Computer Architecture

Abeera Farooq Alam
Muhammad Zain Yousaf
Omer Rastgar

Project : 5 Stage RISC-V Pipeline Processor

CS-353: Computer Architecture
Habib University
31/01/2022

Contents

1 Introduction **3**
1.1 Description 3

2 Task1 **4**
2.1 Objectives 4
2.2 Code 4
2.2.1 Instruction Sort Algorithm 4
2.2.2 Instruction Sort in Assembly Language 4
2.2.3 Implementing Instruction Sort on Single Cycle Processor 6
2.3 Results 22

3 Task2 **23**
3.1 Objectives 23
3.2 Creating Pipeline Registers 23
3.2.1 IF/ID 23
3.2.2 ID/EX 25
3.2.3 EX/MEM 26
3.2.4 MEM/WB 27
3.3 Forwarding Unit 28
3.3.1 Modules for Forward A and B 29
3.3.2 Implementing the Forwarding Unit 30
3.4 Integrating Register Modules and Forwarding Unit 32
3.4.1 Testing 35
3.5 Results 37

4 Task3 **38**
4.1 Objectives 38
4.2 Hazard Detection (Stalling) 39
4.2.1 Implementation of Hazard Detection Block 40
4.2.2 Flushing 40
4.3 Testing the New Pipeline 45

4.4 Testing the Sorting Algorithm 47
4.5 Results 47
4.6 Challenges faced 48

Bibliography **49**

Introduction

1.1 Description

This project aims to convert the single processor built in our labs to a pipelined one. There are essentially three parts we will be performing for this.

- Implementing insertion sort on the single cycle processor.
- Modifying our single cycle processor to a 5 stage pipelined one.
- Introducing hazard detection circuitry i.e: through forwarding, stalling and flushing the pipeline

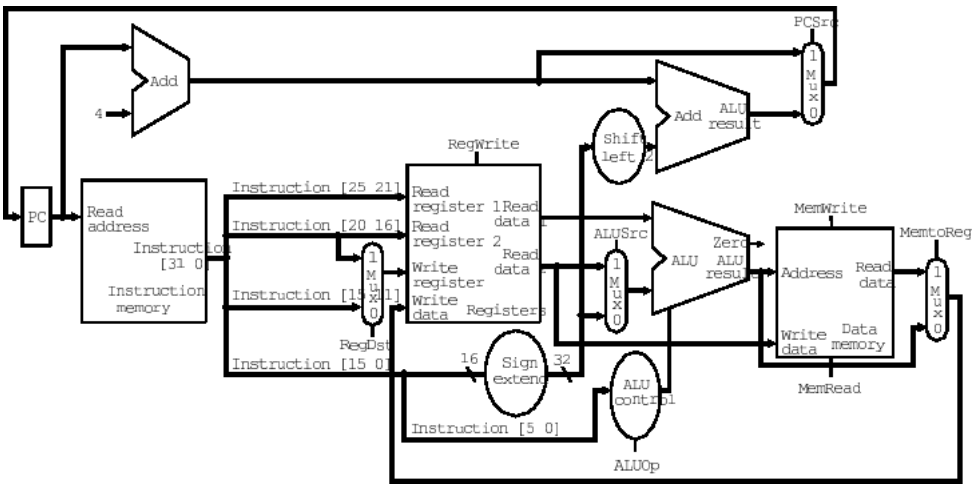


Figure 1.1: Caption

Task1

2.1 Objectives

- Write insertion sort in assembly
- Test insertion sort on
- Make branching logic
- Create test bench
- ensure if the algorithm works on Single cycle processor

2.2 Code

2.2.1 Instruction Sort Algorithm

For this task, we chose to use Instruction Sort as our sorting Algorithm. Below is the algorithm that we followed to implement our code in assembly language.

```
for i : 1 to length(A) - 1
    j = i
    while j > 0 and A[j-1]>A[j]
        swap A[j] and A[j-1]
        j = j - 1
```

2.2.2 Instruction Sort in Assembly Language

The code below performs sorting on array A = [8,5,3,9] where the base address is defined to be 0x100. The array before sorting is shown below.

0x00000018	09	00	00	00
0x00000014	93	09	40	00
0x00000010	03	00	00	00
0x0000000c	13	02	30	00
0x00000008	05	00	00	00
0x00000004	93	01	50	00
0x00000000	08	00	00	00

Figure 2.1: Before Sorting

```
addi x10, x0, 0x000
li x3, 5
li x11, 24 #len of array
li x4, 3
li x16, 9
li x19,4
li x6, 8
li x5,8

sw x6, 0x000(x10)
sw x3, 0x008(x10)
sw x4, 0x0010(x10)
sw x16, 0x018(x10)


Insertion:

li x14, 8#i

for:
addi x18,x14,0  #j

while:
sub x19,x18,x5
add x21,x10,x18
add x20,x10,x19
lw x22, 0(x20)
lw x23, 0(x21)
bgt x22,x23 swap

# loop
addi x14,x14,8
```

```
blt x14,x11, for
beq x0 x0 end

swap:
sw x23, 0(x20)      #A[j-1]= x23
sw x22, 0(x21)     #A[j] = temp
sub x18,x18,x5
beq x18 x0 for
bge x18 x0 while

end:
```

0x00000018	09	00	00	00
0x00000014	93	09	40	00
0x00000010	08	00	00	00
0x0000000c	13	02	30	00
0x00000008	05	00	00	00
0x00000004	93	01	50	00
0x00000000	03	00	00	00

Figure 2.2: Simulation Result

2.2.3 Implementing Instruction Sort on Single Cycle Processor

Decoding Instructions

To run our sorting algorithmm on RISC V Processor, we had to decode our assembly language instructions into binary instructions. From the machine code provided in the simulation tab on https://venus.kvakil.me/, we were able to generate these binary instructions which we then populated into our instruction memory.

Modifications Required for our Single Cycle Processor

Our Single Cycle Processor previously did not cater to bge instructions and so to successfully implement our sorting algorithm, we had to make change

to two of our modules; ALU 64 and Control Unit.
In ALU 64, we added an additional output register "sign". This sign bit is used with the control outputs to decide if the number is less than or greater than the other number being compared.
In the Control Unit, we checked func3 to see which type of branching instruction is being used, and based on that we updated the register values for BEQ, BLT and BGE.

Modified Code for Single Cycle Processor

```
module MUX (A, B, sel, out);

  input [63:0] A;
  input [63:0] B;
  input sel;
  output [63:0] out;

  assign out = (sel == 1'b0)? A : B;

endmodule

module Parser (ins, opcode, rd, funct3, rs1, rs2, funct7);

  input [31:0] ins;

  output [6:0] opcode;
  output [4:0] rd;
  output [2:0] funct3;
  output [4:0] rs1;
  output [4:0] rs2;
  output [6:0] funct7;

  assign opcode = ins[6:0];
  assign rd = ins[11:7];
  assign funct3 = ins[14:12];
  assign rs1 = ins[19:15];
  assign rs2 = ins[24:20];
  assign funct7 = ins[31:25];

endmodule

module ImmGen (ins, imm_data);

  input [31:0] ins;
  output reg [63:0] imm_data;

