Computer Organization and Processor Architecture Project Report

Verilog Design of a RISCV Single Cycle Processor and Performing Bubble and Insertion Sort on it

Bhuvanesh Ganta – B23485
Gnyaneshwar Mulkala – B23496
Godaba Jasmini – B23490
Nanda Kishore Yadla – B23497
Shersingh Meena – B23501
Shubham Meena – B23467
Vishwas Jasuja – B23303

6 May 2025

Contents

1 Introduction

The RISC-V (Reduced Instruction Set Computer – Five) architecture is an open-source, modular instruction set architecture (ISA) designed to support a wide range of computing applications, from embedded systems to high-performance processors. Its simplicity, extensibility, and community-driven development make it an ideal foundation for academic and industrial processor design projects.

This project focuses on the **implementation and simulation of a single-cycle, single core RISC-V processor datapath** in Verilog. The design includes all key stages of instruction execution—*fetch, decode, operand fetch, execution, memory access, and write-back*—and integrates essential components such as the *program counter (PC), instruction memory (IMEM), register file, ALU, data memory (DRAM), immediate generator*, and *control unit (decoder)*.

The primary objective of the project is to construct a functional processor capable of executing a subset of RISC-V instructions, verify its behavior through simulation, and analyze internal signal propagation and control flow using a detailed testbench. Special attention is given to handling control instructions such as `branches` and `jumps`, sign-extending immediates appropriately.

Through this project, we aim to:

- Understand the working of RISC-V architecture at the microarchitectural level
- Gain proficiency in Verilog HDL and digital design practices
- Develop debugging and simulation skills using waveform viewers and testbenches
- Validate the correctness of instruction execution via signal observation and register output tracking

This report documents the design methodology, module-wise breakdown, integration strategy, and simulation outcomes of the processor datapath, offering insights into how modern CPUs handle instruction execution at the hardware level.

2 RISC V Processor

The processor is composed of the following major components:

- **Program Counter (PC)**: Holds the address of the current instruction and handles sequential execution (`PC+4`) and jumps/branches using sign-extended offsets..
- **Instruction Memory (IMEM)**: Stores instructions to be fetched and executed.
- **Decoder**: Generates control signals based on the instruction opcode and funct fields.
- **Register File**: Contains general-purpose registers for data storage and retrieval.
- **ALU (Arithmetic Logic Unit)**: Performs arithmetic and logic operations.
- **Data Memory (DRAM)**: Stores data that can be loaded and stored.
- **Sign Extension Unit**: Extends immediate values from instruction fields. Dynamically adjusts for:
  - **I-type** (12-bit immediates for arithmetic/logical ops)
  - **B-type** (13-bit PC-relative branch offsets, LSB=0)
  - **S-type** (12-bit memory offsets for `LW`/`SW`)
- **Datapath**: Interconnects all the components to form the functional processor.

Each of these components is implemented as a separate Verilog module and integrated into a top-level module to execute instructions.

2.1 Fetch Stage - PC and IMEM

The fetch stage is responsible for retrieving instructions from memory and updating the program counter.

- **PC Module**:

```
module PC(
    input clk, rst, PCsel,
    input [31:0] imm_in,
    output reg [31:0] pc
);
always @(posedge clk or posedge rst) begin
    if (rst) pc <= 32'b0;
    else if (PCsel) pc <= pc + imm_in; // Branch/jump
    else pc <= pc + 4; // Normal increment
end
endmodule
```

Program Counter (PC)

  - Tracks the address of the current instruction in memory.
  - Updates on the clock edge based on:
    - **PC+4**: Default sequential execution (word-aligned, 4-byte increment).
    - **Branch/Jump Target**: Calculated as `PC + sign-extended immediate` (for `BEQ`, `BNE`, `JAL`, etc.).
  - Controlled by `PCsel` (from decoder) and `branch_en` (gated by ALU condition for branches)

- **IMEM (Instruction Memory)**:

```
module IMEM(
    input [31:0] address,
    input clk, rst,
    output wire [31:0] ins
);
reg [31:0] mem[1000:0];
assign ins = mem[address>>2];
endmodule
```

- **Instruction Memory (IMEM)**
  - **Byte-addressable**, 32-bit wide memory storing RISC-V instructions.
  - Uses the PC value as the read address (`address = PC >> 2` for word alignment).
  - Asynchronous read: Outputs the instruction immediately during the clock cycle.
  - Initialized with test programs (e.g., sorting algorithms, max-value search).

2.2 Decode Stage - Decoder

The decoder extracts control signals from the 32-bit instruction to guide the datapath during instruction execution.

```
module decoder(
    input [31:0] instruction,
    output wire PCsel,          // Branch/jump
    output wire [1:0] immsel,   // Immediate type
    output wire regwrite,       // Register write enable
    output wire memwrite,       // Memory write
    output wire [3:0] alucon,   // ALU operation
    output wire alusrc,         // ALU source (reg/imm)
    output wire resultSRC,      // Result source (ALU/mem)
    output wire signext         // Sign extend
);
// Control logic implementation...
endmodule
```

The decoder examines the opcode and function (funct3/funct7) fields of the instruction to generate the following control signals:

- **PCsel**: Determines the source of the next program counter (PC) value. It is asserted (set to 1) for control-flow instructions like `jal`, `jalr`, and conditional branches, indicating that the next PC comes from the ALU result (i.e., a branch target).
- **immsel [1:0]**: Selects the immediate format based on instruction type. For example:
  - 00 – I-type immediate (e.g., `addi`, `lw`)
  - 01 – S-type immediate (e.g., `sw`)
  - 10 – B-type immediate (e.g., `beq`)
  - 11 – U/J-type immediate (e.g., `lui`, `jal`)
- **regwrite**: Enables writing to the register file. It is set for instructions that update a destination register, such as arithmetic instructions, `lw`, and `lui`.
- **memwrite**: Controls writing to memory. It is asserted for store instructions like `sw` and `sb`.
- **alucon [3:0]**: Encodes the specific ALU operation to perform. This is derived from both the opcode and funct fields. Examples:
  - 0000 – ADD
  - 0001 – SUB
  - 0010 – AND
  - 0011 – OR

    - 0100 – XOR
    - 0101 – SLL (Shift Left Logical)
    - 0110 – SRL/SRA (Shift Right Logical/Arithmetic)
    - 0111 – SLT/SLTU (Set Less Than)
- **alusrc**: Selects the second operand of the ALU. If 0, it uses a register value (R-type). If 1, it uses an immediate value (I-type, load/store, etc.).
- **resultSRC**: Chooses the source for the value to write back to the register. If 0, the ALU result is used. If 1, the data comes from memory (for `lw`).
- **signext**: Enables sign-extension of immediate values. It is set for signed immediates (most instructions), but cleared for instructions with zero-extended immediates (like `andi` in certain variants).

