EE533 Lab 5 Report

Team 5 Members:
Jeremy Cai (yujiecai@usc.edu, 5988192173)
Prathamesh Hirekodi (hirekodi@usc.edu, 5404695932)
Yanzhe Li (yanzheli@usc.edu, 6884027371)

Part 1 - Extending Synchronous Adder into an ALU

1.1 ALU OVERVIEW

We implemented a parameterized 64-bit synchronous ALU (data width set by DATA_WIDTH = 64 in define.v; meets "at least 32-bit" requirement). The ALU is split into:

- **ksa.v**: Kogge–Stone adder building block;
- **addsub.v**: Performs addition/subtraction using KSA-based datapath; generates signed overflow for ADD/SUB;
- **alu.v**: Combinational ALU that selects among arithmetic, bitwise, compare, and shift functions;
- **sync_alu.v**: Registers the ALU outputs (Z and overflow) on the rising clock edge (synchronous output stage);

Overflow definition: Signed overflow is reported only for ADD and SUB; for all other ops, overflow outputs 0.

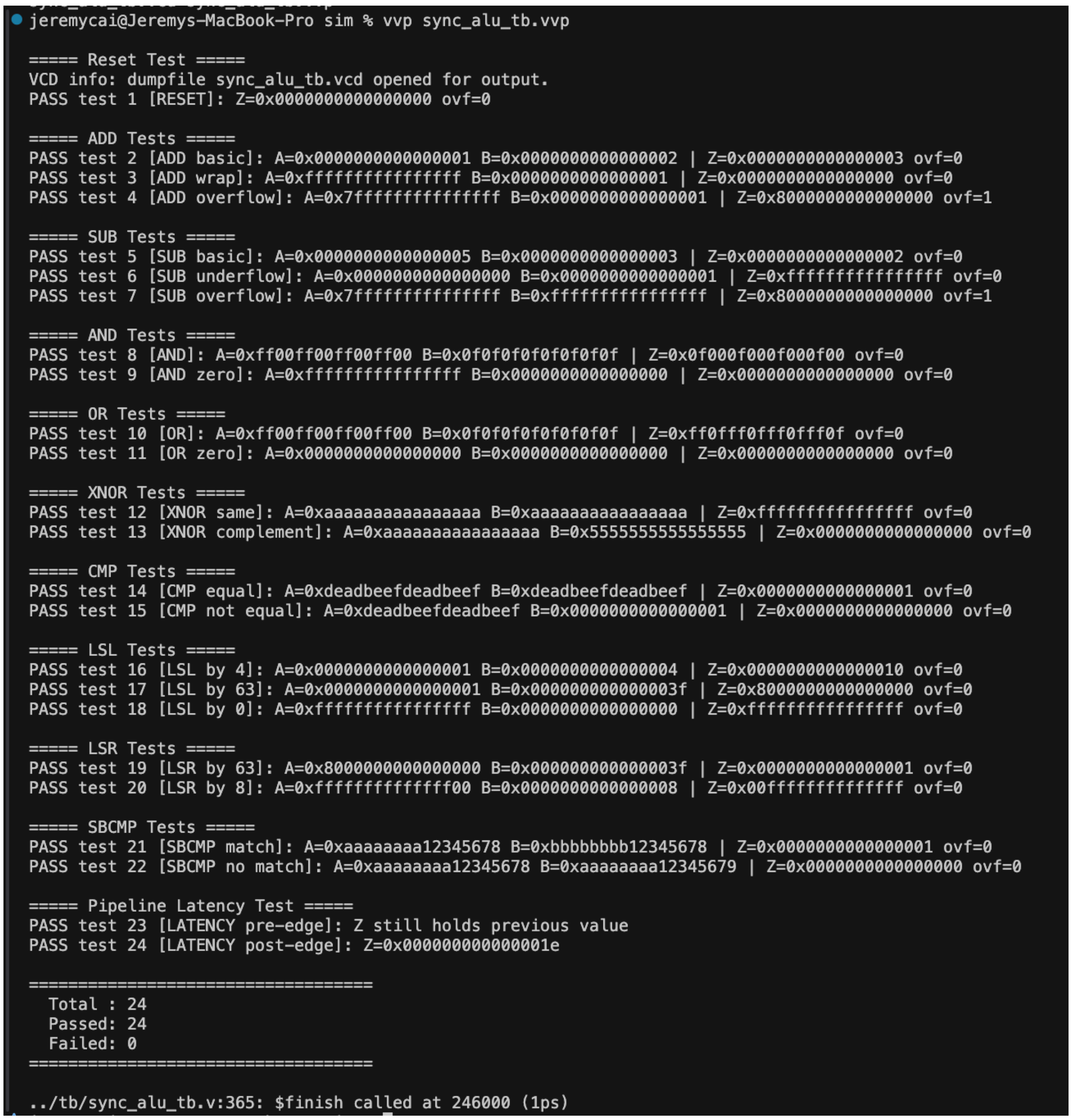
1.2 ALU CONTROL TABLE

	A	B	C	D
1	**aluctrl (4-bit)**	**Mnemonic**	**Operation**	**Notes**
2	4'b0000	ADD	A + B	signed overflow valid
3	4'b0001	SUB	A - B	signed overflow valid
4	4'b0010	AND	A & B	
5	4'b0011	OR	A | B	
6	4'b0100	XNOR	~(A ^ B)	
7	4'b0101	CMP	(A == B) ? 1 : 0	result is 1-bit in LSB (zero-extended)
8	4'b0110	LSL	A << B[5:0]	logical shift left
9	4'b0111	LSR	A >> B[5:0]	logical shift right
10	4'b1000	SBCMP	(A[31:0] == B[31:0]) ? 1 : 0	extra function (network-oriented)
11	4'b1001	LSTC	(reserved)	defined but not used
12	4'b1010	RSTC	(reserved)	defined but not used

1.3 SIMULATION AND VERIFICATION

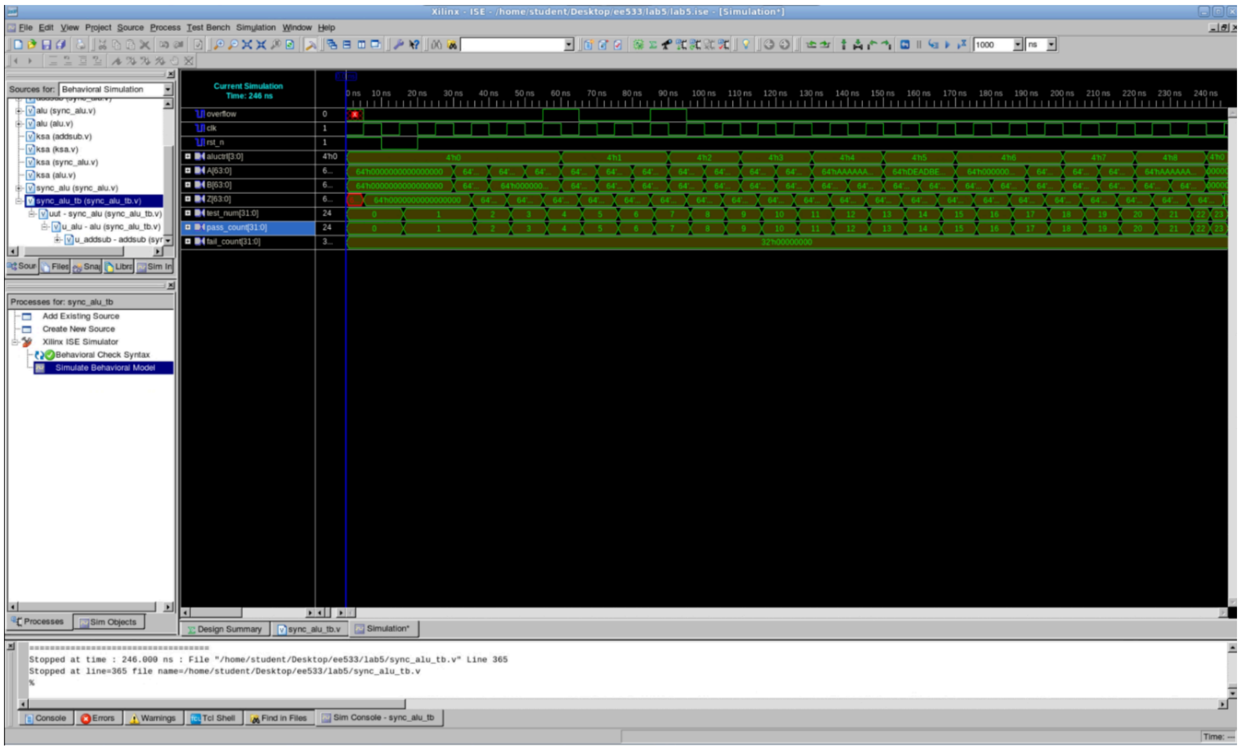
Testbench: sync_alu_tb.v
Verified items:

- **Reset behavior:** registered outputs clear to 0;
- **ADD/SUB:** normal cases + signed overflow corner cases;
- **AND/OR/XNOR/CMP/LSL/LSR/SBCMP:** representative vectors;
- **Latency check:** `Z/overflow` reflect inputs **exactly one clock cycle later** (registered output).



```
jeremycai@Jeremys-MacBook-Pro sim % vvp sync_alu_tb.vvp

===== Reset Test =====
VCD info: dumpfile sync_alu_tb.vcd opened for output.
PASS test 1 [RESET]: Z=0x0000000000000000 ovf=0

===== ADD Tests =====
PASS test 2 [ADD basic]: A=0x0000000000000001 B=0x0000000000000002 | Z=0x0000000000000003 ovf=0
PASS test 3 [ADD wrap]: A=0xffffffffffffffff B=0x0000000000000001 | Z=0x0000000000000000 ovf=0
PASS test 4 [ADD overflow]: A=0x7fffffffffffffff B=0x0000000000000001 | Z=0x8000000000000000 ovf=1

===== SUB Tests =====
PASS test 5 [SUB basic]: A=0x0000000000000005 B=0x0000000000000003 | Z=0x0000000000000002 ovf=0
PASS test 6 [SUB underflow]: A=0x0000000000000000 B=0x0000000000000001 | Z=0xffffffffffffffff ovf=0
PASS test 7 [SUB overflow]: A=0x7fffffffffffffff B=0xffffffffffffffff | Z=0x8000000000000000 ovf=1

===== AND Tests =====
PASS test 8 [AND]: A=0xff00ff00ff00ff00 B=0x0f0f0f0f0f0f0f0f | Z=0x0f000f000f000f00 ovf=0
PASS test 9 [AND zero]: A=0xffffffffffffffff B=0x0000000000000000 | Z=0x0000000000000000 ovf=0

===== OR Tests =====
PASS test 10 [OR]: A=0xff00ff00ff00ff00 B=0x0f0f0f0f0f0f0f0f | Z=0xff0fff0fff0fff0f ovf=0
PASS test 11 [OR zero]: A=0x0000000000000000 B=0x0000000000000000 | Z=0x0000000000000000 ovf=0

===== XNOR Tests =====
PASS test 12 [XNOR same]: A=0xaaaaaaaaaaaaaaaa B=0xaaaaaaaaaaaaaaaa | Z=0xffffffffffffffff ovf=0
PASS test 13 [XNOR complement]: A=0xaaaaaaaaaaaaaaaa B=0x5555555555555555 | Z=0x0000000000000000 ovf=0

===== CMP Tests =====
PASS test 14 [CMP equal]: A=0xdeadbeefdeadbeef B=0xdeadbeefdeadbeef | Z=0x0000000000000001 ovf=0
PASS test 15 [CMP not equal]: A=0xdeadbeefdeadbeef B=0x0000000000000001 | Z=0x0000000000000000 ovf=0

===== LSL Tests =====
PASS test 16 [LSL by 4]: A=0x0000000000000001 B=0x0000000000000004 | Z=0x0000000000000010 ovf=0
PASS test 17 [LSL by 63]: A=0x0000000000000001 B=0x000000000000003f | Z=0x8000000000000000 ovf=0
PASS test 18 [LSL by 0]: A=0xffffffffffffffff B=0x0000000000000000 | Z=0xffffffffffffffff ovf=0

===== LSR Tests =====
PASS test 19 [LSR by 63]: A=0x8000000000000000 B=0x000000000000003f | Z=0x0000000000000001 ovf=0
PASS test 20 [LSR by 8]: A=0xffffffffffffff00 B=0x0000000000000008 | Z=0x00ffffffffffffff ovf=0

===== SBCMP Tests =====
PASS test 21 [SBCMP match]: A=0xaaaaaaaa12345678 B=0xbbbbbbbb12345678 | Z=0x0000000000000001 ovf=0
PASS test 22 [SBCMP no match]: A=0xaaaaaaaa12345678 B=0xaaaaaaaa12345679 | Z=0x0000000000000000 ovf=0

===== Pipeline Latency Test =====
PASS test 23 [LATENCY pre-edge]: Z still holds previous value
PASS test 24 [LATENCY post-edge]: Z=0x000000000000001e

===================================
  Total : 24
  Passed: 24
  Failed: 0
===================================

../tb/sync_alu_tb.v:365: $finish called at 246000 (1ps)
```

Figure 1. sync_alu_tb.vvp console output showing all tests passed



```
==================================
Stopped at time : 246.000 ns : File "/home/student/Desktop/ee533/lab5/sync_alu_tb.v" Line 365
Stopped at line=365 file name=/home/student/Desktop/ee533/lab5/sync_alu_tb.v
%
```

Figure 2. ISE/Vivado waveform showing A/B/aluctrl and the one-cycle-later update of Z/overflow

Part 2 - Register File and Memories

2.1 REGISTER FILE (REGFILE.V) + ILA PROBE PATH

Our register file implements:

- **Data width**: 64-bit (REG_DATA_WIDTH = 64);
- **Register count:** 8 registers (REG_ADDR_WIDTH = 3);
- **Ports:** two combinational read ports (r0addr→r0data, r1addr→r1data) and one synchronous write port (waddr/wdata/wena);
- **Write timing**: Write occurs on rising edge when wena=1;
- **Architectural note**: Register 0 is not hardwired to zero in RTL; in our tests we keep R0 = 0 by never writing to it;

To support internal observability (as recommended), we added a probe path:

- ILA probe: ila_cpu_reg_addr selects a register index; ila_cpu_reg_data outputs the corresponding register value.

In cpu.v, this probe value can be routed to ila_debug_data when ila_debug_sel[4]=1, enabling on-chip ILA capture and/or testbench checking without perturbing the pipeline.

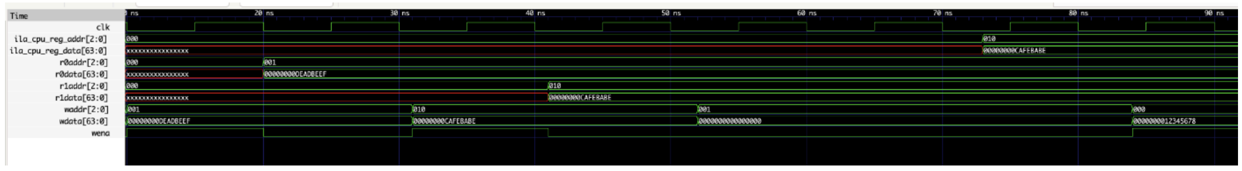


Figure 3. regfile read/write correctness (2 reads + 1 write) and ILA probe reads

2.2 DATA MEMORY (D-MEM)

- **Target hardware:** BRAM-based **dual-port synchronous** memory, **64-bit × 256** entries;
- Simulation model: test_d_mem.v;
- SoC hookup (current): host MMIO and CPU are muxed onto port A (mutual exclusion via req_rdy gating). Port B is reserved for future concurrent access.

2.3 INSTRUCTION MEMORY (I-MEM)

- Target hardware: BRAM-based single-port synchronous memory, 32-bit × 512 entries;
- Simulation model: test_i_mem.v.

