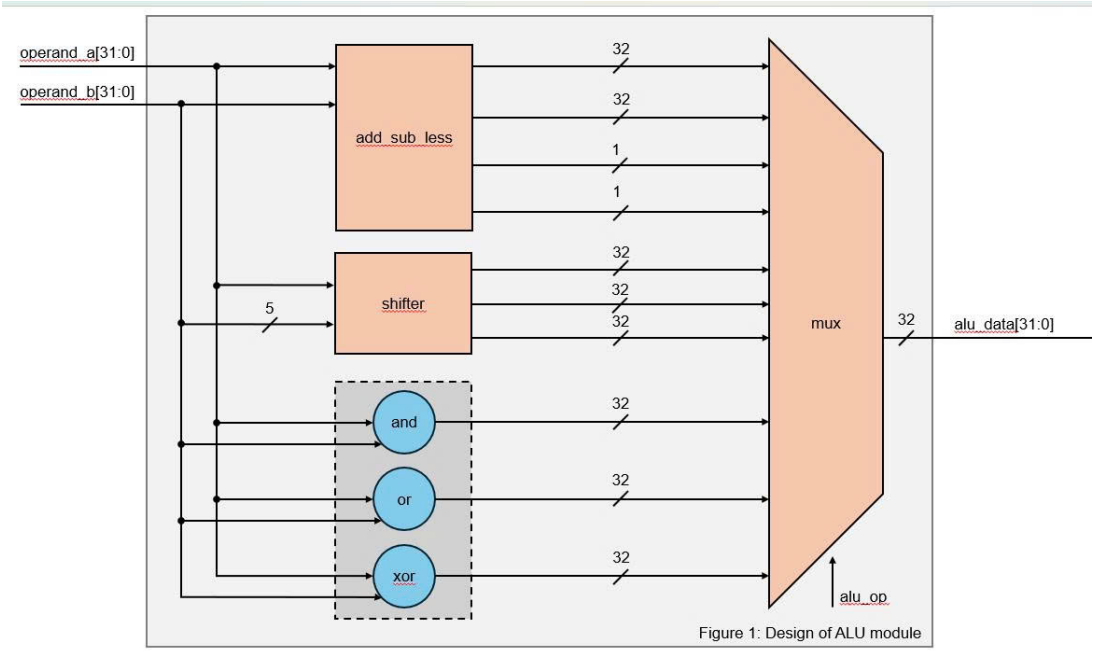

Figure 3.1: *ALU datapath with add/sub/less, shifter, and bitwise logic sub-blocks.*

3.1.2 I/O Interface

Signal	Width	Dir	Description
`i_op_a`	32	In	Operand A (rs1 or PC)
`i_op_b`	32	In	Operand B (rs2 or immediate)
`i_alu_op`	4	In	`{funct7[5], funct3}` opcode select
`o_alu_data`	32	Out	ALU result

Table 3.1: *ALU I/O specification.*

3.2 Branch Comparison Unit (BRU)

3.2.1 Functionality

The branch unit (`brc`) provides domain-correct comparisons for RV32I branch instructions:

- Unsigned less-than is implemented by a 32-bit ripple comparator.
- Equality is computed by XOR-reduction across all bits.
- Signed less-than uses sign-bit logic: if signs differ, negative $<$ positive; otherwise, it reuses the unsigned less-than result.

This structure avoids using high-level relational operators and is fully structural.

At the lowest level, a 1-bit cell (Figure 3.2) receives a pair of bits `a`, `b` together with "pre_less" and "pre_equal" flags coming from more-significant bits. It updates the flags according to:

$$\text{less}_i = \text{pre\_less} \mid (\text{pre\_equal} \;\&\; \sim a_i \;\&\; b_i), \qquad \text{equal}_i = \text{pre\_equal} \;\&\; \sim (a_i \oplus b_i).$$

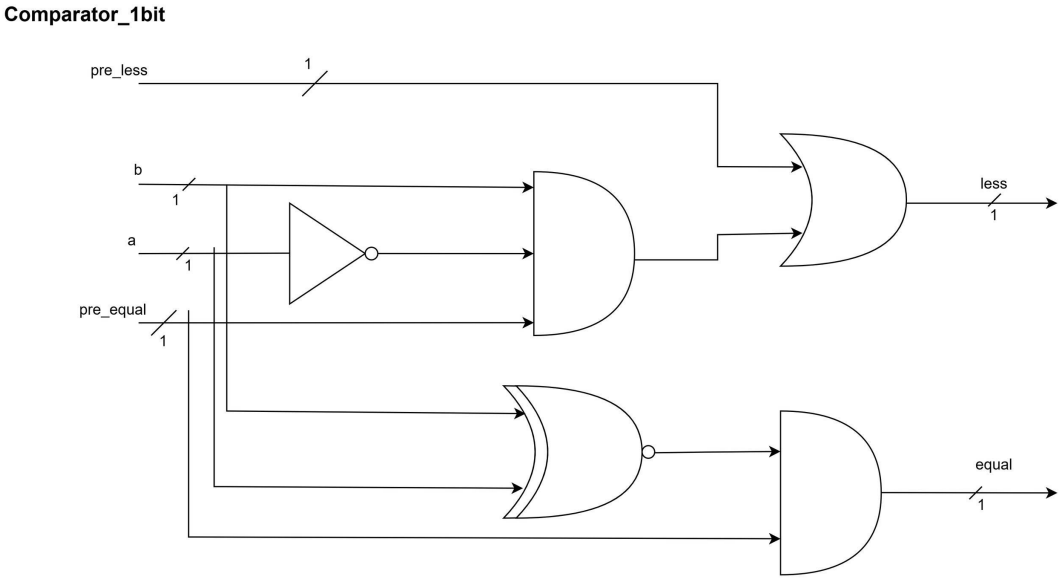

Figure 3.2: *1-bit comparator cell propagating* `pre_less` *and* `pre_equal` *flags.*

Thirty-two of these cells are chained to build the unsigned comparator, as shown in Figure 3.3. Bits are processed from MSB to LSB, so the final `less` and `equal` flags correspond to the full 32-bit unsigned comparison.