2.3 Operand Fetch - Register File

The register file provides read/write access to processor registers.

```
module registerfile(
    input wire clk, rst, regwrite,
    input wire [4:0] read_addr1, read_addr2,
    input wire [4:0] write_addr,
    input wire [31:0] write_data,
    output wire [31:0] read_data1,
    output wire [31:0] read_data2
);
reg [31:0] registers[31:0];
// Asynchronous read
assign read_data1 = (read_addr1 != 0) ? registers[read_addr1] : 32'b0;
assign read_data2 = (read_addr2 != 0) ? registers[read_addr2] : 32'b0;
// Synchronous write
always @(posedge clk or posedge rst) begin
    if (rst) begin /* Initialize registers */ end
    else if (regwrite) registers[write_addr] <= write_data;
end
endmodule
```

Key features:

- Asynchronous read (zero register hardwired to 0)
- Synchronous write on clock edge
- Initialized with test values for bubble sort

2.4 Execute Stage - ALU

The ALU performs arithmetic, logical, and comparison operations, as well as address computations for memory access and control flow.

```
module ALU(
    input [31:0] a,
    input [31:0] b,
    input [3:0] alucon,
    output reg [31:0] result
);
    always @(*) begin
        // Default values
        result = 32'b0;

        case (alucon)
            4'b0010: result = a + b;                    // ADD
```

```
            4'b1010: result = a >> b;                          // SRL (logical right shift
    )
            4'b0110: result = (a == b);                        // BEQ (equal comparison)
            4'b0001: result = a & b;                           // AND
            4'b1000: result = a * b;                           // MUL
            4'b1110: result = (a != b);                        // BNE (not equal)
            4'b0111: result = a + b;                           // Address calculation
            4'b0101: result = a + b;                           // Address calculation
            4'b1111: result = ($signed(a) > $signed(b));       // BLT (signed less than)
            4'b1100: result = ($signed(a) <= $signed(b));      // BGE (signed greater or
    equal)
            4'b1001: result = 32'b1;                            // JAL (unconditional jump
     marker)
            default: result = 32'b0;
        endcase
    end
endmodule
```

Supported operations include:

- **Arithmetic:** ADD, SUB (previously encoded but replaced here by SRL)
- **Logical:** AND, SRL (logical right shift), MUL
- **Comparisons:** BEQ (==), BNE (!=), BLT (`signed <`), BGE (`signed` $\geq$)
- **Control flow and address handling:** ADD-based address calculation, JAL (jump flag)

2.5 Memory Access - DRAM

Data memory for load/store operations.

```
module DRAM(
    input clk, rst, memwrite,
    input [31:0] address,
    input [31:0] write_data,
    output [31:0] read_data
);
reg [31:0] memory [0:255];
// Synchronous write
always @(posedge clk or posedge rst) begin
    if (rst) begin /* Initialize memory */ end
    else if (memwrite) memory[address] <= write_data;
end
// Asynchronous read
assign read_data = memory[address];
endmodule
```

Features:

- 256-word memory space
- Asynchronous read, synchronous write
- Initialized with test data for sorting

2.6 Write Back Stage - Datapath and registerfile

The write back logic selects between ALU result and memory load data.

```
// In dpath module:
assign reg_write_data = resultSRC ? read_data : alu_result;
```

The resultSRC signal from the decoder determines whether to write back:

- ALU result (for arithmetic/logical operations)
- Memory read data (for load instructions)

2.7 Immediate Generation - signextend

Immediate generation .

```
module signextend(
    input [31:0] instruction,
    input signext, [1:0] immsel,
    output wire [31:0] imm_out
);
assign imm_out = (signext) ?
    (immsel == 2'b01) ? /* B-type */ :
    (immsel == 2'b10) ? /* J-type */ :
    /* Other types */ : 32'b0;
endmodule
```

- The `signextend` module extracts and sign-extends immediate values from RISC-V instructions depending on the instruction type, indicated by the 2-bit `immsel` signal.
- If `signext` is enabled (i.e., asserted), the output `imm_out` is computed based on the following `immsel` cases:
  - **B-type (Branch instructions),** `immsel == 2'b01`**:** The offset is formed by combining bits `[31], [7], [30:25], [11:8], 1'b0`. The immediate is sign-extended to 32 bits, with bit [31] replicated for the top 20 bits. The final offset is left-shifted by 1 (via `1'b0` concatenation) to align with instruction word boundaries.
  - **J-type (Jump instructions),** `immsel == 2'b10`**:** The offset is built from `[31], [19:12], [20], [30:21], 1'b0`. If the sign bit `instruction[31]` is 1, the immediate is negated and sign-extended to handle large negative jumps; otherwise, it is extended as a positive offset. This format handles 21-bit signed offsets used in `JAL`-type instructions.
  - **S-type (Store instructions),** `immsel == 2'b11`**:** The immediate is constructed from `[31:25]` and `[11:7]`, forming a 12-bit value sign-extended to 32 bits. This reflects how RISC-V splits the offset between two fields for store instructions.
  - **I-type (Immediate arithmetic/load),** `immsel == 2'b00`**:** The immediate is directly taken from bits `[31:20]` and sign-extended. This is used for arithmetic immediate, load, and JALR instructions.
  - If none of the conditions match, `imm_out` defaults to `32'b1` or `32'b0` depending on `signext`.
- This careful bit-slicing and concatenation ensures that the processor extracts correct offsets for branching, memory access, and jump operations, complying with the RISC-V instruction encoding formats.

3 Overall Integration in Datapath

The datapath integrates all the fundamental components required to execute instructions in a RISC-V-based processor. This includes instruction fetch, decode, execution, memory access, and write-back stages. The integration ensures that control and data signals flow coherently across the modules on every clock cycle, driven by the control logic and the program counter updates.

```
module dpath(
    input clk,
    input rst,
    output [31:0] ins
);
    wire [31:0] pc_out;
    wire [31:0] instruction, imm_out, read_data1, read_data2, alu_result;
    wire [31:0] read_data, reg_write_data;
    wire [3:0] alucon;
    wire alusrc, regwrite, signext, memwrite, resultSRC;
    wire [1:0] immsel;
    wire [4:0] rs1, rs2, rd;
```

```
    wire [31:0] alu_b_input;
    wire PCsel;
    wire branch_en; // New port from ALU: branch condition flag

    assign rs1 = instruction[19:15];
    assign rs2 = instruction[24:20];
    assign rd  = instruction[11:7];

    // Gate branch enable: only assert PCsel when decoder says branch and ALU flag is
    true
    assign branch_en = PCsel & alu_result;

    // PC Module
    PC pc_inst (//);

    // DRAM (Data Memory)
    DRAM dram_inst (//);

    // Instruction Memory
    IMEM imem_inst (
        .address(pc_out),
        .clk(clk),
        .win(32'b0),
        .rst(rst),
        .ins(instruction)
    );

    assign ins = instruction;

    // Register File
    registerfile rf_inst (//);

    // Sign Extend
    signextend se_inst (
        .instruction(instruction),
        .signext(signext),
        .immsel(immsel),
        .imm_out(imm_out)
    );

    // ALU Input Mux
    assign alu_b_input = alusrc ? imm_out : read_data2;

    // ALU with new flag port
    ALU alu_inst (
        .a(read_data1),
        .b(alu_b_input),
        .alucon(alucon),
        .result(alu_result) // condition flag output
    );

    // MUX for register file write data (ALU result or DRAM read_data)
    assign reg_write_data = resultSRC ? read_data : alu_result;

    // Decoder
    decoder decoder_inst ();
endmodule
```

