CE-321L/CS-330L: Computer Architecture
[CA Pipelining]

[Muhammad Daniyal Farooqui, Ikhlas Ahmed T1]

[27/2/2024]
[Miss Maham and Sir Ahmed Ali]

Introduction:

1. So, imagine we have this single cycle processor, which can only execute one instruction at a time. Now, we want to upgrade it to a pipelined processor. Pipelining allows multiple instructions to be processed simultaneously, which increases overall performance.

2. To make this change, we will need to adjust some instructions in our processor, especially to support sorting algorithms. Additionally, we have to identify the different types of hazards and fix them with forwarding stalling and flushing. We will be using the bubble sort algorithm to test the correct pipelining.

3. These are the 4 task we performed in this project and provided with detail report for code, simulation, results and block Diagram.

- **Task 1: Single Cycle Processor with Sorting Algorithm**
- **Task 2: Pipelined Processor Implementation**
- **Task 3: Hazard Detection and Handling**
- **Task 4: Performance Comparison**

Task 1: Single Cycle Processor with Sorting Algorithm

1. We Choose a sorting algorithm bubble sort.
2. We took the sorting algorithm from lab4 and convert it into hexadecimal code.
3. We then modify and combine all the labs and made a Single Cycle Processor (Non-Pipelined) lab 11.
4. We then run the sorting algorithm on the single-cycle processor and verify that the processor sorts the array correctly.

Bubble Sort Code in Hexadecimal:

```
{inst_mem[3], inst_mem[2], inst_mem[1], inst_mem[0]} = 32'h00000913;//1
{inst_mem[7], inst_mem[6], inst_mem[5], inst_mem[4]} = 32'h00000433;//2
{inst_mem[11], inst_mem[10], inst_mem[9], inst_mem[8]} = 32'h04b40863;//3
{inst_mem[15], inst_mem[14], inst_mem[13], inst_mem[12]} = 32'h00800eb3;//4
{inst_mem[19], inst_mem[18], inst_mem[17], inst_mem[16]} = 32'h000409b3;//5
{inst_mem[23], inst_mem[22], inst_mem[21], inst_mem[20]} = 32'h013989b3;//6
{inst_mem[27], inst_mem[26], inst_mem[25], inst_mem[24]} = 32'h013989b3;//7
{inst_mem[31], inst_mem[30], inst_mem[29], inst_mem[28]} = 32'h013989b3;//8
{inst_mem[35], inst_mem[34], inst_mem[33], inst_mem[32]} = 32'h02be8663;//9
{inst_mem[39], inst_mem[38], inst_mem[37], inst_mem[36]} = 32'h001e8e93;//10
{inst_mem[43], inst_mem[42], inst_mem[41], inst_mem[40]} = 32'h00898993;//11
{inst_mem[47], inst_mem[46], inst_mem[45], inst_mem[44]} = 32'h00093d03;//12
{inst_mem[51], inst_mem[50], inst_mem[49], inst_mem[48]} = 32'h0009bd83;//13
{inst_mem[55], inst_mem[54], inst_mem[53], inst_mem[52]} = 32'h01bd4463;//14
{inst_mem[59], inst_mem[58], inst_mem[57], inst_mem[56]} = 32'hfe0004e3;//15
{inst_mem[63], inst_mem[62], inst_mem[61], inst_mem[60]} = 32'h01a002b3;//16
{inst_mem[67], inst_mem[66], inst_mem[65], inst_mem[64]} = 32'h01b93023;//17
{inst_mem[71], inst_mem[70], inst_mem[69], inst_mem[68]} = 32'h0059b023;//18
{inst_mem[75], inst_mem[74], inst_mem[73], inst_mem[72]} = 32'hfc000ce3;//19
{inst_mem[79], inst_mem[78], inst_mem[77], inst_mem[76]} = 32'h00140413;//20
{inst_mem[83], inst_mem[82], inst_mem[81], inst_mem[80]} = 32'h00890913;//21
{inst_mem[87], inst_mem[86], inst_mem[85], inst_mem[84]} = 32'hfa000ae3;//22
```