```

```
37   always @ (ins)
38     begin
39       if (ins[6:5] == 2'b01)
40         begin
41           imm_data[4:0] = ins[11:7];
42           imm_data[11:5] = ins[31:25];
43         end
44
45       else if (ins[6:5] == 2'b00)
46         begin
47           imm_data[11:0] = ins[31:20];
48         end
49
50       else if (ins[6:5] == 2'b11)
51         begin
52           imm_data[3:0] = ins[11:8];
53           imm_data[9:4] = ins[30:25];
54           imm_data[10] = ins[7];
55           imm_data[11] = ins[31];
56         end
57
58       imm_data[63:12] = {52{(ins[31])}};
59     end
60 endmodule
61
62
63 module registerFile(clk, reset, wtData, rs1, rs2, rd,
      regWrite, rd1, rd2,r1,r2,r3,r4,r22,r23,r20,r21,r19,r18);
64
65
66   input clk, reset;
67   input [63:0] wtData;
68   input [4:0] rs1;
69   input [4:0] rs2;
70   input [4:0] rd;
71   input regWrite;
72
73   output reg [63:0] rd1;
74   output reg [63:0] rd2;
75   output reg [63:0] r1,r2,r3,r4,r22,r23,r20,r21,r19,r18;
76
77   reg [63:0] register [31:0];
78
79
80
81
82   initial
83
84     begin
```

```
85
86        register[0] = 64'd0;
87  register[1] = 64'd0;
88  register[2] = 64'd0;
89  register[3] = 64'd0;
90  register[4] = 64'd0;
91  register[5] = 64'd0;
92  register[6] = 64'd0;
93  register[7] = 64'd0;
94  register[8] = 64'd0;
95  register[9] = 64'd0;
96  register[10] = 64'd0;
97  register[11] = 64'd0;
98  register[12] = 64'd0;
99  register[13] = 64'd0;
100 register[14] = 64'd0;
101 register[15] = 64'd0;
102 register[16] = 64'd0;
103 register[17] = 64'd0;
104 register[18] = 64'd0;
105 register[19] = 64'd0;
106 register[20] = 64'd0;
107 register[21] = 64'd0;
108 register[22] = 64'd0;
109 register[23] = 64'd0;
110 register[24] = 64'd0;
111 register[25] = 64'd0;
112 register[26] = 64'd0;
113 register[27] = 64'd0;
114 register[28] = 64'd0;
115 register[29] = 64'd0;
116 register[30] = 64'd0;
117 register[31] = 64'd0;
118
119
120     end
121
122
123
124
125   always @ (*)
126
127     begin
128
129       rd1 = register[rs1];
130
131       rd2 = register[rs2];
132
133
```

```
134
135
136       if (reset == 1'b1)
137
138         begin
139
140           rd1 = 64'd0;
141
142           rd2 = 64'd0;
143
144         end
145
146     end
147
148
149
150   always @ (posedge clk)
151
152     begin
153
154       if (regWrite == 1'b1)
155
156         begin
157
158           register[rd] = wtData;
159
160         end
161
162
163     end
164
165   always @ (*)
166
167     begin
168
169       r1=register[3];
170       r2=register[4];
171       r3=register[16];
172       r4=register[6];
173       r20=register[20];
174       r21=register[21];
175       r22=register[22];
176       r23=register[23];
177       r19= register[19];
178       r18=register[20];
179     end
180
181 endmodule
182
```

```
183
184
185 module ALU_64 (a, b, ALUop, result, zero, sign);
186
187   input [63:0] a;
188   input [63:0] b;
189   input [3:0] ALUop;
190
191   output reg [63:0] result;
192   output reg zero;
193   output reg sign;
194
195   always @ (*)
196     begin
197
198       if (ALUop[3:0] == 4'b0000)
199         begin
200           result = a & b;
201         end
202
203       else if (ALUop[3:0] == 4'b0001)
204         begin
205           result = a | b;
206         end
207
208       else if (ALUop[3:0] == 4'b0010)
209         begin
210           result = a + b;
211         end
212
213       else if (ALUop[3:0] == 4'b0110)
214         begin
215           result = a - b;
216         end
217
218       else if (ALUop[3:0] == 4'b1100)
219         begin
220           result = ~(a | b);
221         end
222
223       zero = (result == 0)? 1'b1: 1'b0;
224
225       sign = result[63];
226
227     end
228
229 endmodule
230
231
```

```
232
233 module Instruction_Memory (Instr_Add, Instruction);
234
235   input [63:0] Instr_Add;
236   output [31:0] Instruction;
237
238   reg [7:0] iMEM [175:0];
239
240   initial
241     begin
242 {iMEM[3], iMEM[2], iMEM[1], iMEM[0]} = 32'h00000513;
243 {iMEM[7], iMEM[6], iMEM[5], iMEM[4]} = 32'h00500193;
244 {iMEM[11], iMEM[10], iMEM[9], iMEM[8]} = 32'h01800593;
245 {iMEM[15], iMEM[14], iMEM[13], iMEM[12]} = 32'h00300213;
246 {iMEM[19], iMEM[18], iMEM[17], iMEM[16]} = 32'h00900813;
247 {iMEM[23], iMEM[22], iMEM[21], iMEM[20]} = 32'h00400993;
248 {iMEM[27], iMEM[26], iMEM[25], iMEM[24]} = 32'h00800313;
249 {iMEM[31], iMEM[30], iMEM[29], iMEM[28]} = 32'h00800293;
250 {iMEM[35], iMEM[34], iMEM[33], iMEM[32]} = 32'h00652023;
251 {iMEM[39], iMEM[38], iMEM[37], iMEM[36]} = 32'h00352423;
252 {iMEM[43], iMEM[42], iMEM[41], iMEM[40]} = 32'h00452823;
253 {iMEM[47], iMEM[46], iMEM[45], iMEM[44]} = 32'h01052c23;
254 {iMEM[51], iMEM[50], iMEM[49], iMEM[48]} = 32'h00800713;
255 {iMEM[55], iMEM[54], iMEM[53], iMEM[52]} = 32'h00070913;
256 {iMEM[59], iMEM[58], iMEM[57], iMEM[56]} = 32'h405909b3;
257 {iMEM[63], iMEM[62], iMEM[61], iMEM[60]} = 32'h01250ab3;
258 {iMEM[67], iMEM[66], iMEM[65], iMEM[64]} = 32'h01350a33;
259 {iMEM[71], iMEM[70], iMEM[69], iMEM[68]} = 32'h000a2b03;
260 {iMEM[75], iMEM[74], iMEM[73], iMEM[72]} = 32'h000aab83;
261 {iMEM[79], iMEM[78], iMEM[77], iMEM[76]} = 32'h016bc863;
262 {iMEM[83], iMEM[82], iMEM[81], iMEM[80]} = 32'h00870713;
263 {iMEM[87], iMEM[86], iMEM[85], iMEM[84]} = 32'hfeb740e3;
264 {iMEM[91], iMEM[90], iMEM[89], iMEM[88]} = 32'h00000c63;
265 {iMEM[95], iMEM[94], iMEM[93], iMEM[92]} = 32'h017a2023;
266 {iMEM[99], iMEM[98], iMEM[97], iMEM[96]} = 32'h016aa023;
267 {iMEM[103], iMEM[102], iMEM[101], iMEM[100]} = 32'h40590933;
268 {iMEM[107], iMEM[106], iMEM[105], iMEM[104]} = 32'hfc0906e3;
269 {iMEM[111], iMEM[110], iMEM[109], iMEM[108]} = 32'hfc0956e3;
270  /*
271  {iMEM[167], iMEM[166], iMEM[165], iMEM[164]} = 32'h00032a03;
272 {iMEM[171], iMEM[170], iMEM[169], iMEM[168]} = 32'h0003aa83;
273 {iMEM[175], iMEM[174], iMEM[173], iMEM[172]} = 32'h00042b03;
274   */
275   end
276
277   assign Instruction[7:0] = iMEM[Instr_Add];
278   assign Instruction[15:8] = iMEM[Instr_Add + 1'b1];
279   assign Instruction[23:16] = iMEM[Instr_Add + 2'b10];
280   assign Instruction[31:24] = iMEM[Instr_Add + 2'b11];
```

```
281
282 endmodule
283
284
285
286 module Data_Memory (Mem_Addr, W_Data, clk, MemWrite, MemRead,
        Read_Data,d1,d2,d3,d4);
287
288   input [63:0] Mem_Addr;
289   input [63:0] W_Data;
290   input clk, MemWrite, MemRead;
291
292   output reg [63:0] Read_Data;
293   output reg [63:0] d1,d2,d3,d4;
294
295   reg [7:0] DMem [63:0];
296
297   initial
298     begin
299       DMem[0] = 8'b00000000;
300       DMem[1] = 8'b00000000;
301       DMem[2] = 8'b00000000;
302       DMem[3] = 8'b00000000;
303       DMem[4] = 8'b00000000;
304       DMem[5] = 8'b00000000;
305       DMem[6] = 8'b00000000;
306       DMem[7] = 8'b00000000;
307       DMem[8] = 8'b00000000;
308       DMem[9] = 8'b00000000;
309       DMem[10] = 8'b00000000;
310       DMem[11] = 8'b00000000;
311       DMem[12] = 8'b00000000;
312       DMem[13] = 8'b00000000;
313       DMem[14] = 8'b00000000;
314       DMem[15] = 8'b00000000;
315       DMem[16] = 8'b00000000;
316       DMem[17] = 8'b00000000;
317       DMem[18] = 8'b00000000;
318       DMem[19] = 8'b00000000;
319       DMem[20] = 8'b00000000;
320       DMem[21] = 8'b00000000;
321       DMem[22] = 8'b00000000;
322       DMem[23] = 8'b00000000;
323       DMem[24] = 8'b00000000;
324       DMem[25] = 8'b00000000;
325       DMem[26] = 8'b00000000;
326       DMem[27] = 8'b00000000;
327       DMem[28] = 8'b00000000;
328       DMem[29] = 8'b00000000;
```

```
329       DMem[30] = 8'b00000000;
330       DMem[31] = 8'b00000000;
331       DMem[32] = 8'b00000000;
332       DMem[33] = 8'b00000000;
333       DMem[34] = 8'b00000000;
334       DMem[35] = 8'b00000000;
335       DMem[36] = 8'b00000000;
336       DMem[37] = 8'b00000000;
337       DMem[38] = 8'b00000000;
338       DMem[39] = 8'b00000000;
339       DMem[40] = 8'b00000000;
340       DMem[41] = 8'b00000000;
341       DMem[42] = 8'b00000000;
342       DMem[43] = 8'b00000000;
343       DMem[44] = 8'b00000000;
344       DMem[45] = 8'b00000000;
345       DMem[46] = 8'b00000000;
346       DMem[47] = 8'b00000000;
347       DMem[48] = 8'b00000000;
348       DMem[49] = 8'b00000000;
349       DMem[50] = 8'b00000000;
350       DMem[51] = 8'b00000000;
351       DMem[52] = 8'b00000000;
352       DMem[53] = 8'b00000000;
353       DMem[54] = 8'b00000000;
354       DMem[55] = 8'b00000000;
355       DMem[56] = 8'b00000000;
356       DMem[57] = 8'b00000000;
357       DMem[58] = 8'b00000000;
358       DMem[59] = 8'b00000000;
359       DMem[60] = 8'b00000000;
360       DMem[61] = 8'b00000000;
361       DMem[62] = 8'b00000000;
362       DMem[63] = 8'b00000000;
363     end
364
365   always @ (posedge clk)
366
367     begin
368
369       if (MemWrite == 1'b1)
370         begin
371
372           DMem[Mem_Addr] = W_Data[7:0];
373           DMem[Mem_Addr + 1'b1] = W_Data[15:8];
374           DMem[Mem_Addr + 2'b10] = W_Data[23:16];
375           DMem[Mem_Addr + 2'b11] = W_Data[31:24];
376           DMem[Mem_Addr + 3'b100] = W_Data[39:32];
377           DMem[Mem_Addr + 3'b101] = W_Data[47:40];
```

```
          DMem[Mem_Addr + 3'b110] = W_Data[55:48];
          DMem[Mem_Addr + 3'b111] = W_Data[63:56];

        end

    end

  always @ (*)

    begin

      if (MemRead == 1'b1)

        begin

          Read_Data[7:0] = DMem[Mem_Addr];
          Read_Data[15:8] = DMem[Mem_Addr + 1'b1];
          Read_Data[23:16] = DMem[Mem_Addr + 2'b10];
          Read_Data[31:24] = DMem[Mem_Addr + 2'b11];
          Read_Data[39:32] = DMem[Mem_Addr + 3'b100];
          Read_Data[47:40] = DMem[Mem_Addr + 3'b101];
          Read_Data[55:48] = DMem[Mem_Addr + 3'b110];
          Read_Data[63:56] = DMem[Mem_Addr + 3'b111];

        end
    end
  always @ (*)
    begin
      d1 = DMem[0];
      d2 = DMem[8];
      d3 = DMem[16];
      d4 = DMem[24];

    end

endmodule


module Program_Counter (clk, reset, PC_In, PC_Out);

  input clk, reset;
  input [63:0] PC_In;

  output reg [63:0] PC_Out;


  always @ (posedge clk or posedge reset)
    begin