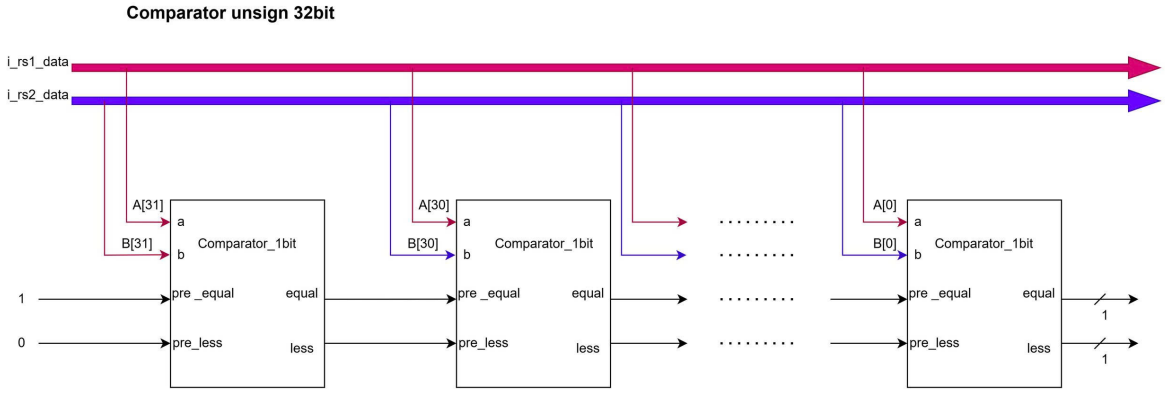

Figure 3.3: *32-bit unsigned comparator built as a ripple of 1-bit cells.*

Signed comparison (Figure 3.4) wraps the unsigned block. The sign bits of `i_rs1_data` and `i_rs2_data` are examined first:

- If the signs are equal, the unsigned comparator result is reused.
- If the signs differ, the negative operand is considered smaller.

Two multiplexers implement this rule and select the proper `less` flag while reusing the same `equal` output for both domains.

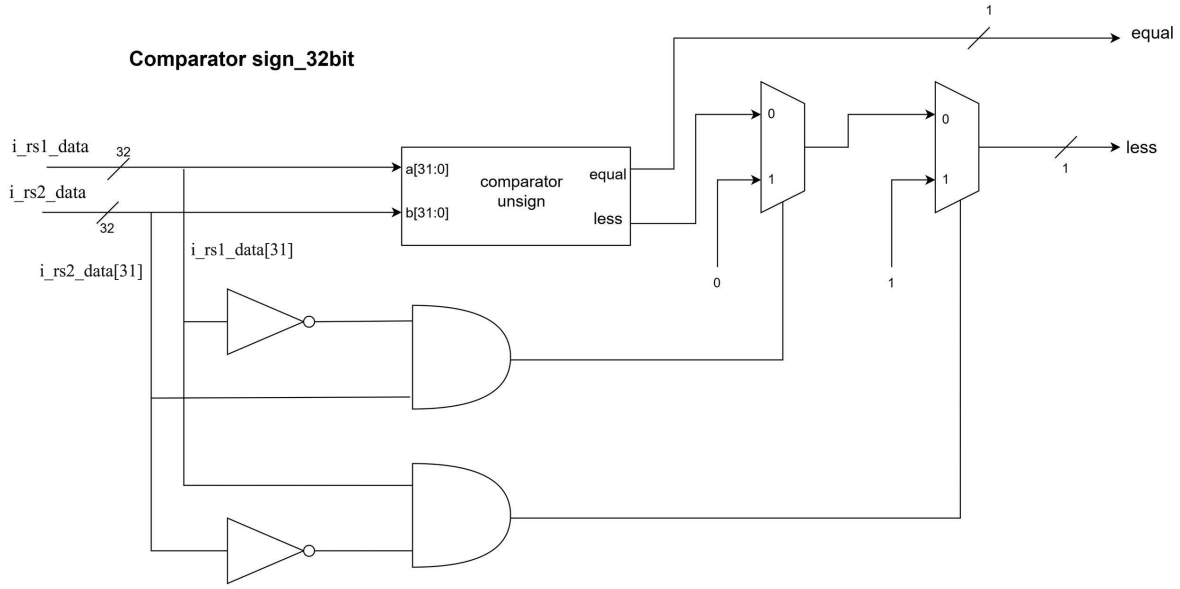

Figure 3.4: *Signed 32-bit comparator wrapping the unsigned comparator and sign-bit logic.*

3.2.2 I/O Interface

Signal	Width	Dir	Description
`i_rs1_data`	32	In	rs1 data
`i_rs2_data`	32	In	rs2 data
`i_br_un`	1	In	1: unsigned domain (BLTU/BGEU), 0: signed (BLT/BGE)
`o_br_less`	1	Out	A < B in selected domain
`o_br_equal`	1	Out	A == B

Table 3.2: *BRU I/O specification.*

3.3 Register File

3.3.1 Functionality

The register file is a 32x32, 2-read/1-write bank with synchronous write and asynchronous read. Register `x0` is hard-wired to zero by tying its storage to a constant 0 and blocking writes to address 0 at the decoder. A one-hot decoder drives per-register write enables, and two 32-to-1 multiplexers implement the read ports. An optional bypass path can be enabled via the `BYPASS_EN` parameter to handle read-after-write hazards in a single cycle.

Figure 3.5 illustrates the structural implementation of the 32x32 register file. A 5-to-32 decoder generates one-hot write enables, each feeding a 32-bit register cell. Two independent 32-to-1 multiplexers implement the asynchronous read ports for `rs1` and `rs2`. Register `x0` is tied to zero by forcing its data input to `32'b0` and disabling its write-enable.