Single Cycle Processor:

```
module riscV_processor(
input clk,
input reset,
output wire [63:0] PC_Out,
output wire [63:0] ao1,
output wire [63:0] ao2,
output wire [63:0] PC_In,
output wire zero,
output wire [31:0] instruction,
output wire [6:0] opcode,
output wire [4:0] rd,
output wire [2:0] funct3,
output wire [4:0] rs1,
output wire [4:0] rs2,
output wire [6:0] funct7,
output wire [63:0]WriteData,
output wire [63:0]ReadData1,
output wire [63:0]ReadData2,
output wire Branch, MemRead, MemToReg, MemWrite, ALUSrc, RegWrite,
output wire [1:0] ALUOp,
output wire [63:0] ImmData,
output wire [63:0] mo,
output wire [3:0] operation,
output wire [63:0] aluo,
output wire [63:0] datamemoryReadData,
output wire [63:0] index1,
output wire [63:0] index2,
output wire [63:0] index3,
output wire [63:0] index4,
output wire [63:0] index5,
output wire [63:0] index6,
output wire [63:0] index7,
output wire [63:0] index8
```

```
);

wire final_branch;
wire [3:0] funct;

// Instruction Fetch
adder a1(PC_Out, 64'd4, ao1);
mux m1(ao1, ao2,(Branch&&final_branch) , PC_In);
program_counter pc(clk, reset, PC_In,PC_Out);
instruction_memory im(PC_Out, instruction);
parser ip(instruction, opcode, rd, funct3, rs1, rs2, funct7);

// Instruction Decode
control_unit cu(opcode, Branch, MemRead, MemToReg, MemWrite, ALUSrc, RegWrite, ALUOp);
register_file rf(WriteData, rs1, rs2, rd, RegWrite, clk, reset, ReadData1, ReadData2);
imm_gen imm_gen(instruction, ImmData);

// Execution
mux m2(ReadData2, ImmData, ALUSrc, mo);
assign funct = {instruction[30], instruction[14:12]};
alu_control alu_c(ALUOp, funct, operation);
adder a2(PC_Out, ImmData*2, ao2);
alu_64bit alu(ReadData1, mo, operation, aluo, zero);

// Memory Access
data_memory dm(aluo, ReadData2, clk, MemWrite, MemRead, datamemoryReadData,index1,index2,index3,i
branching_unit bu(funct3, ReadData1, mo, final_branch);

// Write Back
mux m3(aluo, datamemoryReadData, MemToReg, WriteData);

endmodule
```

Results:

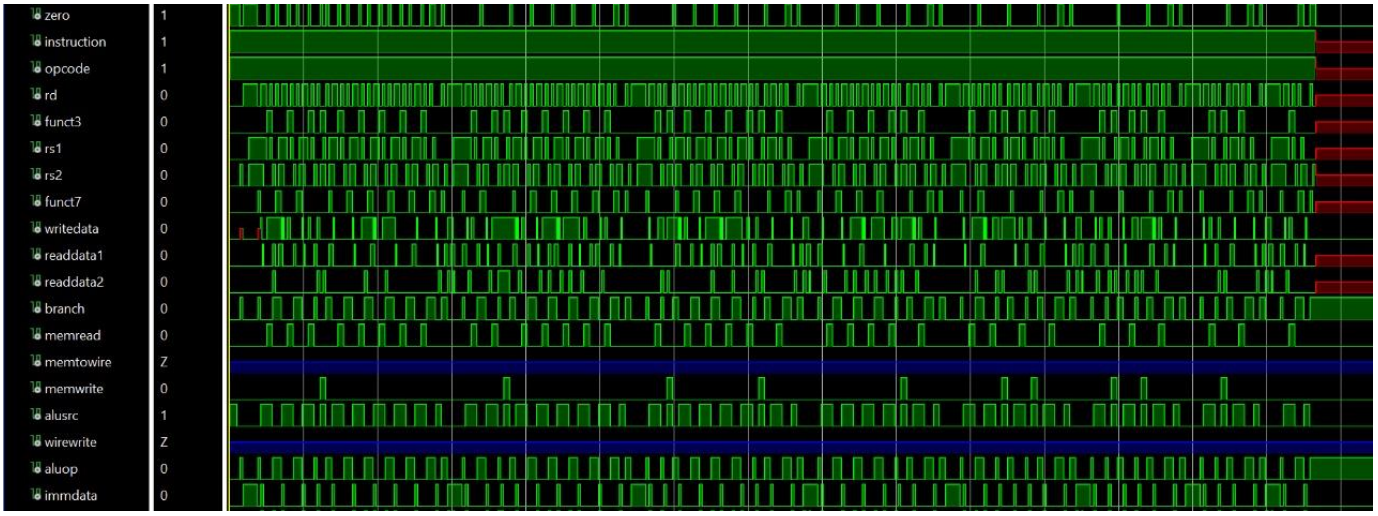

Figure 1

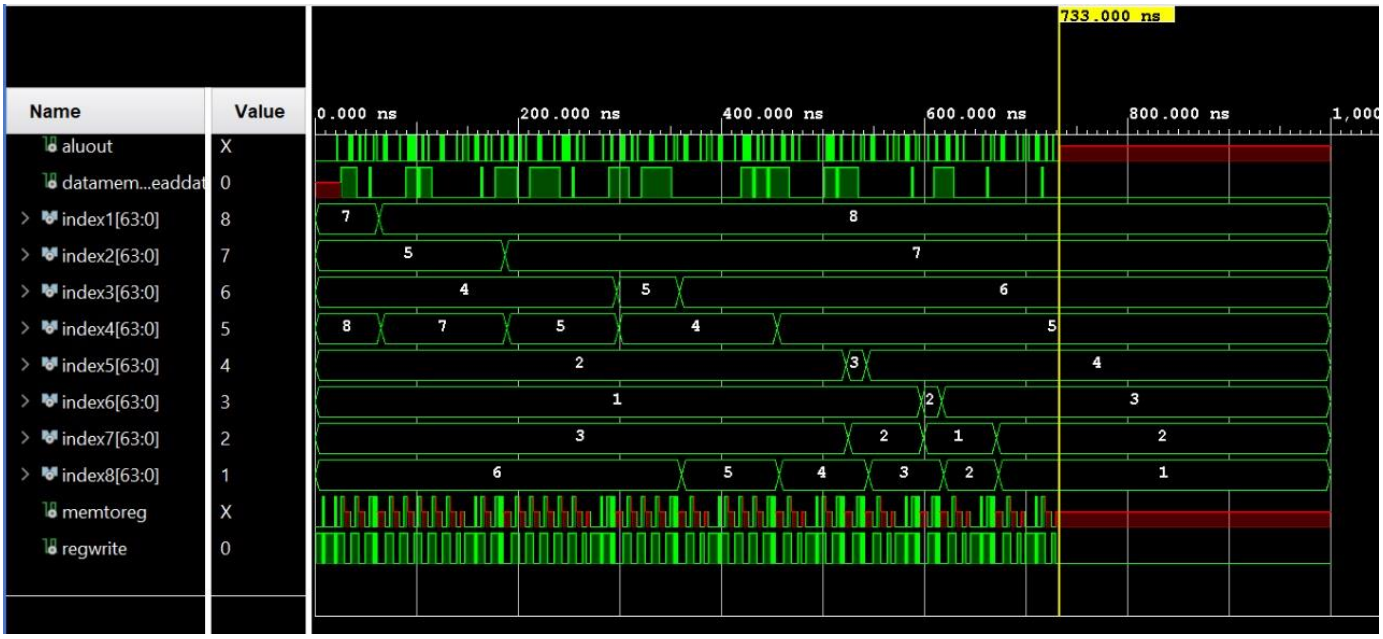

Figure 2

The values are sorted in around 733,000 ns and the sorted values are shown in **Figure 2**.

Block Diagram:

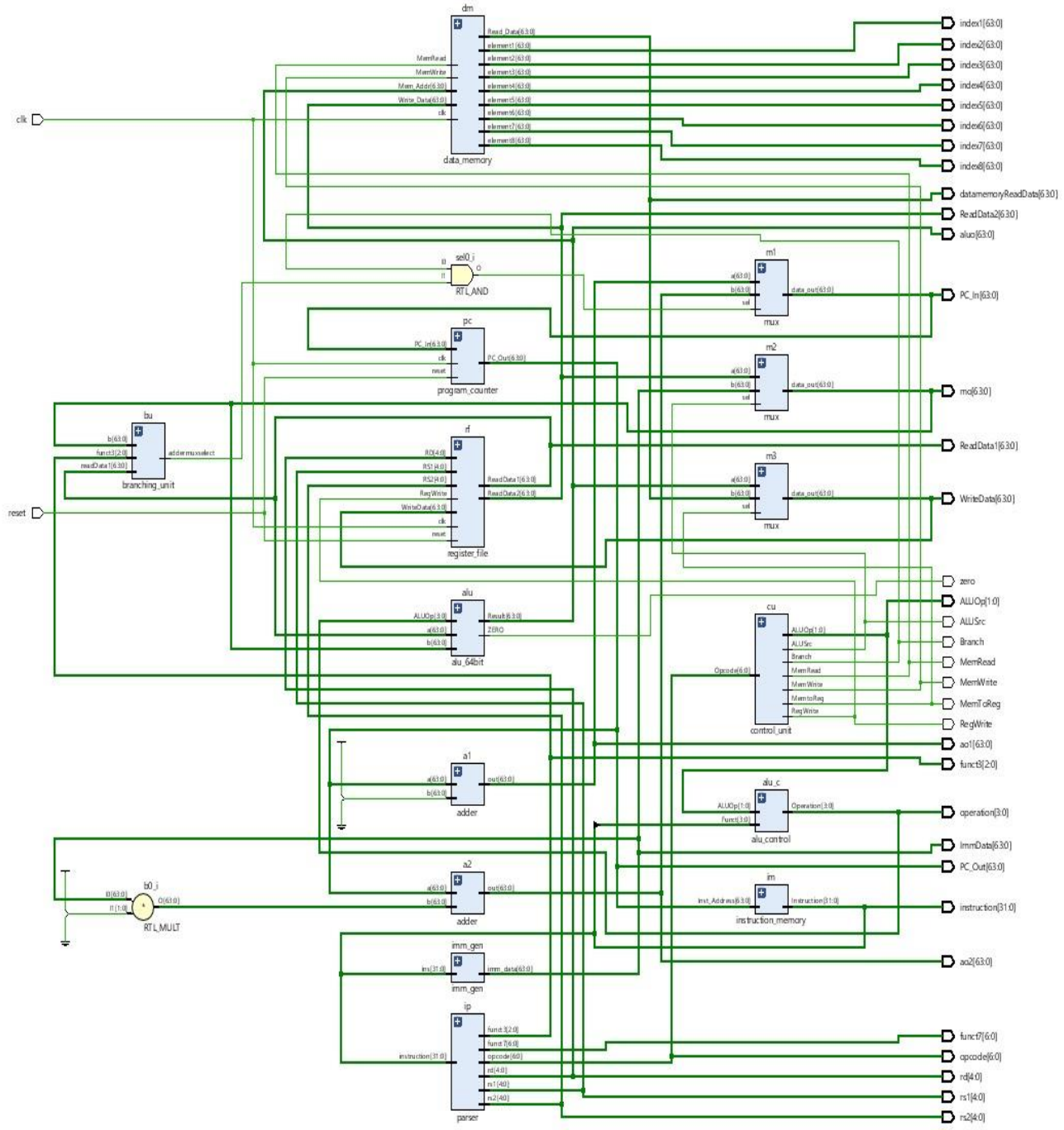

Figure 3

Task 2: Pipelined Processor Implementation

1. We designed the pipelined processor architecture with five stages (Fetch, Decode, Execute, Memory, Writeback).
2. We tested instructions individually to ensure correctness in isolation and also we tested different types of instructions to verify proper functioning.