```

```
427       if (reset == 1'b1)
428         begin
429
430           PC_Out = 64'd0;
431         end
432
433       else
434         begin
435
436           PC_Out = PC_In;
437
438         end
439
440     end
441
442 endmodule
443
444
445
446
447 module Adder (a,b,out);
448
449   input [63:0] a,b;
450   output [63:0] out;
451
452   assign out = a + b;
453
454 endmodule
455
456
457 module Control_Unit (Opcode, funct3, BEQ, BLT, BGE, MemRead,
      MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite);
458
459   input [6:0] Opcode;
460   input [2:0] funct3;
461
462   output reg BEQ, BLT, BGE, MemRead, MemtoReg, MemWrite,
      ALUSrc, RegWrite;
463   output reg [1:0] ALUOp;
464
465   always @ (*)
466     begin
467
468       case(Opcode)
469
470         7'b0110011: begin           // R-Type
471           ALUSrc = 1'b0;
472           MemtoReg = 1'b0;
473           RegWrite = 1'b1;
```

```
474           MemRead = 1'b0;
475           MemWrite = 1'b0;
476           BEQ = 1'b0;
477           BLT = 1'b0;
478           BGE = 1'b0;
479           ALUOp = 2'b10;
480         end
481
482         7'b0000011: begin           // I-Type (Load)
483           ALUSrc = 1'b1;
484           MemtoReg = 1'b1;
485           RegWrite = 1'b1;
486           MemRead = 1'b1;
487           MemWrite = 1'b0;
488           BEQ = 1'b0;
489           BLT = 1'b0;
490           BGE = 1'b0;
491           ALUOp = 2'b00;
492         end
493
494         7'b0100011: begin          // S-Type
495           ALUSrc = 1'b1;
496           MemtoReg = 1'bx;
497           RegWrite = 1'b0;
498           MemRead = 1'b0;
499           MemWrite = 1'b1;
500           BEQ = 1'b0;
501           BLT = 1'b0;
502           BGE = 1'b0;
503           ALUOp = 2'b00;
504         end
505
506         7'b1100011: begin          // B-Type (BEQ)
507
508           case(funct3)
509
510             3'b000: begin
511               ALUSrc = 1'b0;
512               MemtoReg = 1'bx;
513               RegWrite = 1'b0;
514               MemRead = 1'b0;
515               MemWrite = 1'b0;
516               BEQ = 1'b1;
517               BLT = 1'b0;
518               BGE = 1'b0;
519               ALUOp = 2'b01;
520             end
521
522             3'b100: begin
```

```
523                 ALUSrc = 1'b0;
524                 MemtoReg = 1'bx;
525                 RegWrite = 1'b0;
526                 MemRead = 1'b0;
527                 MemWrite = 1'b0;
528                 BEQ = 1'b0;
529                 BLT = 1'b1;
530                 BGE = 1'b0;
531                 ALUOp = 2'b01;
532               end
533
534               3'b101: begin
535                 ALUSrc = 1'b0;
536                 MemtoReg = 1'bx;
537                 RegWrite = 1'b0;
538                 MemRead = 1'b0;
539                 MemWrite = 1'b0;
540                 BEQ = 1'b0;
541                 BLT = 1'b0;
542                 BGE = 1'b1;
543                 ALUOp = 2'b01;
544               end
545             endcase
546
547           end
548
549           7'b0010011: begin          // I-Type (ADDI)
550             ALUSrc = 1'b1;
551             MemtoReg = 1'b0;
552             RegWrite = 1'b1;
553             MemRead = 1'b0;
554             MemWrite = 1'b0;
555             BEQ = 1'b0;
556             BLT = 1'b0;
557             BGE = 1'b0;
558             ALUOp = 2'b00;
559           end
560
561           default: begin            // Default
562             ALUSrc = 1'b0;
563             MemtoReg = 1'b0;
564             RegWrite = 1'b0;
565             MemRead = 1'b0;
566             MemWrite = 1'b0;
567             BEQ = 1'b0;
568             BLT = 1'b0;
569             BGE = 1'b0;
570             ALUOp = 2'b00;
571           end
```

```
572
573       endcase
574
575     end
576
577 endmodule
578
579
580 module ALU_Control (ALUOp, Funct, Operation);
581
582   input [1:0] ALUOp;
583   input [3:0] Funct;
584
585   output reg [3:0] Operation;
586
587   always @ (*)
588     begin
589
590       case(ALUOp)
591
592         2'b00: Operation = 4'b0010;    // I/S-Type
593
594         2'b01: Operation = 4'b0110;    // SB-Type (BEQ)
595
596         2'b10: begin                   // R-Type
597
598           case(Funct)
599
600             4'b0000: Operation = 4'b0010;
601
602             4'b1000: Operation = 4'b0110;
603
604             4'b0111: Operation = 4'b0000;
605
606             4'b0110: Operation = 4'b0001;
607
608             default: Operation = 4'b0000;
609
610           endcase
611
612         end
613
614         default: Operation = 4'b0000;
615
616       endcase
617
618     end
619
620 endmodule
```

```
621
622
623 module RISC_V_Processor (clk, reset);
624
625   input clk, reset;
626
627   wire [63:0] PC_Out;
628   wire [63:0] out;
629   wire [31:0] Instruction;
630   wire [6:0] opcode;
631   wire [4:0] rd;
632   wire [2:0] funct3;
633   wire [4:0] rs1;
634   wire [4:0] rs2;
635   wire [6:0] funct7;
636   wire [63:0] imm_data;
637   wire [63:0] rd1;
638   wire [63:0] rd2;
639   wire BEQ, BLT, BGE, MemRead, MemtoReg, MemWrite, ALUSrc,
       RegWrite;
640   wire [1:0] ALUOp;
641   wire [63:0] out_M1;
642   wire [3:0] Operation;
643   wire [63:0] result;
644   wire zero;
645   wire sign;
646   wire [63:0] out_A2;
647   wire [63:0] out_M2;
648   wire [63:0] out_DM;
649   wire [63:0] wtData;
650   wire [63:0] d1,d2,d3,d4;
651   wire [63:0] r1,r2,r3,r4,r22,r23,r20,r21,r19,r18;
652
653   Program_Counter PC1(.clk(clk), .reset(reset), .PC_In(out_M2
       ), .PC_Out(PC_Out) );
654
655
656   Adder A1 (.a(PC_Out), .b(64'd4), .out(out) );
657
658
659   Instruction_Memory I1(.Instr_Add(PC_Out), .Instruction(
       Instruction) );
660
661   Parser P1( .ins(Instruction), .opcode(opcode), .rd(rd), .
       funct3(funct3), .rs1(rs1), .rs2(rs2), .funct7(funct7) );
662
663   Control_Unit C1(.Opcode(opcode), .funct3(funct3), .BEQ(BEQ)
       , .BLT(BLT), .BGE(BGE), .MemRead(MemRead), .MemtoReg(
       MemtoReg), .ALUOp(ALUOp), .MemWrite(MemWrite), .ALUSrc(
```

```
    ALUSrc), .RegWrite(RegWrite));
664
665   ImmGen G1 (.ins(Instruction), .imm_data(imm_data) );
666
667   registerFile R1( .clk(clk), .reset(reset), .wtData(wtData),
        .rs1(rs1), .rs2(rs2), .rd(rd), .regWrite(RegWrite), .rd1(
      rd1), .rd2(rd2) ,.r1(r1),.r2(r2),.r3(r3),.r4(r4),.r22(r22)
      ,.r23(r23),.r20(r20),.r21(r21),.r19(r19),.r18(r18));
668
669   MUX M1(.A(rd2), .B(imm_data), .sel(ALUSrc), .out(out_M1) );
670
671   ALU_Control C2( .ALUOp(ALUOp), .Funct({Instruction[30],
      funct3}), .Operation(Operation) );
672
673   Adder A2 (.a(PC_Out), .b(imm_data << 1), .out(out_A2) );
674
675   ALU_64 AL( .a(rd1), .b(out_M1), .ALUop(Operation), .result(
      result), .zero(zero), .sign(sign) );
676
677   MUX M2 (.A(out), .B(out_A2), .sel((zero & BEQ) || (sign &
      BLT) || (~sign & BGE)), .out(out_M2));
678
679
680   Data_Memory D1(.Mem_Addr(result), .W_Data(rd2), .clk(clk),
      .MemWrite(MemWrite), .MemRead(MemRead), .Read_Data(out_DM)
      ,.d1(d1),.d2(d2),.d3(d3),.d4(d4));
681
682   MUX M3 (.A(result), .B(out_DM), .sel(MemtoReg), .out(wtData
      ) );
683
684
685 endmodule
```