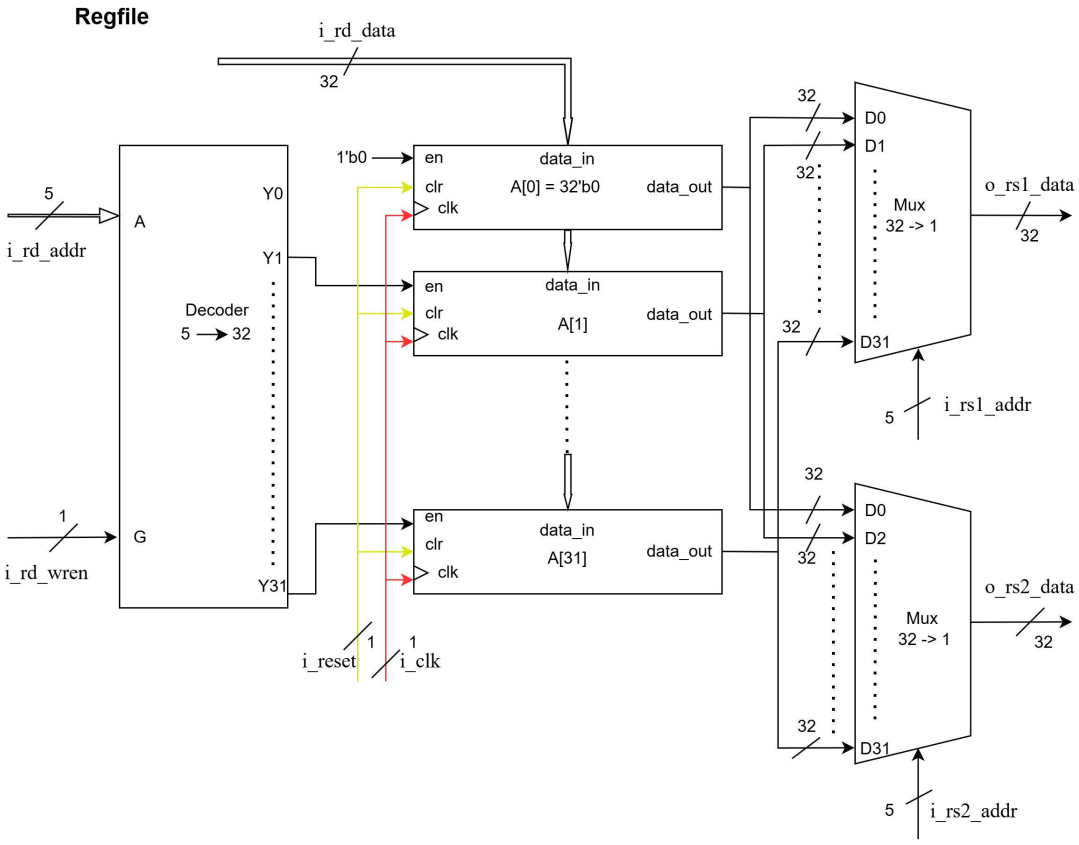

Figure 3.5: *Structural 32x32 register file with 1 write port and 2 read ports.*

3.3.2 I/O Interface

Signal	Width	Dir	Description
`i_clk`	1	In	Clock
`i_rst_n`	1	In	Active-low reset
`i_rd_wren`	1	In	Write enable
`i_rd_addr`	5	In	Write address (rd)
`i_rd_data`	32	In	Write data
`i_rs1_addr`	5	In	Read address 1
`i_rs2_addr`	5	In	Read address 2
`o_rs1_data`	32	Out	Read data 1
`o_rs2_data`	32	Out	Read data 2

Table 3.3: *Register file I/O specification.*

3.4 Immediate Generator (ImmGen)

3.4.1 Functionality

The immediate generator (`immgen`) extracts and sign-extends immediates for all five RV32I formats: I, S, B, U, and J. B- and J-type immediates are generated with the correct bit shuffles and enforced word alignment (least significant bit set to 0). The implementation follows the official RV32I bit layout exactly.

3.4.2 I/O Interface

Signal	Width	Dir	Description
i_instr	32	In	Raw instruction word
o_imm_I	32	Out	I-type immediate
o_imm_S	32	Out	S-type immediate
o_imm_B	32	Out	B-type immediate
o_imm_U	32	Out	U-type immediate
o_imm_J	32	Out	J-type immediate

Table 3.4: *Immediate generator outputs for RV32I formats.*

3.5 Load/Store Unit and MMIO (LSU/MMIO)

3.5.1 Functionality

The LSU/MMIO block implements the entire data-side subsystem grouped in Figure 2.1. Internally, it contains:

- A 2 KiB word-addressed data memory (DMEM) with a 32-bit data path.
- A load/store datapath that performs byte, half-word, and word accesses without using shift operators on the datapath.
- A page-based MMIO decoder that decides whether an address hits DMEM or a MMIO page.
- Memory-mapped registers for LEDs, HEX displays, LCD, switches, and key events.

For store instructions, a helper block `STORE_MERGE` constructs a word-sized byte-lane mask and merges the new byte/half/word into the old word returned from DMEM. For load instructions, `WORD_MISALIGNED` and `LOAD_EXTRACT` build the correct 32-bit value and perform sign/zero extension based on the instruction (LB/LH/LW/LBU/LHU).

Misaligned LW is supported by assembling a 32-bit result from two adjacent words using only slicing and concatenation, still avoiding shift operators. On MMIO addresses, the LSU/MMIO block forwards writes directly into the peripheral registers and multiplexes read data from the corresponding MMIO page instead of DMEM.

3.5.1.1 Top-level LSU/MMIO organization. Figure 3.6 and Figure 3.7 show the split between the top-level LSU/MMIO wrapper and the internal LSU core.

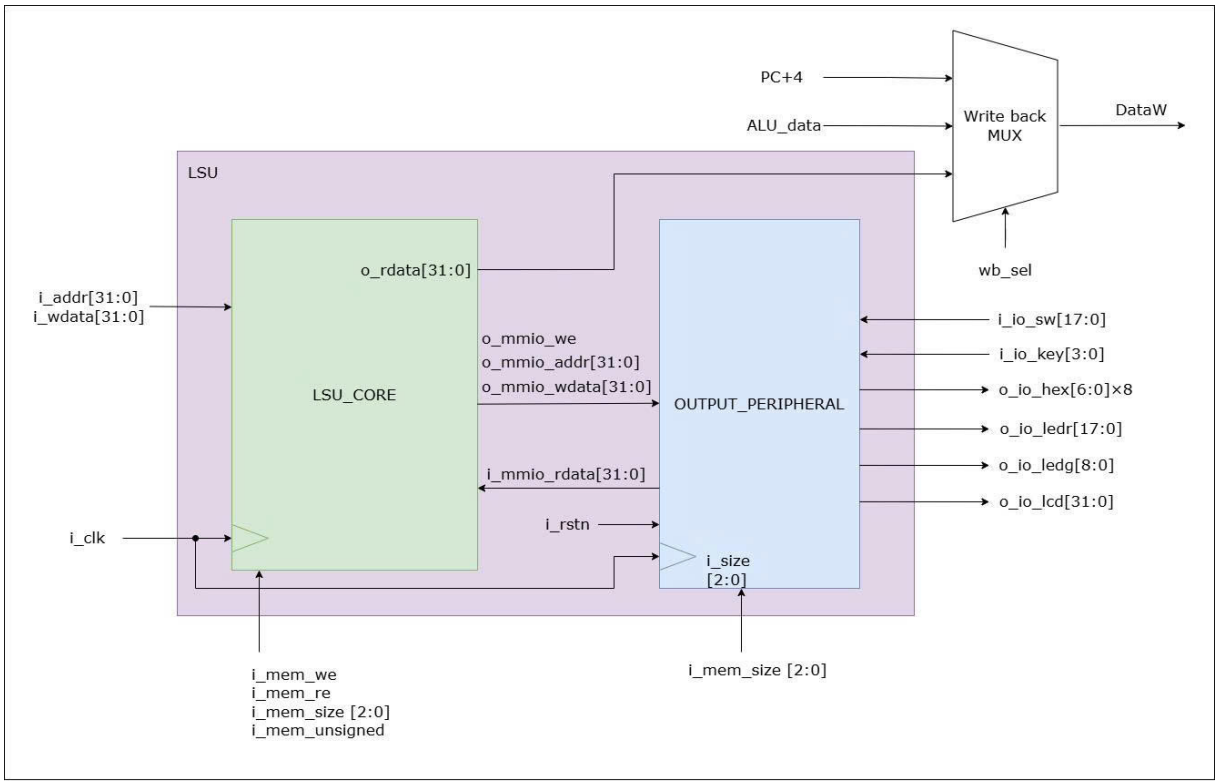

Figure 3.6: *Top-level LSU/MMIO wrapper:* `LSU_CORE` *(DMEM + load/store logic) and* `OUTPUT_PERIPHERAL` *connected to SW/KEY and LED/HEX/LCD.*

