CE-321L/CS-330L: Computer Architecture
Pipelined Processor

Muhammad Moosa Owais, Syed Hamza Raza, Sameez Sarfaraz Sajwani

21-4-2025

Introduction

- Simulate a data sorting algorithm on a processor implemented using RISC-V and Verilog

Methodology

- Task 1 – a single-cycle processor to simulate data sorting
- Task 2 – modify the single-cycle processor and introduce pipelining
- Task 3 – further modify the pipelined processor and introduce hazard control mechanisms (forwarding unit, stalling/flushing the pipeline).

Results

- Show Final simulation
- Describe your results

<u>Task 1</u>

Single-Cycle Bubble Sort

In this exercise we built a simple single-cycle RISC-V datapath that runs a bubble-sort routine on eight byte-spaced elements stored in data memory. The instruction memory is preloaded with machine code for the sort, the data memory is initialized with eight values at addresses 0, 8, 16…56, and eight "element" outputs wire-together each 8-byte group for easy observation. On reset the processor steps through fetch–decode–execute–memory–writeback for every instruction and performs the compare-and-swap passes of the bubble sort.

- **Design Process:** We instantiated IF, ID, EX, MEM and WB stages with a program counter, instruction memory, register file, ALU, data memory and a final multiplexer for writeback. We then wrote a simple testbench to toggle the clock and reset, hooked outputs for the eight data-memory words, and ran the simulation until the sort program completed.
- **Code Changes:** The `Data_Memory` module was initialized in its `initial` block with the eight unsorted bytes at offsets `0, 8, …, 56`. We added eight 64-bit `elementN` outputs that concatenate eight bytes into one word for waveform monitoring. The `Instruction_Memory` was likewise filled with the compiled bubble-sort instructions.
- **Results:** In simulation you should see the eight monitored values start in their original order and end up in ascending order once the sort finishes.

Bubble Sort:

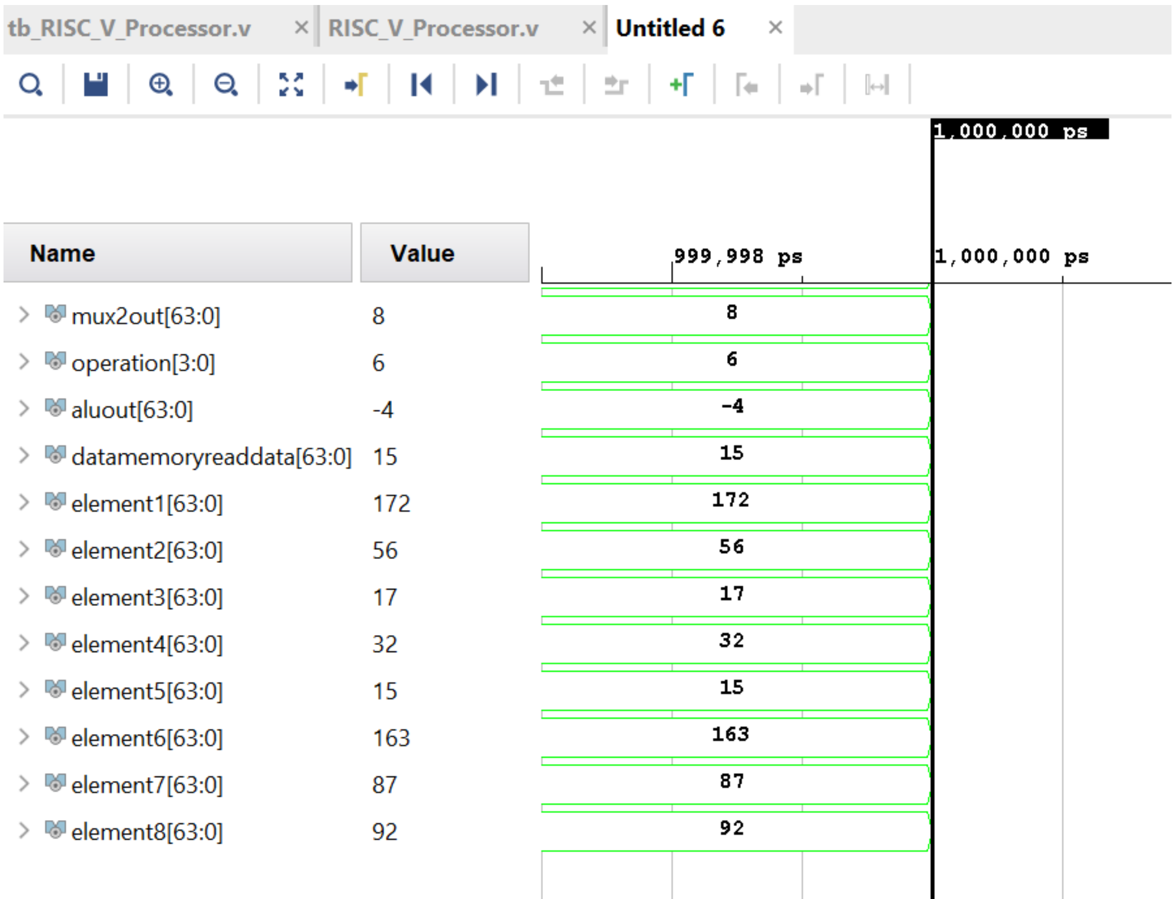
Editor Simulator

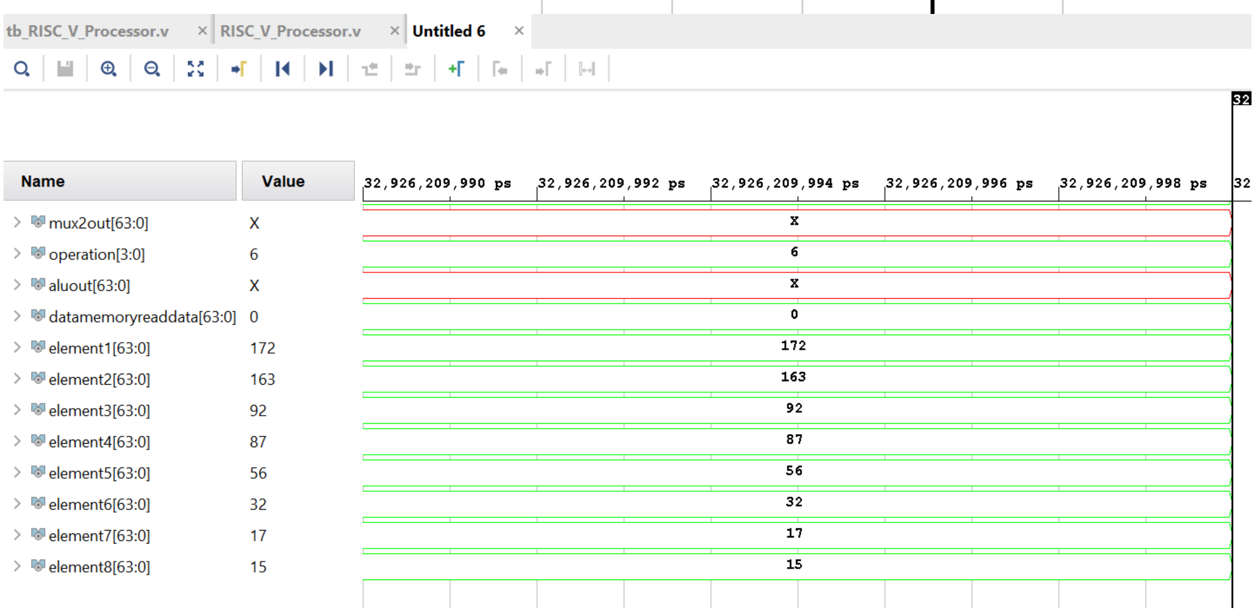
```
    addi  x18, x0, 0        # to track a[i] offset
    add   x8,  x0, x0       # i iterator (starts at 0)
    addi  x11, x0, 10       # loop bound n = 10

outerloop:
    beq   x8,  x11, outerexit  # if i == n, exit
    add   x29, x0, x8          # j iterator = i

    add   x19, x8, x0
    add   x19, x19, x19
    add   x19, x19, x19        # x19 = 4*x8 (byte offset)

innerloop:
    beq   x29, x11, innerexit  # if j == n, inner loop done
    addi  x29, x29, 1          # j++
    addi  x19, x19, 8          # offset += 8

    lw    x26, 0(x18)          # load a[i]
    lw    x27, 0(x19)          # load a[j]

    blt   x26, x27, bubblesort # if a[i] < a[j], swap
    beq   x0,  x0, innerloop   # else continue

bubblesort:
    add   x5,  x0, x26         # temp = a[i]
    sw    x27, 0(x18)          # a[i] = a[j]
    sw    x5,  0(x19)          # a[j] = temp
    beq   x0,  x0, innerloop   # restart inner

innerexit:
    addi  x8,  x8, 1           # i++
    addi  x18, x18, 8          # offset += 8
    beq   x0,  x0, outerloop   # back to outer

outerexit:
    # done
```