Part 3 - Building the Pipeline Datapath (Memory-Interface Skeleton)

3.1 INSTRUCTION FORMAT USED BY OUR CPU

We use the lab's 32-bit "skeleton instruction":

- **WMemEn = instr[31]** (D-Mem write enable (store));
- **WRegEn = instr[30]** (Regfile write enable (load write-back));

Register fields (3-bit addressing, 8 regs):

- **Reg1 = instr[29:27]** (**Store-data** source register (used for store only));
- **Reg2 = instr[26:24]** (**Memory-address** source register (used for load & store));
- **WReg1 = instr[23:21]** (Load destination register);
- Unused = **instr[20:0]** (set to 0).

Operation meaning:

- **Load:** WMemEn=0, WRegEn=1 → read D-Mem at address in Reg2, write to WReg1;
- **Store:** WMemEn=1, WRegEn=0 → write Reg1 value to D-Mem at address in Reg2;
- **NOP:** both enables 0

3.2 PIPELINE STAGES (IF / ID / EX / MEM / WB)

This lab stage implements the memory + regfile interface skeleton (no ALU integration yet; EX is pass-through):

- **IF:** PC (pc.v) supplies I-Mem address. I-Mem is synchronous, so instruction becomes valid after a clock;
- **IF/ID:** pipeline register captures fetched instruction for decode;
- **ID:** decode WMemEn/WRegEn and register indices; regfile outputs operands;
- **EX:** pass-through stage (placeholder for future ALU integration);
- **MEM:** drive D-Mem address/data/write-enable based on decoded control;
- **WB:** for loads, D-Mem read data is written back into regfile at the destination index.

Program termination: cpu_done asserts when PC == 9'h1FF. The remaining I-Mem contents are NOPs (0) to avoid side effects.

3.3 TEST PROGRAM AND EXPECTED RESULTS

Initial D-Mem contents (from d_mem.coe):

- d_mem[0] = 4
- d_mem[4] = 100
- others = 0 / don’t care

I-Mem program (from i_mem.coe / cpu_tb.v initialization):

- Load d_mem[R0] → R2
- Load d_mem[R0] → R3
- NOPs (to allow pipeline write-back to complete)
- Store R2 → d_mem[R3]

Expected final state:

- d_mem[0] remains 4
- d_mem[4] changes from 100 → 4
- registers reflect the loaded values by the time the store executes

3.4 HIGH-LEVEL DATAPATH DIAGRAM

Please note that we made two modifications to the pipeline design provided in the lab manual. In the manual, the instruction memory and data memory is treated as asynchronous-read components, while in BRAM, both read and write are synchronous. Hence, we 1) create a new dummy pipeline stage after instruction memory due to bram configuration 2) data memory output bypasses the MEM/WB pipeline register so that the data is aligned with the control signals. In Lab 6, we will re-evaluate the configuration and re-design the pipeline if necessary.

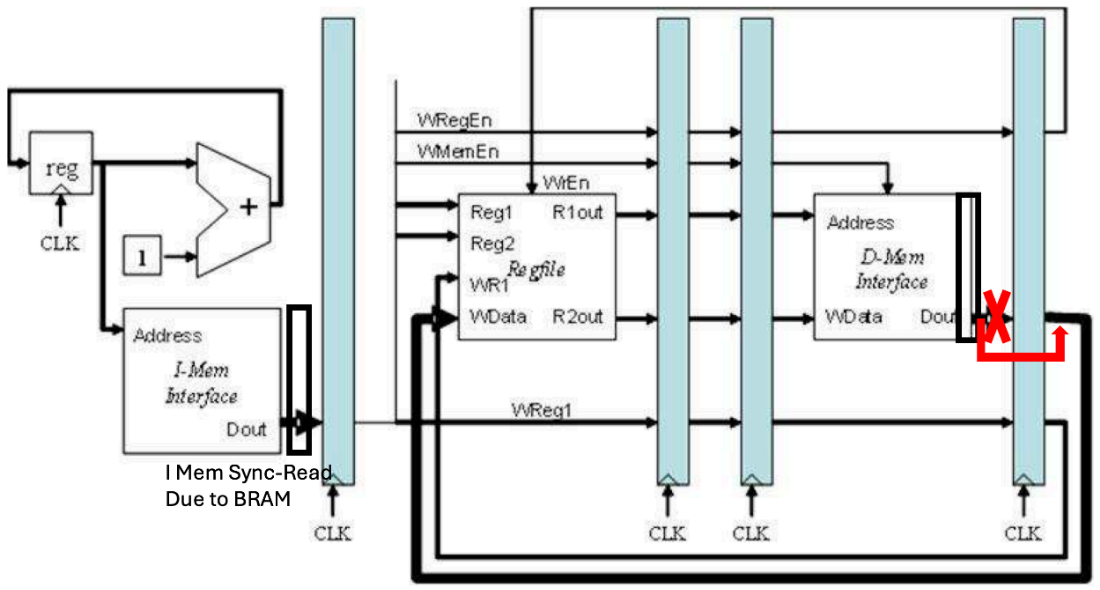
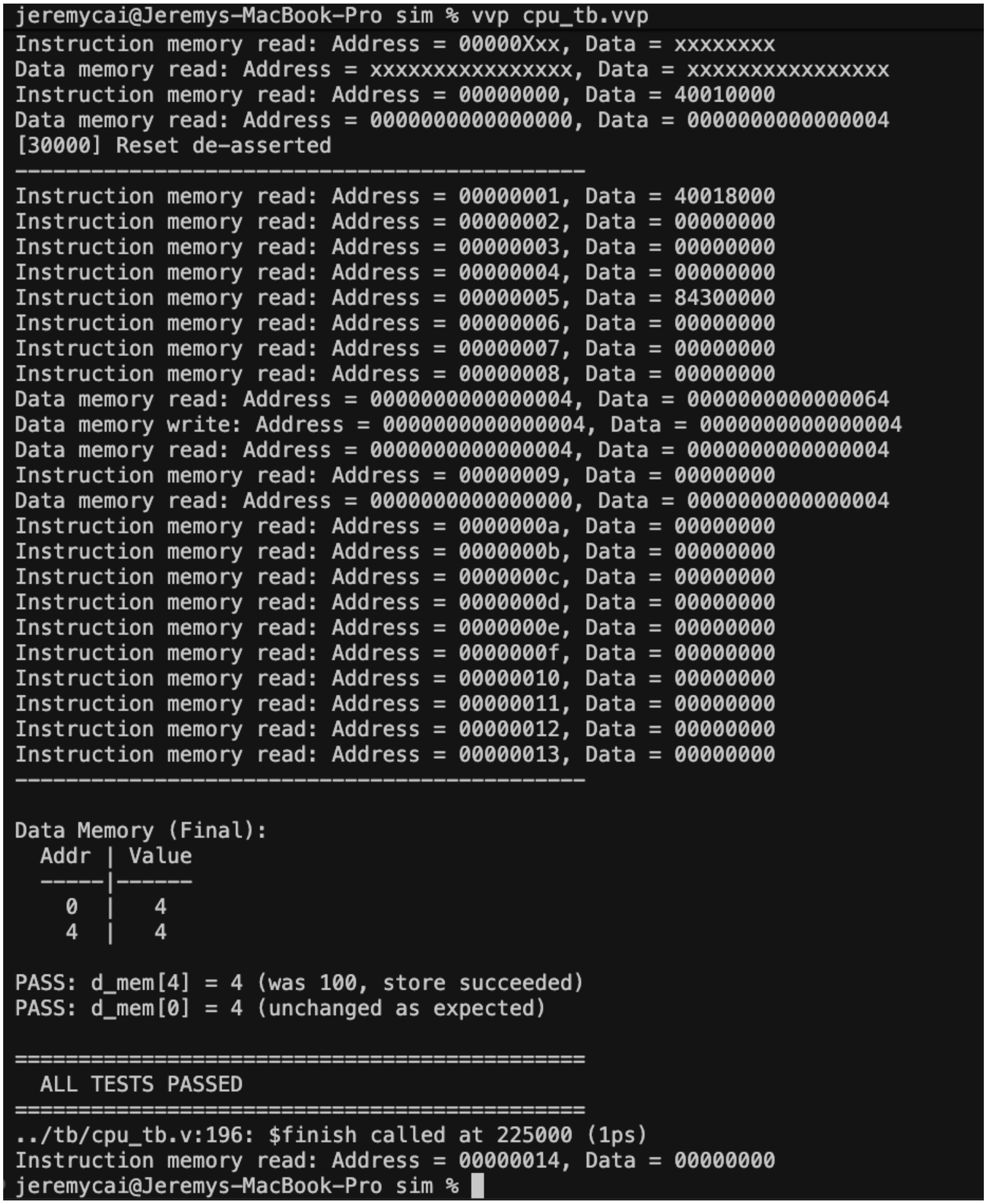


Figure 4. PC → I-Mem → IF/ID → RegFile → (ID/EX, EX/MEM) → D-Mem → MEM/WB → RegFile write-back, with control enables

3.5 SIMULATION RESULTS

We validated the pipeline using:

- cpu_tb.v (CPU-only functional check)
- soc_tb.v (SoC + MMIO programming + CPU run + post-check + ILA observation + random MMIO stress tests)



```
jeremycai@Jeremys-MacBook-Pro sim % vvp cpu_tb.vvp
Instruction memory read: Address = 00000Xxx, Data = xxxxxxxx
Data memory read: Address = xxxxxxxxxxxxxxxx, Data = xxxxxxxxxxxxxxxx
Instruction memory read: Address = 00000000, Data = 40010000
Data memory read: Address = 0000000000000000, Data = 0000000000000004
[30000] Reset de-asserted
---------------------------------------------
Instruction memory read: Address = 00000001, Data = 40018000
Instruction memory read: Address = 00000002, Data = 00000000
Instruction memory read: Address = 00000003, Data = 00000000
Instruction memory read: Address = 00000004, Data = 00000000
Instruction memory read: Address = 00000005, Data = 84300000
Instruction memory read: Address = 00000006, Data = 00000000
Instruction memory read: Address = 00000007, Data = 00000000
Instruction memory read: Address = 00000008, Data = 00000000
Data memory read: Address = 0000000000000004, Data = 0000000000000064
Data memory write: Address = 0000000000000004, Data = 0000000000000004
Data memory read: Address = 0000000000000004, Data = 0000000000000004
Instruction memory read: Address = 00000009, Data = 00000000
Data memory read: Address = 0000000000000000, Data = 0000000000000004
Instruction memory read: Address = 0000000a, Data = 00000000
Instruction memory read: Address = 0000000b, Data = 00000000
Instruction memory read: Address = 0000000c, Data = 00000000
Instruction memory read: Address = 0000000d, Data = 00000000
Instruction memory read: Address = 0000000e, Data = 00000000
Instruction memory read: Address = 0000000f, Data = 00000000
Instruction memory read: Address = 00000010, Data = 00000000
Instruction memory read: Address = 00000011, Data = 00000000
Instruction memory read: Address = 00000012, Data = 00000000
Instruction memory read: Address = 00000013, Data = 00000000
---------------------------------------------

Data Memory (Final):
  Addr | Value
  -----|------
    0  |   4
    4  |   4

PASS: d_mem[4] = 4 (was 100, store succeeded)
PASS: d_mem[0] = 4 (unchanged as expected)

=============================================
  ALL TESTS PASSED
=============================================
../tb/cpu_tb.v:196: $finish called at 225000 (1ps)
Instruction memory read: Address = 00000014, Data = 00000000
jeremycai@Jeremys-MacBook-Pro sim %
```

Figure 5. cpu_tb console log showing store success and final d_mem[4] == 4

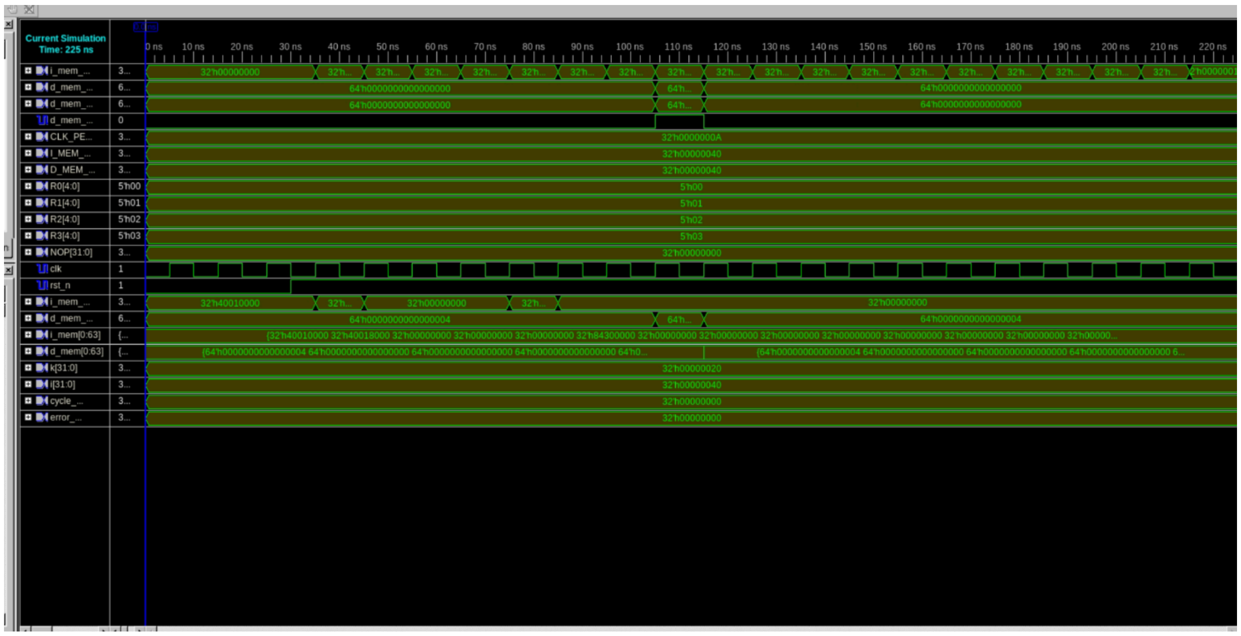
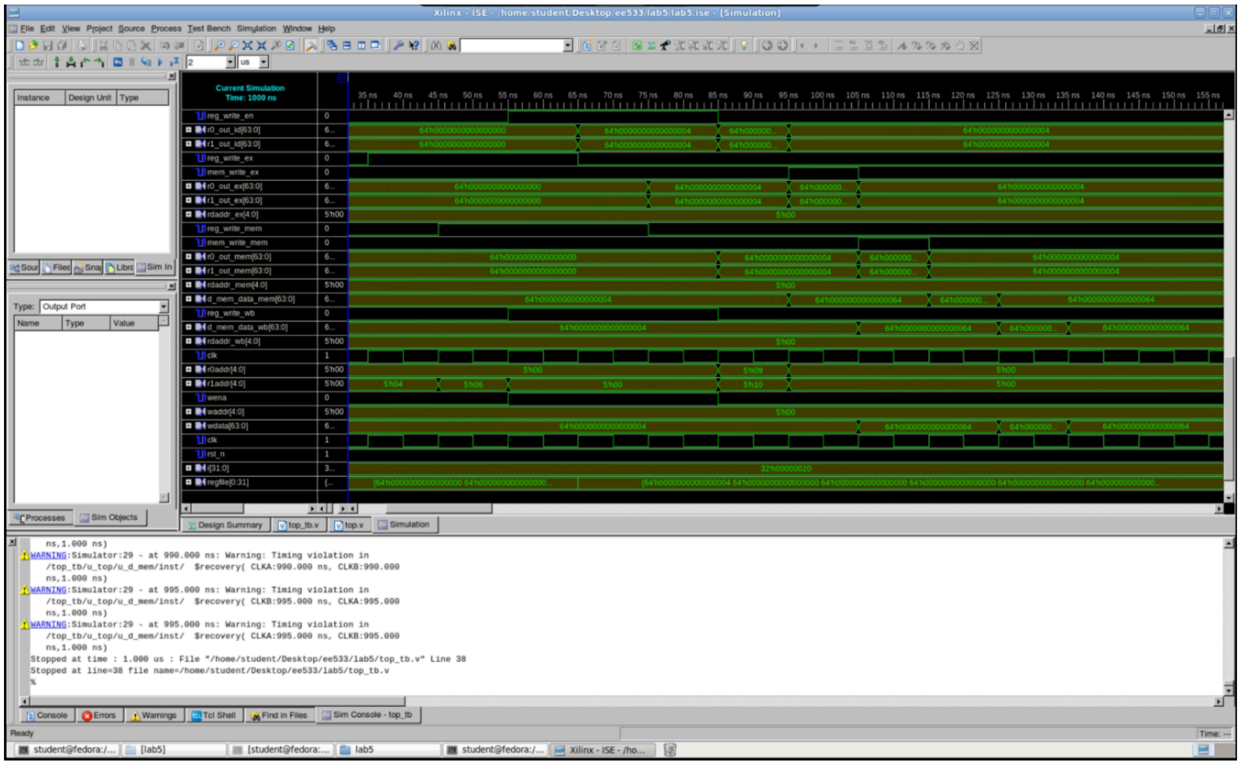