Pipelined Processor:

```
module riscV_processor(
input clk,
input reset,
output wire [63:0] PC_Out,
output wire [63:0] PC_In,
output wire [31:0] instruction,
output wire [4:0] rs1,
output wire [4:0] rs2,
output wire [4:0] rd,
output wire [63:0]WriteData,
output wire [63:0]ReadData1,
output wire [63:0]ReadData2,
output wire [63:0] ImmData,
output wire [63:0] mo,
output wire [63:0] aluo,
output wire zero,
output wire [63:0] ao1,
output wire [63:0] ao2,
output wire [6:0] opcode,
output wire [2:0] funct3,
output wire [6:0] funct7,
output wire Branch, MemRead, MemToReg, MemWrite, ALUSrc, RegWrite,
output wire [1:0] ALUOp,
```

```
output wire [3:0] operation,
output wire [63:0] datamemoryReadData,
output wire [63:0] index1,
output wire [63:0] index2,
output wire [63:0] index3,
output wire [63:0] index4,
output wire [63:0] index5,
output wire [63:0] index6,
output wire [63:0] index7,
output wire [63:0] index8
    );
wire addermuxselect;
wire Branch_finale;
wire [3:0] funct;
wire [63:0] ifidPC_Out;
wire [31:0] ifidinst;
wire [63:0]idexPC_Out,idexReadData1,idexReadData2, idexImmData;
wire [4:0] idexrs1, idexrs2, idexrd;
wire idexBranch, idexMemRead, idexMemToReg, idexMemWrite, idexRegWrite, idexALUSrc;
wire [1:0] idexALUOp;
wire [63:0] exmemadderout;
wire exmemzero;
wire [63:0]   exmemaluo;
```

```
wire [63:0]  exMemWriteDataout;
wire [4:0]  exmemrd;
wire  exmemBranch, exmemMemRead, exmemMemToReg, exmemMemWrite, exmemRegWrite;
wire [63:0] memwbreaddataout, memwbaluo;
wire [4:0] memwbrd;
wire memwbMemToReg, mwmwbRegWrite;
wire [1:0] forwarda, forwardb;
wire [63:0] m3to1out1, m3to1out2;
wire [3:0] idexfunct;

// Instruction Fetch
program_counter pc(clk, reset, PC_In,PC_Out);
instruction_memory im(PC_Out, instruction);
adder a1(PC_Out, 64'd4, ao1);
InsParser ip(ifidinst, opcode, rd, funct3, rs1, rs2, funct7);

// IF/ID
ifidreg IFID(clk, reset, PC_Out, instruction, ifidPC_Out, ifidinst);

//Instruction Decode
Control_Unit cu(opcode, Branch, MemRead, MemToReg, MemWrite, ALUSrc, RegWrite, ALUOp);
registerFile rf(WriteData, rs1, rs2, memwbrd, memwbRegWrite, clk, reset, ReadData1, ReadData2);
ImmGen imm gen(ifidinst, ImmData);
```

```
// ID/EX
idexreg IDEX( clk, reset,  ifidPC_Out,ReadData1,ReadData2, ImmData,   rs1, rs2, rd, {ifidinst[30],ifidinst[14:12] },
    Branch, MemRead, MemToReg, MemWrite, RegWrite, ALUSrc, ALUOp,  idexPC_Out,idexReadData1,idexReadData2, idexImmData,
    idexrs1, idexrs2, idexrd,   idexfunct,   idexBranch, idexMemRead, idexMemToReg, idexMemWrite, idexRegWrite,
    idexALUSrc,   idexALUOp);


// Execution
adder a2(idexPC_Out, idexImmData*2, ao2);
Mux_3to1 mm(idexReadData1, WriteData, exmemaluo, forwarda, m3to1out1);
Mux_3to1 mn(idexReadData2, WriteData, exmemaluo, forwardb, m3to1out2);
Mux m2(m3to1out2, idexImmData, idexALUSrc, mo);
ALU_64_bit alu(m3to1out1, mo, operation, aluo, zero);
ALU_Control alu_c(idexALUOp, idexfunct, operation);

// EX/MEM
exmemreg EXMEM( clk,reset, ao2, aluo, zero, m3to1out2, idexrd, idexBranch, idexMemRead, idexMemToReg, idexMemWrite,
    idexRegWrite, addermuxselect, exmemadderout, exmemzero, exmemaluo, exMemWriteDataout, exmemrd, exmemBranch,
    exmemMemRead, exmemMemToReg, exmemMemWrite, exmemRegWrite, Branch_finale);


// Memory Access + Write Back
Data_Memory dm(exmemaluo, exMemWriteDataout, clk, exmemMemWrite, exmemMemRead,
    datamemoryReadData,index1,index2,index3,index4,index5,index6,index7,index8);
Mux m1(ao1, exmemadderout,(exmemBranch&&Branch_finale) , PC_In);
memwbreg memwb(clk, reset, datamemoryReadData, exmemaluo, exmemrd, exmemMemToReg, exmemRegWrite, memwbreaddataout,
    memwbaluo, memwbrd, memwbMemToReg, memwbRegWrite);
forwardingunit fu(idexrs1, idexrs2, exmemrd, memwbrd, memwbRegWrite, exmemRegWrite, forwarda, forwardb);
branching_unit bu(idexfunct[2:0], idexReadData1, mo, addermuxselect);
Mux m3(memwbaluo, memwbreaddataout, memwbMemToReg, WriteData);

endmodule
```