Test Branch

```
1  module tb();
2
3    reg clk, reset;
4
5    RISC_V_Processor RISCV(.clk(clk), .reset(reset) );
6
7    initial
8      begin
9
10       $dumpfile("dump.vcd");
11       $dumpvars();
12
13       reset = 1'b1;
```

```
14        clk = 1'b0;
15
16        #2
17
18        reset = 1'b0;
19
20      end
21
22    always
23      begin
24        #6
25        clk = ~clk;
26      end
27
28  endmodule
```

2.3 Results

As can be seen from the waveform generated, the final values in d1,d2,d3 and d4 are 3, 5, 8 and 9 respectively. Initially, the first element of the unsorted array, 8, was stored in d1. Upon performing insertion sort, this value got swapped with the value in d2 which was 5. Now d1 = 5 and d2 = 8. This swapping continued until all the elements in the array were sorted in ascending order. The waveform given below shows the elements being swapped and sorted at each time instant.

Simulation results can also be viewed at: `https://www.edaplayground.com/x/sBdk`

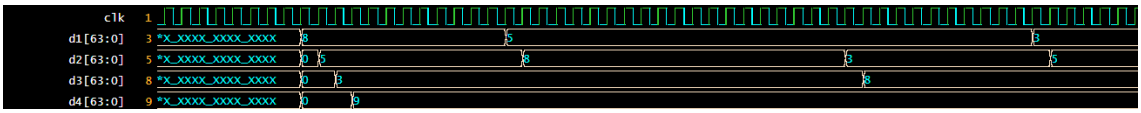


Figure 2.3: EP Wave for Sorting

Task2

3.1 Objectives

The objective of this task is to convert our Single Cycle Processor into a 5 Stage Pipelined Processor.

- Create registers
- Create Control Block
- integration
- Create test bench
- ensure if the algorithm works

3.2 Creating Pipeline Registers

To make a pipelined processor, we need to implement 4 intermediate registers, namely IF/ID, ID/EX, EX/MEM and MEM/WB. Their placement in the processor architecture can be seen below.

3.2.1 IF/ID

The instruction is read from the memory using PC address and is then placed in the IF/ID pipeline register. The PC address gets incremented by 4 and is then back into the PC for the next clock cycle. This PC is also stored in the IF/ID pipeline register for later use if needed, such as for beq instructions. This is because the processor does not know which instruction is being fetched and so any information that has the potential to be needed later is passed down the pipeline. The instruction portion of the IF/ID pipeline register supplies the immediate field (64 bit sign extended), and the registers numbers needed to read the two registers.

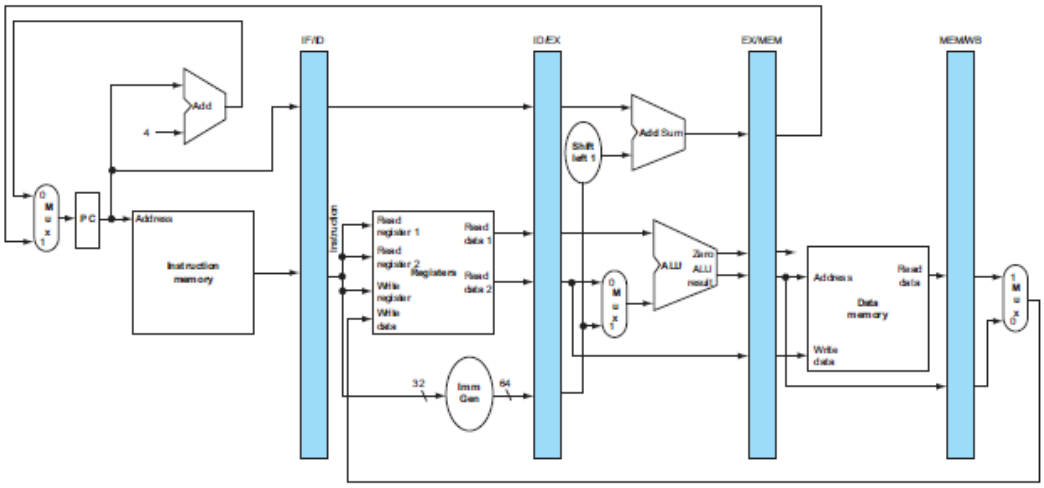


Figure 3.1: 5 Stage Pipelined Processor

```
1
2  module Reg_IF_ID (clk, reset, PC_In, ins, PC_Out, ins_out);
3
4    input clk, reset;
5    input [63:0] PC_In;
6    input [31:0] ins;
7
8    output reg [63:0] PC_Out;
9    output reg [31:0] ins_out;
10
11   always @ (posedge clk or posedge reset)
12     begin
13
14       if (reset == 1'b1)
15         begin
16
17           PC_Out = 64'd0;
18           ins_out = 32'd0;
19
20         end
21
22       else
23         begin
24
25           PC_Out = PC_In;
26           ins_out = ins;
27
28         end
29     end
```

```
endmodule
```

3.2.2 ID/EX

All three values from IF/ID are stored in the ID/EX pipeline register, along with the PC address. If reset is enabled, the output values are all set to 0, else the same input values are passed as output.

```


module Reg_ID_EX (clk, reset, WB_In, WB_Out, M_In, M_Out,
    EX_In, EX_Out, PC_In, PC_Out, RD1_In, RD1_Out, RD2_In,
    RD2_Out, Imm_In, Imm_Out, Funct_In, Funct_Out, Wr_Reg_In,
    Wr_Reg_Out, Rs1_In, Rs1_Out, Rs2_In, Rs2_Out);

  input clk, reset;
  input [63:0] PC_In, RD1_In, RD2_In, Imm_In;
  input [3:0] Funct_In;
  input [4:0] Wr_Reg_In, Rs1_In, Rs2_In;
  input [1:0] WB_In;
  input [4:0] M_In;
  input [2:0] EX_In;

  output reg [63:0] PC_Out, RD1_Out, RD2_Out, Imm_Out;
  output reg [3:0] Funct_Out;
  output reg [4:0] Wr_Reg_Out, Rs1_Out, Rs2_Out;
  output reg [1:0] WB_Out;
  output reg [4:0] M_Out;
  output reg [2:0] EX_Out;

  always @ (posedge clk or posedge reset)
    begin

      if (reset == 1'b1)
        begin

          PC_Out = 64'd0;
          RD1_Out = 64'd0;
          RD2_Out = 64'd0;
          Imm_Out = 64'd0;
          Funct_Out = 4'd0;
          Wr_Reg_Out = 5'd0;
          Rs1_Out = 5'd0;
          Rs2_Out = 5'd0;
          WB_Out = 2'd0;
          M_Out = 5'd0;
          EX_Out = 3'd0;

```

```
        end

      else
        begin

          PC_Out = PC_In;
          RD1_Out = RD1_In;
          RD2_Out = RD2_In;
          Imm_Out = Imm_In;
          Funct_Out = Funct_In;
          Wr_Reg_Out = Wr_Reg_In;
          Rs1_Out = Rs1_In;
          Rs2_Out = Rs2_In;
          WB_Out = WB_In;
          M_Out = M_In;
          EX_Out = EX_In;

        end

    end

endmodule
```