Figure 6. cpu_tb/ISE showing instruction fetch, write-back completing before the store, and d_mem[4] updating to 4



```
    ns,1.000 ns)
WARNING:Simulator:29 - at 990.000 ns: Warning: Timing violation in
    /top_tb/u_top/u_d_mem/inst/  $recovery( CLKA:990.000 ns, CLKB:990.000
    ns,1.000 ns)
WARNING:Simulator:29 - at 995.000 ns: Warning: Timing violation in
    /top_tb/u_top/u_d_mem/inst/  $recovery( CLKB:995.000 ns, CLKA:995.000
    ns,1.000 ns)
WARNING:Simulator:29 - at 995.000 ns: Warning: Timing violation in
    /top_tb/u_top/u_d_mem/inst/  $recovery( CLKA:995.000 ns, CLKB:995.000
    ns,1.000 ns)
Stopped at time : 1.000 us : File "/home/student/Desktop/ee533/lab5/top_tb.v" Line 38
Stopped at line=38 file name=/home/student/Desktop/ee533/lab5/top_tb.v
%
```

Figure 7. Simulation results with Memory IP .coe file initialization

Part 4 Integrating the Pipeline into NetFPGA (SoC + MMIO)

4.1 PLACEMENT AND INTERACTION

Per the handout, the processor subsystem is controlled through the **NetFPGA register interface**. In our codebase, this is realized by:

- soc.v: the CPU + memories + request/response MMIO channel + isolation logic;
- soc_driver.v: bridges SW/HW register transactions to the SoC MMIO channel (and exposes status/counters);
- top.v: ties the SW/HW register interface, soc_driver, soc, and debug blocks together.

4.2 Memory programming interface

soc.v implements a request/response MMIO channel to program/read I-Mem & D-Mem and to start the CPU.
Request/response signals:

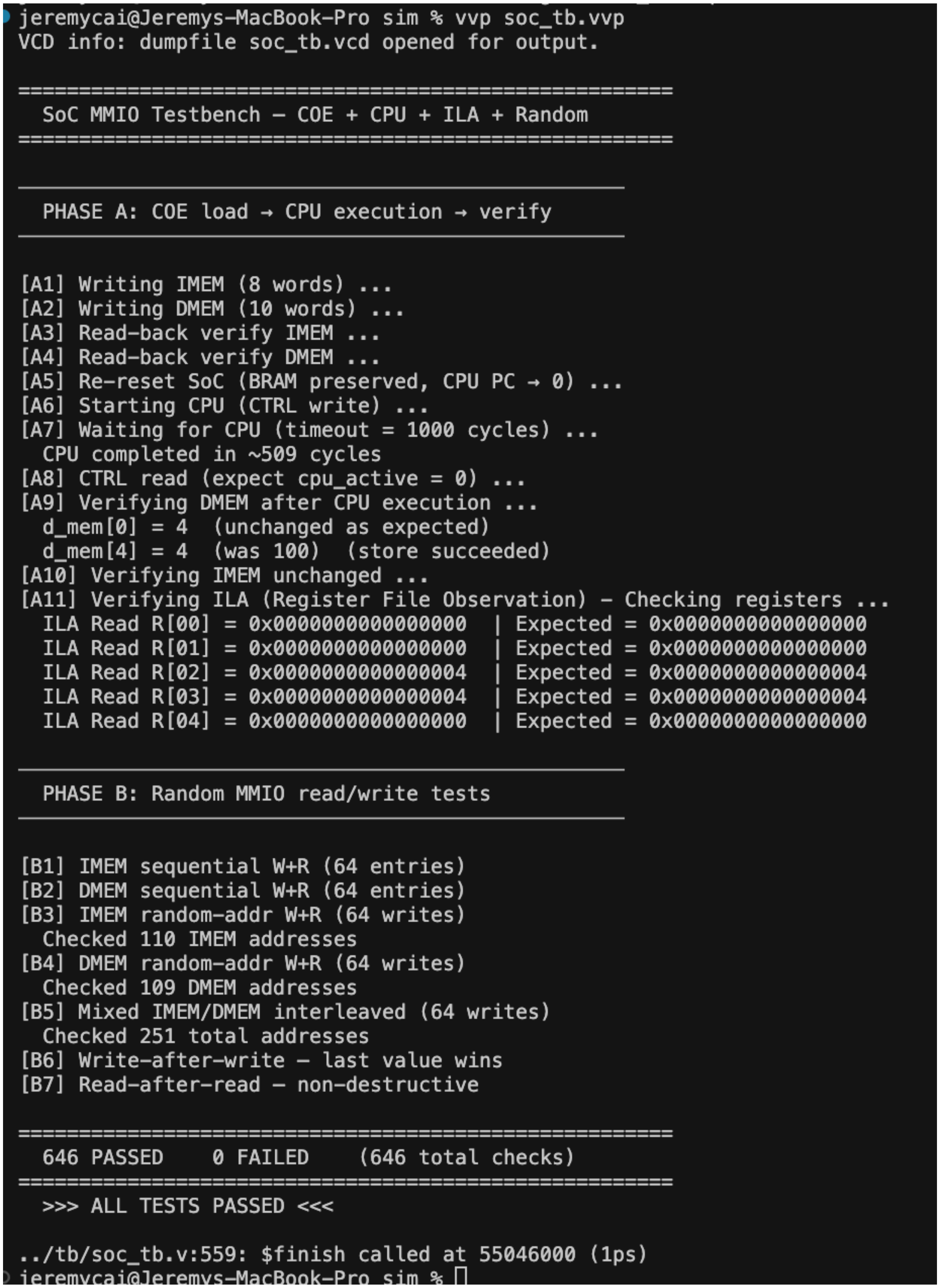
- Request: req_cmd (0=read, 1=write), req_addr, req_data, req_val, req_rdy
- Response: resp_cmd, resp_addr, resp_data, resp_val, resp_rdy

Address map decode (req_addr[31:30]):

- 00: IMEM space (512 entries, index uses low PC_WIDTH bits)
- 01: CTRL space (write triggers CPU start; read reports active state)
- 10: DMEM space (256 entries, index uses low DMEM_ADDR_WIDTH bits)
- 11: reserved (returns 0)

Isolation / concurrency policy (as implemented):

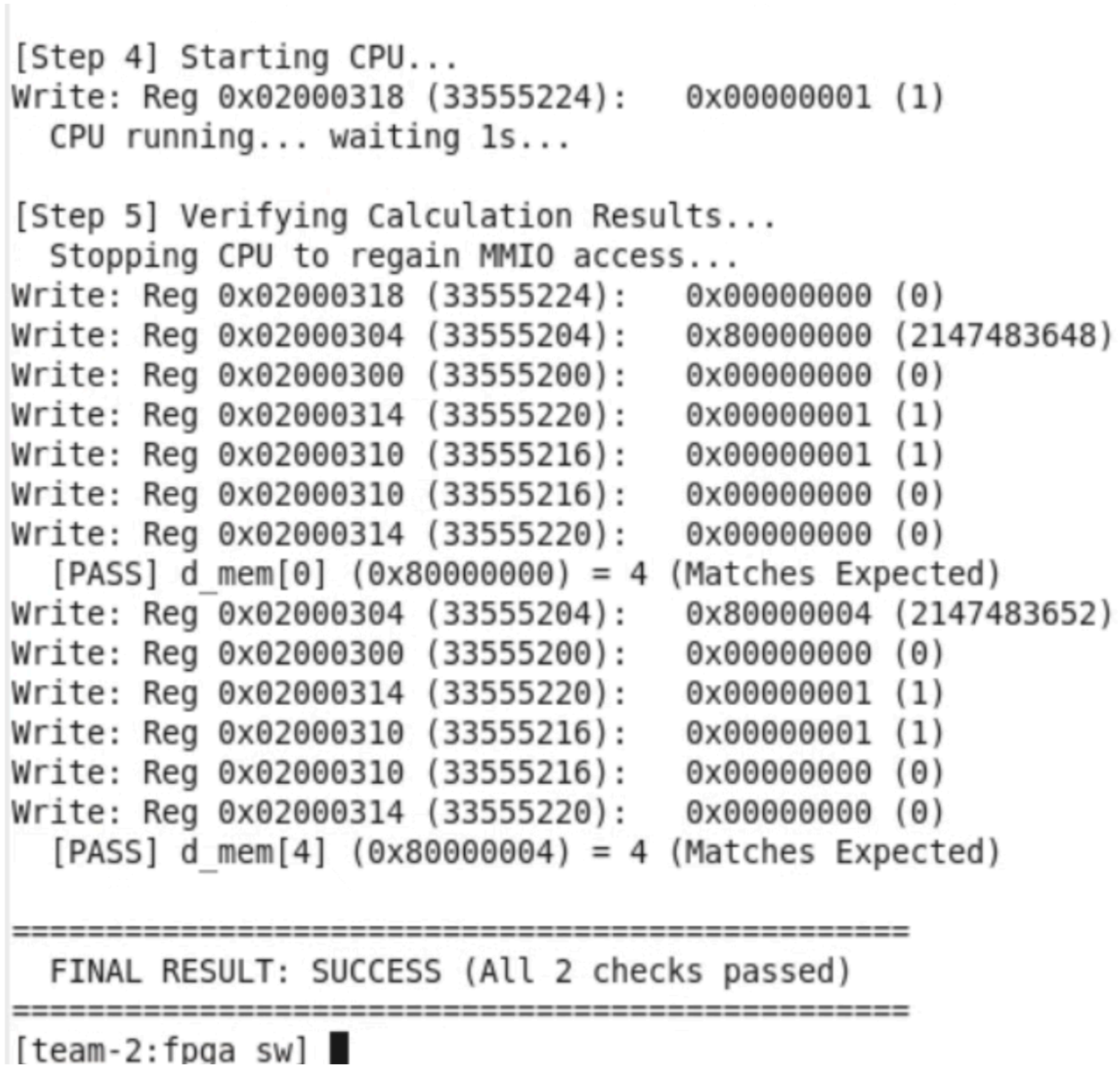
- system_active = cpu_active | start
- while system_active=1, req_rdy=0 (host transactions blocked to avoid IMEM/DMEM contention)
- CPU reset is gated as cpu_rst_n = rst_n & system_active (CPU held in reset when not active)
- when cpu_done asserts, cpu_active clears and host MMIO is re-enabled for result readback



```
jeremycai@Jeremys-MacBook-Pro sim % vvp soc_tb.vvp
VCD info: dumpfile soc_tb.vcd opened for output.

======================================================
  SoC MMIO Testbench — COE + CPU + ILA + Random
======================================================

──────────────────────────────────────────────────
  PHASE A: COE load → CPU execution → verify
──────────────────────────────────────────────────

[A1] Writing IMEM (8 words) ...
[A2] Writing DMEM (10 words) ...
[A3] Read-back verify IMEM ...
[A4] Read-back verify DMEM ...
[A5] Re-reset SoC (BRAM preserved, CPU PC → 0) ...
[A6] Starting CPU (CTRL write) ...
[A7] Waiting for CPU (timeout = 1000 cycles) ...
  CPU completed in ~509 cycles
[A8] CTRL read (expect cpu_active = 0) ...
[A9] Verifying DMEM after CPU execution ...
  d_mem[0] = 4  (unchanged as expected)
  d_mem[4] = 4  (was 100)  (store succeeded)
[A10] Verifying IMEM unchanged ...
[A11] Verifying ILA (Register File Observation) — Checking registers ...
  ILA Read R[00] = 0x0000000000000000  | Expected = 0x0000000000000000
  ILA Read R[01] = 0x0000000000000000  | Expected = 0x0000000000000000
  ILA Read R[02] = 0x0000000000000004  | Expected = 0x0000000000000004
  ILA Read R[03] = 0x0000000000000004  | Expected = 0x0000000000000004
  ILA Read R[04] = 0x0000000000000000  | Expected = 0x0000000000000000

──────────────────────────────────────────────────
  PHASE B: Random MMIO read/write tests
──────────────────────────────────────────────────

[B1] IMEM sequential W+R (64 entries)
[B2] DMEM sequential W+R (64 entries)
[B3] IMEM random-addr W+R (64 writes)
  Checked 110 IMEM addresses
[B4] DMEM random-addr W+R (64 writes)
  Checked 109 DMEM addresses
[B5] Mixed IMEM/DMEM interleaved (64 writes)
  Checked 251 total addresses
[B6] Write-after-write — last value wins
[B7] Read-after-read — non-destructive

======================================================
  646 PASSED    0 FAILED    (646 total checks)
======================================================
  >>> ALL TESTS PASSED <<<

../tb/soc_tb.v:559: $finish called at 55046000 (1ps)
jeremycai@Jeremys-MacBook-Pro sim %
```

Figure 8. Simulation results with ILA interface for register file access

We modify the ids.v design into pipeline.v with modifications in project.xml and pipeline.xml. We use perl script to create an automatic testbench to automate the data read and write through the MMIO interface. We are able to obtain the final passing result



```
[Step 4] Starting CPU...
Write: Reg 0x02000318 (33555224):   0x00000001 (1)
  CPU running... waiting 1s...

[Step 5] Verifying Calculation Results...
  Stopping CPU to regain MMIO access...
Write: Reg 0x02000318 (33555224):   0x00000000 (0)
Write: Reg 0x02000304 (33555204):   0x80000000 (2147483648)
Write: Reg 0x02000300 (33555200):   0x00000000 (0)
Write: Reg 0x02000314 (33555220):   0x00000001 (1)
Write: Reg 0x02000310 (33555216):   0x00000001 (1)
Write: Reg 0x02000310 (33555216):   0x00000000 (0)
Write: Reg 0x02000314 (33555220):   0x00000000 (0)
  [PASS] d_mem[0] (0x80000000) = 4 (Matches Expected)
Write: Reg 0x02000304 (33555204):   0x80000004 (2147483652)
Write: Reg 0x02000300 (33555200):   0x00000000 (0)
Write: Reg 0x02000314 (33555220):   0x00000001 (1)
Write: Reg 0x02000310 (33555216):   0x00000001 (1)
Write: Reg 0x02000310 (33555216):   0x00000000 (0)
Write: Reg 0x02000314 (33555220):   0x00000000 (0)
  [PASS] d_mem[4] (0x80000004) = 4 (Matches Expected)

================================================
  FINAL RESULT: SUCCESS (All 2 checks passed)
================================================
[team-2:fpga sw] █
```