3.1 Data Flow

- **Instruction Fetch:** The Program Counter (PC) module holds the address of the current instruction. Based on the branch condition and control signal `PCsel`, it either increments normally or jumps to a new address determined by the immediate field.
- **Instruction Decode:** The instruction fetched from the Instruction Memory (`IMEM`) is decoded into its respective fields: `rs1`, `rs2`, and `rd`, which are used to read from and write to the Register

File. The `decoder` module generates control signals like `regwrite`, `memwrite`, `alusrc`, `PCsel`, etc., based on the opcode and function fields.

- **Register File:** Two source operands are read from the Register File using `rs1` and `rs2`. These are provided as inputs to the ALU or, in the case of a store instruction, to the data memory. The result from the ALU or data memory is written back to the register specified by `rd`.
- **Immediate Generation:** The `signextend` module extracts and appropriately sign-extends immediate values from the instruction. The immediate type is selected using `immsel` and optionally extended based on `signext`.
- **ALU Operations:** The ALU receives its first operand from the register file. The second operand is selected via a multiplexer controlled by `alusrc`, allowing it to choose between another register value or the immediate. The operation performed by the ALU is controlled by the `alucon` signal.
- **Branch Logic:** The output of the ALU is also used as a branch condition flag. The final decision to update the PC is based on a gated signal `branch_en`, which is the logical AND of `PCsel` and the ALU result, ensuring that branching occurs only when the condition is satisfied.
- **Memory Access and Write-Back:** The Data Memory (`DRAM`) is accessed if the instruction is a load or store. In the case of a store, data is written from the register file into memory. In the case of a load, data is fetched from memory. The `resultSRC` control signal selects whether the ALU result or the memory data should be written back into the destination register.

3.2 Control Flow

- A gating mechanism is implemented in the datapath to control branching behavior precisely.
- The decoder generates a signal `PCsel` to indicate whether a branch instruction is present.
- The actual branch decision is made using a gated signal `branch_en`, which is the logical AND of `PCsel` and the `ALU` result.
- The `ALU` result serves as a branch condition flag (e.g., zero, less than), indicating if the branch condition is satisfied.
- This mechanism ensures that branching only occurs if both the control signal and the condition flag are true, avoiding incorrect jumps.

This modular and well-coordinated datapath design allows sequential instruction processing while maintaining flexibility for control flow changes, such as branches and jumps. It highlights how control signals from the decoder orchestrate the actions of all functional units in a cycle-accurate manner.

4 From C Code to Assembly to Machine-Level Code: Bubble Sort

This section demonstrates the transition from a high-level C program to RISC-V assembly code and finally to machine-level code, using a bubble sort implementation as an example. The program sorts an array of integers in ascending order, printing the array before and after sorting. The array used is `[25, 509, 8, 31, 11, 0, 66]` with a size of 7 elements.

4.1 C Code

The C code defines the array, prints it, performs bubble sort by swapping the maximum element to the end in each pass, and prints the sorted array. The output format includes space-separated numbers with a newline at the end of each array print.

```c
#include <stdio.h>

void print_array(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

void array_pairs(int *arr, int size) {
    for (int i = 0; i < size - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            // Swap to move larger element to the right
            int temp = arr[i];
            arr[i] = arr[i + 1];
            arr[i + 1] = temp;
        }
    }
}

int main() {
    int arr[] = {25, 509, 8, 31, 11, 0, 66};
    int size = 7;

    // Print initial array
    print_array(arr, size);

    // Perform bubble sort (size passes)
    for (int i = 0; i < size; i++) {
        array_pairs(arr, size);
    }

    // Print sorted array
    print_array(arr, size);

    return 0;
}
```

Listing 1: C Code for Bubble Sort

4.2 RISC-V Assembly Code

The RISC-V assembly code implements the same bubble sort algorithm, tailored for the RARS RISC V simulator. It uses RV32I instructions and system calls (`a7=1` for printing integers, `a7=11` for printing characters, `a7=10` for exit). Descriptive register names (e.g., $array_start$, $current_value$) and detailed comments are included for clarity. The code avoids macros, includes stack management, and sorts in ascending order.

```
.data
arr: .word 25, 509, 8, 31, 11, 0, 66  # Array of 32-bit integers
arrS: .word 7                          # Array size (7 elements)

.text
MAIN:
    # Allocate stack space and save registers
    addi sp, sp, -32
    sw ra, 28(sp)          # Save return address
    sw s0, 24(sp)          # Save array_start
    sw s1, 20(sp)          # Save array_end
    sw s2, 16(sp)          # Save array_size
    sw s10, 12(sp)         # Save loop_counter
```

```
    # Initialize array pointers
    la s0, arr              # array_start = address of arr
    lw s2, arrS             # array_size = 7
    slli t0, s2, 2          # t0 = array_size * 4 (bytes per element)
    add s1, s0, t0          # array_end = array_start + array_size * 4

    # Print initial array
    mv a0, s0               # a0 = array_start
    mv a1, s1               # a1 = array_end
    call PRINT_ARRAY

    # Initialize loop counter for bubble sort passes
    mv s10, zero            # loop_counter = 0
MAIN_LOOP:
    # Check if all passes are done
    beq s10, s2, MAIN_DONE # if loop_counter == array_size, exit loop
    mv a0, s0               # a0 = array_start
    mv a1, s1               # a1 = array_end
    call ARRAY_PAIRS        # Perform one pass of bubble sort
    addi s10, s10, 1        # loop_counter++
    j MAIN_LOOP             # Repeat for next pass

MAIN_DONE:
    # Print sorted array
    mv a0, s0               # a0 = array_start
    mv a1, s1               # a1 = array_end
    call PRINT_ARRAY

    # Restore registers and deallocate stack
    lw s10, 12(sp)          # Restore loop_counter
    lw s2, 16(sp)           # Restore array_size
    lw s1, 20(sp)           # Restore array_end
    lw s0, 24(sp)           # Save array_start
    lw ra, 28(sp)           # Restore return address
    addi sp, sp, 32
    j EXIT                  # Exit program

# Prints all elements of the array
PRINT_ARRAY:
    # Allocate stack space and save registers
    addi sp, sp, -16
    sw ra, 12(sp)           # Save return address
    sw a0, 8(sp)            # Save array_start argument
    sw a1, 4(sp)            # Save array_end argument

    # Initialize pointers for printing
    mv t0, a0               # current_ptr = array_start
    mv t1, a1               # end_ptr = array_end
PRINT_LOOP:
    # Check if done printing
    beq t0, t1, PRINT_DONE # if current_ptr == end_ptr, exit loop
    lw t2, 0(t0)            # current_value = *current_ptr
    mv a0, t2               # a0 = current_value for printing
```

```
    li a7, 1                # System call 1: print integer
    ecall
    li a7, 11               # System call 11: print character
    li a0, 32               # a0 = ASCII space (32)
    ecall
    addi t0, t0, 4          # current_ptr += 4 (next element)
    j PRINT_LOOP            # Repeat for next element

PRINT_DONE:
    # Print newline
    li a7, 11               # System call 11: print character
    li a0, 10               # a0 = ASCII newline (10)
    ecall

    # Restore registers and deallocate stack
    lw a1, 4(sp)            # Restore array_end argument
    lw a0, 8(sp)            # Restore array_start argument
    lw ra, 12(sp)           # Restore return address
    addi sp, sp, 16
    jr ra                   # Return to caller

# Performs one pass of bubble sort, swapping adjacent elements to move maximum to end
ARRAY_PAIRS:
    # Initialize pointers for the pass
    mv t0, a0               # current_ptr = array_start
    addi t1, a1, -4         # last_pair_ptr = array_end - 4 (second-to-last element)
PAIRS_LOOP:
    # Check if done with pass
    beq t0, t1, PAIRS_DONE # if current_ptr == last_pair_ptr, exit loop
    lw t2, 0(t0)            # current_value = *current_ptr
    lw t3, 4(t0)            # next_value = *(current_ptr + 4)
    blt t3, t2, PAIRS_SWAP # if next_value < current_value, swap to move larger element right
    j PAIRS_OK              # No swap needed
PAIRS_SWAP:
    sw t2, 4(t0)            # *(current_ptr + 4) = current_value
    sw t3, 0(t0)            # *current_ptr = next_value
PAIRS_OK:
    addi t0, t0, 4          # current_ptr += 4 (next element)
    j PAIRS_LOOP            # Repeat for next pair

PAIRS_DONE:
    jr ra                   # Return to caller

EXIT:
    li a7, 10               # System call 10: exit
    ecall
```