3.2.3 EX/MEM

The effective address is placed in the EX/MEM pipeline register.

```
module Reg_EX_MEM (clk, reset, WB_In, WB_Out, M_In, M_Out,
    PC_In, PC_Out, Zero_In, Zero_Out, Result_In, Result_Out,
    RD2_In, RD2_Out, Wr_Reg_In, Wr_Reg_Out, Sign_In, Sign_Out)
    ;

  input clk, reset, Sign_In;
  input [1:0] WB_In;
  input [4:0] M_In;
  input [63:0] PC_In;
  input Zero_In;
  input [63:0] Result_In;
  input [63:0] RD2_In;
  input [4:0] Wr_Reg_In;

  output reg [1:0] WB_Out;
  output reg [4:0] M_Out;
  output reg [63:0] PC_Out;
  output reg Zero_Out, Sign_Out;
  output reg [63:0] Result_Out;
  output reg [63:0] RD2_Out;
  output reg [4:0] Wr_Reg_Out;
```

```
19
20  always @ (posedge clk or posedge reset)
21    begin
22
23      if (reset == 1'b1)
24        begin
25
26          WB_Out = 2'd0;
27          M_Out = 5'd0;
28          PC_Out = 64'd0;
29          Zero_Out = 1'b0;
30          Result_Out = 64'd0;
31          RD2_Out = 64'd0;
32          Wr_Reg_Out = 5'd0;
33          Sign_Out = 1'b0;
34
35        end
36
37      else
38        begin
39
40          WB_Out = WB_In;
41          M_Out = M_In;
42          PC_Out = PC_In;
43          Zero_Out = Zero_In;
44          Result_Out = Result_In;
45          RD2_Out = RD2_In;
46          Wr_Reg_Out = Wr_Reg_In;
47          Sign_Out = Sign_In;
48
49        end
50    end
51 endmodule
```

3.2.4 MEM/WB

The register containing the data to be stored was read in an earlier stage and stored in ID/EX. The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address into EX/MEM.

```
1 module Reg_MEM_WB (clk, reset, WB_In, WB_Out, RD_In, RD_Out,
     Result_In, Result_Out, Wr_Reg_In, Wr_Reg_Out);
2
3   input clk, reset;
4   input [1:0] WB_In;
5   input [63:0] RD_In;
6   input [63:0] Result_In;
```

```
7   input [4:0] Wr_Reg_In;
8
9   output reg [1:0] WB_Out;
10  output reg [63:0] RD_Out;
11  output reg [63:0] Result_Out;
12  output reg [4:0] Wr_Reg_Out;
13
14  always @ (posedge clk or posedge reset)
15    begin
16
17      if (reset == 1'b1)
18        begin
19
20          WB_Out = 2'd0;
21          RD_Out = 64'd0;
22          Result_Out = 64'd0;
23          Wr_Reg_Out = 5'd0;
24
25        end
26
27      else
28        begin
29
30          WB_Out = WB_In;
31          RD_Out = RD_In;
32          Result_Out = Result_In;
33          Wr_Reg_Out = Wr_Reg_In;
34
35        end
36
37    end
38
39  endmodule
```

3.3 Forwarding Unit

The forwarding control will be in the EX stage, because the ALU forwarding multiplexors are found in that stage. Thus, we must pass the operand register numbers from the ID stage via the ID/EX pipeline register to determine whether to forward values. Before forwarding, the ID/EX register had no need to include space to hold the rs1 and rs2 fields. Hence, they were added to ID/EX. Forwarding is based on the following table. It indicates from where the first and second ALU operand are forwarded from. Forward A is the control output signal for the first operand whereas Forward B is for the second operand.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Figure 3.2: Control Values for Forwarding Multiplexers

3.3.1 Modules for Forward A and B

```
1
2  module Forward_A (rd1, wtData, prev_result, sel_A, out_FwA);
3
4    input [63:0] rd1, wtData, prev_result;
5    input [1:0] sel_A;
6
7    output reg [63:0] out_FwA;
8
9    always @ (*)
10     begin
11
12       case(sel_A)
13
14         2'b00: out_FwA = rd1;
15         2'b01: out_FwA = wtData;
16         2'b10: out_FwA = prev_result;
17
18         default: out_FwA = 64'd0;
19
20       endcase
21
22     end
23
24 endmodule
```

```
25
26
27 module Forward_B (rd2, wtData, prev_result, sel_B, out_FwB);
28
29   input [63:0] rd2, wtData, prev_result;
30   input [1:0] sel_B;
31
32   output reg [63:0] out_FwB;
33
34   always @ (*)
35     begin
36
37       case(sel_B)
38
39         2'b00: out_FwB = rd2;
40         2'b01: out_FwB = wtData;
41         2'b10: out_FwB = prev_result;
42
43         default: out_FwB = 64'd0;
44
45       endcase
46
47     end
48
49 endmodule
```

3.3.2 Implementing the Forwarding Unit

Conditions for detecting data hazards and resolving them

```
1 if (EX/MEM.RegWrite
2 and (EX/MEM.RegisterRd     0)
3 and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 10
4 if (EX/MEM.RegWrite
5 and (EX/MEM.RegisterRd     0)
6 and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
```

This case forwards the result from the previous instruction to either input of the ALU. If the previous instruction is going to write to the register file, and the write register number matches the read register number of ALU inputs A or B, provided it is not register 0, then steer the multiplexor to pick the value instead from the pipeline register EX/MEM.

```
1
2 if (MEM/WB.RegWrite and (MEM/WB.RegisterRd     0) and not(EX/
    MEM.RegWrite and (EX/MEM.RegisterRd     0) and (EX/MEM.
    RegisterRd = ID/EX.RegisterRs1)) and (MEM/WB.RegisterRd =
    ID/EX.RegisterRs1))
```

```
    ForwardA = 01
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd     0) and not(EX/
   MEM.RegWrite and (EX/MEM.RegisterRd     0) and (EX/MEM.
   RegisterRd = ID/EX.RegisterRs2)) and (MEM/WB.RegisterRd =
   ID/EX.RegisterRs2))
    ForwardB = 01
```

Based on these forwarding conditions, we developed our own forwarding unit.

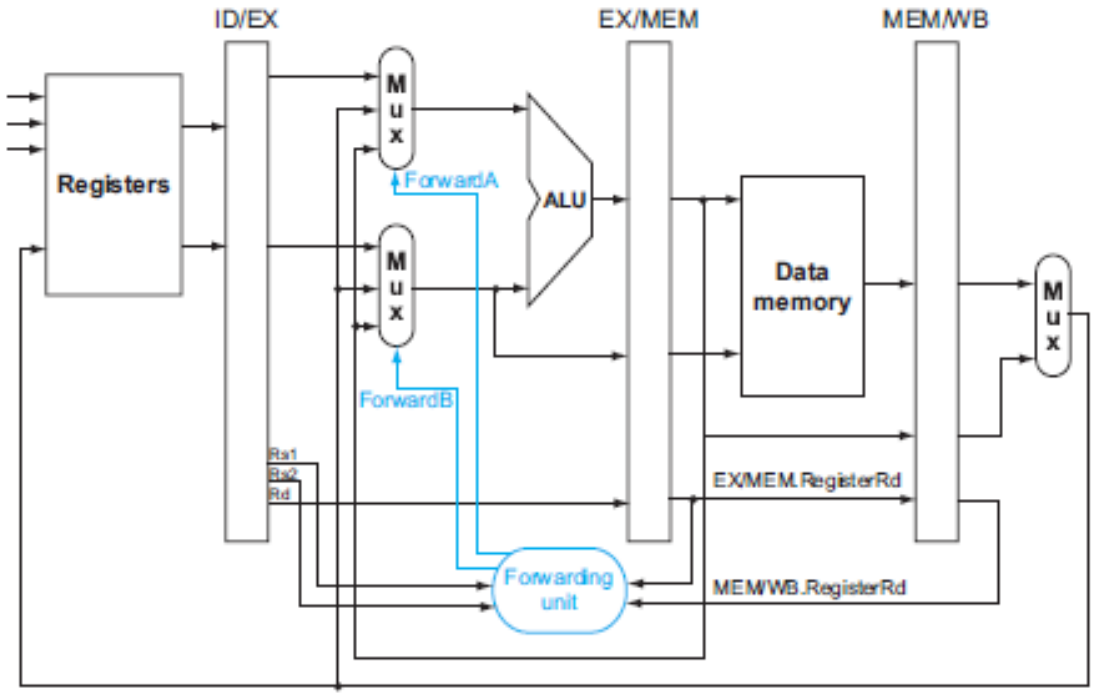


Figure 3.3: Forwarding Unit Added to the Architecture

```

module Forwarding_Unit (rs1, rs2, EM_rd, MW_rd, EM_regwt,
   MW_regwt, sel_A, sel_B);

  input [4:0] rs1, rs2, EM_rd, MW_rd;
  input EM_regwt, MW_regwt;

  output reg [1:0] sel_A, sel_B;

  always @ (*)
    begin