Figure 9. Final verification for the CPU pipeline functionality

4.3 SOC BLOCK DIAGRAM

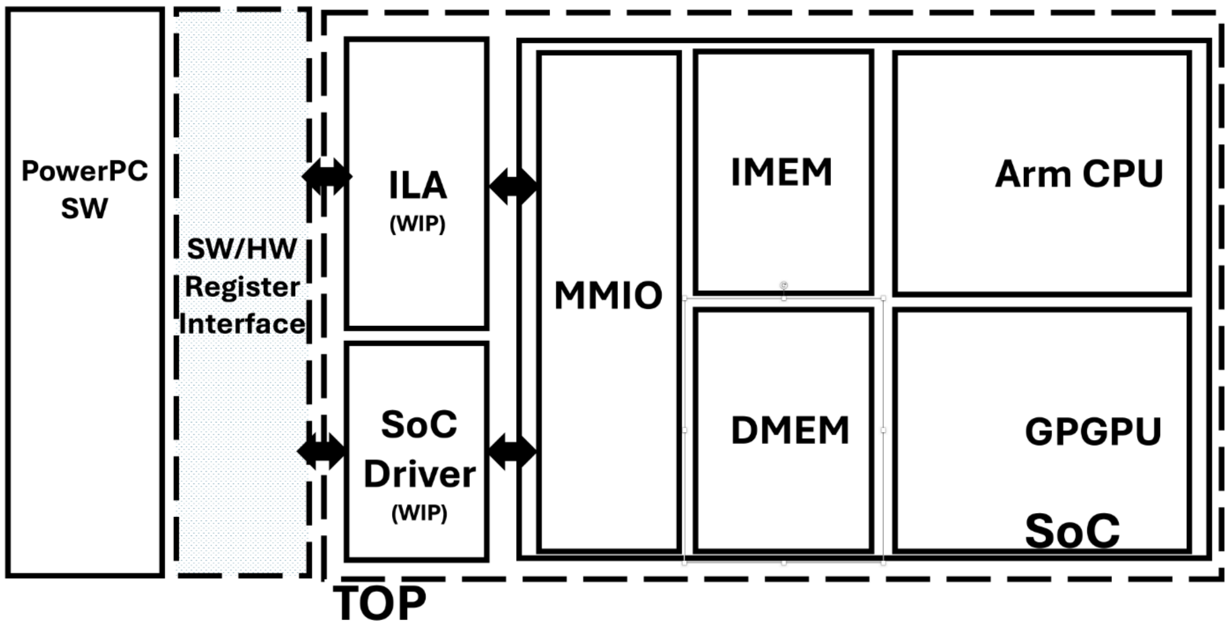


Figure 10. Full design architecture

Part 5 Generated Verilog Files, GitHub Repository and Team Contributions

GitHub Repository: https://github.com/jeremycai99/EE533-Labs/tree/main/Lab5

Contribution Summary:

- Jeremy Cai: CPU pipeline design and memory interface RTL design, and netFPGA deployment
- Prathamesh Hirekodi: Testbench design and verification
- Yanzhe Li: ILA initial design and Investigate feasibility of armv4t instruction set under current pipeline configuration