4.3 Explanation of the Transition

- **C Code**: The C code (Listing 1) provides a high-level implementation, using loops and function calls. The $array_pairs$ function performs one pass of bubble sort, and $print_array$ handles output.
- **Assembly Code**: The RISC-V assembly code (above) translates the C logic into low-level instructions, managing registers ($array_start$, $current_value$), stack frames, and system calls for I/O.

– **Machine-Level Code**: The machine code, which is the binary representation of the assembly instructions, will be shown in a screenshot you provide, illustrating addresses, hexadecimal codes, basic instructions, and source lines.

– **Register States**: The register states at the end of execution will be shown in a screenshot you provide, detailing the values of registers like `a7`, `t0`, `a0`, and the program counter (`pc`).

The main bubble sort routine begins at address `0x004000EC` and implements the classic bubble sort algorithm:

```
0x004000EC: 0x017B0663 (beq x22, x23, 0x00400100)  # Outer loop condition
0x004000F0: 0x000B2E03 (lw x28, 0(x22))            # Load arr[j]
0x004000F4: 0x004B2E83 (lw x29, 4(x22))            # Load arr[j+1]
0x004000F8: 0x01CE4663 (blt x28, x29, 0x00400108)   # Compare arr[j] < arr[j+1]
0x004000FC: 0x00C0006F (jal x0, 0x00400108)        # Skip swap if ordered
0x00400100: 0x004B2E23 (sw x28, 4(x22))            # Swap operation
0x00400104: 0x01DB2023 (sw x29, 0(x22))            # Swap operation
0x00400108: 0x004B0B13 (addi x22, x22, 4)          # Move to next element
0x0040010C: 0xFE1FF06F (jal x0, 0x004000EC)        # Repeat inner loop
```

Listing 2: Bubble Sort Main Loop

4.4 Algorithm Breakdown

4.4.1 Register Usage

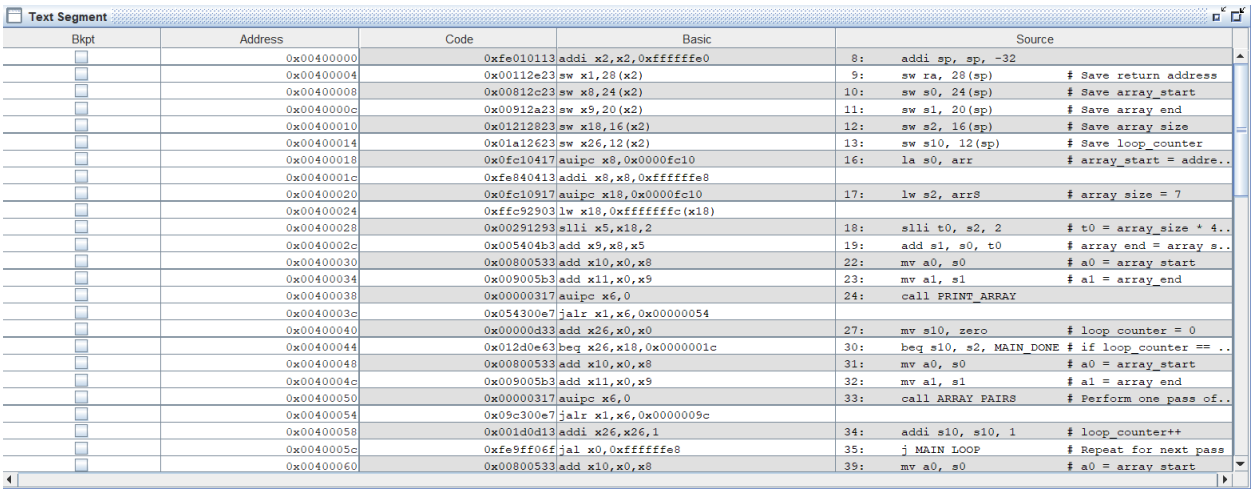
– `x22 (s6)`: Current array pointer (equivalent to `j` in high-level code)

– `x23 (s7)`: Array end pointer (points to last element)