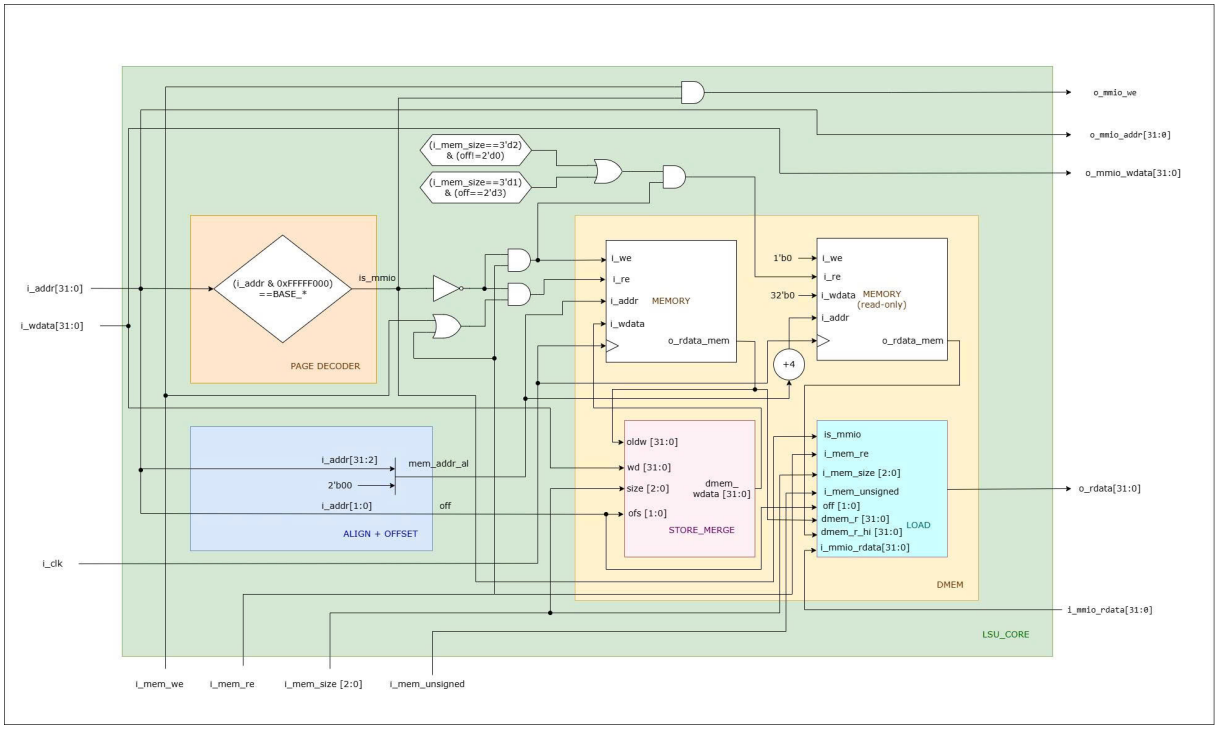

Figure 3.7: *Internal structure of* `LSU_CORE`*: page decoder, align+offset, DMEM memory macro, and store/load helper blocks.*

At a high level, the LSU core first checks whether the incoming address is MMIO or DMEM, then computes an aligned word address and a 2-bit byte offset. The aligned address drives the DMEM macro

and the store-merge logic, while the offset controls the load-extract and misaligned-LW blocks. A final multiplexer selects between the "DMEM load result" and the "MMIO load result" to form `o_rdata`.

3.5.1.2 Load datapath. The detailed load datapath is shown in Figure 3.8. For DMEM accesses, two helper blocks are used:

- `WORD_MISALIGNED` assembles a 32-bit word from two adjacent DMEM reads (`hi` and `lo`) when a misaligned LW crosses a word boundary.
- `LOAD_EXTRACT` selects the target byte/half/word based on `i_mem_size` and `addr[1:0]`, and performs sign or zero extension according to `i_mem_unsigned`.

For MMIO loads, the same `LOAD_EXTRACT` logic is reused, but the input word comes from the MMIO read-data path instead of DMEM.

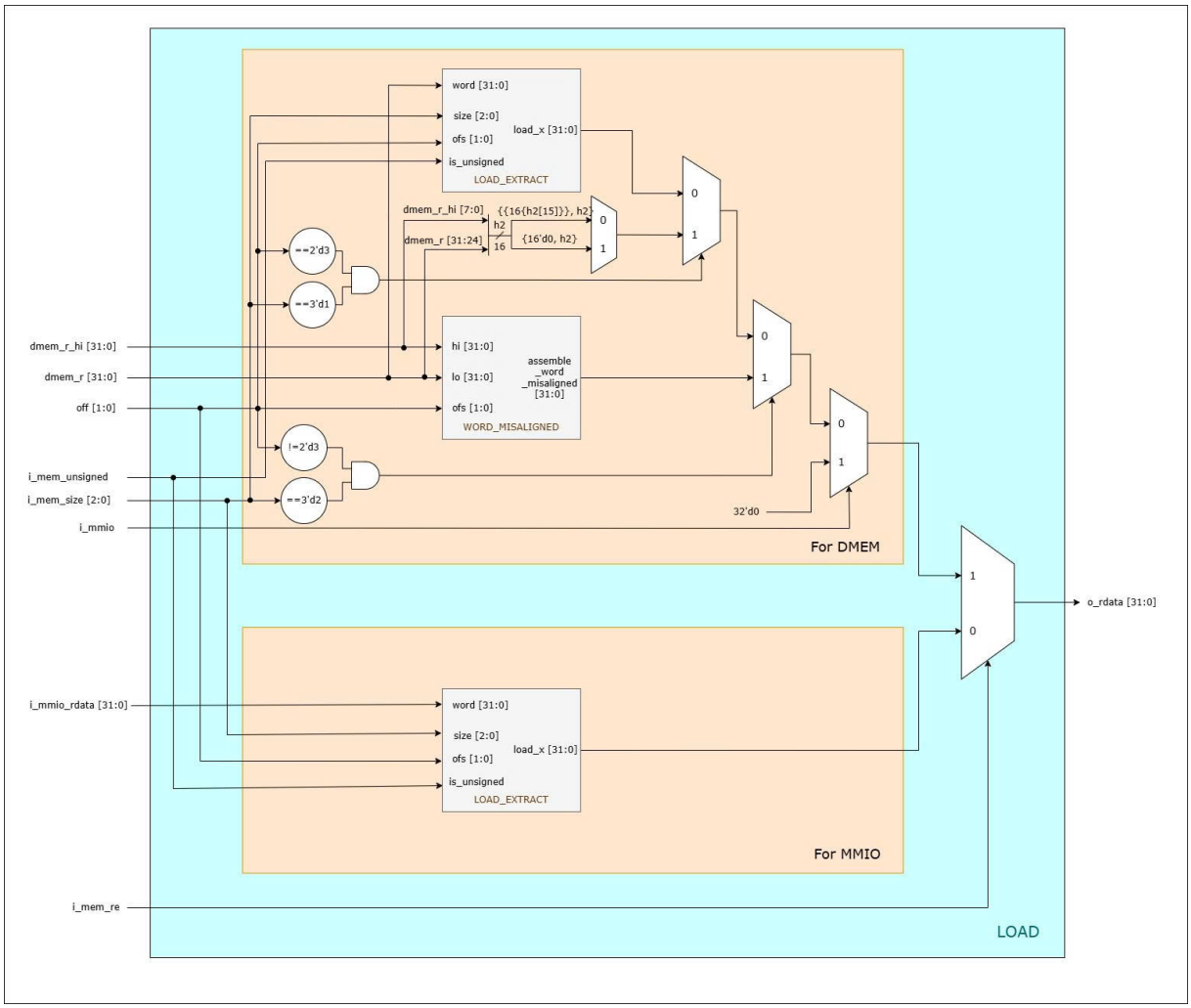

Figure 3.8: *Load datapath inside the LSU: DMEM path with misaligned-word handling and MMIO path sharing the same extract logic.*