```

```
13      if (EM_regwt == 1'b1 && EM_rd != 4'b0000)
14        begin
15
16          if (EM_rd == rs1)
17            begin
18              sel_A = 2'b10;
19            end
20
21          else if (EM_rd == rs2)
22            begin
23              sel_B = 2'b10;
24            end
25        end
26
27
28      else if (MW_regwt == 1'b1 && MW_rd != 4'b0000)
29        begin
30
31          if (EM_rd != rs1 && MW_rd == rs1)
32            begin
33              sel_A = 2'b01;
34            end
35
36          else if (EM_rd != rs2 && MW_rd == rs2)
37            begin
38              sel_B = 2'b01;
39            end
40        end
41
42      else
43        begin
44          sel_A = 2'b00;
45          sel_B = 2'b00;
46        end
47
48    end
49
50  endmodule
```

3.4 Integrating Register Modules and Forwarding Unit

```
1  module RISC_V_Processor (clk, reset);
2
3    input clk, reset;
4
5    wire [63:0] PC_Out;
6    wire [63:0] out;
```

```
7   wire [31:0] Instruction;
8   wire [6:0] opcode;
9   wire [4:0] rd;
10  wire [2:0] funct3;
11  wire [4:0] rs1;
12  wire [4:0] rs2;
13  wire [6:0] funct7;
14  wire [63:0] imm_data;
15  wire [63:0] rd1;
16  wire [63:0] rd2;
17  wire BEQ, BLT, BGE, MemRead, MemtoReg, MemWrite, ALUSrc,
     RegWrite;
18  wire [1:0] ALUOp;
19  wire [63:0] out_M1;
20  wire [3:0] Operation;
21  wire [63:0] result;
22  wire zero;
23  wire sign;
24  wire [63:0] out_A2;
25  wire [63:0] out_M2;
26  wire [63:0] out_DM;
27  wire [63:0] out_FwA, out_FwB;
28  wire [63:0] wtData;
29  wire [63:0] d1,d2,d3,d4;
30  wire [63:0] r1,r2,r3,r4,r22,r23,r20,r21,r19,r18;
31
32  wire [63:0] REG1_PCout;
33  wire [31:0] REG1_INSout;
34  wire [1:0] WB_Out, sel_A, sel_B;
35  wire [4:0] M_Out;
36  wire [2:0] EX_Out;
37  wire [63:0] REG2_PCout;
38  wire [63:0] REG2_RD1out;
39  wire [63:0] REG2_RD2out;
40  wire [63:0] Imm_Out;
41  wire [3:0] Funct_Out;
42  wire [4:0] REG2_Wr_Reg_Out, Rs1_Out, Rs2_Out;
43  wire [1:0] WBout_S3;
44  wire [4:0] Mout_S3;
45  wire [63:0] PCout_S3;
46  wire Zero_Out, sign_out;
47  wire [63:0] Result_Out;
48  wire [63:0] RD2out_S3;
49  wire [4:0] RegOut_S3;
50  wire [1:0] WBout_S4;
51  wire [63:0] RDout_S4;
52  wire [63:0] Resultout_S4;
53  wire [4:0] RegOut_S4;
54
```

```
55  Program_Counter PC1(.clk(clk), .reset(reset), .PC_In(out_M2
     ), .PC_Out(PC_Out) );
56
57
58  Adder A1 (.a(PC_Out), .b(64'd4), .out(out) );
59
60
61  Instruction_Memory I1(.Instr_Add(PC_Out), .Instruction(
     Instruction) );
62
63  Reg_IF_ID S1 (.clk(clk), .reset(reset), .PC_In(PC_Out), .
     ins(Instruction), .PC_Out(REG1_PCout), .ins_out(
     REG1_INSout) );
64
65  Parser P1( .ins(REG1_INSout), .opcode(opcode), .rd(rd), .
     funct3(funct3), .rs1(rs1), .rs2(rs2), .funct7(funct7) );
66
67  Control_Unit C1(.Opcode(opcode), .funct3(funct3), .BEQ(BEQ)
     , .BLT(BLT), .BGE(BGE), .MemRead(MemRead), .MemtoReg(
     MemtoReg), .ALUOp(ALUOp), .MemWrite(MemWrite), .ALUSrc(
     ALUSrc), .RegWrite(RegWrite));
68
69  ImmGen G1 (.ins(REG1_INSout), .imm_data(imm_data) );
70
71  registerFile R1( .clk(clk), .reset(reset), .wtData(wtData),
       .rs1(rs1), .rs2(rs2), .rd(RegOut_S4), .regWrite(WBout_S4
     [1]), .rd1(rd1), .rd2(rd2) ,.r1(r1),.r2(r2),.r3(r3),.r4(r4
     ),.r22(r22),.r23(r23),.r20(r20),.r21(r21),.r19(r19),.r18(
     r18));
72
73  Reg_ID_EX S2 (.clk(clk), .reset(reset), .WB_In({RegWrite,
     MemtoReg}), .WB_Out(WB_Out), .M_In({BEQ, BLT, BGE, MemRead
     , MemWrite}), .M_Out(M_Out), .EX_In({ALUOp, ALUSrc}), .
     EX_Out(EX_Out), .PC_In(REG1_PCout), .PC_Out(REG2_PCout), .
     RD1_In(rd1), .RD1_Out(REG2_RD1out), .RD2_In(rd2), .RD2_Out
     (REG2_RD2out), .Imm_In(imm_data), .Imm_Out(Imm_Out), .
     Funct_In({REG1_INSout[30], funct3}), .Funct_Out(Funct_Out)
     , .Wr_Reg_In(rd), .Wr_Reg_Out(REG2_Wr_Reg_Out), .Rs1_In(
     rs1), .Rs1_Out(Rs1_Out), .Rs2_In(rs2), .Rs2_Out(Rs2_Out) )
     ;
74
75
76  Forwarding_Unit FU (.rs1(Rs1_Out), .rs2(Rs2_Out), .EM_rd(
     RegOut_S3), .MW_rd(RegOut_S4), .EM_regwt(WB_Out[1]), .
     MW_regwt(WBout_S4[1]), .sel_A(sel_A), .sel_B(sel_B) );
77
78  Forward_A FA (.rd1(REG2_RD1out), .wtData(wtData), .
     prev_result(Result_Out), .sel_A(sel_A), .out_FwA(out_FwA)
     );
```

```
79
80    Forward_B FB (.rd2(REG2_RD2out), .wtData(wtData), .
       prev_result(Result_Out), .sel_B(sel_B), .out_FwB(out_FwB)
       );
81
82    MUX M1(.A(out_FwB), .B(Imm_Out), .sel(EX_Out[0]), .out(
       out_M1));
83
84    ALU_Control C2( .ALUOp(EX_Out[2:1]), .Funct(Funct_Out), .
       Operation(Operation) );
85
86    Adder A2 (.a(REG2_PCout), .b(Imm_Out << 1), .out(out_A2) );
87
88    ALU_64 AL( .a(out_FwA), .b(out_M1), .ALUop(Operation), .
       result(result), .zero(zero), .sign(sign) );
89
90    Reg_EX_MEM S3 (.clk(clk), .reset(reset), .WB_In(WB_Out), .
       WB_Out(WBout_S3), .M_In(M_Out), .M_Out(Mout_S3), .PC_In(
       out_A2), .PC_Out(PCout_S3), .Zero_In(zero), .Zero_Out(
       Zero_Out), .Result_In(result), .Result_Out(Result_Out), .
       RD2_In(REG2_RD2out), .RD2_Out(RD2out_S3), .Wr_Reg_In(
       REG2_Wr_Reg_Out), .Wr_Reg_Out(RegOut_S3), .Sign_In(sign),
       .Sign_Out(sign_out) );
91
92    MUX M2 (.A(out), .B(PCout_S3), .sel((Zero_Out & Mout_S3[4])
        || (sign_out & Mout_S3[3]) || (~sign_out & Mout_S3[2])),
       .out(out_M2));
93
94
95    Data_Memory D1(.Mem_Addr(Result_Out), .W_Data(RD2out_S3), .
       clk(clk), .MemWrite(Mout_S3[0]), .MemRead(Mout_S3[1]), .
       Read_Data(out_DM),.d1(d1),.d2(d2),.d3(d3),.d4(d4));
96
97    Reg_MEM_WB S4 (.clk(clk), .reset(reset), .WB_In(WBout_S3),
       .WB_Out(WBout_S4), .RD_In(out_DM), .RD_Out(RDout_S4), .
       Result_In(Result_Out), .Result_Out(Resultout_S4), .
       Wr_Reg_In(RegOut_S3), .Wr_Reg_Out(RegOut_S4) );
98
99    MUX M3 (.A(Resultout_S4), .B(RDout_S4), .sel(WBout_S4[0]),
       .out(wtData) );
100
101
102 endmodule
```