– `x28 (t3)`: Temporary register for `arr[j]`

– `x29 (t4)`: Temporary register for `arr[j+1]`

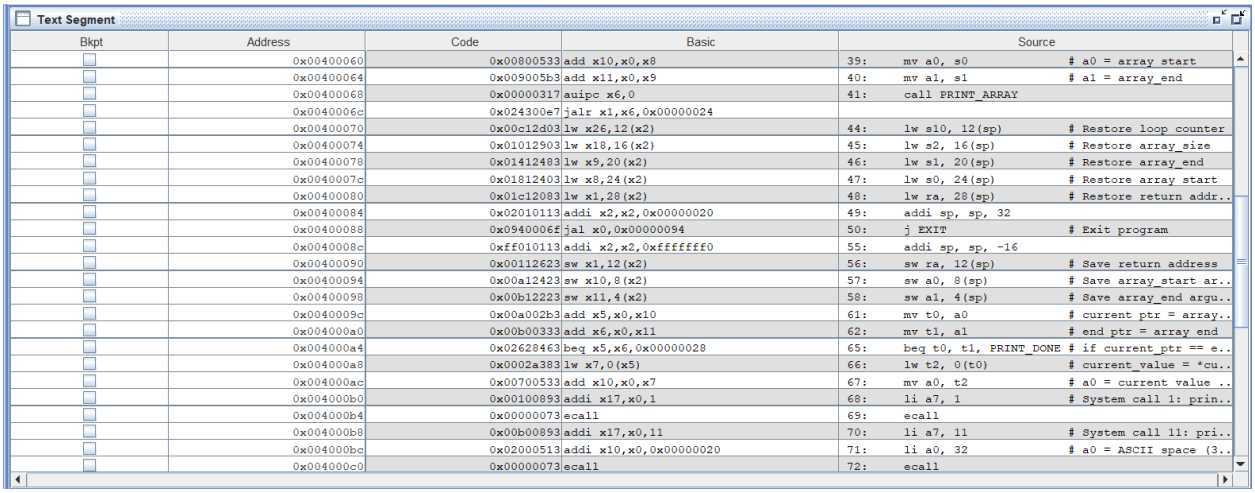
1. **Outer Loop Setup** (preceding code):
   – Initializes `x22` to array start
   – Sets `x23` to array end - 4 (pre `0x004000EC`)
2. **Element Comparison**:
$$\texttt{blt x28, x29, skip\_swap} \tag{1}$$
3. **Swap Operation** (when `arr[j] > arr[j+1]`):
$$\begin{aligned}&\texttt{sw x28, 4(x22)} \quad \text{(Store arr[j] in arr[j+1] position)}\\ &\texttt{sw x29, 0(x22)} \quad \text{(Store arr[j+1] in arr[j] position)}\end{aligned}$$
4. **Pointer Advancement**:
$$\texttt{addi x22, x22, 4} \quad \text{(Move to next 32-bit element)} \tag{2}$$



Text Segment

Bkpt	Address	Code	Basic	Source		
	0x00400000	0xfe010113	addi x2,x2,0xfffffff0	8:	addi sp, sp, -32	
	0x00400004	0x00112e23	sw x1,28(x2)	9:	sw ra, 28(sp)	# Save return address
	0x00400008	0x00812c23	sw x8,24(x2)	10:	sw s0, 24(sp)	# Save array_start
	0x0040000c	0x00912a23	sw x9,20(x2)	11:	sw s1, 20(sp)	# Save array end
	0x00400010	0x01212823	sw x18,16(x2)	12:	sw s2, 16(sp)	# Save array size
	0x00400014	0x01a12623	sw x26,12(x2)	13:	sw s10, 12(sp)	# Save loop_counter
	0x00400018	0x0fc10417	auipc x8,0x0000fc10	16:	la s0, arr	# array_start = addre..
	0x0040001c	0xfe840413	addi x8,x8,0xffffffe8			
	0x00400020	0x0fc10917	auipc x18,0x0000fc10	17:	lw s2, arrS	# array size = 7
	0x00400024	0xffc92903	lw x18,0xfffffffc(x18)			
	0x00400028	0x00291293	slli x5,x18,2	18:	slli t0, s2, 2	# t0 = array_size * 4..
	0x0040002c	0x005404b3	add x9,x8,x5	19:	add s1, s0, t0	# array end = array s..
	0x00400030	0x00800533	add x10,x0,x8	22:	mv a0, s0	# a0 = array start
	0x00400034	0x009005b3	add x11,x0,x9	23:	mv a1, s1	# a1 = array_end
	0x00400038	0x00000317	auipc x6,0	24:	call PRINT_ARRAY	
	0x0040003c	0x054300e7	jalr x1,x6,0x00000054			
	0x00400040	0x00000d33	add x26,x0,x0	27:	mv s10, zero	# loop counter = 0
	0x00400044	0x012d0e63	beq x26,x18,0x0000001c	30:	beq s10, s2, MAIN_DONE	# if loop_counter == ..
	0x00400048	0x00800533	add x10,x0,x8	31:	mv a0, s0	# a0 = array_start
	0x0040004c	0x009005b3	add x11,x0,x9	32:	mv a1, s1	# a1 = array end
	0x00400050	0x00000317	auipc x6,0	33:	call ARRAY PAIRS	# Perform one pass of..
	0x00400054	0x09c300e7	jalr x1,x6,0x0000009c			
	0x00400058	0x001d0d13	addi x26,x26,1	34:	addi s10, s10, 1	# loop_counter++
	0x0040005c	0xfe9ff06f	jal x0,0xffffffe8	35:	j MAIN LOOP	# Repeat for next pass
	0x00400060	0x00800533	add x10,x0,x8	39:	mv a0, s0	# a0 = array start

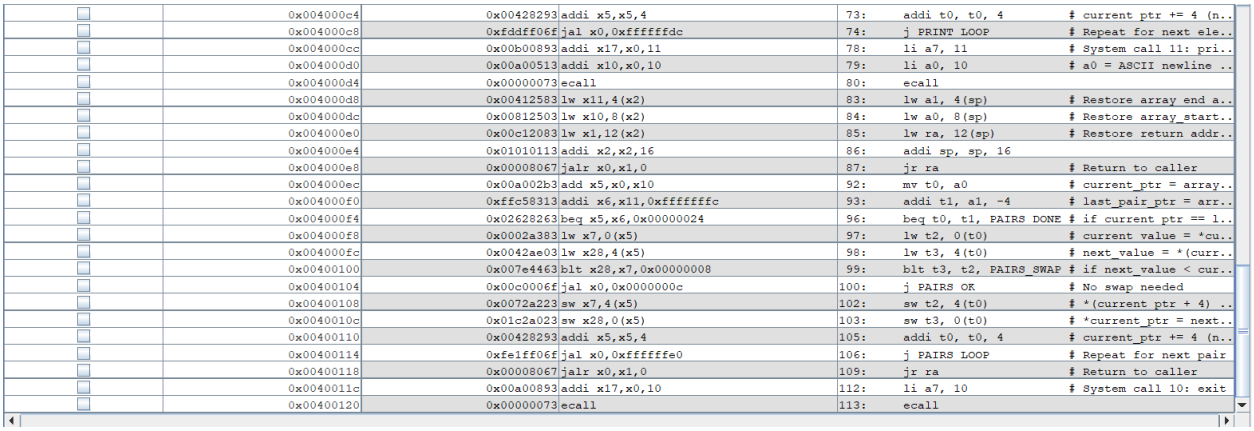
Figure 1: Machine-Level Code for Bubble Sort