3.5.1.3 Store path (`STORE_MERGE`). As shown in Figure 3.9, the store datapath never uses shift operators. The `STORE_MERGE` block takes the old 32-bit word from DMEM (`oldw`), the new store data (`wd`), the access size (`size[2:0]`: byte / half / word), and the byte offset `ofs[1:0]` and builds:

- A *mask* word that selects which byte lanes must be updated (e.g. `0x000000FF`, `0x0000FFFF`, `0xFFFFFFFF`), chosen by the upper multiplexers from `size` and `ofs`.

- A *data* word that replicates the input byte/half/word into the correct position using only concatenation and multiplexers (no shifts on the datapath).

The final DMEM write data is computed structurally as

$$\texttt{dmem\_wdata} = (\texttt{oldw} \,\&\, \texttt{!mask}) \;|\; (\texttt{data} \,\&\, \texttt{mask}),$$

so bytes outside the selected lanes are preserved, while SB/SH/SW update exactly one, two, or four byte lanes in the target word.

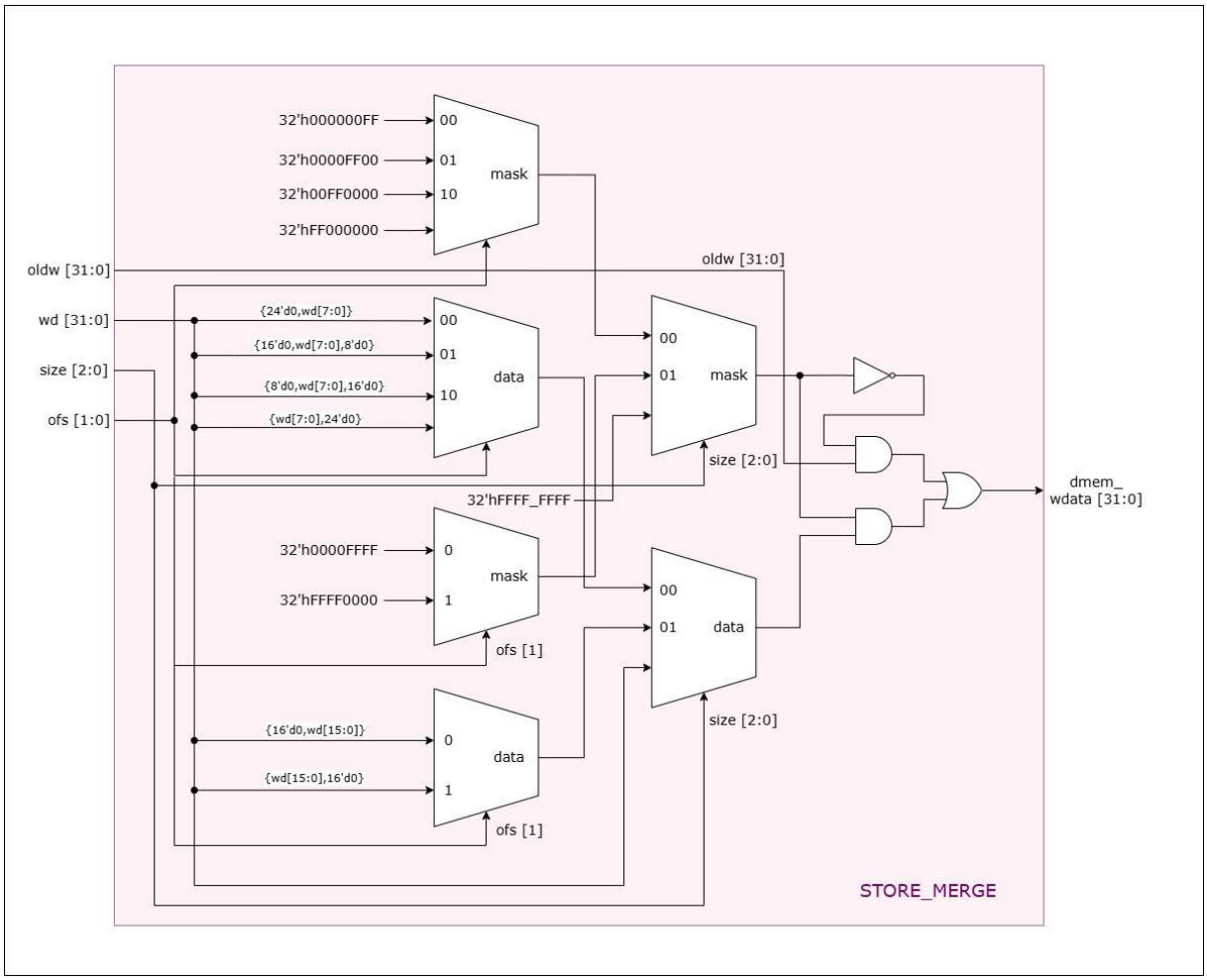

Figure 3.9: *STORE_MERGE: construct byte/half/word masks and merged write data for SB/SH/SW.*

3.5.1.4 Load helpers (`WORD_MISALIGNED` and `LOAD_EXTRACT`). For loads, the LSU splits the work into two helper blocks illustrated in Figure 3.10:

- `WORD_MISALIGNED` takes two adjacent 32-bit words (`hi` and `lo`) plus the byte offset `ofs[1:0]` and assembles a correctly rotated 32-bit value when a misaligned LW crosses a word boundary. The rotation is implemented entirely with multiplexers that select different byte groupings from `hi` and `lo`.
- `LOAD_EXTRACT` takes a 32-bit aligned word and uses `size[2:0]` and `ofs[1:0]` to select:
  - One of the four bytes (`word[7:0]`, `[15:8]`, `[23:16]`, `[31:24]`) for byte loads;
  - One of the two half-words (`word[15:0]` or `word[31:16]`) for half-word loads;
  - Or the full 32-bit word for LW.

– The selected byte/half is then sign-extended or zero-extended based on `is_unsigned` (LB/LH vs. LBU/LHU), again using concatenation and multiplexers instead of shift or arithmetic operators.