Results:

Isolation:

Add x9, x9, x8 = 10 + 9 = 19

```
// add x9, x9, x8
{inst_mem[3], inst_mem[2], inst_mem[1], inst_mem[0]} = 32'h008484B3; // 00000000100001001000010010110011
```

Figure 4

So, the values in the registered are stored in the form of "i+1" that means if we are referring x9 register then the value in that register will be 9 + 1 = 10, similarly the value in x8 register will be 8 +1 = 9.

The **figure 4** shows results if we run a instruction in isolation. So, the instruction add x9, x9, x8 is loading value 10 and 9 and in the very next cycle outputting 10 + 9 = 19.

Together:

```
// sub x1, x2, x3
{inst_mem[7], inst_mem[6], inst_mem[5], inst_mem[4]} = 32'h403100B3; // 01000000001100010000000010110011
// addi x4, x5, 6
{inst_mem[11], inst_mem[10], inst_mem[9], inst_mem[8]} = 32'h00628213; // 00000000011000101000001000010011
// add x9, x9, x8
{inst_mem[3], inst_mem[2], inst_mem[1], inst_mem[0]} = 32'h008484B3; // 00000000100001001000010010110011
```

Figure 5

Figure 5 shows result when we run multiple instructions together so the first instruction which is executing is add x9, x9, x8 then sub x1, x2, x3 and finally addi x4, x5, 6.

So 10 + 9 = 19, 3-4 =-1 and 6+6 =12 is shown in results subsequently.

Block Diagram:

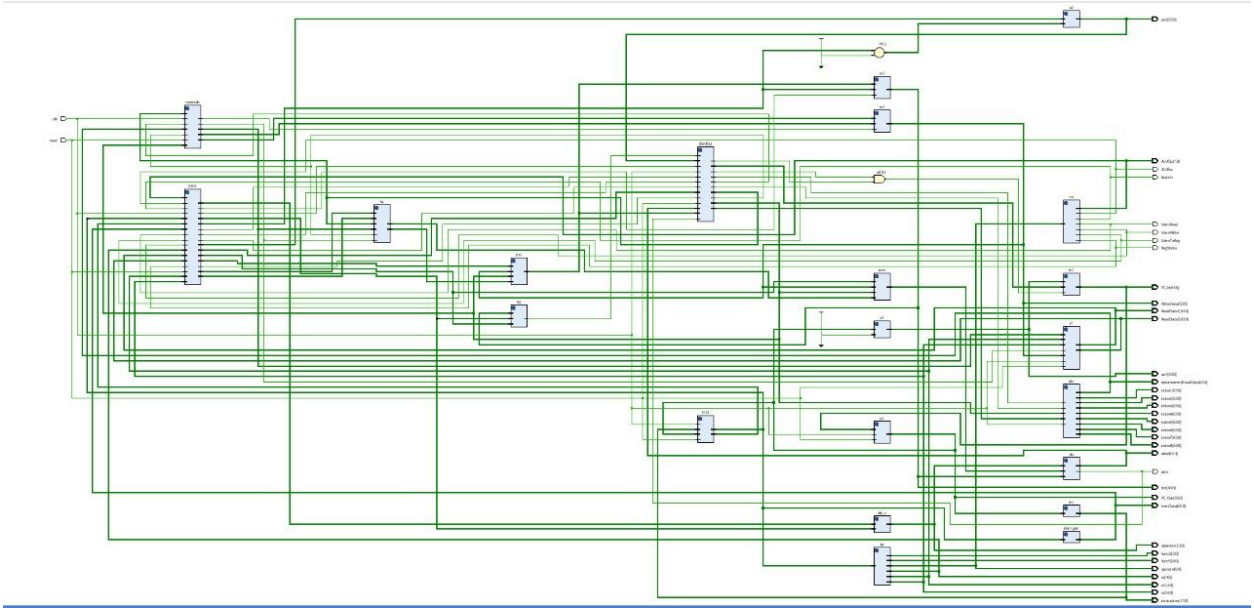

Figure 6

Task 3: Hazard Detection and Handling

1. We identify potential hazards like data hazards, control hazards, structural hazards in the pipelined processor.
2. We added forwarding, stalling, flushing and tested the handling of hazards to ensure correctness.

- Structural hazards result from not having enough hardware available to execute multiple instructions simultaneously.
  - These are avoided by adding more functional units (e.g., more adders or memories) or by redesigning the pipeline stages.
- Data hazards can occur when instructions need to access registers that haven't been updated yet.
  - Hazards from R-type instructions can be avoided with forwarding.
  - Loads can result in a "true" hazard, which must stall the pipeline.
- Control hazards arise when the CPU cannot determine which instruction to fetch next.
  - We can minimize delays by doing branch tests earlier in the pipeline.
  - We can also take a chance and predict the branch direction, to make the most of a bad situation.