Text Segment

Bkpt	Address	Code	Basic	Source		
	0x00400060	0x00800533	add x10,x0,x8	39:	mv a0, s0	# a0 = array start
	0x00400064	0x009005b3	add x11,x0,x9	40:	mv a1, s1	# a1 = array_end
	0x00400068	0x00000317	auipc x6,0	41:	call PRINT_ARRAY	
	0x0040006c	0x024300e7	jalr x1,x6,0x00000024			
	0x00400070	0x00c12d03	lw x26,12(x2)	44:	lw s10, 12(sp)	# Restore loop counter
	0x00400074	0x01012903	lw x18,16(x2)	45:	lw s2, 16(sp)	# Restore array_size
	0x00400078	0x01412483	lw x9,20(x2)	46:	lw s1, 20(sp)	# Restore array_end
	0x0040007c	0x01812403	lw x8,24(x2)	47:	lw s0, 24(sp)	# Restore array start
	0x00400080	0x01c12083	lw x1,28(x2)	48:	lw ra, 28(sp)	# Restore return addr..
	0x00400084	0x02010113	addi x2,x2,0x00000020	49:	addi sp, sp, 32	
	0x00400088	0x0940006f	jal x0,0x00000094	50:	j EXIT	# Exit program
	0x0040008c	0xff010113	addi x2,x2,0xfffffff0	55:	addi sp, sp, -16	
	0x00400090	0x00112623	sw x1,12(x2)	56:	sw ra, 12(sp)	# Save return address
	0x00400094	0x00a12423	sw x10,8(x2)	57:	sw a0, 8(sp)	# Save array_start ar..
	0x00400098	0x00b12223	sw x11,4(x2)	58:	sw a1, 4(sp)	# Save array_end argu..
	0x0040009c	0x00a002b3	add x5,x0,x10	61:	mv t0, a0	# current ptr = array..
	0x004000a0	0x00b00333	add x6,x0,x11	62:	mv t1, a1	# end ptr = array end
	0x004000a4	0x02628463	beq x5,x6,0x00000028	65:	beq t0, t1, PRINT_DONE	# if current_ptr == e..
	0x004000a8	0x0002a383	lw x7,0(x5)	66:	lw t2, 0(t0)	# current_value = *cu..
	0x004000ac	0x00700533	add x10,x0,x7	67:	mv a0, t2	# a0 = current value ..
	0x004000b0	0x00100893	addi x17,x0,1	68:	li a7, 1	# System call 1: prin..
	0x004000b4	0x00000073	ecall	69:	ecall	
	0x004000b8	0x00b00893	addi x17,x0,11	70:	li a7, 11	# System call 11: pri..
	0x004000bc	0x02000513	addi x10,x0,0x00000020	71:	li a0, 32	# a0 = ASCII space (3..
	0x004000c0	0x00000073	ecall	72:	ecall	

Figure 2: Machine-Level Code for Bubble Sort



	0x004000c4	0x00428293	addi x5,x5,4	73:	addi t0, t0, 4	# current ptr += 4 (n..
	0x004000c8	0xfdddf06f	jal x0,0xfffffdc	74:	j PRINT LOOP	# Repeat for next ele..
	0x004000cc	0x00b00893	addi x17,x0,11	78:	li a7, 11	# System call 11: pri..
	0x004000d0	0x00a00513	addi x10,x0,10	79:	li a0, 10	# a0 = ASCII newline ..
	0x004000d4	0x00000073	ecall	80:	ecall	
	0x004000d8	0x00412583	lw x11,4(x2)	83:	lw a1, 4(sp)	# Restore array end a..
	0x004000dc	0x00812503	lw x10,8(x2)	84:	lw a0, 8(sp)	# Restore array_start..
	0x004000e0	0x00c12083	lw x1,12(x2)	85:	lw ra, 12(sp)	# Restore return addr..
	0x004000e4	0x01010113	addi x2,x2,16	86:	addi sp, sp, 16	
	0x004000e8	0x00008067	jalr x0,x1,0	87:	jr ra	# Return to caller
	0x004000ec	0x00a002b3	add x5,x0,x10	92:	mv t0, a0	# current_ptr = array..
	0x004000f0	0xffc58313	addi x6,x11,0xfffffffc	93:	addi t1, a1, -4	# last_pair_ptr = arr..
	0x004000f4	0x02628263	beq x5,x6,0x00000024	96:	beq t0, t1, PAIRS DONE	# if current ptr == l..
	0x004000f8	0x0002a383	lw x7,0(x5)	97:	lw t2, 0(t0)	# current value = *cu..
	0x004000fc	0x0042ae03	lw x28,4(x5)	98:	lw t3, 4(t0)	# next_value = *(curr..
	0x00400100	0x007e4463	blt x28,x7,0x00000008	99:	blt t3, t2, PAIRS_SWAP	# if next_value < cur..
	0x00400104	0x00c0006f	jal x0,0x0000000c	100:	j PAIRS OK	# No swap needed
	0x00400108	0x0072a223	sw x7,4(x5)	102:	sw t2, 4(t0)	# *(current ptr + 4) ..
	0x0040010c	0x01c2a023	sw x28,0(x5)	103:	sw t3, 0(t0)	# *current_ptr = next..
	0x00400110	0x00428293	addi x5,x5,4	105:	addi t0, t0, 4	# current_ptr += 4 (n..
	0x00400114	0xfe1ff06f	jal x0,0xfffffe0	106:	j PAIRS LOOP	# Repeat for next pair
	0x00400118	0x00008067	jalr x0,x1,0	109:	jr ra	# Return to caller
	0x0040011c	0x00a00893	addi x17,x0,10	112:	li a7, 10	# System call 10: exit
	0x00400120	0x00000073	ecall	113:	ecall	

Figure 3: Machine-Level Code for Bubble Sort

Control and Status

Registers | Floating Point

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7fffeffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x1001001c
t1	6	0x1001001c
t2	7	0x000001fd
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x10010000
a1	11	0x1001001c
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x0000000a
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x000001fd
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00400124