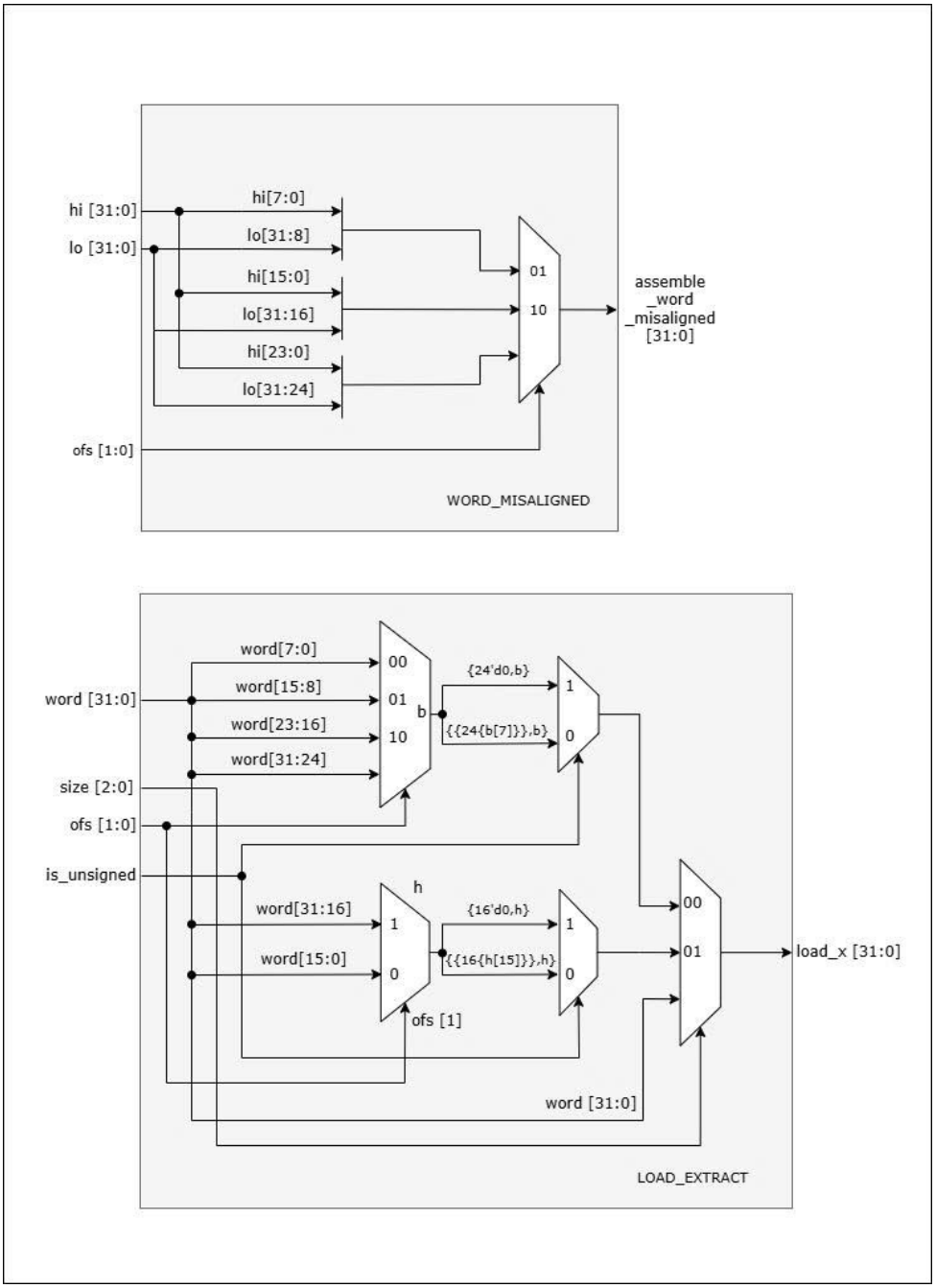

Figure 3.10: *`WORD_MISALIGNED` and `LOAD_EXTRACT`: assemble misaligned words and perform sign/zero extension for LB/LH/LW/LBU/LHU.*

Together, these helper blocks allow the LSU to implement LB/LH/LW/LBU/LHU and misaligned LW purely with combinational logic and byte slicing, while keeping the main LSU core schematic in Figure 3.7 relatively simple.

3.5.1.5 MMIO output peripherals. On the MMIO side, the LSU connects to a dedicated `output_peripherals` block that implements the memory-mapped registers for seven-segment displays (HEX), LEDR/LEDG, LCD, switches, and key events. From the core's point of view, these are just

MMIO words located in specific pages; all decoding of fields and display control is handled inside `output_peripherals`.

The block supports three display modes for the HEX outputs:

- **Manual HEX mode:** the CPU writes raw 7-segment patterns into `HEX_LO` and `HEX_HI`. Each nibble controls one digit directly, giving full control over the segments.
- ***AUTO_SW* mode:** the current switch value (from the `SW` MMIO page) is automatically converted to decimal and shown on the HEX displays, without software doing any explicit BCD conversion.
- ***AUTO_CPU* mode:** a 32-bit CPU-provided value in `HEX_BIN` is automatically converted to decimal and displayed. An optional "signed" bit enables showing a minus sign on `HEX7` when the value is negative.

The auto-decimal path reuses a sequential binary-to-BCD converter and an 8-digit 7-segment encoder, implemented in the `bin2bcd` and `hex_auto_glue` modules. The LSU simply forwards MMIO writes into these registers and returns their current contents on MMIO reads; all user-visible formatting of HEX/LED/LCD is encapsulated inside the `output_peripherals` block.

3.5.2 I/O Interface

Signal	Width	Dir	Description
`i_clk`	1	In	Clock
`i_mem_we`	1	In	Memory write enable
`i_mem_re`	1	In	Memory read enable
`i_mem_size`	3	In	0: byte, 1: half, 2: word
`i_mem_unsigned`	1	In	Unsigned load (LBU/LHU)
`i_addr`	32	In	Byte address
`i_wdata`	32	In	Store data
`o_rdata`	32	Out	Load data (from DMEM or MMIO)
`o_mmio_we`	1	Out	MMIO write enable
`o_mmio_addr`	32	Out	MMIO address
`o_mmio_wdata`	32	Out	MMIO write data
`i_mmio_rdata`	32	In	MMIO read data

Table 3.5: *LSU/MMIO I/O specification.*

3.6 Control Logic

3.6.1 Functionality

The control logic decodes the RV32I instruction fields (`opcode`, `funct3`, `funct7`) and generates:

- ALU operation code `o_alu_op`.
- Immediate selection via the ImmGen outputs.
- LSU size and unsigned flags for loads/stores.
- Branch domain (`o_br_un`) and branch type.
- JAL/JALR and generic branch enables.
- Write-back select between ALU, MEM, and special flows (LUI/AUIPC/JAL/JALR).

Write-back priority is implemented as:

$$\texttt{JALR/JAL} > \texttt{MEM} > \texttt{AUIPC} > \texttt{LUI} > \texttt{ALU},$$

ensuring that control-flow instructions always return `pc+4` to `rd`, while loads override ALU results when active.