We attach all verilog files for this project for reference. Please see Lab4 GitHub repo for more details (including testbench files)

```
/* file: addsub.v
 Description: This file implements the addition and subtraction operations for the ALU
 This design support datawidth up to 64 bits and overflow detection for both addition
 This module does not handle carry in.
 Author: Jeremy Cai
 Date: Feb. 8, 2026
 Version: 1.0
 */
`ifndef ADDSUB_V
`define ADDSUB_V
`include "define.v"
`include "ksa.v"
module addsub (
    input wire [`DATA_WIDTH-1:0] operand_a,     // First operand
    input wire [`DATA_WIDTH-1:0] operand_b,     // Second operand
    input wire sub,                             // Subtract control signal (1 for subt
    output wire [`DATA_WIDTH-1:0] result,       // Result of addition or subtraction
    output wire overflow,                       // Overflow flag for addition and subt
    output wire carry_out                       // Carry out for addition (not used fo
);
wire [63:0] operand_a_ext; // Extended first operand for addition/subtraction
wire [63:0] operand_b_ext; // Extended second operand for addition/subtraction
assign operand_a_ext = {{(64-`DATA_WIDTH){operand_a[`DATA_WIDTH-1]}}, operand_a};
assign operand_b_ext = {{(64-`DATA_WIDTH){operand_b[`DATA_WIDTH-1]}}, operand_b};
wire [63:0] operand_b_ext_mod; // Modified second operand for subtraction
// Modify operand_b for subtraction (two's complement)
assign operand_b_ext_mod = sub ? ~operand_b_ext : operand_b_ext;
wire [63:0] sum_ext; // Extended sum output from KSA
wire cout_ext; // Carry out from KSA. No usage.
// Instantiate the Kogge-Stone Adder
ksa u_ksa (
    .operand_a(operand_a_ext),
    .operand_b(operand_b_ext_mod),
    .cin(sub), // Carry input is 1 for subtraction (to add the two's complement)
    .sum(sum_ext),
    .cout(carry_out) //For alu C flag
);
assign result = sum_ext[`DATA_WIDTH-1:0]; // Truncate the result to the defined data w
// Overflow detection for addition and subtraction
assign overflow = (sub) ? ((operand_a[`DATA_WIDTH-1] != operand_b[`DATA_WIDTH-1]) && (
                    ((operand_a[`DATA_WIDTH-1] == operand_b[`DATA_WIDTH-1]) && (result
endmodule
`endif //ADDSUB_V
```

```
/* file: alu.v
 Description: This file implements the main ALU module,
 which performs various arithmetic and logical operations
 based on the provided operation code.
 Author: Jeremy Cai
 Date: Feb. 9, 2026
 Version: 1.0 (for Lab 5 only)
 */
`ifndef ALU_V
`define ALU_V
`include "define.v"
`include "addsub.v"
module alu (
    input wire [`DATA_WIDTH-1:0] operand_a, // First operand
    input wire [`DATA_WIDTH-1:0] operand_b, // Second operand
    input wire [`ALU_OP_WIDTH-1:0] alu_op,  // ALU data processing operation code
    output reg [`DATA_WIDTH-1:0] result,    // Result of the ALU operation
    output wire alu_overflow             // Overflow flag for addition and subtractior
);
wire [`DATA_WIDTH-1:0] addsub_result; // Register to hold the result from add/sub modu
wire addsub_overflow, addsub_carry_out; // Overflow and carry out flags from add/sub m
wire sub_en = (alu_op == `ALU_OP_SUB);
addsub u_addsub (
    .operand_a(operand_a),
    .operand_b(operand_b),
    .sub(sub_en), // Control signal for subtraction
    .result(addsub_result), // Output result
    .overflow(addsub_overflow), // Overflow flag (V)
    .carry_out(addsub_carry_out) // Carry out for addition (C)
);
always @(*) begin
    case (alu_op)
        `ALU_OP_ADD: result = addsub_result; // ADD
        `ALU_OP_SUB: result = addsub_result; // SUB
        `ALU_OP_AND: result = operand_a & operand_b; // AND
        `ALU_OP_OR:  result = operand_a | operand_b; // OR
        `ALU_OP_XNOR: result = ~(operand_a ^ operand_b); // XNOR
        `ALU_OP_CMP: result = (operand_a == operand_b); // CMP
        `ALU_OP_LSL: result = operand_a << operand_b[5:0]; // LSL (using lower 6 bits
        `ALU_OP_LSR: result = operand_a >> operand_b[5:0]; // LSR (using lower 6 bits
        // Not standard ALU operation and not used by Arm ISA
        `ALU_OP_SBCMP: result = (operand_a[31:0] == operand_b[31:0]) ? 1 : 0; // Subst
        // `ALU_OP_LSTC: result = (operand_a << operand_b[5:0]) == operand_b ? 1 : 0;
        // `ALU_OP_RSTC: result = (operand_a >> operand_b[5:0]) == operand_b ? 1 : 0;
        default: result = 0; // Default case to handle undefined operation codes
    endcase
end
assign alu_overflow = ((alu_op == `ALU_OP_ADD || alu_op == `ALU_OP_SUB) && addsub_over
endmodule
`endif //ALU_V
```

```
/* file: ksa.v
 Description: This file implements the 64-bit kogge-stone adder for the ALU.
 Please note that this implementation is for 64 only because of the stages required.
 Author: Jeremy Cai
 Date: Feb. 8, 2026
 Version: 1.0
 */

`ifndef KSA_V
`define KSA_V

`include "define.v"

module ksa (
    input wire [63:0] operand_a,     // First operand
    input wire [63:0] operand_b,     // Second operand
    input wire cin,                  // Carry input for addition
    output wire [63:0] sum,          // Sum output
    output wire cout                 // Carry output for addition
);

//NOTICE: THIS IS A FIXED VALUE FOR ADDER!
//Higher hierarchical logic should padding if needed, and this module will only handle
localparam N = 65; // Total width including carry.

//Generate and Propagate signals
wire [N-1:0] g0, p0;
wire [N-1:0] g1, p1;
wire [N-1:0] g2, p2;
wire [N-1:0] g3, p3;
wire [N-1:0] g4, p4;
wire [N-1:0] g5, p5;
wire [N-1:0] g6, p6;
wire [N-1:0] g7, p7;

// Initial Generate and Propagate
assign g0[0] = cin;
assign p0[0] = 1'b0; // No propagate for carry input

// Stage 0
genvar i0;
generate
    for (i0 = 1; i0 < N; i0 = i0 + 1) begin
        assign g0[i0] = operand_a[i0-1] & operand_b[i0-1]; // Generate
        assign p0[i0] = operand_a[i0-1] ^ operand_b[i0-1]; // Propagate
    end
endgenerate

// Stage 1
genvar i1;
generate
    for (i1 = 0; i1 < N; i1 = i1 + 1) begin
        if (i1 < 1) begin
            assign g1[i1] = g0[i1];
            assign p1[i1] = p0[i1];
        end else begin
            assign g1[i1] = g0[i1] | (p0[i1] & g0[i1-1]);
            assign p1[i1] = p0[i1] & p0[i1-1];
        end
    end
endgenerate

// Stage 2
genvar i2;
generate
    for (i2 = 0; i2 < N; i2 = i2 + 1) begin
        if (i2 < 2) begin
            assign g2[i2] = g1[i2];
            assign p2[i2] = p1[i2];
        end else begin
            assign g2[i2] = g1[i2] | (p1[i2] & g1[i2-2]);
            assign p2[i2] = p1[i2] & p1[i2-2];
        end
    end
endgenerate

// Stage 3
genvar i3;
generate
    for (i3 = 0; i3 < N; i3 = i3 + 1) begin
        if (i3 < 4) begin
        assign g3[i3] = g2[i3];
        assign p3[i3] = p2[i3];
        end else begin
        assign g3[i3] = g2[i3] | (p2[i3] & g2[i3-4]);
        assign p3[i3] = p2[i3] & p2[i3-4];
        end
    end
endgenerate

// Stage 4
genvar i4;
generate
    for (i4 = 0; i4 < N; i4 = i4 + 1) begin
        if (i4 < 8) begin
        assign g4[i4] = g3[i4];
        assign p4[i4] = p3[i4];
        end else begin
        assign g4[i4] = g3[i4] | (p3[i4] & g3[i4-8]);
        assign p4[i4] = p3[i4] & p3[i4-8];
        end
    end
endgenerate

// Stage 5
genvar i5;
generate
    for (i5 = 0; i5 < N; i5 = i5 + 1) begin
        if (i5 < 16) begin
        assign g5[i5] = g4[i5];
        assign p5[i5] = p4[i5];
        end else begin
        assign g5[i5] = g4[i5] | (p4[i5] & g4[i5-16]);
        assign p5[i5] = p4[i5] & p4[i5-16];
        end
    end
endgenerate

// Stage 6
genvar i6;
generate
    for (i6 = 0; i6 < N; i6 = i6 + 1) begin
        if (i6 < 32) begin
        assign g6[i6] = g5[i6];
        assign p6[i6] = p5[i6];
        end else begin
        assign g6[i6] = g5[i6] | (p5[i6] & g5[i6-32]);
        assign p6[i6] = p5[i6] & p5[i6-32];
        end
    end
endgenerate

// Stage 7
genvar i7;
generate
    for (i7 = 0; i7 < N; i7 = i7 + 1) begin
        if (i7 < 64) begin
        assign g7[i7] = g6[i7];
        assign p7[i7] = p6[i7];
        end else begin
        assign g7[i7] = g6[i7] | (p6[i7] & g6[i7-64]);
        assign p7[i7] = p6[i7] & p6[i7-64];
        end
    end
endgenerate

genvar k;
generate
    for (k = 0; k < 64; k = k + 1) begin
        assign sum[k] = p0[k+1] ^ g7[k];
    end
endgenerate

assign cout = g7[64];

endmodule

`endif //KSA_V
```

```
/* file: sync_alu.v
 Description: This file implements the integration of asynchronous ALU
 operations with a register at output to create a synchronous ALU module.
 Date: Feb. 8, 2026
 Version: 1.0
 */
`ifndef SYNC_ALU_V
`define SYNC_ALU_V
`include "define.v"
`include "alu.v"
module sync_alu (
    input wire clk, // Clock signal
    input wire rst_n, // Active low reset signal
    input wire [`DATA_WIDTH-1:0] A, // First operand
    input wire [`DATA_WIDTH-1:0] B, // Second operand
    input wire [`ALU_OP_WIDTH-1:0] aluctrl,  // ALU operation code
    output reg [`DATA_WIDTH-1:0] Z,    // Result of the ALU operation
    output reg overflow        // Overflow flag for addition and subtraction
);
wire [`DATA_WIDTH-1:0] alu_result; // Register to hold the ALU result
wire alu_overflow; // Register to hold the ALU overflow flag
alu u_alu (
    .operand_a(A),
    .operand_b(B),
    .alu_op(aluctrl),
    .result(alu_result),
    .alu_overflow(alu_overflow)
);
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        Z <= 0; // Reset the output register
        overflow <= 0; // Reset the overflow flag
    end else begin
        Z <= alu_result; // Update the output register with the ALU result
        overflow <= alu_overflow; // Update the overflow flag
    end
end
endmodule
`endif //SYNC_ALU_V
```

```
/* file: pc.v
 Description: Program counter module for the pipeline CPU design
 Author: Jeremy Cai
 Date: Feb. 10, 2026
 Version: 1.0
 */
`ifndef PC_V
`define PC_V
`include "define.v"
module pc (
    input wire clk,
    input wire rst_n,
    input wire en, // Enable signal for updating the PC

    output reg [`PC_WIDTH-1:0] pc_out // Current value of the program counter
);
// PC update logic: synchronous update on clock edge
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        pc_out <= 0; // Reset PC to 0 on reset
    end else if (en) begin
        pc_out <= pc_out + 1; // Increment PC by 1 each cycle if enabled
    end
end
endmodule
`endif // PC_V
```

```
/* file: ppl_reg.v
 Description: Configurable register moduld for pipeline registers
 Author: Jeremy Cai
 Date: Feb. 8, 2026
 Version: 1.0
 */
`ifndef PPL_REG_V
`define PPL_REG_V
`include "define.v"
module ppl_reg #(parameter NUM_REG = 32)(
    input wire clk, // Clock signal
    input wire rst_n, // Active low reset signal
    input wire en, // Enable signal for updating the register
    input wire [NUM_REG-1:0] D, // Data input
    output reg [NUM_REG-1:0] Q // Data output
);
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        Q <= 0; // Reset the register output to 0
    end else if (en) begin
        Q <= D; // Update the register output with the input data
    end
end
endmodule
`endif //PPL_REG_V
```

```
/* file: regfile.v
 Description: Register file module for the pipeline CPU design
 Author: Jeremy Cai
 Date: Feb. 9, 2026
 Version: 1.0
 */
`ifndef REGFILE_V
`define REGFILE_V
`include "define.v"
module regfile(
    input clk, // Clock signal. No reset signal needed
    input [`REG_ADDR_WIDTH-1:0] r0addr, // Source register 1 address
    input [`REG_ADDR_WIDTH-1:0] r1addr, // Source register 2 address
    input wena, // Write enable signal
    input [`REG_ADDR_WIDTH-1:0] waddr, // Destination register address
    input [`REG_DATA_WIDTH-1:0] wdata, // Data to write to the register
    output wire [`REG_DATA_WIDTH-1:0] r0data, // Data read from source register 1
    output wire [`REG_DATA_WIDTH-1:0] r1data, // Data read from source register 2

    // ILA probe signals for debugging
    input [`REG_ADDR_WIDTH-1:0] ila_cpu_reg_addr, // ILA probe address input
    output wire [`REG_DATA_WIDTH-1:0] ila_cpu_reg_data // ILA probe data output
);
reg [`REG_DATA_WIDTH-1:0] regs [0:(1<<`REG_ADDR_WIDTH)-1]; // Register file array
// Register read logic: combinational read
    // do not need to qualify with waddr != 0 since Arm doesn't restrict writes to x0
assign r0data = regs[r0addr]; // Register x0 is hardwired to 0
assign r1data = regs[r1addr]; // Register x0 is hardwired to 0
assign ila_cpu_reg_data = regs[ila_cpu_reg_addr]; // ILA probe data output
// Register write logic: synchronous write on clock edge
always @(posedge clk) begin
    if (wena) begin // do not need to qualify with waddr != 0 since Arm doesn't restri
        regs[waddr] <= wdata;
    end
end
endmodule
`endif //REGFILE_V
```

```
/* file: cpu.v
 Description: CPU module for the pipeline CPU design
 Author: Jeremy Cai
 Date: Feb. 14, 2026
 Version: 1.3
 */
`ifndef CPU_V
`define CPU_V
`include "define.v"
`include "pc.v"
`include "ppl_reg.v"
```

```verilog
`include "regfile.v"
module cpu (
    input wire clk,
    input wire rst_n,
    // Instruction memory interface
    input wire [`INSTR_WIDTH-1:0] i_mem_data_i, // Instruction memory data input
    output wire [`PC_WIDTH-1:0] i_mem_addr_o,   // Instruction memory address output
    // Data memory interface
    input wire [`DATA_WIDTH-1:0] d_mem_data_i,  // Data memory data input (64-bit)
    output wire [`DMEM_ADDR_WIDTH-1:0] d_mem_addr_o, // Data memory address output
    output wire [`DATA_WIDTH-1:0] d_mem_data_o, // Data memory data output (64-bit)
    output wire d_mem_wen_o,                    // Data memory write enable
    output wire cpu_done,                       // Signal to indicate CPU completion
    // ========================================================================
    // ILA Debug Interface
    // ========================================================================
    // Multiplexed Debug Port (Full 64-bit width)
    // [4] = 0: System Debug (Selects via [3:0])
    // [4] = 1: Register File Debug (Address via [2:0])
    input wire [4:0] ila_debug_sel,
    output wire [`DATA_WIDTH-1:0] ila_debug_data
);
// CPU internal signal definitions
// IF stage signals
wire [`PC_WIDTH-1:0] pc_if;
wire pc_en;
assign pc_en = 1'b1;
// CPU Done signal: Active when PC reaches max value (all 1s)
assign cpu_done = (pc_if == {`PC_WIDTH{1'b1}});
// Instruction memory interface signals
wire [`INSTR_WIDTH-1:0] instr_if;
// ID stage signals
wire [`INSTR_WIDTH-1:0] instr_id;
wire reg_write_id;
wire mem_write_id;
wire [`REG_ADDR_WIDTH-1:0] rdaddr_id;
// Register file interface signals
wire [`REG_ADDR_WIDTH-1:0] r0addr_id;
wire [`REG_ADDR_WIDTH-1:0] r1addr_id;
wire [`REG_ADDR_WIDTH-1:0] reg_write_addr;
wire [`REG_DATA_WIDTH-1:0] reg_write_data;
wire reg_write_en;
wire [`REG_DATA_WIDTH-1:0] r0_out_id;
wire [`REG_DATA_WIDTH-1:0] r1_out_id;
// EX stage signals
wire reg_write_ex;
wire mem_write_ex;
wire [`REG_DATA_WIDTH-1:0] r0_out_ex;
wire [`REG_DATA_WIDTH-1:0] r1_out_ex;
wire [`REG_ADDR_WIDTH-1:0] rdaddr_ex;
// MEM stage signals
wire reg_write_mem;
wire mem_write_mem;
wire [`REG_DATA_WIDTH-1:0] r0_out_mem;
wire [`REG_DATA_WIDTH-1:0] r1_out_mem;
wire [`REG_ADDR_WIDTH-1:0] rdaddr_mem;
// WB stage signals
wire reg_write_wb;
wire [`REG_ADDR_WIDTH-1:0] rdaddr_wb;
// Debug signals
wire [`REG_DATA_WIDTH-1:0] debug_reg_out;

// Start core pipeline logic implementation
/* IF STAGE */
pc u_pc (
    .clk(clk),
    .rst_n(rst_n),
    .en(pc_en),
    .pc_out(pc_if)
);
assign i_mem_addr_o = pc_if;
assign instr_if = i_mem_data_i;
/* IF/ID pipeline register */
// Width: Instruction Width (32)
ppl_reg #(.NUM_REG(`INSTR_WIDTH)) if_id_reg (
    .clk(clk),
    .rst_n(rst_n),
    .en(1'b1),
    .D(instr_if),
    .Q(instr_id)
);
/* ID STAGE */
// Control Signals
assign mem_write_id = instr_id[31];
assign reg_write_id = instr_id[30];
// Decoding based on REG_ADDR_WIDTH = 3
assign r0addr_id = instr_id[26:24]; // Source Address (e.g., for Load addr / Store add
assign r1addr_id = instr_id[29:27]; // Data Source (e.g., for Store data)
assign rdaddr_id = instr_id[23:21]; // Destination Register
regfile u_regfile (
    .clk(clk),
    .r0addr(r0addr_id),
    .r1addr(r1addr_id),
    .waddr(reg_write_addr),
    .wdata(reg_write_data),
    .wena(reg_write_en),
    .r0data(r0_out_id),
    .r1data(r1_out_id),
    // Map lower bits of debug selector to register address