Figure 4: Register States During Execution

4.5 Insertion Sort (810 cycles)

Figure 5: Insertion Sort waveform pattern

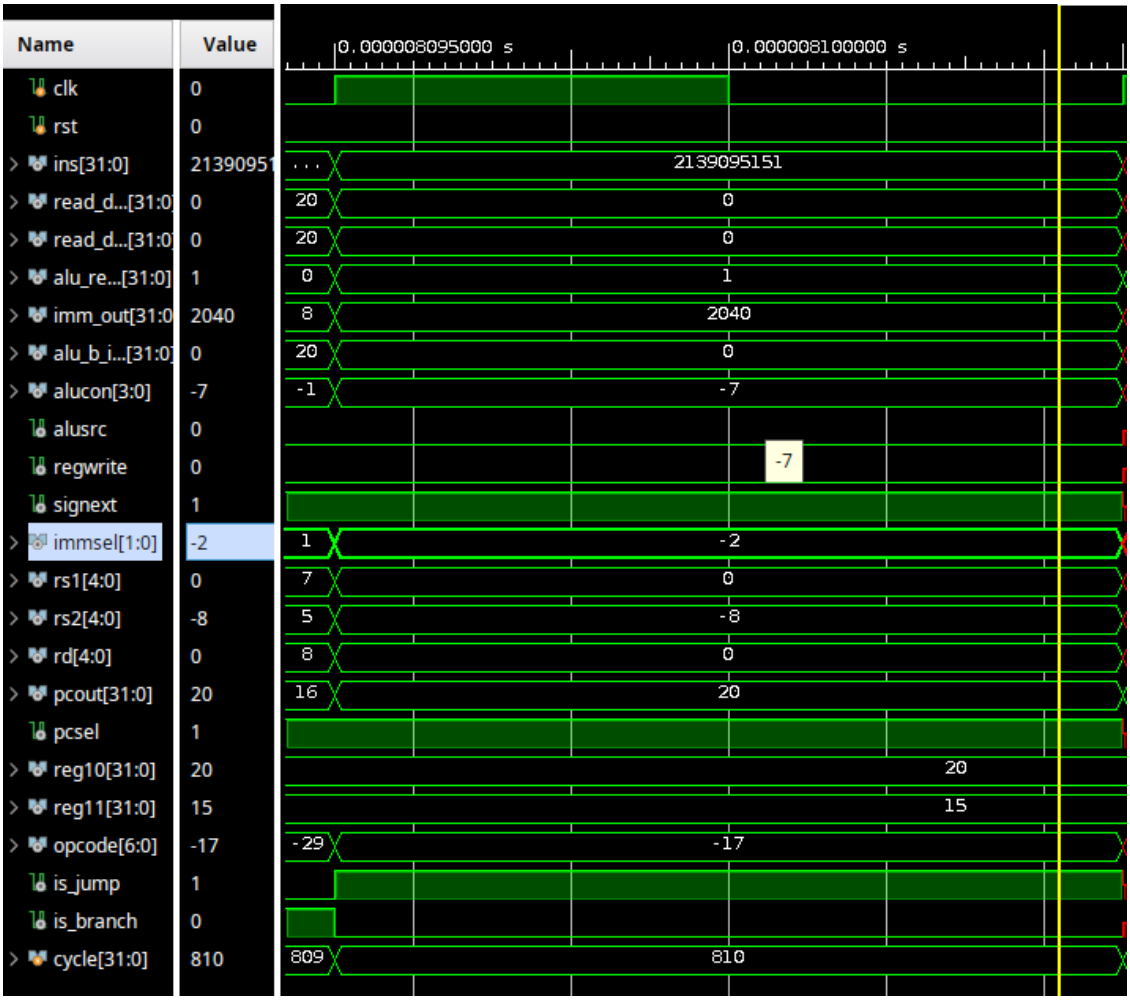


Figure 6: Insertion Sort execution pattern showing 810 cycles

4.5.1 Pattern Characteristics

- **Targeted Element Placement**:
  - Each element is sequentially inserted into its correct position
  - Visible as isolated spikes in regwrite and memwrite signals
- **Signal Behavior**:
  - pcout: Linear progression with occasional jumps
  - alucon: Brief bursts of comparison operations
  - imm_out: Sparse immediate value usage

4.6 Bubble Sort (1887 cycles)

Figure 7: Bubble Sort waveform pattern (maximum-based)

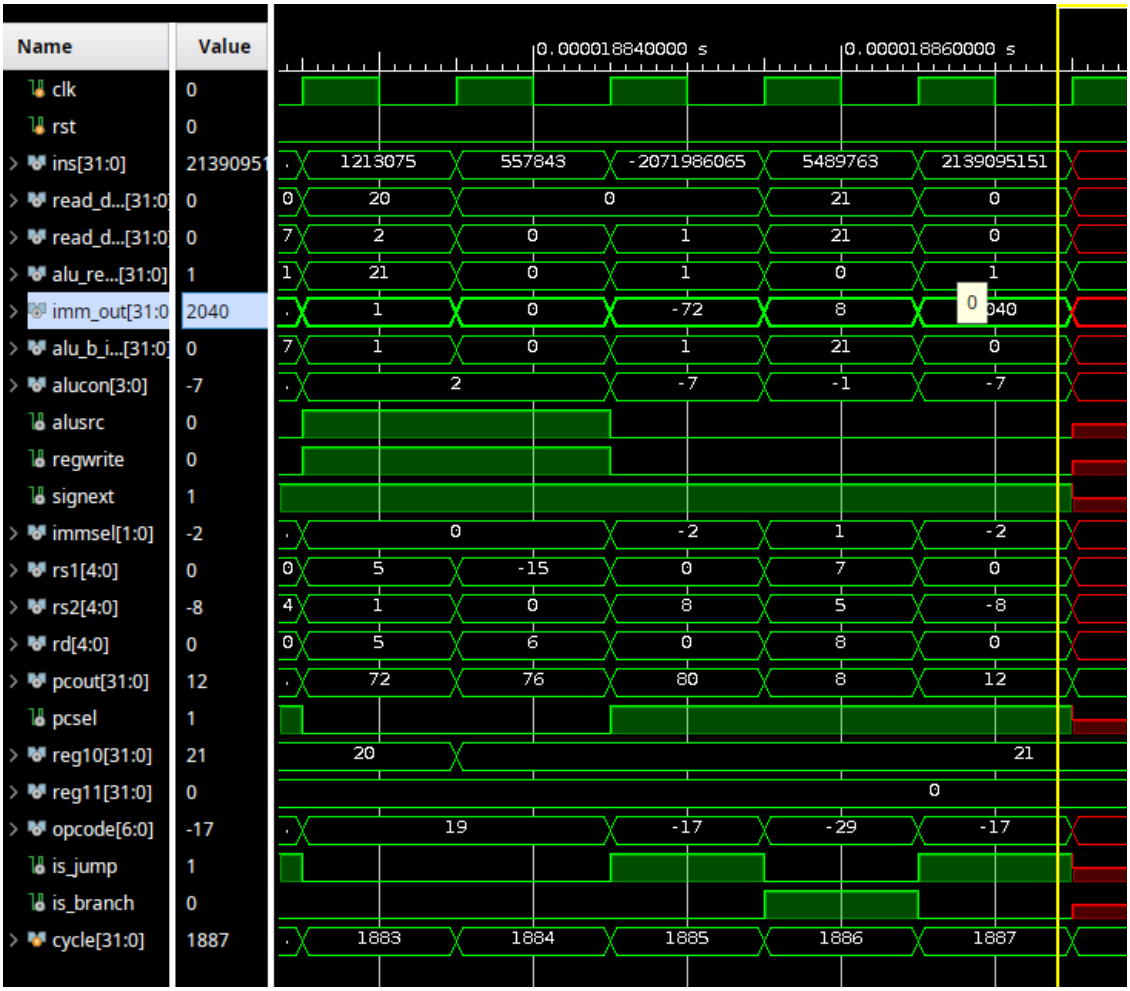


Figure 8: Bubble Sort execution pattern showing 1887 cycles (maximum-based)