3.4.1 Testing

```
1 module tb();
```

```
2
3   reg clk, reset;
4
5   RISC_V_Processor RISCV(.clk(clk), .reset(reset) );
6
7   initial
8     begin
9
10      $dumpfile("dump.vcd");
11      $dumpvars();
12
13      reset = 1'b1;
14      clk = 1'b0;
15
16      #2
17
18      reset = 1'b0;
19
20    end
21
22  always
23    begin
24      #5
25      clk = ~clk;
26    end
27
28
29 endmodule
```

3.5 Results

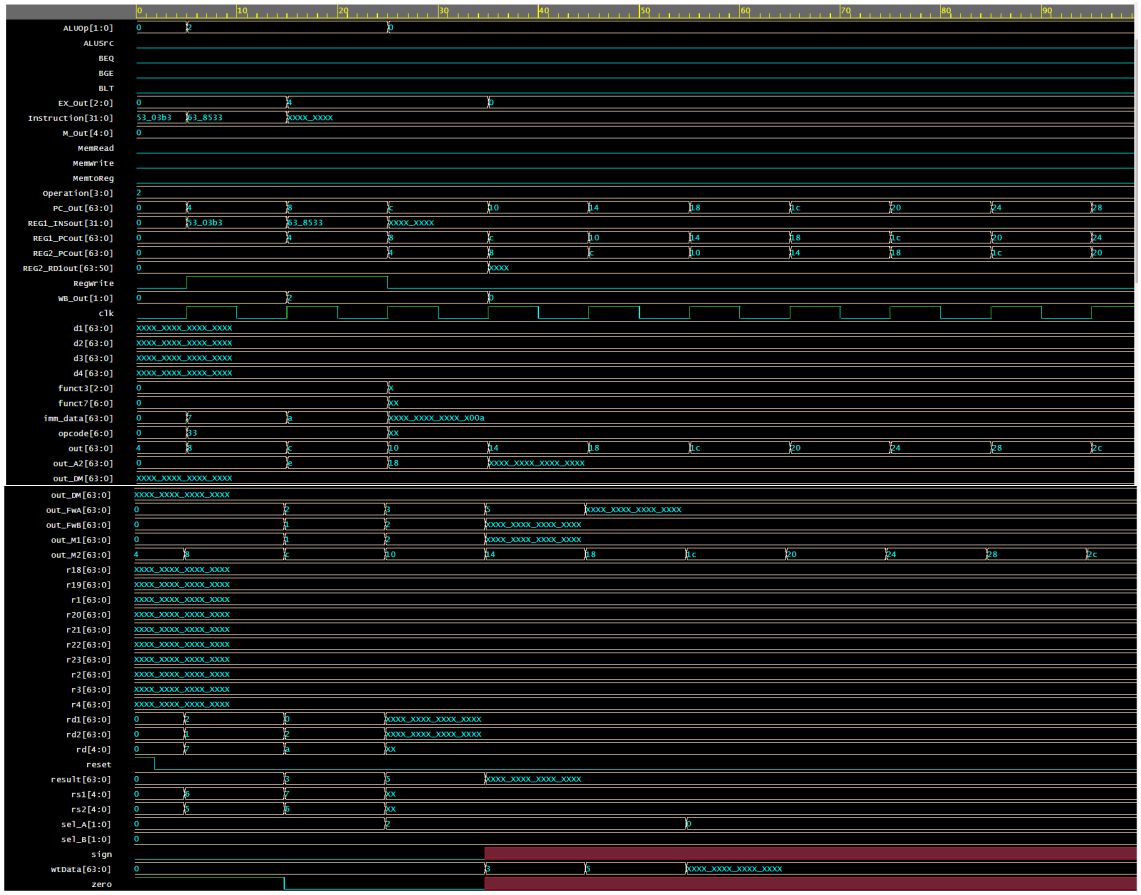


Figure 3.4: Simulation Results

Simulation results can also be viewed at: `https://www.edaplayground.com/x/MY3s`

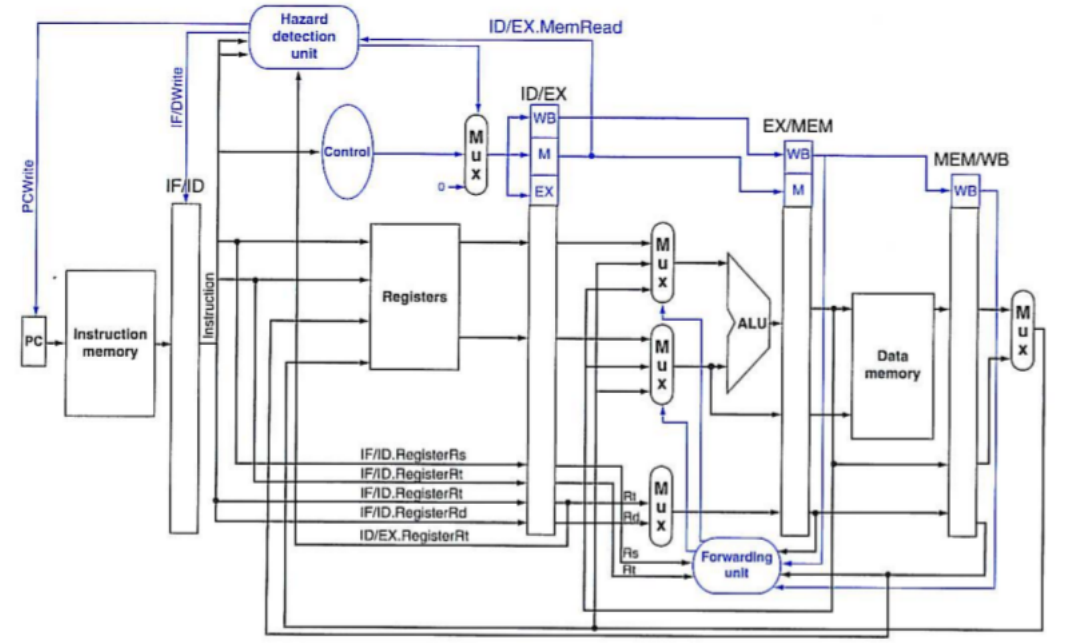
Instructions:
ADD x7, x6, x5
ADD x10, x7, x6

x6 and x5 holds 2 and 1 values respectively. We can see that the result is 3 and then 5, which is correct and as expected. x7 will hold the value 2+1 = 3. Then, x10 will be assigned 3+2 = 5. It means that forwarding unit works fine!

Task3

4.1 Objectives

- Optimize insertion sort code to remove data hazards
- Making stall hazard block
- Making Flushing hazard block
- ensure if the algorithm works



IGURE 4.60 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and ıe forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—ıis drawing gives the essence of the forwarding hardware requirements.

Figure 4.1: Architecture for Hazard detection

4.2 Hazard Detection (Stalling)

Forwarding cannot be used when an instruction tries to read a register following a load instruction that writes the same register. There should be a mechanism that must stall the pipeline for the combination of load followed by an instruction that reads its result. Hence, in addition to a forwarding unit, we need a hazard detection unit. It operates during the ID stage so that it can insert the stall between the load and the instruction dependent on it. Checking for load instructions, the control for the hazard detection unit is this condition:

```
1 if (ID/EX.MemRead and ((ID/EX.RegisterRd = IF/ID.RegisterRs1)
      or (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
2 stall the pipeline
```

Stalling the pipeline sets all seven control signals to 0 and doesn't change the program counter and IF-ID pipelined register, it acts like an nop instruction.

4.2.1 Implementation of Hazard Detection Block

```
module Hazard_detection_block(rst,clk, ID_EX_memRead,
   Instructions,ID_EX_registerRt,IF_ID_Write,PC_Write,
   MUX_Control_zero);
  input clk,rst;
  input ID_EX_memRead;
  input [31:0] Instructions;
  input [4:0] ID_EX_registerRt;
  output reg IF_ID_Write;
  output reg PC_Write;
  output reg MUX_Control_zero;

  always @(*)
    begin
      if((ID_EX_memRead) & ((ID_EX_registerRt == Instructions
   [19:15]) | (ID_EX_registerRt == Instructions[24:20])))
        begin

          IF_ID_Write= 0;
          PC_Write=0 ;
          MUX_Control_zero=1;
     end else
         begin
           IF_ID_Write = 1;
           PC_Write =1'b1 ;
           MUX_Control_zero =0;
       end

         end


endmodule
```

4.2.2 Flushing

We also introduced an additional block to perform flushing. If a branch is detected, then the register contents are set to 0 for the stages that were preloaded before branch was fully executed.

```

module Hazard_flush(clk,rst,M_EX_MEM,flush_If_ID_register,
   flush_ID_EX_register,flush_EX_MEM_register);
input clk,rst;
input M_EX_MEM;
 output reg flush_If_ID_register;
  output reg flush_ID_EX_register;
  output reg flush_EX_MEM_register;
```

```
  always @(posedge clk, posedge rst)
  begin
  if (M_EX_MEM)
    begin

    flush_If_ID_register = 1'b1;
    flush_ID_EX_register = 1'b1;
    flush_EX_MEM_register= 1'b1;
    end
  else
    begin
    flush_If_ID_register = 1'b0;
    flush_ID_EX_register = 1'b0;
    flush_EX_MEM_register= 1'b0;
    end
  end


endmodule
```

To implement flush, we had to make changes to our pipeline registers.