    .ila_cpu_reg_addr(ila_debug_sel[`REG_ADDR_WIDTH-1:0]),
    .ila_cpu_reg_data(debug_reg_out)
);
// ID/EX pipeline register
// Width: 1(RegW) + 1(MemW) + 64(Data0) + 64(Data1) + 3(RdAddr)
ppl_reg #(.NUM_REG(1+1+`REG_DATA_WIDTH+`REG_DATA_WIDTH+`REG_ADDR_WIDTH)) id_ex_reg (
    .clk(clk),
    .rst_n(rst_n),
    .en(1'b1),
    .D({reg_write_id, mem_write_id, r0_out_id, r1_out_id, rdaddr_id}),
    .Q({reg_write_ex, mem_write_ex, r0_out_ex, r1_out_ex, rdaddr_ex})
);
/* EX STAGE */
// EX/MEM pipeline register
// Width: 1(RegW) + 1(MemW) + 64(ALU/Data0) + 64(Data1) + 3(RdAddr)
ppl_reg #(.NUM_REG(1+1+`REG_DATA_WIDTH+`REG_DATA_WIDTH+`REG_ADDR_WIDTH)) ex_mem_reg (
    .clk(clk),
    .rst_n(rst_n),
    .en(1'b1),
    .D({reg_write_ex, mem_write_ex, r0_out_ex, r1_out_ex, rdaddr_ex}),
    .Q({reg_write_mem, mem_write_mem, r0_out_mem, r1_out_mem, rdaddr_mem})
);
/* MEM STAGE */
// Connect data memory interface signals
// Note: r0_out_mem is 64-bit, but d_mem_addr_o is 8-bit. Truncation is implicit or ex
assign d_mem_addr_o = r0_out_mem[`DMEM_ADDR_WIDTH-1:0];
assign d_mem_data_o = r1_out_mem; // Data to write
assign d_mem_wen_o = mem_write_mem;
// MEM/WB pipeline register
// Width: 1(RegW) + 3(RdAddr)
ppl_reg #(.NUM_REG(1+`REG_ADDR_WIDTH)) mem_wb_reg (
    .clk(clk),
    .rst_n(rst_n),
    .en(1'b1),
    .D({reg_write_mem, rdaddr_mem}),
    .Q({reg_write_wb, rdaddr_wb})
);
/* WB STAGE */
assign reg_write_data = d_mem_data_i; // Data from memory load
assign reg_write_addr = rdaddr_wb;
assign reg_write_en = reg_write_wb;
// Newly added ILA Debug Interface logic
// Now using full `DATA_WIDTH` (64 bits) for output.
// Signals smaller than 64 bits are zero-padded.
always @(*) begin
    if (ila_debug_sel[4]) begin
        // Mode 1: Register File Debug (MSB = 1)
        // Address is taken from ila_debug_sel[2:0] which is already wired to regfile
        ila_debug_data = debug_reg_out;
    end else begin
        // Mode 0: System Debug (MSB = 0)
        case (ila_debug_sel[3:0])
            // 0: Program Counter (Fetch) - 9 bits
            4'd0: ila_debug_data = { {`DATA_WIDTH-`PC_WIDTH{1'b0}}, pc_if };

            // 1: Instruction (Decode) - 32 bits
            4'd1: ila_debug_data = { {`DATA_WIDTH-`INSTR_WIDTH{1'b0}}, instr_id };

            // 2: Register Read Data A (Decode) - Full 64 bits
            4'd2: ila_debug_data = r0_out_id;

            // 3: Register Read Data B (Decode) - Full 64 bits
            4'd3: ila_debug_data = r1_out_id;

            // 4: EX Stage Result / Address (Execute) - Full 64 bits
            4'd4: ila_debug_data = r0_out_ex;
            // 5: EX Stage Write Data (Execute) - Full 64 bits
            4'd5: ila_debug_data = r1_out_ex;

            // 6: Writeback Data / Memory Read Data (Writeback) - Full 64 bits
            4'd6: ila_debug_data = d_mem_data_i;

            // 7: Control Signals Vector
            // [0]: Reg Write ID, [1]: Mem Write ID, [2]: Mem Write Mem, [3]: Reg Writ
            4'd7: ila_debug_data = { {`DATA_WIDTH-4{1'b0}}, reg_write_wb, mem_write_me
            // 8: Destination Register Address (Decode) - 3 bits
            4'd8: ila_debug_data = { {`DATA_WIDTH-`REG_ADDR_WIDTH{1'b0}}, rdaddr_id };
            // 9: Destination Register Address (Writeback) - 3 bits
            4'd9: ila_debug_data = { {`DATA_WIDTH-`REG_ADDR_WIDTH{1'b0}}, rdaddr_wb };
            default: ila_debug_data = {`DATA_WIDTH{1'b1}}; // All 1s for invalid selec
        endcase
    end
end
endmodule
`endif // CPU_V
```

```
/* file: soc.v
 Description: SoC top module for the pipeline CPU design with memory access interface
 Author: Jeremy Cai
 Date: Feb. 14, 2026
 Version: 1.5
 */

`ifndef SOC_V
`define SOC_V

`include "define.v"

// CPU and test memory modules
// Remove from soc.v when synthesis
// ---------------------------
// `include "test_i_mem.v"
// `include "test_d_mem.v"
// ---------------------------

`include "i_mem.v"
`include "d_mem.v"

`include "cpu.v"

module soc (
    input wire clk,
    input wire rst_n,

    input wire req_cmd, //Request command 0 for read and 1 for write
    input wire [`MMIO_ADDR_WIDTH-1:0] req_addr, //Fixed request address width
    input wire [`MMIO_DATA_WIDTH-1:0] req_data, //Data to write for write requests
    input wire req_val,
    output wire req_rdy,
```

```verilog
    output wire resp_cmd, //Response command 0 for read response and 1 for write respo
    output wire [`MMIO_ADDR_WIDTH-1:0] resp_addr, //Response address
    output wire [`MMIO_DATA_WIDTH-1:0] resp_data, //Response data
    output wire resp_val,
    input wire resp_rdy,

    // External start signal (Active High, Level Sensitive)
    input wire start,

    // Expanded Debug Interface
    // Bit 4: 0 = System Debug, 1 = Register File Debug
    // Bits 3-0: Selection index or Register Address
    input wire [4:0] ila_debug_sel,
    output wire [`DATA_WIDTH-1:0] ila_debug_data // Full 64-bit debug data output
);

//  Address Region Decode — req_addr[31:30]
//    2'b00  (0x0000_0000)  → IMEM   (blocked while CPU active)
//    2'b01  (0x4000_0000)  → CTRL   write: start CPU (internal latch)
//                                    read : {31'b0, system_active}
//    2'b10  (0x8000_0000)  → DMEM   (Port B — safe any time)
//    2'b11                 → reserved (returns 0)

localparam REGION_IMEM = 2'b00;
localparam REGION_CTRL = 2'b01;
localparam REGION_DMEM = 2'b10;
localparam REGION_RESERVED = 2'b11;

//MMIO interface FSM states
localparam STATE_IDLE = 2'b00;
localparam STATE_ACCESS = 2'b01;
localparam STATE_RESP = 2'b10;

reg [1:0] current_state, next_state;

//MMIO interface registers
reg req_cmd_reg;
reg [`MMIO_ADDR_WIDTH-1:0] req_addr_reg;
reg [`MMIO_DATA_WIDTH-1:0] req_data_reg;
reg [1:0] req_region_reg;

reg resp_pending;
reg resp_val_reg;
reg resp_cmd_reg;
reg [`MMIO_ADDR_WIDTH-1:0] resp_addr_reg;
reg [`MMIO_DATA_WIDTH-1:0] resp_data_reg;

//CPU control signals
reg cpu_active; // Internal register for MMIO-triggered start
wire system_active; // Combined active signal (MMIO latch OR external start pin)
wire cpu_done; // Indicates whether the CPU has completed execution

// Combine external start pin with internal MMIO active latch
assign system_active = cpu_active | start;
```

```
//MMIO interface handshaking signals
wire req_fire = req_val && req_rdy; // Indicates a valid request handshake
wire req_is_ctrl = (req_region_reg == REGION_CTRL); // Indicates if the request is tar
wire req_is_imem = (req_region_reg == REGION_IMEM); // Indicates if the request is tar
wire req_is_dmem = (req_region_reg == REGION_DMEM); // Indicates if the request is tar

// The two-port data memory has the advantage to access data memory without stalling t
// However, we block access if the CPU is running (via MMIO or external Start pin) to
assign req_rdy = (current_state == STATE_IDLE) && (~system_active);

assign resp_val = resp_val_reg;
assign resp_cmd = resp_cmd_reg;
assign resp_addr = resp_addr_reg;
assign resp_data = resp_data_reg;

//Host interface active flags
wire host_active = (current_state == STATE_ACCESS) || (current_state == STATE_RESP); /

wire imem_host_active = host_active && req_is_imem;
wire dmem_host_active = host_active && req_is_dmem;
wire ctrl_host_active = host_active && req_is_ctrl;

//Instruction memory muxing
wire [`PC_WIDTH-1:0] cpu_imem_addr;
wire [`INSTR_WIDTH-1:0] imem_dout;

reg [`PC_WIDTH-1:0] imem_addr_mux;
reg [`INSTR_WIDTH-1:0] imem_din_mux;
reg imem_we_mux;

// Data memory address and data muxing logic
wire [`DMEM_ADDR_WIDTH-1:0] cpu_dmem_addr;
wire [`DATA_WIDTH-1:0]      cpu_dmem_wdata;
wire                        cpu_dmem_wen;
wire [`DATA_WIDTH-1:0]      dmem_douta;   // Port A read  → CPU
wire [`DATA_WIDTH-1:0]      dmem_doutb;

reg [`DMEM_ADDR_WIDTH-1:0] dmem_addr_mux;
reg [`DATA_WIDTH-1:0] dmem_din_mux;
reg dmem_we_mux;

// Instruction memory address muxing logic
always @(*) begin
    if (imem_host_active) begin
        imem_addr_mux = req_addr_reg[(`PC_WIDTH-1):0];
        imem_din_mux = req_data_reg[(`INSTR_WIDTH-1):0];
        imem_we_mux = req_cmd_reg && ((current_state == STATE_ACCESS));
    end else begin
        imem_addr_mux = cpu_imem_addr;
        imem_din_mux = `INSTR_WIDTH'b0;
        imem_we_mux = 1'b0;
    end
end
```

```
// Data memory muxing logic
always @(*) begin
    if (dmem_host_active) begin
        dmem_addr_mux = req_addr_reg[(`DMEM_ADDR_WIDTH-1):0];
        dmem_din_mux = req_data_reg[(`DATA_WIDTH-1):0];
        dmem_we_mux = req_cmd_reg && ((current_state == STATE_ACCESS));
    end else begin
        dmem_addr_mux = cpu_dmem_addr;
        dmem_din_mux = cpu_dmem_wdata;
        dmem_we_mux = cpu_dmem_wen;
    end
end


// CPU start and done logic (MMIO based)
// Note: This logic only controls the internal latch 'cpu_active'.
// The actual CPU reset is controlled by 'system_active' which includes the 'start' pi
wire start_pulse = (current_state == STATE_ACCESS) && req_is_ctrl && req_cmd_reg && !s

always @ (posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        cpu_active <= 1'b0;
    end else begin
        if (start_pulse) begin
            cpu_active <= 1'b1; // Latch active on MMIO command
        end else if (cpu_done) begin
            cpu_active <= 1'b0; // Clear latch when CPU finishes
        end
    end
end

//MMIO interface FSM design
always @(*) begin
    next_state = current_state;
    case (current_state)
        STATE_IDLE: if (req_fire) next_state = STATE_ACCESS;
        STATE_ACCESS: next_state = STATE_RESP;
        STATE_RESP: if (resp_val_reg && resp_rdy) next_state = STATE_IDLE;
        default: next_state = STATE_IDLE;
    endcase
end

// MMIO interface sequential logic
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        current_state <= STATE_IDLE;
        req_cmd_reg <= 1'b0;
        req_addr_reg <= {`MMIO_ADDR_WIDTH{1'b0}};
        req_data_reg <= {`MMIO_DATA_WIDTH{1'b0}};
        req_region_reg <= 2'b00;
        resp_pending <= 1'b0;
        resp_val_reg <= 1'b0;
        resp_cmd_reg <= 1'b0;
        resp_addr_reg <= {`MMIO_ADDR_WIDTH{1'b0}};
        resp_data_reg <= {`MMIO_DATA_WIDTH{1'b0}};
    end else begin
        current_state <= next_state;
        if (req_fire) begin
            req_cmd_reg <= req_cmd;
            req_addr_reg <= req_addr;
            req_data_reg <= req_data;
            req_region_reg <= req_addr[31:30];
            resp_pending <= 1'b1;
            resp_val_reg <= 1'b0;
        end

        if (resp_val_reg && resp_rdy) begin
            resp_val_reg <= 1'b0;
            resp_pending <= 1'b0;
        end

        if (current_state == STATE_RESP && resp_pending && ~resp_val_reg) begin
            resp_val_reg <= 1'b1;
            resp_cmd_reg <= req_cmd_reg;
            resp_addr_reg <= req_addr_reg;

            // Generate response based on the request region
            case (req_region_reg)
                REGION_IMEM: resp_data_reg <= {{(`MMIO_DATA_WIDTH-`INSTR_WIDTH){1'b0}}
                REGION_DMEM: resp_data_reg <= dmem_douta;
                // Return system_active (includes external start pin status)
                REGION_CTRL: resp_data_reg <= {{(`MMIO_DATA_WIDTH-1){1'b0}}, system_ac
                default: resp_data_reg <= {`MMIO_DATA_WIDTH{1'b0}};
            endcase
        end
    end
end

// CPU Reset Logic:
// The CPU is active (out of reset) if global reset is high AND (MMIO started it OR ex
wire cpu_rst_n = rst_n & system_active;

cpu u_cpu (
    .clk(clk),
    .rst_n(cpu_rst_n),
    .i_mem_data_i(imem_dout),
    .i_mem_addr_o(cpu_imem_addr),
    .d_mem_addr_o(cpu_dmem_addr),
    .d_mem_data_i(dmem_douta),
    .d_mem_data_o(cpu_dmem_wdata),
    .d_mem_wen_o(cpu_dmem_wen),
    .cpu_done(cpu_done),

    .ila_debug_sel(ila_debug_sel),
    .ila_debug_data(ila_debug_data)
);

test_i_mem u_i_mem (
    .clk(clk),
    .din(imem_din_mux),
    .addr(imem_addr_mux),
    .we(imem_we_mux),
    .dout(imem_dout)
);

test_d_mem u_d_mem (
    .clka(clk),
    .dina(dmem_din_mux),
    .addra(dmem_addr_mux),
    .wea(dmem_we_mux),
    .douta(dmem_douta),
    // Port B reserved for GPU access in future labs
    .clkb(clk),
    .dinb({`DATA_WIDTH{1'b0}}),
    .addrb({`DMEM_ADDR_WIDTH{1'b0}}),
    .web(1'b0),
    .doutb()
);

endmodule

`endif // SOC_V
```

```
/* file: define.v
 Description: This file contains global definitions and parameters for the project.
 Author: Jeremy Cai
 Date: Feb. 9, 2026
 Version: 1.0
 */
//Data width definition
`define DATA_WIDTH 64 //Maximum 64 due to KSA design
`define ALU_OP_WIDTH 4
//Instruction width definition
`define INSTR_WIDTH 32
//Data memory parameters
`define DMEM_ADDR_WIDTH 8 //8 bits for 256 entries of data memory
//ALU operation codes (4 bits)
//Support list: ADD SUB AND OR XNOR CMP LSL LSR SBCMP LSTC RSTC
`define ALU_OP_ADD 4'b0000
`define ALU_OP_SUB 4'b0001
`define ALU_OP_AND 4'b0010
`define ALU_OP_OR  4'b0011
`define ALU_OP_XNOR 4'b0100
`define ALU_OP_CMP 4'b0101
`define ALU_OP_LSL 4'b0110
`define ALU_OP_LSR 4'b0111
// Not standard ALU operation and not used by Arm ISA
`define ALU_OP_SBCMP 4'b1000
`define ALU_OP_LSTC 4'b1001
`define ALU_OP_RSTC 4'b1010
//Register file parameters
`define REG_ADDR_WIDTH 3 //3 bits for 8 registers (lab 5)
`define REG_DATA_WIDTH 64 //64-bit registers
//Program counter parameters
```

```
`define PC_WIDTH 9 //9-bit program counter
//MMIO interface parameters
`define MMIO_ADDR_WIDTH 32 //32 bits for MMIO address space
`define MMIO_DATA_WIDTH 64 //64 bits for MMIO data width
```

```
#!/usr/bin/perl -w
#
# file: pipelinereg
# Description: Script to control the Pipeline SoC via NetFPGA Register Interface.
# Updated for Lab 5 with Reset Logic and detailed Pass/Fail reporting.
#

use strict;

# ================================================================
#  Register Address Map
# ================================================================
# Software Registers (Host -> FPGA)
my $PIPELINE_SW_REQ_CMD_REG       = 0x2000300;
my $PIPELINE_SW_REQ_ADDR_REG      = 0x2000304;
my $PIPELINE_SW_REQ_DATA_LO_REG   = 0x2000308;
my $PIPELINE_SW_REQ_DATA_HI_REG   = 0x200030c;
my $PIPELINE_SW_REQ_VAL_REG       = 0x2000310;
my $PIPELINE_SW_RESP_RDY_REG      = 0x2000314;
my $PIPELINE_SW_CPU_START_REG     = 0x2000318;
my $PIPELINE_SW_ILA_DEBUG_SEL_REG = 0x200031c;

# Hardware Registers (FPGA -> Host)
my $PIPELINE_HW_REQ_RDY_REG       = 0x2000320;
my $PIPELINE_HW_RESP_CMD_REG      = 0x2000324;
my $PIPELINE_HW_RESP_ADDR_REG     = 0x2000328;
my $PIPELINE_HW_RESP_DATA_LO_REG  = 0x200032c;
my $PIPELINE_HW_RESP_DATA_HI_REG  = 0x2000330;
my $PIPELINE_HW_RESP_VAL_REG      = 0x2000334;
my $PIPELINE_HW_ILA_DEBUG_LO_REG  = 0x2000338;
my $PIPELINE_HW_ILA_DEBUG_HI_REG  = 0x200033c;

# ================================================================
#  SoC Internal Base Addresses
# ================================================================
my $IMEM_BASE = 0x00000000;
my $CTRL_BASE = 0x40000000;
my $DMEM_BASE = 0x80000000;

# ================================================================
#  COE Data (Program & Initial Data)
# ================================================================
my @imem_coe = (
    0x40400000, 0x40600000, 0x00000000, 0x00000000,
    0x00000000, 0x93000000, 0x00000000, 0x00000000
);

my @dmem_coe = (
    0x00000004, 0x00000000, 0x00000000, 0x00000000,
    0x00000064, 0x00000000, 0x00000000, 0x00000000,
    0x00000000, 0x00000000
);

# Expected results after CPU execution
my %expected_results = (
    0 => 4,    # Address Offset 0 : Expect 4
    4 => 4     # Address Offset 4 : Expect 4
);

# ================================================================
#  Low Level Register Access
# ================================================================

sub regwrite {
    my ($addr, $value) = @_;
    # Using 2>/dev/null to suppress noise
    my $cmd = sprintf("regwrite 0x%08x 0x%08x 2>/dev/null", $addr, $value);
    system($cmd);
}

sub regread {
    my ($addr) = @_;
    my $cmd = sprintf("regread 0x%08x 2>/dev/null", $addr);
    my @out = `$cmd`;
    my $result = 0;
    foreach my $line (@out) {
        if ($line =~ m/Reg (0x[0-9a-f]+) \((\d+)\):\s+(0x[0-9a-f]+) \((\d+)\)/i) {
            $result = hex($3);
            last;
        }
    }
    return $result;
}

