A BASIC PROCESSING-IN-MEMORY (PIM) SYSTEM

Modern VLSI Design

Mohammad Parsa Bashari 400104812
Pariya Hajipour 400109605

Fall 2024

Contents

1 Architecture Overview

The Processing-In-Memory (PIM) system is designed to integrate memory and computational capabilities within a unified architecture. The system consists of three primary modules: **SRAM**, **ALU**, and **Control Unit (CU)**, which are interconnected to perform operations efficiently. This section provides an architectural overview of the PIM system, emphasizing its modular design and data flow.

The architecture diagram shown in Figure 1 illustrates the interaction between the key components: the SRAM module, the Arithmetic Logic Unit (ALU), and the Control Unit (CU). The architecture ensures the seamless integration of memory, computation, and control, allowing the PIM system to execute arithmetic, logical, and memory operations effectively.

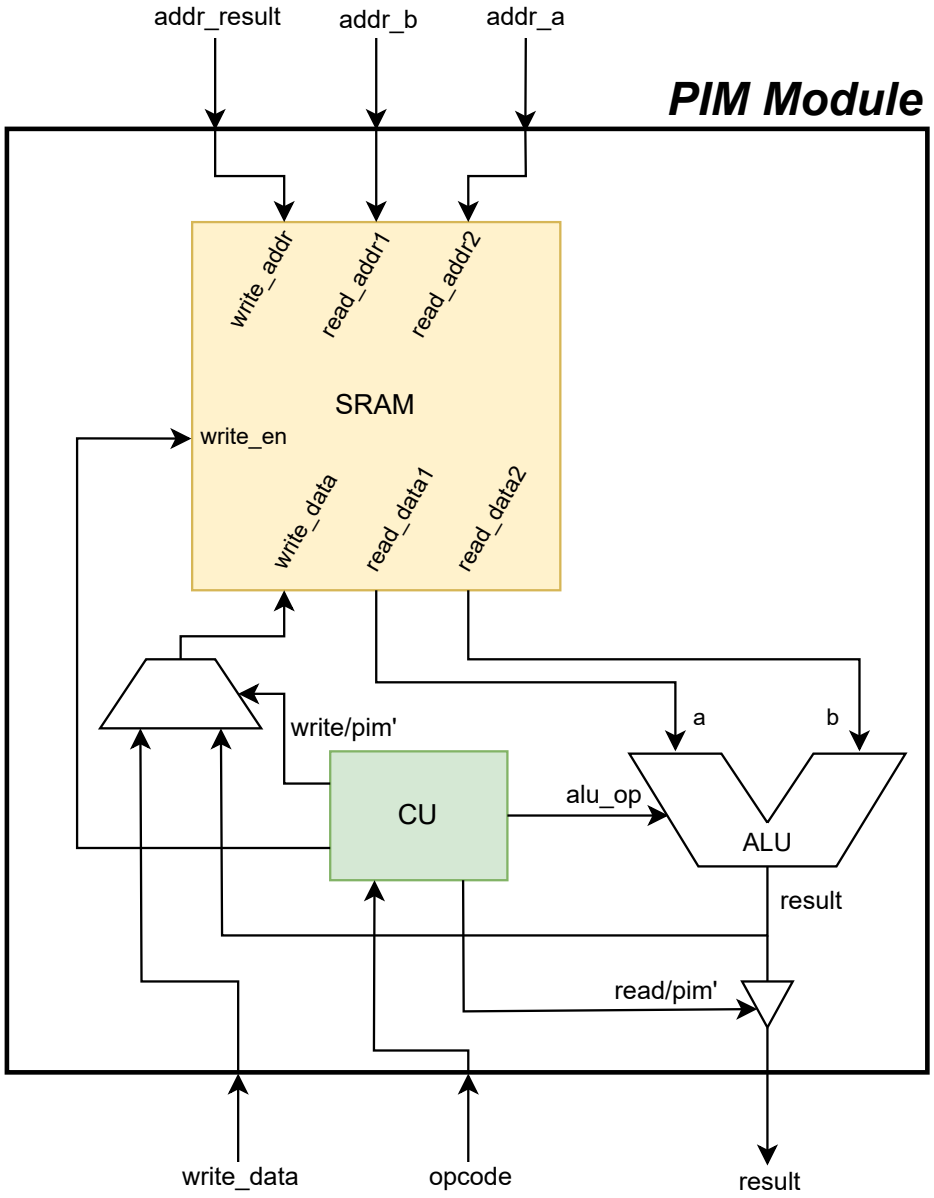

Figure 1: PIM System Architecture

1.1 Components

- **Data Flow and Storage:** The SRAM module serves as the primary memory unit, storing operands, intermediate data, and results. It supports simultaneous dual-read operations from two addresses (`addr_a` and `addr_b`) and a single write operation to a specified address (`addr_result`). Data flows into the ALU from the SRAM, where operands (`a` and `b`) are processed based on the operation specified by the Control Unit.
- **Computation:** The ALU performs arithmetic and logical operations on the inputs (`a` and `b`) based on the control signal (`alu_op`). The result of the operation is either written back to the SRAM or made available at the system's output (`result`).
- **Control:** The Control Unit (CU) decodes the `opcode` signal to generate control signals for the SRAM, ALU, and the overall system. It determines the type of operation (e.g., memory write, memory read, arithmetic operation) and configures the ALU and SRAM accordingly.
- **Instruction Handling:** Based on the `opcode`, the system can perform read operations from memory (`read_pim_not`), execute write operations to memory (`write_pim_not`), and execute PIM-enabled arithmetic or logical operations using the ALU (`alu_op`).

1.2 Key Interactions

- **SRAM and ALU:** The operands (`a` and `b`) are fetched from SRAM based on addresses provided (`addr_a` and `addr_b`). After computation, the result is either written to memory at `addr_result` or directed to the output.
- **SRAM and CU:** The Control Unit generates signals (`write_enable`, `write_pim_not`, `read_pim_not`) to manage memory operations, ensuring proper synchronization during reads and writes.
- **CU and ALU:** The Control Unit configures the ALU operation via the `alu_op` signal. This ensures that the correct arithmetic or logical operation is performed based on the instruction.