Hazards Solving Code:

```
module riscV_processor(
   input clk,
   input reset,
   input wire[63:0] element1,
   input wire[63:0] element2,
   input wire[63:0] element3,
   input wire[63:0] element4,
   input wire[63:0] element5,
   input wire[63:0] element6,
   input wire[63:0] element7,
   input wire[63:0] element8,
   input stall, flush);

   wire branch;
   wire memread;
   wire memtoreg;
   wire memwrite;
   wire ALUsrc;
   wire regwrite;
   wire [1:0] ALUop;
   wire regwrite_memwb_out;
   wire [63:0] readdata1, readdata2;
   wire [63:0] r8, r19, r20, r21, r22;
   wire [63:0] write_data;
   wire [63:0] pc_in;
   wire [63:0] pc_out;
   wire [63:0] adderout1;
   wire [63:0] adderout2;
   wire [31:0] instruction;
   wire[31:0] inst_ifid_out;
   wire [6:0] opcode;
   wire [4:0] rd, rs1, rs2;
   wire [2:0] funct3;
```

```
wire [6:0] funct7;
wire [63:0] imm_data;
wire [63:0] random;
wire [63:0] a1;
wire [4:0] RS1;
wire [4:0] RS2;
wire [4:0] RD;
wire [63:0] d, M1, M2;
wire Branch;
wire Memread;
wire Memtoreg;
wire Memwrite;
wire Regwrite;
wire Alusrc;
wire [1:0] aluop;
wire [3:0] funct4_out;
wire [63:0] threeby1_out1;
wire[63:0] threeby1_out2;
wire[63:0]  alu_64_b;
wire [63:0] write_Data;
wire [63:0] exmem_out_adder;
wire exmem_out_zero;
wire [63:0] exmem_out_result;
wire [4:0] exmemrd;
wire BRANCH,MEMREAD,MEMTOREG,MEMEWRITE,REGWRITE;
wire [63:0] AluResult;
wire zero;
wire [3:0] operation;
wire [63:0] readdata;
wire[63:0] muxin1,muxin2;
wire [4:0] memwbrd;
wire memwb_memtoreg;
wire memwb_regwrite;
```

```
wire [4:0] memwbrd;
wire memwb_memtoreg;
wire memwb_regwrite;
wire [1:0] forwardA;
wire [1:0] forwardB;
wire addermuxselect;
wire branch_final;


pipeline_flush p_flush (branch_final & BRANCH, flush);
hazard_detection_unit hu (Memread, inst_ifid_out, RD, stall);
program_counter pc (pc_in, stall, clk, reset, pc_out);
instruction_memory im (pc_out, instruction);
adder adder1 (pc_out, 64'd4, adderout1);

 IFID i1 (clk, reset, stall, instruction, pc_out, inst_ifid_out, random, flush);

Parser ip (inst_ifid_out, opcode, rd, funct3, rs1, rs2, funct7);

CU cu (opcode, branch, memread, memtoreg, memwrite, ALUsrc, regwrite, ALUop, stall);

data_extractor immextr (inst_ifid_out, imm_data);


registerFile regfile (clk, reset, rs1, rs2, memwbrd, write_data, memwb_regwrite, readdata1, readdata2,
  r8, r19, r20, r21, r22);

IDEX i2 (clk, flush, reset, {inst_ifid_out[30],inst_ifid_out[14:12]}, random, readdata1,
  readdata2, imm_data, rs1, rs2, rd, branch, memread, memtoreg, memwrite, ALUsrc, regwrite, ALUop,
  a1, RS1, RS2, RD, d, M1, M2, funct4_out, Branch, Memread, Memtoreg, Memwrite,
  Regwrite, Alusrc, aluop);

adder adder2 (a1, d << 1, adderout2);
```

```
adder adder2 (a1, d << 1, adderout2);

ThreebyOneMux m1 (write_data, exmem_out_result, forwardA, threeby1_out1);

ThreebyOneMux m2
(
  M2,write_data,exmem_out_result,forwardB,threeby1_out2
);

twox1Mux mux1
(
  threeby1_out2,d,Alusrc,alu_64_b
);

Alu64 alu
(
  threeby1_out1,
  alu_64_b,
  operation,
  AluResult,
  zero
);

 alu_control ac
(
  aluop,
  funct4_out,
  operation
);
```

```
EXMEM i3
(
  clk,reset,adderout2,AluResult,zero,flush,
  threeby1_out2,RD, addermuxselect,
  Branch,Memread,Memtoreg,Memwrite,Regwrite,
  exmem_out_adder,exmem_out_zero,exmem_out_result,write_Data,
  exmemrd,BRANCH,MEMREAD,MEMTOREG,MEMEWRITE,REGWRITE, branch_final
);

data_memory datamem
(
  write_Data,
  exmem_out_result,
  MEMEWRITE,
  clk,
  MEMREAD,
  readdata,
  element1,
  element2,
  element3,
  element4,
  element5,
  element6,
  element7,
  element8
);
```

```
twox1Mux mux2
(
  adderout1,exmem_out_adder,BRANCH & branch_final,pc_in
);


MEMWB i4

(
  clk,reset,readdata,
  exmem_out_result,exmemrd,MEMTOREG,REGWRITE,
  muxin1,muxin2,memwbrd,memwb_memtoreg,memwb_regwrite
);

 twox1Mux mux3
(
  muxin2,muxin1,memwb_memtoreg,write_data
);


 ForwardingUnit f1
(
  RS1,RS2,exmemrd,
  memwbrd,memwb_regwrite,
  REGWRITE,
  forwardA,forwardB
);


 branching_unit branc
 (
   funct4_out[2:0],M1,alu_64_b,addermuxselect
 );


endmodule
```