Machine Code:

Machine Code	Basic Code	Original Code
0x00000913	addi x18 x0 0	addi x18, x0, 0 # to track a[i] offset
0x00000433	add x8 x0 x0	add x8, x0, x0 # i iterator (starts at 0)
0x00a00593	addi x11 x0 10	addi x11, x0, 10 # loop bound n = 10
0x04b40663	beq x8 x11 76	beq x8, x11, outerexit # if i == n, exit
0x00800eb3	add x29 x0 x8	add x29, x0, x8 # j iterator = i
0x000409b3	add x19 x8 x0	add x19, x8, x0
0x013989b3	add x19 x19 x19	add x19, x19, x19
0x013989b3	add x19 x19 x19	add x19, x19, x19 # x19 = 4*x8 (byte offset)
0x02be8663	beq x29 x11 44	beq x29, x11, innerexit # if j == n, inner loop done
0x001e8e93	addi x29 x29 1	addi x29, x29, 1 # j++
0x00898993	addi x19 x19 8	addi x19, x19, 8 # offset += 8
0x00092d03	lw x26 0(x18)	lw x26, 0(x18) # load a[i]
0x0009ad83	lw x27 0(x19)	lw x27, 0(x19) # load a[j]
0x01bd4463	blt x26 x27 8	blt x26, x27, bubblesort # if a[i] < a[j], swap
0xfe0004e3	beq x0 x0 -24	beq x0, x0, innerloop # else continue
0x01a002b3	add x5 x0 x26	add x5, x0, x26 # temp = a[i]
0x01b92023	sw x27 0(x18)	sw x27, 0(x18) # a[i] = a[j]
0x0059a023	sw x5 0(x19)	sw x5, 0(x19) # a[j] = temp
0xfc000ce3	beq x0 x0 -40	beq x0, x0, innerloop # restart inner
0x00140413	addi x8 x8 1	addi x8, x8, 1 # i++
0x00890913	addi x18 x18 8	addi x18, x18, 8 # offset += 8
0xfa000ce3	beq x0 x0 -72	beq x0, x0, outerloop # back to outer

Results:


tb_RISC_V_Processor.v
RISC_V_Processor.v
Untitled 6
1,000,000 ps
Name
Value
999,998 ps
1,000,000 ps
mux2out[63:0] 8
operation[3:0] 6
aluout[63:0] -4
datamemoryreaddata[63:0] 15
element1[63:0] 172
element2[63:0] 56
element3[63:0] 17
element4[63:0] 32
element5[63:0] 15
element6[63:0] 163
element7[63:0] 87
element8[63:0] 92


tb_RISC_V_Processor.v
RISC_V_Processor.v
Untitled 6
Name
Value
32,926,209,990 ps
32,926,209,992 ps
32,926,209,994 ps
32,926,209,996 ps
32,926,209,998 ps
mux2out[63:0] X
operation[3:0] 6
aluout[63:0] X
datamemoryreaddata[63:0] 0
element1[63:0] 172
element2[63:0] 163
element3[63:0] 92
element4[63:0] 87
element5[63:0] 56
element6[63:0] 32
element7[63:0] 17
element8[63:0] 15

Task 2

Pipelined RISC-V Bubble Sort

Here we converted the single-cycle design into a classic 5-stage pipeline by inserting IF/ID, ID/EX, EX/MEM and MEM/WB registers and steering control signals through them. The same bubble-sort code and data initialization are used, but the pipeline introduces read-after-write hazards between successive load, compare and store instructions.