2 Code Overview

2.1 SRAM Module

The module operates on a clock signal and takes control inputs such as the write enable (`we`) signal. When this signal is asserted, the data specified by `write_data` is written to the address `write_addr` at the rising edge of the clock. In addition, the SRAM drives the output ports `read_data1` and `read_data2` with the contents of memory stored at `read_addr1` and `read_addr2`, respectively. This asynchronous dual-read capability enhances parallelism by enabling simultaneous access to two memory locations.

```
module SRAM #(parameter ADDR_WIDTH = 10, DATA_WIDTH = 32) (
    input wire clk, we,
    input wire [ADDR_WIDTH-1:0] write_addr, read_addr1, read_addr2,
    input wire [DATA_WIDTH-1:0] write_data,
    output wire [DATA_WIDTH-1:0] read_data1, read_data2
);
    reg [DATA_WIDTH-1:0] mem [0:(1<<ADDR_WIDTH)-1];

    always @(posedge clk) begin
        if (we)
            mem[write_addr] <= write_data;
    end

    assign read_data1 = mem[read_addr1];
    assign read_data2 = mem[read_addr2];
endmodule
```

Code Snippet 1: SRAM Module Implementation

2.2 ALU Module

The ALU operates combinationally, providing results immediately based on the input operands and the selected operation. For example, when `op` is `00`, the ALU adds the operands; when `op` is `01`, it performs subtraction. Similarly, logical operations such as AND and OR are performed for `10` and `11`, respectively.

```
module ALU #(parameter DATA_WIDTH = 32)  (
    input wire [DATA_WIDTH-1:0] a, b,
    input wire [1:0] op,
    output reg [DATA_WIDTH-1:0] result
);
    always @(*) begin
        case (op)
            2'b00: result = a + b;
            2'b01: result = a - b;
            2'b10: result = a & b;
            2'b11: result = a | b;
            default: result = {DATA_WIDTH{1'b0}};
        endcase
    end
endmodule
```

Code Snippet 2: ALU Module Implementation

2.3 CU Module

The CU ensures synchronization and coordination between the ALU and SRAM, enabling seamless data flow and execution of instructions. The 3-bit `opcode` determines the specific operation to be performed. For example, the CU derives the ALU operation control signals (`alu_op`) from the lower two bits of the `opcode`. Additionally, the CU identifies specialized operations such as PIM-specific writes and reads using predefined opcode values. The write enable (`write_enable`) signal is also generated by the CU, enabling or disabling memory write operations as needed.

```
module CU (
    input wire [2:0] opcode,
    output wire write_pim_not, read_pim_not, write_enable,
    output wire [1:0] alu_op
);
    assign alu_op = opcode[1:0];
    assign write_pim_not = opcode == 3'b100;
    assign read_pim_not = opcode == 3'b111;
    assign write_enable = (opcode[2] == 0) || write_pim_not;
endmodule
```