3.6.2 I/O Interface

Signal	Width	Dir	Description
i_instr	32	In	Raw instruction
o_reg_we	1	Out	Register write enable
o_alu_op	4	Out	ALU opcode
o_br_un	1	Out	Unsigned branch domain
o_br_funct3	3	Out	Branch function (BEQ/BNE/...)
o_use_imm_b	1	Out	Use immediate as ALU B operand
o_mem_we	1	Out	Memory write enable
o_mem_re	1	Out	Memory read enable
o_mem_size	3	Out	Access size (B/H/W)
o_mem_unsigned	1	Out	Unsigned load
o_is_jal	1	Out	JAL instruction
o_is_jalr	1	Out	JALR instruction
o_is_branch	1	Out	Branch instruction
o_wb_sel_mem	1	Out	WB from MEM instead of ALU
o_is_lui	1	Out	LUI instruction
o_is_auipc	1	Out	AUIPC instruction

Table 3.6: *Control logic I/O specification.*

3.7 Single Cycle

3.7.1 Functionality

The `single_cycle` module is the top-level RV32I core used in Milestone 2. It instantiates all previously described building blocks: instruction memory, control logic, register file, immediate generator, ALU, branch comparator, LSU/MMIO, output peripherals, and an optional timer wrapper. At a high level, it implements the classic single-cycle datapath shown in Figure 2.1, where one instruction is fetched, decoded, executed, and written back in a single clock cycle.

The core is organized as follows:

- **PC and IMEM:** a 32-bit program counter `r_pc` is updated every cycle according to the PC update logic (JALR/JAL/branch/PC+4). The instruction memory `inst_mem` is a synchronous ROM indexed by `r_pc` and returns the 32-bit instruction `w_instr`.
- **Decode and register file:** the control unit `control_logic` decodes `w_instr` to generate all control signals (ALU opcode, branch type, LSU controls, WB selection, JAL/JALR flags, etc.). The register file decodes `rs1`, `rs2`, and `rd` from the instruction and provides two asynchronous read ports and one synchronous write port. Writes to `x0` are suppressed.
- **Immediate generation and ALU:** the `immgen` module produces the I/S/B/U/J immediates. The ALU operand A is selected between `r_pc` and `rs1` depending on whether the instruction is a branch/jump/AUIPC; operand B is selected between `rs2` and one of the immediates (I/S/B/U/J) according to the instruction type. The ALU then performs the requested arithmetic/logic operation and produces `w_alu_y`.
- **Branch comparison and PC update:** the branch comparator `brc` computes signed/unsigned less-than and equality between `rs1` and `rs2`. Using `funct3`, the core decides whether a branch is taken and applies the PC update priority (JALR > JAL > branch-taken > PC+4). The `add_sub` adder computes `pc+4` structurally.
- **LSU/MMIO and peripherals:** the LSU takes the effective address from `w_alu_y` and the store data from `rs2`, then either accesses DMEM or forwards the access into the MMIO space. The `output_peripherals` block implements the HEX/LED/LCD/SW/KEY registers and drives the board-facing signals. Internally, the LED buses are 18 bits (LEDR) and 9 bits (LEDG); the core widens them to 32 bits (`o_io_ledr`, `o_io_ledg`) for easier testbench observation.

- **Optional timer:** a small `timer_wrapper` instance shares the same MMIO address space. Its read data `w_timer_rdata` is selected over peripheral data `w_periph_rdata` when the address matches the `BASE_TIMER0` page. In the current demo, the timer is not used by software and does not appear in the datapath diagram.
- **Write-back and debug:** the write-back mux selects between ALU result, LSU/MMIO load data, and `pc+4` for JAL/JALR, following the write-back priority described earlier. The selected value `w_wd` is written into `rd` when `w_reg_we` is asserted. For verification and FPGA demo, the core also exposes `o_pc_debug = r_pc` and a constant `o_insn_vld = 1'b1` as a "valid instruction" strobe.

Overall, `single_cycle` ties together the datapath and control modules into a complete RV32I core that strictly obeys the single-cycle execution model and the structural-RTL constraints of the course.

3.7.2 I/O Interface

Signal	Width	Dir	Description
`i_clk`	1	In	Core clock (from `singlecycle_wrapper`)
`i_rstn`	1	In	Active-low synchronous reset
`i_io_sw`	32	In	Switch inputs (mapped from `SW[17:0]`)
`i_io_key`	4	In	Key inputs (mapped from `KEY[3:0]`)
`o_io_hex0`	7	Out	Seven-segment segments for HEX0 (active-low)
`o_io_hex1`	7	Out	Seven-segment segments for HEX1 (active-low)
`o_io_hex2`	7	Out	Seven-segment segments for HEX2 (active-low)
`o_io_hex3`	7	Out	Seven-segment segments for HEX3 (active-low)
`o_io_hex4`	7	Out	Seven-segment segments for HEX4 (active-low)
`o_io_hex5`	7	Out	Seven-segment segments for HEX5 (active-low)
`o_io_hex6`	7	Out	Seven-segment segments for HEX6 (active-low)
`o_io_hex7`	7	Out	Seven-segment segments for HEX7 (active-low)
`o_io_ledr`	32	Out	LEDR bus (internal 18 bits, zero-extended to 32 bits)
`o_io_ledg`	32	Out	LEDG bus (internal 9 bits, zero-extended to 32 bits)
`o_io_lcd`	32	Out	LCD MMIO data bus (for `singlecycle_wrapper`)
`o_pc_debug`	32	Out	Current program counter `r_pc` (for debug)
`o_insn_vld`	1	Out	Instruction valid strobe (constant 1 in single-cycle)

Table 3.7: *`single_cycle` core I/O specification.*

3.8 Top-Level Single Cycle Wrapper

3.8.1 Functionality

The `singlecycle_wrapper` module is the DE2-facing top level. It:

- Divides the 50 MHz board clock to a 25 MHz core clock.
- Synchronizes the active-low reset from `KEY0`.
- Connects switches and keys to the core's MMIO input buses.
- Maps the core's LEDR/LEDG/HEX/LCD outputs down to the physical pins.

3.8.2 I/O Interface

Signal	Width	Dir	Description
CLOCK_50	1	In	50 MHz system clock
KEY	4	In	KEY0 = active-low reset, others for demo control
SW	18	In	Switch inputs
LEDR	18	Out	Red LEDs
LEDG	9	Out	Green LEDs
HEX0..HEX7	7	Out	Seven-segment segments (active-low)
LCD_DATA	8	Out	LCD data bus
LCD_RS,RW,EN,ON	1 each	Out	LCD control lines

Table 3.8: *Top-level DE2 wrapper I/O summary.*

4 Implementation Notes

4.1 Datapath Constraints

All arithmetic is explicitly built from structural components:

- The 32-bit adder/subtractor uses a chain of 1-bit full adders, without + or - on the datapath.
- The barrel shifter is implemented as five layers of multiplexers for 1, 2, 4, 8, and 16-bit shifts.
- SLT/SLTU reuse subtractor flags (sign, overflow, borrow) instead of a separate comparator, though a separate branch comparator exists for branch decisions.

Immediate fields strictly follow the RV32I specification for I/S/B/U/J and are always sign-extended and aligned as required.