- **Design Process:** We added pipeline-register modules (`ifidreg`, `idexreg`, `exmemreg`, `memwbreg`) between each stage, passed all control and data signals through them, and updated the top-level to drive those registers on every clock edge. The rest of the datapath (PC, IM, CU, RF, ALU, DM) was left largely unchanged.
- **Code Changes:** New pipeline-register files were created, and the top-level `RISC_V_Processor_task2.v` was rewritten to connect stage outputs to the next stage's registers rather than directly to the downstream functional units.
- **Results:** The sort never actually completes correctly in simulation—the eight "element" outputs remain in roughly their original order. This is because the design has no hazard detection or forwarding logic, so dependent instructions in the bubble-sort inner loop read stale register or memory values and the compare-and-swap sequence never executes on the correct data.

Pipeline Registers:

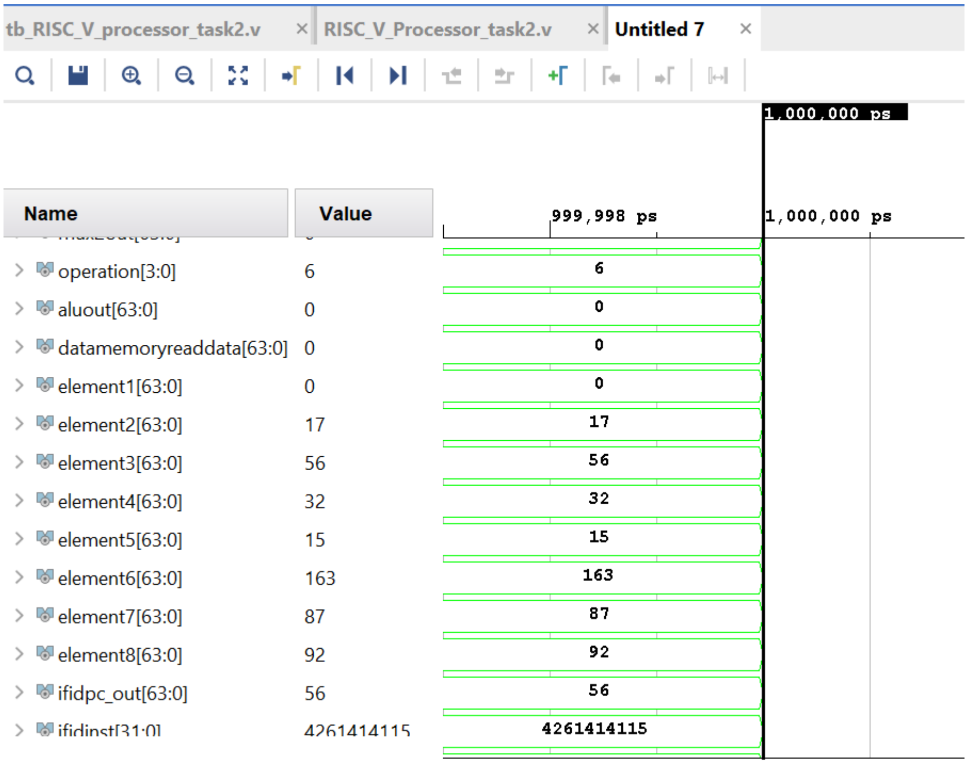
```
module ifidreg(
input clk,
input reset,
input [63:0] pc_out,
input [31:0] instruction,
output reg [63:0] ifidpc_out,
output reg [31:0] ifidinst);

always @(posedge clk) begin
    if (reset == 1'b1) begin
        ifidpc_out = 0;
        ifidinst = 0;
        end
    else begin
        ifidpc_out = pc_out;
        ifidinst = instruction;
        end
end

endmodule
```