Code Snippet 3: CU Module Implementation

2.4 PIM Module

The `PIM Module` is the top-level module that integrates the SRAM, ALU, and CU to form a functional processing-in-memory system. It defines the external interface, including inputs such as clock signal (`clk`), start signal (`start`), addresses for operands and results, the operation code (`opcode`), and data for write operations (`write_data`). The primary output of the system is the computation result (`result`).

```
module PIM_Module #(parameter DATA_WIDTH=32, ADDR_WIDTH=10) (
    input wire clk,
    input wire [ADDR_WIDTH-1:0] addr_a, addr_b, addr_result, // operand addresses
    input wire [2:0] opcode, // 100: write 'write_data' to 'addr_result', 111: read 'addr_a'
    input wire [DATA_WIDTH-1:0] write_data,
    output wire [DATA_WIDTH-1:0] result
);

    wire [DATA_WIDTH-1:0] a, b, mem_write_data, alu_result;
    wire [1:0] alu_op;
    wire we, write_pim_not, read_pim_not;

    assign mem_write_data = write_pim_not ? write_data : alu_result;
    assign result = read_pim_not ? a : {DATA_WIDTH{1'bz}};

    SRAM sram (
        .clk(clk),
        .we(we),
        .write_addr(addr_result),
        .write_data(mem_write_data),
        .read_addr1(addr_a),
        .read_addr2(addr_b),
        .read_data1(a),
        .read_data2(b)
    );


```

```
28
29      ALU alu (
30          .a(a),
31          .b(b),
32          .op(alu_op),
33          .result(alu_result)
34      );
35
36      CU cu (
37              .opcode(opcode),
38              .alu_op(alu_op),
39              .write_pim_not(write_pim_not),
40              .read_pim_not(read_pim_not),
41              .write_enable(we)
42      );
43
44  endmodule
```

Code Snippet 4: PIM Module Implementation

The module fetches operands from the SRAM using the addresses `addr_a` and `addr_b`, as directed by the control signals generated by the CU. Based on the operation specified by the `opcode`, the operands are processed through the ALU. The result of the computation is either written back to memory or made available as output, depending on the control logic.

For PIM-specific operations, the CU identifies specialized instructions, such as writing a predefined value to memory or reading data directly from a specified memory location. These specialized operations allow the system to handle unique memory-processing tasks effectively.

2.5 Conclusion

The PIM system combines modular components—SRAM, ALU, and CU—to provide a versatile and efficient processing-in-memory solution. The SRAM ensures efficient data access, the ALU delivers computational power, and the CU coordinates operations seamlessly. This design is well-suited for applications that require high-performance memory and processing integration. Through its modularity and clear signal flow, the system achieves an optimal balance of complexity and functionality, making it a compelling solution for modern computing challenges.

3 Testbench Overview

The testbench validates the functionality of the Processing-In-Memory (PIM) system by simulating a wide range of operations, including memory read and write, arithmetic and logical computations, and edge cases such as overflow and underflow. The simulation outputs include a table of results and a waveform, both of which confirm that the system operates as designed.

The simulation begins by initializing all inputs and issuing a `NOP` (No Operation) instruction to stabilize the system. The testbench then progresses through a series of operations to assess the behavior of the PIM system. Basic memory operations are tested first by writing specific values to addresses in SRAM and verifying their correctness through subsequent read operations. For instance, the value `20` is written to the lowest address `0x000`, and a read operation confirms that the retrieved value matches the expected result. Similarly, the value `91` is written to the highest address `0x3FF`, and its correctness is verified.

```
{clk, addr_a, addr_b, addr_result, write_data} = 0;
opcode = 3'b101;  // no op

// write 20 to 0x000
#4
opcode = 3'b100;
addr_result = 10'h0;
write_data = 32'd20;
// read 0x000
#10
opcode = 3'b111;
addr_a = 10'h0;

// write 91 to 0x3FF
#10
opcode = 3'b100;
addr_result = 10'h3FF;
write_data = 32'd91;
// read 0x3FF
#10
opcode = 3'b111;
addr_a = 10'h3FF;
```