4.2 Memory and MMIO Organization

Instruction memory (inst_mem) is implemented as an 8 KiB ROM initialized from isa_4b.hex. The PC uses byte addresses, but the ROM index discards [1:0] to enforce word alignment.

Data memory is a 2 KiB word-addressed RAM with a 32-bit data path. The LSU/MMIO block computes a word-aligned address {addr[31:2], 2'b00} and an internal byte offset addr[1:0]. Stores use lane masks for SB/SH/SW; loads use extract/extend helpers and a second word read for misaligned LW. A page-based MMIO decode (using PAGE_MASK) forwards accesses in MMIO regions to the output-peripheral and SW/KEY registers instead of DMEM.

5 Verification Strategy

5.1 Unit-Level Verification

5.1.1 ALU

Directed and randomized tests cover all ALU ops (ADD/SUB, logical ops, shifts, SLT/SLTU), including edge values such as 0, INT_MIN, and INT_MAX. Overflow behavior is observed via the subtractor's overflow flag and compare outputs.

5.1.2 BRU

The BRU is tested with:

- Mixed-sign pairs (e.g., −1 vs. 0, 0 vs. -1).
- Same-sign ranges to ensure ordering is preserved.
- Equality patterns (all zeros, all ones, alternating bits).

5.1.3 LSU/MMIO

The LSU/MMIO block is stressed with:

- SB/SH/SW and LB/LH/LW/LBU/LHU across all four byte offsets.
- Misaligned LW cases that cross word boundaries.
- MMIO-only accesses to ensure DMEM is not touched.

5.2 Core-Level Verification

At the core level, an instruction ROM image (e.g., `isa_4b.hex`) is loaded into IMEM. The testbench monitors:

- `pc` and `insn_vld` to check the PC trace.
- MMIO writes to HEX/LED/LCD to detect PASS/FAIL patterns.
- Register file write-back (rd, data) to validate instruction semantics.

Branches, jumps, and memory operations are checked against a compact golden model. Writes to `x0` are explicitly verified to be ignored.

5.3 Methodology

Self-checking SystemVerilog testbenches are used, with assertions on:

- PC update priority (JALR/JAL/branch/PC+4).
- Write-back source selection (ALU vs. MEM vs. PC+imm).
- MMIO page decoding and LSU alignment behavior.

Waveform checkpoints are captured for the first 10–20 instructions to debug bring-up, after which longer randomized streams are run.

6 FPGA Demo on DE2

6.1 Implementation Flow

1. Assign DE2 pins for `CLOCK_50`, `KEY[3:0]`, `SW[17:0]`, `LEDR`, `LEDG`, `HEX0..HEX7`, and LCD signals in Quartus.
2. Add the IMEM `.hex` file and set it as a ROM initialization file (“Read During Compilation”).
3. Synthesize, fit, and program the DE2 board with the `singlecycle_wrapper` top-level.
4. Press `KEY0` (active-low) to reset and release the core into normal execution.

6.2 Adder/Subtractor Demo Using SW and KEY

For the Milestone 2 demo on the DE2 board, we implemented a simple accumulator-based adder/subtractor application:

- The 18 switches `SW[17:0]` encode an unsigned integer in binary.
- An accumulator register `S` lives in the CPU and is continuously displayed on the seven-segment displays via the HEX auto-decimal path.
- `KEY1` is mapped (through the MMIO KEY/KEY_EV registers) to an “add” event: when the key is pressed and released, the software reads the current switch value and computes

$$S \leftarrow S + \mathrm{SW\_value}.$$

- `KEY2` similarly generates a "subtract" event:

$$S \leftarrow S - \text{SW\_value}.$$

The software running on the `single_cycle` core uses MMIO to:

1. Read the current switch value from the `BASE_SW` page.
2. Detect key-release events via the `KEY_EV` register and clear them (write-1-to-clear).
3. Update the accumulator in a general-purpose register.
4. Write `S` into `HEX_BIN` and configure `HEX_CTRL` to $AUTO\_CPU$ mode, with the sign bit enabled so negative values show a minus sign on `HEX7`.

The `output_peripherals` block and `hex_auto_glue` then automatically convert `S` from binary to BCD and drive all eight HEX displays in active-low format, so the user sees the accumulator value directly in decimal while toggling switches and pressing `KEY1`/`KEY2` to accumulate or subtract.

The project files are available at:
`https://drive.google.com/drive/folders/1WvyIv2kPWU9NIynI8XhEQ4yQ1n6eaKKN?usp=drive_link`

7 Evaluation

7.1 Correctness

Unit-level and core-level tests cover all RV32I instruction groups required in Milestone 2, including:

- Arithmetic and logical R/I-type instructions.
- Load/store instructions with all byte offsets (including misaligned LW).
- Branch and jump instructions with both signed and unsigned comparisons.
- MMIO accesses to LEDs, HEX, LCD, timer, switches, and keys.

The design passes these tests, including corner cases such as mixed-signed branch comparisons, overflow observation, and misaligned loads.

7.2 Performance (Single-Cycle Limits)

Because this is a single-cycle design, the worst-case combinational path is:

$$\text{RegFile read} \rightarrow \text{ALU (SUB/SLT)} \rightarrow \text{branch compare / PC adder} \rightarrow \text{IMEM}.$$

The ripple-carry adder and subtractor dominate the latency, as expected. This sets an upper bound on the clock frequency. A deeper pipelined design with a more aggressive adder (e.g., carry-select or Kogge–Stone) could achieve a higher F_{max}, but that would violate the single-cycle constraint for Milestone 2.

7.3 Resource Usage (Quartus)

Metric	Value
Max Frequency (F_{max})	*29.17 MHz*
Logic Utilization (LEs)	*24,315 / 33,216 LEs (73%)*
Total Registers	*17,796*
Memory Bits	*0 / 483,840 bits (0%)*

Table 7.1: *Post-fit summary for the Milestone-2 single-cycle core on DE2.*

8 Future Work

Future extensions of this design include:

- Replacing the ripple adder with a faster carry-select or prefix adder to increase F_{max}.
- Switching to a synchronous-read register file with explicit bypassing to ease timing closure.
- Introducing an iterative (multi-cycle) shifter to reduce area if multi-cycle operations are allowed.
- Exploring tighter reuse of subtractor flags for both ALU comparisons and branch decisions to simplify verification.
- Migrating this Milestone-2 core into a 5-stage pipeline (Milestone 3) while preserving the MMIO interface and DE2 demo programs.

9 Conclusion

We implemented a clean, textbook-aligned single-cycle RV32I core with a practical MMIO subsystem and visible I/O on the DE2 board. The RTL strictly adheres to the course constraints (no high-level arithmetic or shift operators on the datapath), is modular for unit testing, and has been validated through both directed tests and an end-to-end DE2 demo (adder/subtractor using SW and KEY). This Milestone-2 design provides a solid foundation for future work such as pipelining, higher-performance adder structures, and more complex demo applications.

References

[1] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/-Software Interface, RISC-V Edition*. Morgan Kaufmann, 2017.