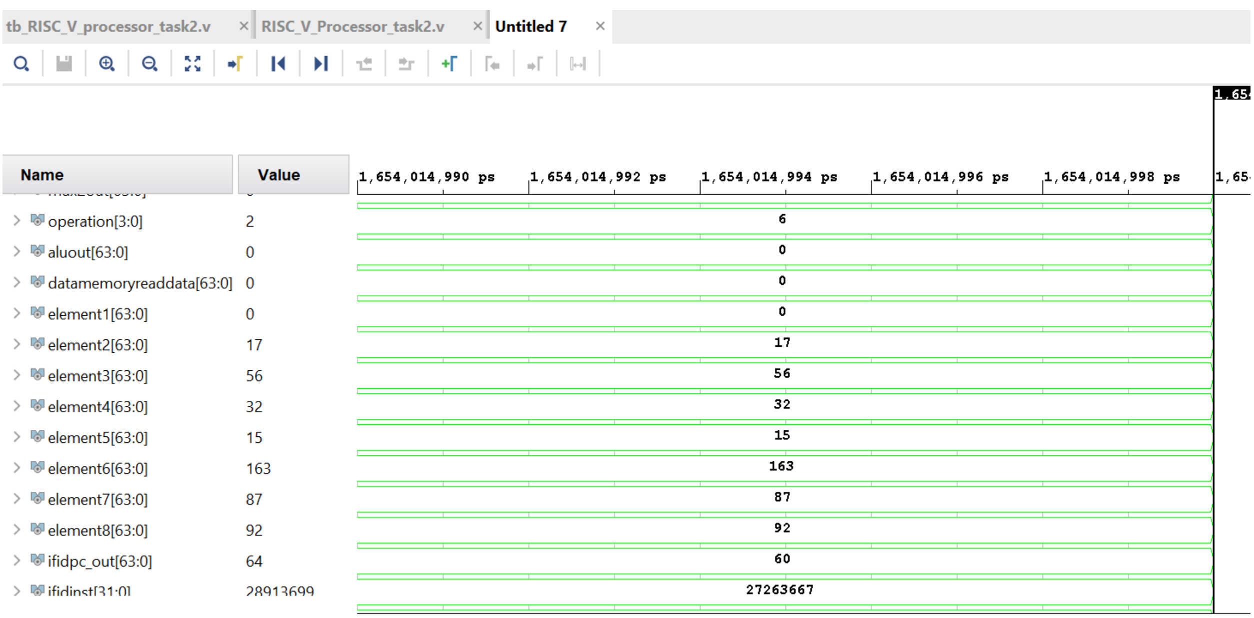
```
module idexreg(
input clk,
input reset,
input  [63:0] ifidpc_out, readdata1, readdata2, imm,
input  [4:0] rs1, rs2, rd,
input  [3:0] funct3,
input  branch, memread, memtoreg, memwrite, regwrite, alusrc,
input  [1:0] aluop,
output reg [63:0]idexpc_out, idexreaddata1, idexreaddata2, ideximm,
output reg [4:0] idexrs1, idexrs2, idexrd,
output reg [3:0] idexfunct3,
output reg idexbranch, idexmemread, idexmemtoreg, idexmemwrite, idexregwrite, idexalusrc,
output reg [1:0] idexaluop
    );

always @(posedge clk) begin
    if (reset == 1'b1)begin
        idexpc_out = 0;
        idexreaddata1 = 0;
        idexreaddata2 = 0;
        ideximm = 0;
        idexrs1 = 0;
        idexrs2 = 0;
        idexrd = 0;
        idexfunct3 = 0;
        idexbranch = 0;
        idexmemread = 0;
        idexmemtoreg = 0;
        idexmemwrite = 0;
        idexregwrite = 0;
        idexalusrc = 0;
        idexaluop = 0;
        end
    else begin
        idexpc_out = ifidpc_out;
        idexreaddata1 = readdata1;
        idexreaddata2 = readdata2;
        ideximm = imm;
        idexrs1 = rs1;
        idexrs2 = rs2;
        idexrd = rd;
        idexfunct3 = funct3;
        idexbranch = branch;
        idexmemread = memread;
        idexmemtoreg = memtoreg;
        idexmemwrite = memwrite;
        idexregwrite = memwrite;
        idexalusrc = alusrc;
        idexaluop = aluop;
        end

    end
endmodule
```

```
module exmemreg(
  input clk,reset,
  input [63:0] adderout,
  input [63:0] resultinalu,
  input zeroin,
  input [63:0] writedatain,
  input [4:0] rdin,
  input branchin,memreadin, memtoregin, memwritein, regwritein,
  input addermuxselectin,
  output reg [63:0] exmemadderout,
  output reg exmemzero,
  output reg [63:0] exmemresultoutalu,
  output reg [63:0] exmemwritedataout,
  output reg [4:0] exmemrd,
  output reg exmembranch, exmemmemread, exmemmemtoreg, exmemmemwrite, exmemregwrite,
  output reg exmemaddermuxselect);

  always @(posedge clk)
    begin
      if (reset == 1'b1)
        begin
          exmemadderout = 64'b0;
          exmemzero = 1'b0;
          exmemresultoutalu = 63'b0;
          exmemwritedataout = 64'b0;
          exmemrd = 5'b0;
          exmembranch = 1'b0;
          exmemmemread = 1'b0;
          exmemmemtoreg =1'b0;
          exmemmemwrite = 1'b0;
          exmemregwrite = 1'b0;
          exmemaddermuxselect = 1'b0;
        end
      else
        begin
          exmemadderout = adderout;
          exmemzero = zeroin;
          exmemresultoutalu = resultinalu;
          exmemwritedataout = writedatain;
          exmemrd = rdin;
          exmembranch = branchin;
          exmemmemread = memreadin;
          exmemmemtoreg = memtoregin;
          exmemmemwrite = memwritein;
          exmemregwrite = regwritein;
          exmemaddermuxselect = addermuxselectin;
        end
    end
endmodule
```

```
module memwbreg(
input clk,reset,
  input [63:0] read_data_in,
  input [63:0] result_alu_in,
  input [4:0] Rd_in,
  input memtoreg_in, regwrite_in,
  output reg [63:0] readdata,
  output reg [63:0] result_alu_out,
  output reg [4:0] rd,
  output reg Memtoreg, Regwrite
);

  always @(posedge clk)
    begin
      if (reset == 1'b1)
        begin
          readdata = 63'b0;
          result_alu_out = 63'b0;
          rd = 5'b0;
          Memtoreg = 1'b0;
          Regwrite = 1'b0;

        end
      else
        begin
         readdata = read_data_in;
          result_alu_out = result_alu_in;
          rd = Rd_in;
          Memtoreg = memtoreg_in;
          Regwrite = regwrite_in;
        end
    end
endmodule
```