# ================================================================
#  MMIO Transaction Logic
# ================================================================

# Wait for hw_req_rdy (Bit 0 of HW_REQ_RDY_REG)
sub wait_req_rdy {
    my $timeout = 5000;
    while ($timeout > 0) {
        my $val = regread($PIPELINE_HW_REQ_RDY_REG);
        if (($val & 0x1) == 0x1) { return 1; }
        $timeout--;
        select(undef, undef, undef, 0.001);
    }
    print "  [ERROR] Timeout waiting for req_rdy (Bus might be busy or CPU running)\n'
    return 0;
}

# Wait for hw_resp_val (Bit 0 of HW_RESP_VAL_REG)
sub wait_resp_val {
    my $timeout = 5000;
    while ($timeout > 0) {
        my $val = regread($PIPELINE_HW_RESP_VAL_REG);
        if (($val & 0x1) == 0x1) { return 1; }
        $timeout--;
        select(undef, undef, undef, 0.001);
    }
    print "  [ERROR] Timeout waiting for resp_val\n";
    return 0;
}

sub mmio_write {
    my ($addr, $data) = @_;

    if (!wait_req_rdy()) { return; }

    regwrite($PIPELINE_SW_REQ_ADDR_REG, $addr);
    regwrite($PIPELINE_SW_REQ_DATA_LO_REG, $data);
    regwrite($PIPELINE_SW_REQ_DATA_HI_REG, 0x0);

    # Command: WRITE = 1
    regwrite($PIPELINE_SW_REQ_CMD_REG, 0x1);

    # Ready for response
    regwrite($PIPELINE_SW_RESP_RDY_REG, 0x1);

    # Trigger Request
    regwrite($PIPELINE_SW_REQ_VAL_REG, 0x1);

    if (!wait_resp_val()) {
        regwrite($PIPELINE_SW_REQ_VAL_REG, 0x0);
        regwrite($PIPELINE_SW_RESP_RDY_REG, 0x0);
        return;
    }

    # Handshake complete
    regwrite($PIPELINE_SW_REQ_VAL_REG, 0x0);
    regwrite($PIPELINE_SW_RESP_RDY_REG, 0x0);
}

sub mmio_read {
    my ($addr) = @_;

    if (!wait_req_rdy()) { return 0; }

    regwrite($PIPELINE_SW_REQ_ADDR_REG, $addr);

    # Command: READ = 0
    regwrite($PIPELINE_SW_REQ_CMD_REG, 0x0);

    # Ready for response
    regwrite($PIPELINE_SW_RESP_RDY_REG, 0x1);

    # Trigger Request
    regwrite($PIPELINE_SW_REQ_VAL_REG, 0x1);
    if (!wait_resp_val()) {
        regwrite($PIPELINE_SW_REQ_VAL_REG, 0x0);
        regwrite($PIPELINE_SW_RESP_RDY_REG, 0x0);
        return 0;
    }

    my $data_lo = regread($PIPELINE_HW_RESP_DATA_LO_REG);

    # Cleanup
    regwrite($PIPELINE_SW_REQ_VAL_REG, 0x0);
    regwrite($PIPELINE_SW_RESP_RDY_REG, 0x0);

    return $data_lo;
}

# ================================================================
#  Phase Tasks
# ================================================================

sub reset_all {
    print "\n[Reset] Resetting Control Registers...\n";

    # 1. Stop the CPU
    regwrite($PIPELINE_SW_CPU_START_REG, 0x0);

    # 2. Clear MMIO Handshake signals (in case script was interrupted previously)
    regwrite($PIPELINE_SW_REQ_VAL_REG, 0x0);
    regwrite($PIPELINE_SW_RESP_RDY_REG, 0x0);

    # 3. Clear Command/Address registers (Optional, for cleanliness)
    regwrite($PIPELINE_SW_REQ_CMD_REG, 0x0);
    regwrite($PIPELINE_SW_REQ_ADDR_REG, 0x0);
    regwrite($PIPELINE_SW_REQ_DATA_LO_REG, 0x0);

    print "  Reset Complete. CPU Stopped. MMIO Idle.\n";
    select(undef, undef, undef, 0.1); # Small delay to let HW settle
}

sub load_imem {
    print "\n[Step 1] Writing IMEM...\n";
    for (my $i = 0; $i < scalar(@imem_coe); $i++) {
        mmio_write($IMEM_BASE | $i, $imem_coe[$i]);
    }
    print "  IMEM Load Complete.\n";
}

sub load_dmem {
    print "\n[Step 2] Writing DMEM...\n";
    for (my $i = 0; $i < scalar(@dmem_coe); $i++) {
        mmio_write($DMEM_BASE | $i, $dmem_coe[$i]);
    }
    print "  DMEM Load Complete.\n";
}

sub verify_load {
    print "\n[Step 3] Verifying Memory...\n";
    my $errors = 0;

    # Verify IMEM
    for (my $i = 0; $i < scalar(@imem_coe); $i++) {
        my $val = mmio_read($IMEM_BASE | $i);
        if ($val != $imem_coe[$i]) {
            printf("  [FAIL] IMEM[%d]: Exp 0x%08x, Got 0x%08x\n", $i, $imem_coe[$i], $
            $errors++;
        }
    }

    # Verify DMEM
    for (my $i = 0; $i < scalar(@dmem_coe); $i++) {
        my $val = mmio_read($DMEM_BASE | $i);
        if ($val != $dmem_coe[$i]) {
            printf("  [FAIL] DMEM[%d]: Exp 0x%08x, Got 0x%08x\n", $i, $dmem_coe[$i], $
            $errors++;
        }
    }

    if ($errors == 0) {
        print "  [PASS] Memory verification successful (No errors).\n";
    } else {
        print "  [FAIL] Memory verification failed with $errors errors.\n";
    }
}

sub start_cpu_mmio {
    print "\n[Step 4] Starting CPU...\n";
    # Assert Start (1)
    regwrite($PIPELINE_SW_CPU_START_REG, 0x1);
}

sub stop_cpu {
    print "  Stopping CPU to regain MMIO access...\n";
    # De-assert Start (0)
    regwrite($PIPELINE_SW_CPU_START_REG, 0x0);
    select(undef, undef, undef, 0.1); # Allow hardware to settle
}

sub check_results {
    print "\n[Step 5] Verifying Calculation Results...\n";

    # IMPORTANT: We stop the CPU to ensure the MMIO interface
    # can access the memory bus without contention.
    stop_cpu();

    my $fail_count = 0;
    my $total_checks = 0;

    # Iterate through expected results defined at top of script
    foreach my $offset (sort { $a <=> $b } keys %expected_results) {
        $total_checks++;
        my $addr = $DMEM_BASE | $offset;
        my $expected = $expected_results{$offset};

        my $actual = mmio_read($addr);

        if ($actual == $expected) {
            printf("  [PASS] d_mem[%d] (0x%08x) = %d (Matches Expected)\n", $offset, $
        } else {
            printf("  [FAIL] d_mem[%d] (0x%08x) = %d (Expected: %d)\n", $offset, $addr
            $fail_count++;
        }
    }

    print "\n==================================================\n";
    if ($fail_count == 0) {
        print "  FINAL RESULT: SUCCESS (All $total_checks checks passed)\n";
    } else {
        print "  FINAL RESULT: FAILURE ($fail_count out of $total_checks checks failed
    }
    print "==================================================\n";
}

# ================================================================
#  Main
# ================================================================

my $cmd = $ARGV[0] || "help";

if ($cmd eq "lab5") {
    reset_all();
    load_imem();
    load_dmem();
    verify_load();
    start_cpu_mmio();
    print "  CPU running... waiting 1s...\n";
    sleep(1);
    check_results();
} elsif ($cmd eq "load") {
    reset_all();
    load_imem();
    load_dmem();
    verify_load();
} elsif ($cmd eq "check") {
    check_results();
} elsif ($cmd eq "reset") {
    reset_all();
} else {
    print "Usage: pipelinereg [lab5 | load | check | reset]\n";
}
```

```
/*
   File: pipeline.v
   Description: Wrapper for the SoC.
   Updated: Split control signals into 1-to-1 registers.
*/
```

```
`include "define.v"

module pipeline
   #(
      parameter DATA_WIDTH = 64,
      parameter CTRL_WIDTH = DATA_WIDTH/8,
      parameter UDP_REG_SRC_WIDTH = 2
   )
   (
      // --- Register interface
      input                               reg_req_in,
      input                               reg_ack_in,
      input                               reg_rd_wr_L_in,
      input  [`UDP_REG_ADDR_WIDTH-1:0]    reg_addr_in,
      input  [`CPCI_NF2_DATA_WIDTH-1:0]   reg_data_in,
      input  [UDP_REG_SRC_WIDTH-1:0]      reg_src_in,

      output                              reg_req_out,
      output                              reg_ack_out,
      output                              reg_rd_wr_L_out,
      output  [`UDP_REG_ADDR_WIDTH-1:0]   reg_addr_out,
      output  [`CPCI_NF2_DATA_WIDTH-1:0]  reg_data_out,
      output  [UDP_REG_SRC_WIDTH-1:0]     reg_src_out,

      // misc
      input                                reset,
      input                                clk
   );

   //------------------------- Signals-------------------------------
   // Software registers (Host -> FPGA)
   // 1-to-1 mapping: Each signal gets a full 32-bit register
    wire [31:0] sw_req_cmd;         // Reg 0: [0] req_cmd
    wire [31:0] sw_req_addr;        // Reg 1: req_addr
    wire [31:0] sw_req_data_lo;     // Reg 2: req_data[31:0]
    wire [31:0] sw_req_data_hi;     // Reg 3: req_data[63:32]
    wire [31:0] sw_req_val;         // Reg 4: [0] req_val
    wire [31:0] sw_resp_rdy;        // Reg 5: [0] resp_rdy
    wire [31:0] sw_cpu_start;       // Reg 6: [0] start
    wire [31:0] sw_ila_debug_sel;   // Reg 7: [4:0] ila_debug_sel

    // Hardware registers (FPGA -> Host)
    reg [31:0] hw_req_rdy;          // Reg 0: [0] req_rdy
    reg [31:0] hw_resp_cmd;         // Reg 1: [0] resp_cmd
    reg [31:0] hw_resp_addr;        // Reg 2: resp_addr
    reg [31:0] hw_resp_data_lo;     // Reg 3: resp_data[31:0]
    reg [31:0] hw_resp_data_hi;     // Reg 4: resp_data[63:32]
    reg [31:0] hw_resp_val;         // Reg 5: [0] resp_val
    reg [31:0] hw_ila_debug_lo;     // Reg 6: ila_debug_data[31:0]
    reg [31:0] hw_ila_debug_hi;     // Reg 7: ila_debug_data[63:32]

    wire [63:0] req_data_64 = {sw_req_data_hi, sw_req_data_lo};

    wire                            soc_req_rdy;
    wire                                soc_resp_cmd;
    wire [31:0]                         soc_resp_addr;
    wire [63:0]                         soc_resp_data;
    wire                                soc_resp_val;
    wire [63:0]                         soc_ila_debug_data;

   //------------------------- Modules-------------------------------

    soc u_soc (
        .clk                (clk),
        .rst_n              (~reset),

        // ---- MMIO Request ----
        .req_cmd            (sw_req_cmd[0]),
        .req_addr           (sw_req_addr),
        .req_data           (req_data_64),
        .req_val            (sw_req_val[0]),
        .req_rdy            (soc_req_rdy),

        // ---- MMIO Response ----
        .resp_cmd           (soc_resp_cmd),
        .resp_addr          (soc_resp_addr),
        .resp_data          (soc_resp_data),
        .resp_val           (soc_resp_val),
        .resp_rdy           (sw_resp_rdy[0]),

        // ---- CPU control ----
        .start              (sw_cpu_start[0]),

        // ---- ILA debug ----
        .ila_debug_sel      (sw_ila_debug_sel[4:0]),
        .ila_debug_data     (soc_ila_debug_data)
    );

    // Logic to update hardware registers
    always @(posedge clk) begin
        if (reset) begin
            hw_req_rdy          <= 32'b0;
            hw_resp_cmd         <= 32'b0;
            hw_resp_addr        <= 32'b0;
            hw_resp_data_lo     <= 32'b0;
            hw_resp_data_hi     <= 32'b0;
            hw_resp_val         <= 32'b0;
            hw_ila_debug_lo     <= 32'b0;
            hw_ila_debug_hi     <= 32'b0;
        end
        else begin
            // Map single bits to LSB of 32-bit registers
            hw_req_rdy          <= {31'b0, soc_req_rdy};
            hw_resp_cmd         <= {31'b0, soc_resp_cmd};
            hw_resp_val         <= {31'b0, soc_resp_val};

            // Map full buses
            hw_resp_addr        <= soc_resp_addr;
            hw_resp_data_lo     <= soc_resp_data[31:0];
            hw_resp_data_hi   <= soc_resp_data[63:32];
            hw_ila_debug_lo   <= soc_ila_debug_data[31:0];
            hw_ila_debug_hi   <= soc_ila_debug_data[63:32];
        end
    end

    generic_regs
    #(
        .UDP_REG_SRC_WIDTH   (UDP_REG_SRC_WIDTH),
        .TAG                 (`PIPELINE_BLOCK_ADDR),
        .REG_ADDR_WIDTH      (`PIPELINE_REG_ADDR_WIDTH),
        .NUM_COUNTERS        (0),
        .NUM_SOFTWARE_REGS   (8),    // Increased to 8
        .NUM_HARDWARE_REGS   (8)     // Increased to 8
    ) module_regs (
        .reg_req_in        (reg_req_in),
        .reg_ack_in        (reg_ack_in),
        .reg_rd_wr_L_in    (reg_rd_wr_L_in),
        .reg_addr_in       (reg_addr_in),
        .reg_data_in       (reg_data_in),
        .reg_src_in        (reg_src_in),

        .reg_req_out       (reg_req_out),
        .reg_ack_out       (reg_ack_out),
        .reg_rd_wr_L_out   (reg_rd_wr_L_out),
        .reg_addr_out      (reg_addr_out),
        .reg_data_out      (reg_data_out),
        .reg_src_out       (reg_src_out),

        .counter_updates   (),
        .counter_decrement (),

        // --- Software Registers (Host -> Hardware) ---
        // Concatenation order: Reg 7 down to Reg 0
        .software_regs     ({sw_ila_debug_sel,    // Reg 7
                             sw_cpu_start,         // Reg 6
                             sw_resp_rdy,          // Reg 5
                             sw_req_val,           // Reg 4
                             sw_req_data_hi,       // Reg 3
                             sw_req_data_lo,       // Reg 2
                             sw_req_addr,          // Reg 1
                             sw_req_cmd}),         // Reg 0

        // --- Hardware Registers (Hardware -> Host) ---
        // Concatenation order: Reg 7 down to Reg 0
        .hardware_regs     ({hw_ila_debug_hi,     // Reg 7
                             hw_ila_debug_lo,      // Reg 6
                             hw_resp_val,          // Reg 5
                             hw_resp_data_hi,      // Reg 4
                             hw_resp_data_lo,      // Reg 3
                             hw_resp_addr,         // Reg 2
                             hw_resp_cmd,          // Reg 1
                             hw_req_rdy}),         // Reg 0

        .clk               (clk),
         .reset              (reset)
    );

endmodule
```

```
///////////////////////////////////////////////////////////////////////////////
// vim:set shiftwidth=3 softtabstop=3 expandtab:
// $Id: user_data_path.v 4385 2008-08-05 02:10:01Z grg $
//
// Module: user_data_path.v
// Project: NF2.1
// Description: contains all the user instantiated modules
//
///////////////////////////////////////////////////////////////////////////////
`timescale 1ns/1ps

/******************************************************
 * Even numbered ports are IO sinks/sources
 * Odd numbered ports are CPU ports corresponding to
 * IO sinks/sources to rpovide direct access to them
 ******************************************************/
module user_data_path
  #(parameter DATA_WIDTH = 64,
    parameter CTRL_WIDTH=DATA_WIDTH/8,
    parameter UDP_REG_SRC_WIDTH = 2,
    parameter NUM_OUTPUT_QUEUES = 8,
    parameter NUM_INPUT_QUEUES = 8,
    parameter SRAM_DATA_WIDTH = DATA_WIDTH+CTRL_WIDTH,
    parameter SRAM_ADDR_WIDTH = 19)

   (
    input  [DATA_WIDTH-1:0]            in_data_0,
    input  [CTRL_WIDTH-1:0]            in_ctrl_0,
    input                              in_wr_0,
    output                             in_rdy_0,

    input  [DATA_WIDTH-1:0]            in_data_1,
    input  [CTRL_WIDTH-1:0]            in_ctrl_1,
    input                              in_wr_1,
    output                             in_rdy_1,

    input  [DATA_WIDTH-1:0]            in_data_2,
    input  [CTRL_WIDTH-1:0]            in_ctrl_2,
    input                              in_wr_2,
    output                             in_rdy_2,

    input  [DATA_WIDTH-1:0]            in_data_3,
    input  [CTRL_WIDTH-1:0]            in_ctrl_3,
    input                              in_wr_3,
    output                             in_rdy_3,

    input  [DATA_WIDTH-1:0]            in_data_4,
    input  [CTRL_WIDTH-1:0]            in_ctrl_4,
    input                              in_wr_4,
    output                              in_rdy_4,

    input  [DATA_WIDTH-1:0]             in_data_5,
    input  [CTRL_WIDTH-1:0]             in_ctrl_5,
    input                               in_wr_5,
    output                              in_rdy_5,

    input  [DATA_WIDTH-1:0]             in_data_6,
    input  [CTRL_WIDTH-1:0]             in_ctrl_6,
    input                               in_wr_6,
    output                              in_rdy_6,

    input  [DATA_WIDTH-1:0]             in_data_7,
    input  [CTRL_WIDTH-1:0]             in_ctrl_7,
    input                               in_wr_7,
    output                              in_rdy_7,

/****  not used
    // --- Interface to SATA
    input  [DATA_WIDTH-1:0]             in_data_5,
    input  [CTRL_WIDTH-1:0]             in_ctrl_5,
    input                               in_wr_5,
    output                              in_rdy_5,

    // --- Interface to the loopback queue
    input  [DATA_WIDTH-1:0]             in_data_6,
    input  [CTRL_WIDTH-1:0]             in_ctrl_6,