Code Snippet 5: TB Module Initialization in Critical Addresses

Arithmetic operations are tested next. The testbench instructs the ALU to perform addition, subtraction, bitwise AND, and OR operations using operands stored in SRAM. The results are written back to specific memory addresses, and their correctness is verified by reading and displaying the stored values. For example, the sum of the values at addresses `0x000` and `0x3FF` is stored at address `0x100` and confirmed to be `111`. Similarly, subtraction, AND, and OR operations produce results of `-71`, `16`, and `95`, respectively.

```
// store mem[0x000] + mem[0x3FF] at address 0x100
#10
opcode = 3'b000; // add opcode
addr_a = 10'h000;
addr_b = 10'h3FF;
addr_result = 10'h100;

// store mem[0x000] - mem[0x3FF] at address 0x101
#10
opcode = 3'b001; // subtract opcode
addr_result = 10'h101;

// store mem[0x000] & mem[0x3FF] at address 0x102
#10
opcode = 3'b010; // subtract opcode
addr_result = 10'h102;

// store mem[0x000] | mem[0x3FF] at address 0x103
#10
opcode = 3'b011; // subtract opcode
addr_result = 10'h103;
```

Code Snippet 6: TB Module Operations

The testbench also tests edge cases involving maximum and minimum integer values. At address `0x001`, the maximum integer value (`2147483647`) is stored and retrieved correctly. At address `0x002`, the minimum integer value (`-2147483648`) is handled in a similar manner. The system's handling of arithmetic overflow and underflow is also tested. For instance, adding the maximum integer value (`2147483647`) to another operand results in an overflow, producing a value of `-2147483629`, which is stored at address `0x105`. Similarly, subtracting the minimum integer value (`-2147483648`) from another operand results in an underflow, producing `2147483628`, stored at address `0x106`.

```
// write 0x7FFFFFFF to 0x001
#10
opcode = 3'b100;
addr_result = 10'h001;
write_data = 32'h7FFFFFFF;

// read 0x001
#10
opcode = 3'b111;
addr_a = 10'h001;

// write 0x80000000 to 0x002
#10
opcode = 3'b100;
addr_result = 10'h002;
write_data = 32'h80000000;

// read 0x002
#10
opcode = 3'b111;
addr_a = 10'h002;

// store mem[0x000] + mem[0x001] at address 0x105
#10
opcode = 3'b000;
addr_b = 10'h001;
addr_result = 10'h105;

// store mem[0x002] - mem[0x000] at address 0x106
#10
opcode = 3'b001;
addr_a = 10'h002;
addr_b = 10'h0;
addr_result = 10'h106;
```

Code Snippet 7: TB Module Critical Values, Overflow, and Underflow handling

The testbench verifies the system's behavior when encountering an invalid opcode. When an invalid operation code is provided, the system ensures no unintended behavior occurs by setting the signal `write_enable` to zero in CU. The output at the designated address reflects no meaningful data, confirming that the system does not perform any changes to the specified location.

```
// store invalid opcode at address 0x104
#10
opcode = 3'b110; // invalid opcode
addr_result = 10'h104;
```

Code Snippet 8: TB Module Invalid Upcode Handling

The results of these operations are captured both in a tabular format (Figure 2) and as a waveform (Figure 3). The tabular output provides a summary of the results, including memory addresses, stored or computed values, and the corresponding operation descriptions. For instance, the value at address `0x100` is confirmed to be the result of an addition operation, while the value at address `0x105` demonstrates the effect of an overflow condition. The waveform offers a time-domain visualization of the testbench execution, showing how signals such as `clk`, `addr_a`, `addr_b`, `addr_result`, `write_data`, `opcode`, and `result` evolve over time. It corroborates the tabular results and provides further evidence of correct system behavior.

```
VSIM 82> run
# value at address 0x000 is          20 (a)
# value at address 0x3ff is          91 (b)
# value at address 0x001 is  2147483647 (max int)
# value at address 0x002 is -2147483648 (min int)
# value at address 0x100 is         111 => a + b
# value at address 0x101 is         -71 => a - b
# value at address 0x102 is          16 => a & b
# value at address 0x103 is          95 => a | b
# value at address 0x104 is           x => invalid opcode (no op)
# value at address 0x105 is -2147483629 => a + max int (overflow)
# value at address 0x106 is  2147483628 => min int - a (underflow)
```

Figure 2: Simulation Results: Tabular Output

Figure 3: Simulation Waveform: Signal Timing

The clock signal in the waveform toggles consistently, ensuring proper synchronization across the system. Memory addresses and data values transition appropriately as operations are performed, and the opcode signal directs the system to execute specific tasks. The result signal accurately reflects the outcome of memory reads or ALU computations at each step.