Results:





Task 3

Pipelined RISC-V Bubble Sort with Hazard Detection and Control

In this section, we integrated hazard detection mechanisms into the pipelined processor developed by the end of Task 2.

- **Design Process:** Hazards such as data, structural, and control hazards are managed in the code by implementing hazard detection circuitry and introducing pipeline stalling mechanisms. These hazards typically arise due to instruction dependencies or the need to forward data between pipeline stages. To handle this, we implemented a hazard detection unit that monitors such situations and determines whether to stall the pipeline or forward the required data. It does so by generating appropriate control signals for the forwarding unit and triggering pipeline stalls or flushes when necessary.
- **Code Changes:** modules were created to detect and resolve hazards associated with pipelining and integrated into the top module from Task 2. We implemented a module for the hazard detection unit and another module for the forwarding unit. We also implemented a module for flushing the pipeline when a branch is detected (*pipeline_flush).*

Hazard detection unit:

```
module hazard_detection_unit (
    input wire          Memread,
    input wire [4:0]    Rd,
    input wire [31:0]   inst,
    output reg          stall
);

  wire [4:0] Rs1, Rs2;

  // Extract source register fields from the instruction
  assign Rs1 = inst[19:15];
  assign Rs2 = inst[24:20];

  always @(*) begin
    // Detect hazard: if Memread and Rd matches any of the source registers
    if (Memread && ((Rd == Rs1) || (Rd == Rs2)))
      stall = 1'b1;
    else
      stall = 1'b0;
  end

endmodule
```