Results:

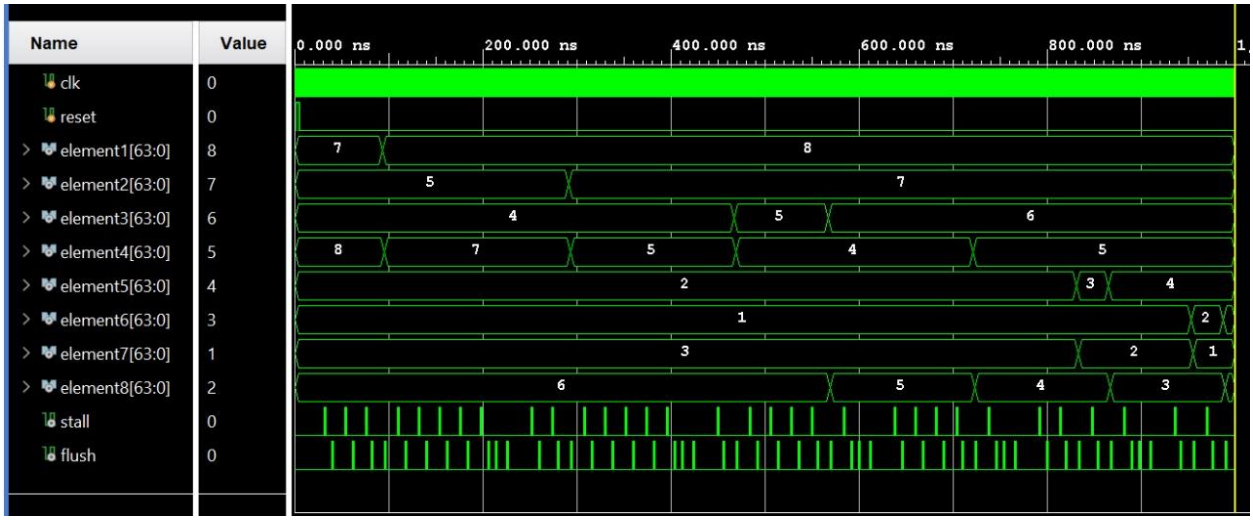

Figure 7

In **Figure 7** the values are sorted in around 1,000,000 ns.

Block Diagram:

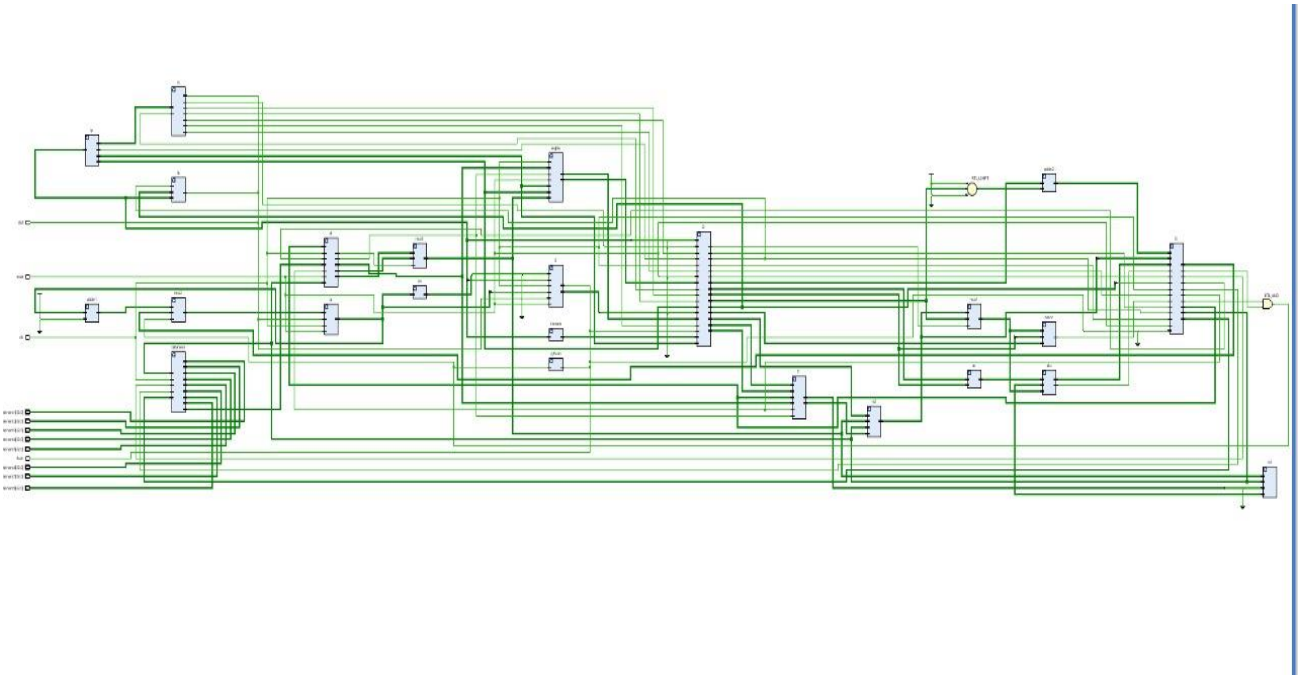

Figure 8

Task 4: Performance Comparison

1. We ran the array sorting program on both the single-cycle and pipelined processors amd measured the execution time for each processor.