    input                               in_wr_6,
    output                              in_rdy_6,

    // --- Interface to a user queue
    input  [DATA_WIDTH-1:0]             in_data_7,
    input  [CTRL_WIDTH-1:0]             in_ctrl_7,
    input                               in_wr_7,
    output                              in_rdy_7,
*****/

    output  [DATA_WIDTH-1:0]            out_data_0,
    output  [CTRL_WIDTH-1:0]            out_ctrl_0,
    output                              out_wr_0,
    input                               out_rdy_0,

    output  [DATA_WIDTH-1:0]            out_data_1,
    output  [CTRL_WIDTH-1:0]            out_ctrl_1,
    output                              out_wr_1,
    input                               out_rdy_1,

    output  [DATA_WIDTH-1:0]            out_data_2,
    output  [CTRL_WIDTH-1:0]            out_ctrl_2,
    output                              out_wr_2,
    input                               out_rdy_2,

    output  [DATA_WIDTH-1:0]            out_data_3,
    output  [CTRL_WIDTH-1:0]            out_ctrl_3,
    output                              out_wr_3,
    input                               out_rdy_3,

    output  [DATA_WIDTH-1:0]            out_data_4,
    output  [CTRL_WIDTH-1:0]            out_ctrl_4,
    output                              out_wr_4,
    input                               out_rdy_4,

    output  [DATA_WIDTH-1:0]            out_data_5,
    output  [CTRL_WIDTH-1:0]            out_ctrl_5,
    output                              out_wr_5,
    input                               out_rdy_5,

    output  [DATA_WIDTH-1:0]            out_data_6,
    output  [CTRL_WIDTH-1:0]            out_ctrl_6,
    output                              out_wr_6,
    input                               out_rdy_6,

    output  [DATA_WIDTH-1:0]            out_data_7,
    output  [CTRL_WIDTH-1:0]            out_ctrl_7,
    output                              out_wr_7,
    input                               out_rdy_7,

/****  not used
    // --- Interface to SATA
    output  [DATA_WIDTH-1:0]            out_data_5,
    output  [CTRL_WIDTH-1:0]            out_ctrl_5,
    output                              out_wr_5,
    input                               out_rdy_5,

    // --- Interface to the loopback queue
    output  [DATA_WIDTH-1:0]            out_data_6,
    output  [CTRL_WIDTH-1:0]            out_ctrl_6,
    output                              out_wr_6,
    input                               out_rdy_6,

    // --- Interface to a user queue
    output  [DATA_WIDTH-1:0]            out_data_7,
    output  [CTRL_WIDTH-1:0]            out_ctrl_7,
    output                              out_wr_7,
    input                               out_rdy_7,
*****/

     // interface to SRAM
     output [SRAM_ADDR_WIDTH-1:0]       wr_0_addr,
     output                             wr_0_req,
     input                              wr_0_ack,
     output [SRAM_DATA_WIDTH-1:0]       wr_0_data,

     input                              rd_0_ack,
     input  [SRAM_DATA_WIDTH-1:0]       rd_0_data,
     input                              rd_0_vld,
     output [SRAM_ADDR_WIDTH-1:0]       rd_0_addr,
     output                             rd_0_req,

     // interface to DRAM
     /* TBD */

     // register interface
     input                              reg_req,
     output                             reg_ack,
     input                              reg_rd_wr_L,
     input [`UDP_REG_ADDR_WIDTH-1:0]    reg_addr,
     output [`CPCI_NF2_DATA_WIDTH-1:0]  reg_rd_data,
     input [`CPCI_NF2_DATA_WIDTH-1:0]   reg_wr_data,

     // misc
     input                              reset,
     input                              clk);


   function integer log2;
      input integer number;
      begin
         log2=0;
         while(2**log2<number) begin
            log2=log2+1;
         end
      end
   endfunction // log2

   //---------- Internal parameters -----------

   localparam NUM_IQ_BITS = log2(NUM_INPUT_QUEUES);

   localparam IN_ARB_STAGE_NUM = 2;
   localparam OP_LUT_STAGE_NUM = 4;
   localparam OQ_STAGE_NUM     = 6;

   //-------- Input arbiter wires/regs -------
   wire                             in_arb_in_reg_req;
   wire                             in_arb_in_reg_ack;
   wire                             in_arb_in_reg_rd_wr_L;
   wire [`UDP_REG_ADDR_WIDTH-1:0]   in_arb_in_reg_addr;
   wire [`CPCI_NF2_DATA_WIDTH-1:0]  in_arb_in_reg_data;
   wire [UDP_REG_SRC_WIDTH-1:0]     in_arb_in_reg_src;

   //------- output port lut wires/regs ------
   wire [CTRL_WIDTH-1:0]            op_lut_in_ctrl;
   wire [DATA_WIDTH-1:0]            op_lut_in_data;
   wire                             op_lut_in_wr;
   wire                             op_lut_in_rdy;

   wire                             op_lut_in_reg_req;
   wire                             op_lut_in_reg_ack;
   wire                             op_lut_in_reg_rd_wr_L;
   wire [`UDP_REG_ADDR_WIDTH-1:0]   op_lut_in_reg_addr;
   wire [`CPCI_NF2_DATA_WIDTH-1:0]  op_lut_in_reg_data;
   wire [UDP_REG_SRC_WIDTH-1:0]     op_lut_in_reg_src;

   //------- pipeline wires/regs ------
```

```
    wire [CTRL_WIDTH-1:0]             pipeline_in_ctrl;
    wire [DATA_WIDTH-1:0]             pipeline_in_data;
    wire                              pipeline_in_wr;
    wire                              pipeline_in_rdy;

    wire                              pipeline_in_reg_req;
    wire                              pipeline_in_reg_ack;
    wire                              pipeline_in_reg_rd_wr_L;
    wire [`UDP_REG_ADDR_WIDTH-1:0]   pipeline_in_reg_addr;
    wire [`CPCI_NF2_DATA_WIDTH-1:0]  pipeline_in_reg_data;
    wire [UDP_REG_SRC_WIDTH-1:0]     pipeline_in_reg_src;

    //------- output queues wires/regs ------
    wire [CTRL_WIDTH-1:0]             oq_in_ctrl;
    wire [DATA_WIDTH-1:0]             oq_in_data;
    wire                              oq_in_wr;
    wire                              oq_in_rdy;

    wire                              oq_in_reg_req;
    wire                              oq_in_reg_ack;
    wire                              oq_in_reg_rd_wr_L;
    wire [`UDP_REG_ADDR_WIDTH-1:0]   oq_in_reg_addr;
    wire [`CPCI_NF2_DATA_WIDTH-1:0]  oq_in_reg_data;
    wire [UDP_REG_SRC_WIDTH-1:0]     oq_in_reg_src;

    //-------- UDP register master wires/regs -------
    wire                              udp_reg_req_in;
    wire                              udp_reg_ack_in;
    wire                              udp_reg_rd_wr_L_in;
    wire [`UDP_REG_ADDR_WIDTH-1:0]   udp_reg_addr_in;
    wire [`CPCI_NF2_DATA_WIDTH-1:0]  udp_reg_data_in;
    wire [UDP_REG_SRC_WIDTH-1:0]     udp_reg_src_in;


    //--------- Connect the data path -----------

    input_arbiter
      #(.DATA_WIDTH(DATA_WIDTH),
        .CTRL_WIDTH(CTRL_WIDTH),
        .UDP_REG_SRC_WIDTH (UDP_REG_SRC_WIDTH),
        .STAGE_NUMBER(IN_ARB_STAGE_NUM))
    input_arbiter
      (
     .out_data             (op_lut_in_data),
     .out_ctrl             (op_lut_in_ctrl),
     .out_wr               (op_lut_in_wr),
     .out_rdy              (op_lut_in_rdy),

       // --- Interface to the input queues
     .in_data_0            (in_data_0),
     .in_ctrl_0            (in_ctrl_0),
     .in_wr_0              (in_wr_0),
     .in_rdy_0             (in_rdy_0),

     .in_data_1            (in_data_1),
     .in_ctrl_1             (in_ctrl_1),
     .in_wr_1               (in_wr_1),
     .in_rdy_1              (in_rdy_1),

     .in_data_2             (in_data_2),
     .in_ctrl_2             (in_ctrl_2),
     .in_wr_2               (in_wr_2),
     .in_rdy_2              (in_rdy_2),

     .in_data_3             (in_data_3),
     .in_ctrl_3             (in_ctrl_3),
     .in_wr_3               (in_wr_3),
     .in_rdy_3              (in_rdy_3),

     .in_data_4             (in_data_4),
     .in_ctrl_4             (in_ctrl_4),
     .in_wr_4               (in_wr_4),
     .in_rdy_4              (in_rdy_4),

     .in_data_5             (in_data_5),
     .in_ctrl_5             (in_ctrl_5),
     .in_wr_5               (in_wr_5),
     .in_rdy_5              (in_rdy_5),

     .in_data_6             (in_data_6),
     .in_ctrl_6             (in_ctrl_6),
     .in_wr_6               (in_wr_6),
     .in_rdy_6              (in_rdy_6),

     .in_data_7             (in_data_7),
     .in_ctrl_7             (in_ctrl_7),
     .in_wr_7               (in_wr_7),
     .in_rdy_7              (in_rdy_7),

       // --- Register interface
     .reg_req_in            (in_arb_in_reg_req),
     .reg_ack_in            (in_arb_in_reg_ack),
     .reg_rd_wr_L_in        (in_arb_in_reg_rd_wr_L),
     .reg_addr_in           (in_arb_in_reg_addr),
     .reg_data_in           (in_arb_in_reg_data),
     .reg_src_in            (in_arb_in_reg_src),

     .reg_req_out           (op_lut_in_reg_req),
     .reg_ack_out           (op_lut_in_reg_ack),
     .reg_rd_wr_L_out       (op_lut_in_reg_rd_wr_L),
     .reg_addr_out          (op_lut_in_reg_addr),
     .reg_data_out          (op_lut_in_reg_data),
     .reg_src_out           (op_lut_in_reg_src),

       // --- Misc
     .reset                 (reset),
     .clk                   (clk)
     );

    output_port_lookup
```

```
     #(.DATA_WIDTH(DATA_WIDTH),
       .CTRL_WIDTH(CTRL_WIDTH),
       .UDP_REG_SRC_WIDTH (UDP_REG_SRC_WIDTH),
       .INPUT_ARBITER_STAGE_NUM(IN_ARB_STAGE_NUM),
       .STAGE_NUM(OP_LUT_STAGE_NUM),
       .NUM_OUTPUT_QUEUES(NUM_OUTPUT_QUEUES),
       .NUM_IQ_BITS(NUM_IQ_BITS))
   output_port_lookup
     (.out_data            (pipeline_in_data),
     .out_ctrl             (pipeline_in_ctrl),
     .out_wr               (pipeline_in_wr),
     .out_rdy              (pipeline_in_rdy),

      // --- Interface to the rx input queues
     .in_data              (op_lut_in_data),
     .in_ctrl              (op_lut_in_ctrl),
     .in_wr                (op_lut_in_wr),
     .in_rdy               (op_lut_in_rdy),

      // --- Register interface
     .reg_req_in           (op_lut_in_reg_req),
     .reg_ack_in           (op_lut_in_reg_ack),
     .reg_rd_wr_L_in       (op_lut_in_reg_rd_wr_L),
     .reg_addr_in          (op_lut_in_reg_addr),
     .reg_data_in          (op_lut_in_reg_data),
     .reg_src_in           (op_lut_in_reg_src),

     .reg_req_out          (pipeline_in_reg_req),
     .reg_ack_out          (pipeline_in_reg_ack),
     .reg_rd_wr_L_out      (pipeline_in_reg_rd_wr_L),
     .reg_addr_out         (pipeline_in_reg_addr),
     .reg_data_out         (pipeline_in_reg_data),
     .reg_src_out          (pipeline_in_reg_src),

      // --- Misc
     .clk                  (clk),
     .reset                (reset));

   pipeline #(
      .DATA_WIDTH(DATA_WIDTH),
      .CTRL_WIDTH(CTRL_WIDTH),
      .UDP_REG_SRC_WIDTH (UDP_REG_SRC_WIDTH)
      // .INPUT_ARBITER_STAGE_NUM(IN_ARB_STAGE_NUM),
      // .NUM_OUTPUT_QUEUES(NUM_OUTPUT_QUEUES),
      // .NUM_IQ_BITS(NUM_IQ_BITS)
   ) pipeline (

      // --- Register interface
      .reg_req_in                       (pipeline_in_reg_req),
      .reg_ack_in                       (pipeline_in_reg_ack),
      .reg_rd_wr_L_in                   (pipeline_in_reg_rd_wr_L),
      .reg_addr_in                      (pipeline_in_reg_addr),
      .reg_data_in                      (pipeline_in_reg_data),
      .reg_src_in                       (pipeline_in_reg_src),
      .reg_req_out                       (oq_in_reg_req),
      .reg_ack_out                       (oq_in_reg_ack),
      .reg_rd_wr_L_out                   (oq_in_reg_rd_wr_L),
      .reg_addr_out                      (oq_in_reg_addr),
      .reg_data_out                      (oq_in_reg_data),
      .reg_src_out                       (oq_in_reg_src),


      // --- Misc
      .clk                               (clk),
      .reset                             (reset)
   );

   output_queues
     #(.DATA_WIDTH(DATA_WIDTH),
       .CTRL_WIDTH(CTRL_WIDTH),
       .UDP_REG_SRC_WIDTH (UDP_REG_SRC_WIDTH),
       .OP_LUT_STAGE_NUM(OP_LUT_STAGE_NUM),
       .NUM_OUTPUT_QUEUES(NUM_OUTPUT_QUEUES),
       .STAGE_NUM(OQ_STAGE_NUM),
       .SRAM_ADDR_WIDTH(SRAM_ADDR_WIDTH))
   output_queues
     (// --- data path interface
    .out_data_0       (out_data_0),
    .out_ctrl_0       (out_ctrl_0),
    .out_wr_0         (out_wr_0),
    .out_rdy_0        (out_rdy_0),

    .out_data_1       (out_data_1),
    .out_ctrl_1       (out_ctrl_1),
    .out_wr_1         (out_wr_1),
    .out_rdy_1        (out_rdy_1),

    .out_data_2       (out_data_2),
    .out_ctrl_2       (out_ctrl_2),
    .out_wr_2         (out_wr_2),
    .out_rdy_2        (out_rdy_2),

    .out_data_3       (out_data_3),
    .out_ctrl_3       (out_ctrl_3),
    .out_wr_3         (out_wr_3),
    .out_rdy_3        (out_rdy_3),

    .out_data_4       (out_data_4),
    .out_ctrl_4       (out_ctrl_4),
    .out_wr_4         (out_wr_4),
    .out_rdy_4        (out_rdy_4),

    .out_data_5       (out_data_5),
    .out_ctrl_5       (out_ctrl_5),
    .out_wr_5         (out_wr_5),
    .out_rdy_5        (out_rdy_5),

    .out_data_6       (out_data_6),
    .out_ctrl_6       (out_ctrl_6),
    .out_wr_6         (out_wr_6),
    .out_rdy_6        (out_rdy_6),

    .out_data_7       (out_data_7),
    .out_ctrl_7       (out_ctrl_7),
    .out_wr_7         (out_wr_7),
    .out_rdy_7        (out_rdy_7),

      // --- Interface to the previous module
    .in_data          (oq_in_data),
    .in_ctrl          (oq_in_ctrl),
    .in_rdy           (oq_in_rdy),
    .in_wr            (oq_in_wr),

      // --- Register interface
    .reg_req_in       (oq_in_reg_req),
    .reg_ack_in       (oq_in_reg_ack),
    .reg_rd_wr_L_in   (oq_in_reg_rd_wr_L),
    .reg_addr_in      (oq_in_reg_addr),
    .reg_data_in      (oq_in_reg_data),
    .reg_src_in       (oq_in_reg_src),

    .reg_req_out      (udp_reg_req_in),
    .reg_ack_out      (udp_reg_ack_in),
    .reg_rd_wr_L_out  (udp_reg_rd_wr_L_in),
    .reg_addr_out     (udp_reg_addr_in),
    .reg_data_out     (udp_reg_data_in),
    .reg_src_out      (udp_reg_src_in),

      // --- SRAM sm interface
    .wr_0_addr        (wr_0_addr),
    .wr_0_req         (wr_0_req),
    .wr_0_ack         (wr_0_ack),
    .wr_0_data        (wr_0_data),
    .rd_0_ack         (rd_0_ack),
    .rd_0_data        (rd_0_data),
    .rd_0_vld         (rd_0_vld),
    .rd_0_addr        (rd_0_addr),
    .rd_0_req         (rd_0_req),


      // --- Misc
    .clk              (clk),
    .reset            (reset));


   //---------------------------------------------------
   //
   // --- User data path register master
   //
   //     Takes the register accesses from core,
   //     sends them around the User Data Path module
   //     ring and then returns the replies back
   //     to the core
   //
   //-----------------------------------------------------

   udp_reg_master #(
      .UDP_REG_SRC_WIDTH (UDP_REG_SRC_WIDTH)
   ) udp_reg_master (
      // Core register interface signals
      .core_reg_req                         (reg_req),
      .core_reg_ack                         (reg_ack),
      .core_reg_rd_wr_L                     (reg_rd_wr_L),

      .core_reg_addr                        (reg_addr),

      .core_reg_rd_data                     (reg_rd_data),
      .core_reg_wr_data                     (reg_wr_data),

      // UDP register interface signals (output)
      .reg_req_out                          (in_arb_in_reg_req),
      .reg_ack_out                          (in_arb_in_reg_ack),
      .reg_rd_wr_L_out                      (in_arb_in_reg_rd_wr_L),

      .reg_addr_out                         (in_arb_in_reg_addr),
      .reg_data_out                         (in_arb_in_reg_data),

      .reg_src_out                          (in_arb_in_reg_src),

      // UDP register interface signals (input)
      .reg_req_in                           (udp_reg_req_in),
      .reg_ack_in                           (udp_reg_ack_in),
      .reg_rd_wr_L_in                       (udp_reg_rd_wr_L_in),

      .reg_addr_in                          (udp_reg_addr_in),
      .reg_data_in                          (udp_reg_data_in),

      .reg_src_in                           (udp_reg_src_in),

      //
      .clk                                  (clk),
      .reset                                (reset)
   );


endmodule // user_data_path
```