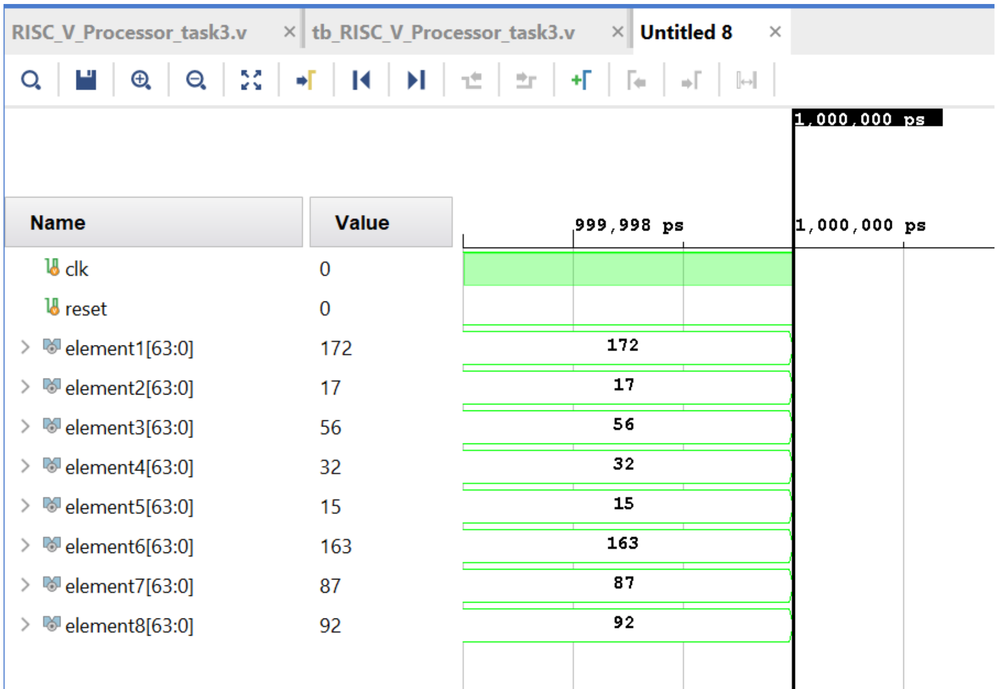
Forwarding unit:

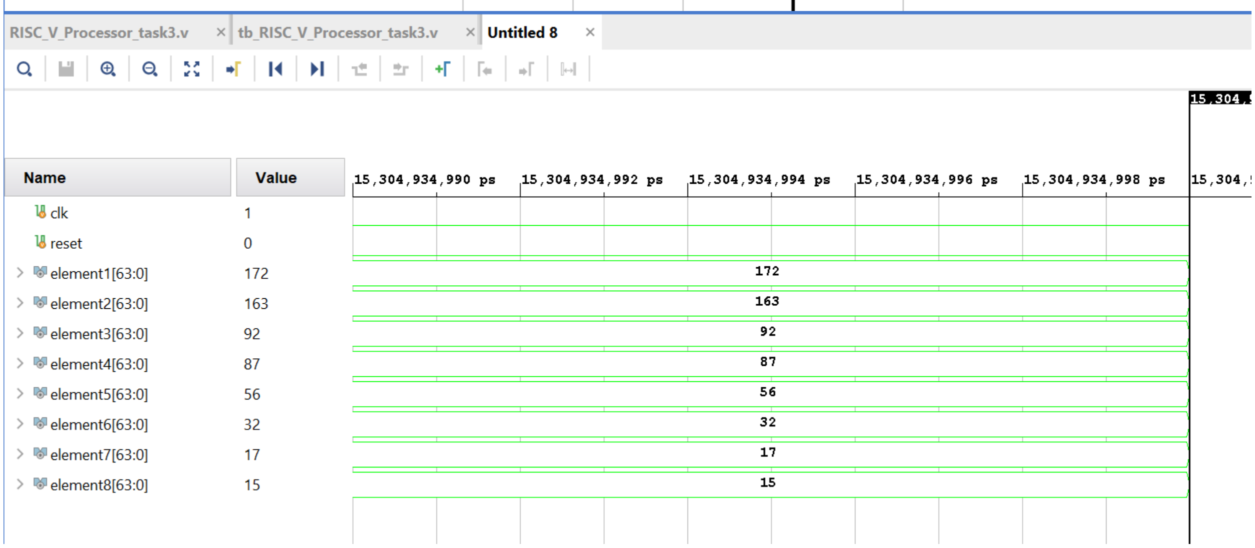
```
module ForwardingUnit (
    input  wire [4:0] RS_1,        // ID/EX.RegisterRs1
    input  wire [4:0] RS_2,        // ID/EX.RegisterRs2
    input  wire [4:0] rdMem,       // EX/MEM.RegisterRd
    input  wire [4:0] rdWb,        // MEM/WB.RegisterRd
    input  wire       regWrite_Mem, // EX/MEM.RegWrite
    input  wire       regWrite_Wb,  // MEM/WB.RegWrite
    output reg  [1:0] Forward_A,
    output reg  [1:0] Forward_B
);

  always @(*) begin
    // Forwarding logic for source register RS_1
    if (regWrite_Mem && (rdMem != 0) && (rdMem == RS_1)) begin
      Forward_A = 2'b10;  // Forward from EX/MEM
    end
    else if (regWrite_Wb && (rdWb != 0) && (rdWb == RS_1) &&
             !(regWrite_Mem && (rdMem != 0) && (rdMem == RS_1))) begin
      Forward_A = 2'b01;  // Forward from MEM/WB
    end
    else begin
      Forward_A = 2'b00;  // No forwarding
    end

    // Forwarding logic for source register RS_2
    if (regWrite_Mem && (rdMem != 0) && (rdMem == RS_2)) begin
      Forward_B = 2'b10;  // Forward from EX/MEM
    end
    else if (regWrite_Wb && (rdWb != 0) && (rdWb == RS_2) &&
             !(regWrite_Mem && (rdMem != 0) && (rdMem == RS_2))) begin
      Forward_B = 2'b01;  // Forward from MEM/WB
    end
    else begin
      Forward_B = 2'b00;  // No forwarding
    end
  end

endmodule
```

Pipeline flushing:

```
module pipeline_flush
  (
    input branch,
    output reg Flush
  );

  initial
    begin
      Flush = 1'b0;
    end

  always @(*)
    begin
      if (branch == 1'b1)
        Flush = 1'b1;
      else
        Flush = 1'b0;
    end

endmodule
```

Results:





Task 4 (Performance Comparison):

The pipelined RISC-V processor completes the Bubble Sort algorithm in 800 nanoseconds, whereas the single-cycle (non-pipelined) processor takes about 1000 nanoseconds to perform the same task. This illustrates the efficiency advantage of pipelining.

In a non-pipelined (single-cycle) processor, instructions are executed one at a time, from start to finish. The processor must wait for one instruction to fully complete before beginning the next. This can lead to inefficient use of resources, as some components of the processor remain idle during certain phases of execution.

In contrast, a pipelined processor divides instruction execution into multiple stages (such as Fetch, Decode, Execute, etc.) and allows multiple instructions to be in different stages at the same time. This overlapping reduces idle time and improves overall speed, as each part of the processor is constantly active.

Challenges

Discuss the challenges faced and how you solved it.

- We expanded the byte-address space from 64 to 256 entries so each of our eight 8-byte elements fits without overlapping or out-of-bounds errors. With only 64 bytes, addresses beyond 0–63 would produce X's; 256 bytes gives plenty of room for all elements plus future growth.

Task Division

State the task division and contribution of each member.

- **Task 1 – Hamza**
- **Task 2 – Moosa**
- **Task 3 – Sameez**
- **Task 4 – Sameez**

Conclusions

- How did your project turn out?
- Briefly state why it was or was not successful.

Project turned out successful, we were able to create a single cycle processor that can implement the bubble sorting algorithm effectively, and the made it a pielined processor coupled with hazard detection, making it more efficient.

Appendices

https://github.com/samsajwani1234/CA-Lab-Project