In Task 1 our sorting is completed in around 733,000 ns while after we run it on pipelined single-cycle processor the time is reduced to around 1,000,000 ns.

Speed down = 733,000/1,000,000 = 0.733

Challenges:

Task 1:

1. Converting the Bubble Sort Algorithm into hexadecimal, it was very time consuming and a single mistake can make the code go bananas.

2. Additionally, the time of every cycle was going beyond 1,000,000 and sorting was not completed.

Task 2:

1. We encode the instruction wrong which took a lot of time to solve.
2. Additionally, we had difficulty in reading the simulation because we didn't efficiently made signals visible in output. Basically, we first only used clock and rest as output and taking other values as reg internally, which created problems in the output.

Task 3:

1. Hazard detection theory was unclear and took us time to first understand the logic, plus the implementation was also very confusing to put in code and make sure everything worked
2. The time required to complete the sorting was longer then non-pipelined version that confused us further.

Task Division:

Task 1 = Daniyal

Task 2 = Ikhlas

Task 3 = Ikhlas, Daniyal

Report = Ikhlas, Daniyal

Conclusions:

- The time it takes to sort the values in Task 1 was faster even though the code was not pipelined but it took more time to sort in task 3 with all the data hazards been taken care of.
- The sorting is done in task 3 is 26.7 % slower.
- We think this is because we have a lot of branches, that results in a lot of wasted Clock Cycles because we flush at every branch. Due to this this cost us a lot more time than Task 1.
- If we decrease the clock to 0.4 or 0.5 in task 3 then our time taken to sort becomes 590,500 ns.

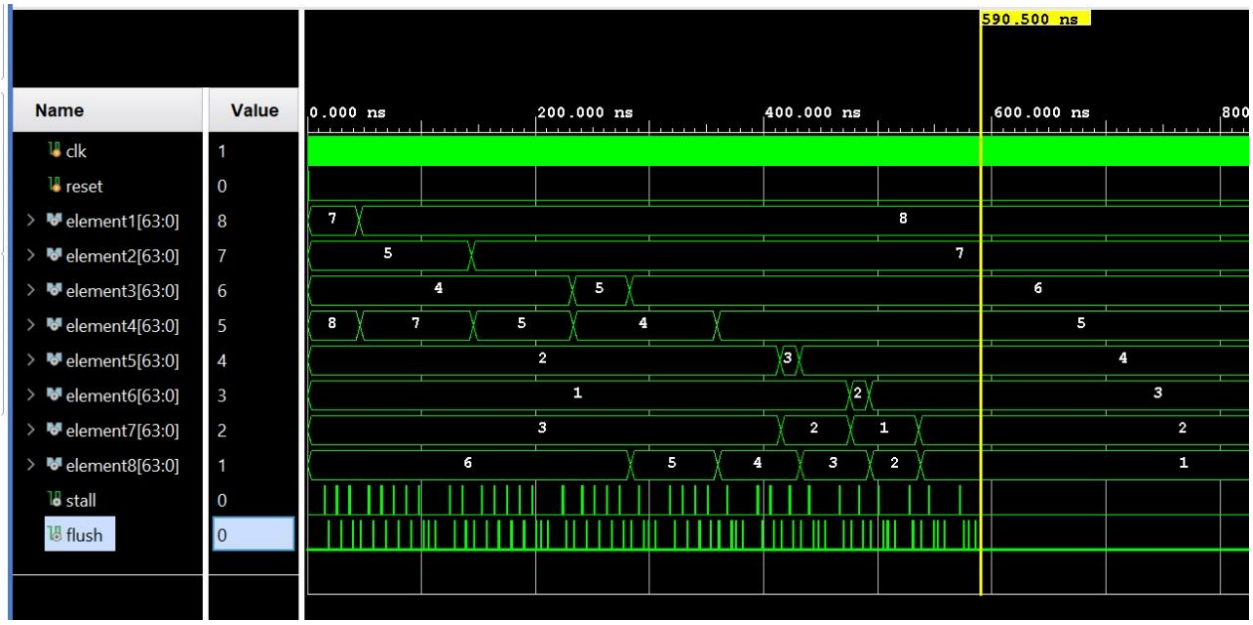

Figure 9

- In **Figure 9** we kept the clock at 0.5 then the time to sort in Task 3 is coming at around 590,000 ns. This way the speed up will be as follows:

- Speed up = 733,000/590,500 = 1.24

Appendices:

- **GitHub Link**

https://github.com/i-ahmed25/CA_riscV_processor