4.6.1 Pattern Characteristics

- **Maximum-First Approach**:
  - Systematically pushes maximum values to the end
  - Visible as regular pulses in memwrite signals
- **Signal Behavior**:
  - pcout: Circular patterns showing repeated passes
  - alucon: Continuous comparison operations (mostly BLT/BGT)
  - rs1/rs2: Adjacent register pairs being compared

5 Comparative Analysis

Characteristic	Insertion Sort	Bubble Sort
Total Cycles	810	1887
Comparison Pattern	Targeted	Exhaustive
Memory Writes	Sparse	Regular
Best For	Nearly-sorted data	Educational purposes
Key Mechanism	Insertion	Maximum Propagation

5.1 Key Differences

- **Cycle Count**:

$$\frac{\text{Bubble}}{\text{Insertion}} = \frac{1887}{810} \approx 2.33\times \tag{3}$$

- **Swap Patterns**:
  - Insertion: Localized element shifts
  - Bubble: Adjacent swaps propagating maxima
- **Implementation Notes**:
  - Bubble Sort's maximum-based approach requires complete passes
  - Insertion Sort benefits from partially sorted input

6 Simulation Outputs

This section presents a detailed walkthrough of the datapath simulation using a testbench in Verilog. Each subsection corresponds to a key stage in the instruction execution pipeline of a single-cycle processor, namely instruction fetch, decode, execution, memory access, and write-back. The Verilog testbench has been designed to observe internal signals and verify the correct behavior of the processor modules.

6.1 Program Counter and Instruction Fetch

The program counter (PC) holds the address of the current instruction to be executed. On every rising edge of the clock signal, if the system is not under reset, the PC updates either sequentially or based on the branch decision. The instruction at the memory location pointed by the PC is fetched into the `ins` register.

The testbench monitors the PC value and instruction as follows:

```
assign pcout = uut.pc_inst.pc;
assign ins = uut.ins;
```

The value of `pcout` is printed to observe the control flow, particularly during jumps or branches. The instruction fetch mechanism ensures correct sequencing of instruction execution, critical to flow control and branching logic.

6.2 Instruction Decode and Control Signal Generation

Once the instruction is fetched, it is decoded to determine the operation type. The decoder extracts the opcode (bits `[6:0]`) and generates control signals necessary for guiding the datapath elements like the ALU, memory, and register file.

The control signals monitored in the testbench include: - `regwrite`: enables writing back to the register file. - `alusrc`: selects between immediate or register operand for the ALU. - `signext`: enables sign extension for immediate values. - `immsel`: selects the format/type of immediate value.

```
assign opcode = ins[6:0];
assign regwrite = uut.regwrite;
assign alusrc   = uut.alusrc;
assign signext  = uut.signext;
assign immsel   = uut.immsel;
```

In addition, branching and jumping conditions are checked using:

```
wire is_jump   = (opcode == 7'b1101111); // JAL
wire is_branch = (opcode == 7'b1100011); // BEQ, BNE, etc.
```

These decoded signals help identify control flow instructions and manipulate the PC accordingly.

6.3 Register File Operand Fetch

The source registers (`rs1`, `rs2`) and destination register (`rd`) are identified by decoding the instruction fields. The register file outputs the data stored in `rs1` and `rs2`, which are then passed to the ALU for computation.

The testbench extracts and displays these values:

```
assign rs1 = uut.rs1;
assign rs2 = uut.rs2;
assign read_data1 = uut.rf_inst.read_data1;
assign read_data2 = uut.rf_inst.read_data2;
```

These signals allow us to verify that the correct operands are being read for ALU operations and that the source fields are being decoded properly from the instruction.

6.4 ALU Execution

The ALU (Arithmetic Logic Unit) performs computation based on the instruction type and ALU control signals. The operands are `read_data1` and `alu_b_input`, where the second operand may be an immediate value or another register depending on `alusrc`.

```
assign alu_b_input = uut.alu_b_input;
assign alucon = uut.alucon;
assign alu_result = uut.alu_result;
```

The `alucon` signal determines the specific operation (e.g., addition, subtraction, AND, OR). The `alu_result` is used either for memory addressing (load/store) or directly written back to the register file.

6.5 Memory Access

For load and store instructions (e.g., `lw`, `sw`), memory access is needed. The memory address is computed by the ALU and is used to either fetch data from memory or store data to it. The control signal `memwrite` determines whether a write operation is to be performed.

Although direct memory contents are not printed in the testbench, memory operations are inferred from instruction types and control signals:

```
// Memory access control signal (from decoder)
output wire memwrite;
```

Tracking the memory access stage is crucial for verifying correct data transfer between the processor and memory.

6.6 Write Back

After ALU execution or memory access, the result must be written back to the destination register (`rd`) if `regwrite` is asserted. In this testbench, specific registers such as `x10` and `x11` are observed for debugging and validation purposes.

```
assign reg10 = uut.rf_inst.registers[5]; // x10
assign reg11 = uut.rf_inst.registers[6]; // x11
```

Monitoring these values helps verify that write-back occurs correctly, and that the instruction's effects are reflected in the processor state.

6.7 Immediate Generation and Integration

Many instructions (e.g., `addi`, `lw`, `beq`) require immediate values. The immediate is extracted from the instruction based on its type and sign-extended if necessary. The decoder and sign extension unit generate `imm_out` which is forwarded to the ALU.

```
assign imm_out = uut.imm_out;
```

Accurate immediate generation is critical for arithmetic operations, branch offsets, and memory addressing. The simulation checks the integrity and correctness of immediate values with each instruction.

6.8 Conclusion of Simulation

The testbench includes a simulation loop that triggers on each rising edge of the clock. In each cycle, it prints all relevant datapath signals including the PC, instruction, decoded registers, immediate, ALU result, and register file outputs.

```
always @(posedge clk) begin
    if (!rst) begin
        $display("Cycle: %0d", cycle);
        $display("PC: 0x%08h", pcout);
        $display("Instruction: %b", ins);
        $display("Opcode: %b", opcode);
        ...
        cycle = cycle + 1;
    end
end
```

This simulation output helps trace instruction execution step-by-step, validate control signal correctness, and ensure that the datapath behaves as intended.

7 Individual Contribution to the Project

- **Godaba Jasmini**: Responsible for writing the testbench.
- **Vishwas Jasuja**: Implemented the instructions for finding the maximum element and the `signextend` module.
- **Bhuvanesh Ganta**: Designed the datapath and worked on instruction control for the counter program.
- **Nanda Kishor Yadla**: Focused on machine code generation from assembly code of Bubble Sort Algorithm.
- **Gnaneshwar Mulkulla**: Developed the `PC` (Program Counter) and `ALU` modules.
- **Shersingh meena**: Created the `DRAM` and `Register File` modules.
- **Shubham Meena**: Implemented the `Decoder` module for instruction parsing and control signal generation.