IF-ID Register

```
module Reg_IF_ID (flush_If_ID_register,IF_ID_Write,clk, reset
   , PC_In, ins, PC_Out, ins_out);

  input clk, reset;
  input [63:0] PC_In;
  input [31:0] ins;
  input IF_ID_Write,flush_If_ID_register;

  output reg [63:0] PC_Out;
  output reg [31:0] ins_out;

  always @ (posedge clk or posedge reset)
    begin

      if (reset == 1'b1 | flush_If_ID_register==1)
        begin

          PC_Out = 64'd0;
          ins_out = 32'd0;

        end

      else
        begin
          if(IF_ID_Write)
```

```
            begin
          PC_Out = PC_In;
          ins_out = ins;
            end
        end
    end
endmodule
```

ID-EX

```

module Reg_ID_EX (flush_ID_EX_register,MUX_Control_zero,clk,
    reset, WB_In, WB_Out, M_In, M_Out, EX_In, EX_Out, PC_In,
    PC_Out, RD1_In, RD1_Out, RD2_In, RD2_Out, Imm_In, Imm_Out,
     Funct_In, Funct_Out, Wr_Reg_In, Wr_Reg_Out, Rs1_In,
    Rs1_Out, Rs2_In, Rs2_Out);

  input clk, reset;
  input [63:0] PC_In, RD1_In, RD2_In, Imm_In;
  input [3:0] Funct_In;
  input [4:0] Wr_Reg_In, Rs1_In, Rs2_In;
  input [1:0] WB_In;
  input [4:0] M_In;
  input [2:0] EX_In;
  input MUX_Control_zero,flush_ID_EX_register;

  output reg [63:0] PC_Out, RD1_Out, RD2_Out, Imm_Out;
  output reg [3:0] Funct_Out;
  output reg [4:0] Wr_Reg_Out, Rs1_Out, Rs2_Out;
  output reg [1:0] WB_Out;
  output reg [4:0] M_Out;
  output reg [2:0] EX_Out;

  always @ (posedge clk or posedge reset)
    begin

      if (reset == 1'b1 | MUX_Control_zero == 1|
    flush_ID_EX_register==1)
        begin

          PC_Out = 64'd0;
          RD1_Out = 64'd0;
          RD2_Out = 64'd0;
          Imm_Out = 64'd0;
          Funct_Out = 4'd0;
          Wr_Reg_Out = 5'd0;
          Rs1_Out = 5'd0;
          Rs2_Out = 5'd0;
```

```
          WB_Out = 2'd0;
          M_Out = 5'd0;
          EX_Out = 3'd0;

        end

      else
        begin

          PC_Out = PC_In;
          RD1_Out = RD1_In;
          RD2_Out = RD2_In;
          Imm_Out = Imm_In;
          Funct_Out = Funct_In;
          Wr_Reg_Out = Wr_Reg_In;
          Rs1_Out = Rs1_In;
          Rs2_Out = Rs2_In;
          WB_Out = WB_In;
          M_Out = M_In;
          EX_Out = EX_In;

        end

    end

endmodule
```

EX-MEM Register

```
module Reg_EX_MEM (flush_EX_MEM_register,clk, reset, WB_In,
    WB_Out, M_In, M_Out, PC_In, PC_Out, Zero_In, Zero_Out,
    Result_In, Result_Out, RD2_In, RD2_Out, Wr_Reg_In,
    Wr_Reg_Out, Sign_In, Sign_Out);

  input clk, reset, Sign_In;
  input [1:0] WB_In;
  input [4:0] M_In;
  input [63:0] PC_In;
  input Zero_In;
  input [63:0] Result_In;
  input [63:0] RD2_In;
  input [4:0] Wr_Reg_In;
  input flush_EX_MEM_register;

  output reg [1:0] WB_Out;
  output reg [4:0] M_Out;
  output reg [63:0] PC_Out;
  output reg Zero_Out, Sign_Out;
```

```
  output reg [63:0] Result_Out;
  output reg [63:0] RD2_Out;
  output reg [4:0] Wr_Reg_Out;


  always @ (posedge clk or posedge reset)
    begin

      if (reset == 1'b1 | flush_EX_MEM_register==1'b1 )
        begin

          WB_Out = 2'd0;
          M_Out = 5'd0;
          PC_Out = 64'd0;
          Zero_Out = 1'b0;
          Result_Out = 64'd0;
          RD2_Out = 64'd0;
          Wr_Reg_Out = 5'd0;
          Sign_Out = 1'b0;

        end

      else
        begin

          WB_Out = WB_In;
          M_Out = M_In;
          PC_Out = PC_In;
          Zero_Out = Zero_In;
          Result_Out = Result_In;
          RD2_Out = RD2_In;
          Wr_Reg_Out = Wr_Reg_In;
          Sign_Out = Sign_In;

        end
    end
endmodule
```

MEM-WB Register

```
module Reg_MEM_WB (flush,clk, reset, WB_In, WB_Out, RD_In,
    RD_Out, Result_In, Result_Out, Wr_Reg_In, Wr_Reg_Out);

  input clk, reset;
  input [1:0] WB_In;
  input [63:0] RD_In;
  input [63:0] Result_In;
  input [4:0] Wr_Reg_In;
```

```
  input flush;

  output reg [1:0] WB_Out;
  output reg [63:0] RD_Out;
  output reg [63:0] Result_Out;
  output reg [4:0] Wr_Reg_Out;

  always @ (posedge clk or posedge reset)
    begin

      if (reset == 1'b1 | flush ==1)
        begin

          WB_Out = 2'd0;
          RD_Out = 64'd0;
          Result_Out = 64'd0;
          Wr_Reg_Out = 5'd0;

        end

      else
        begin

          WB_Out = WB_In;
          RD_Out = RD_In;
          Result_Out = Result_In;
          Wr_Reg_Out = Wr_Reg_In;

        end

    end

endmodule
```

4.3 Testing the New Pipeline

```
lw x4 0x1(zero)
add x3 x4 x5
```

We are loading from memory 0x00001 in to register 4. Then we are adding x4 and x5 together. The image shows the result in R3 which is 6 as x4 contains 4 and x5 contain 2

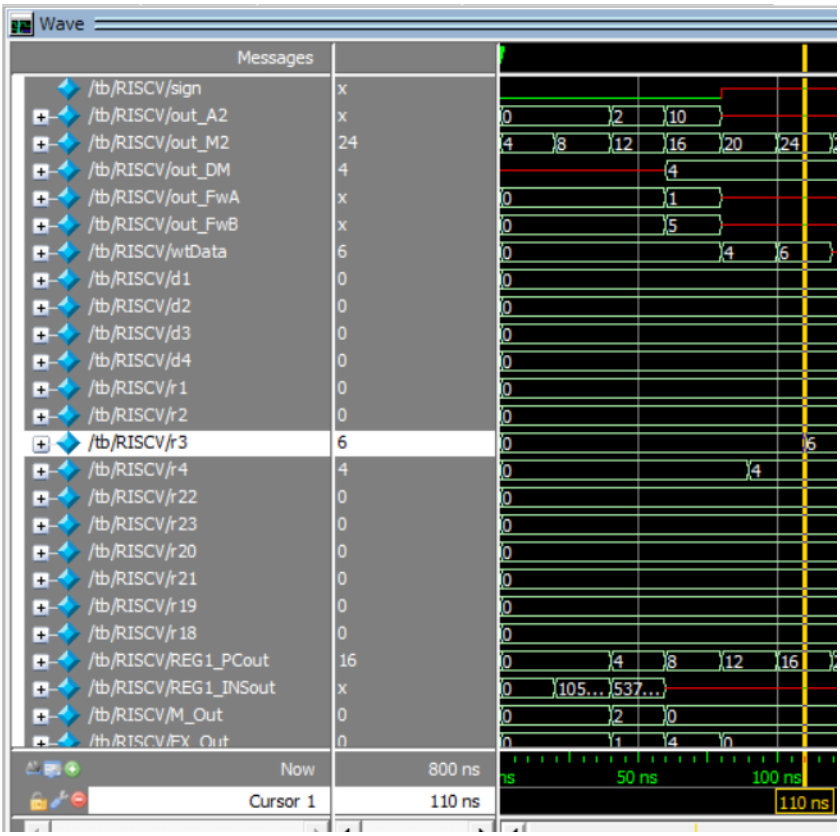


Figure 4.2: Testing for Stall condition

```
add x5 x4 x6
beq zero zero skip
addi x3 zero 3
skip:
addi x4 zero 4
```

The branch instruction is added above an addi stage. hence if flushing takes place we will not see 3 in x3. The result would be 4 in R4 as shown in the figure below.

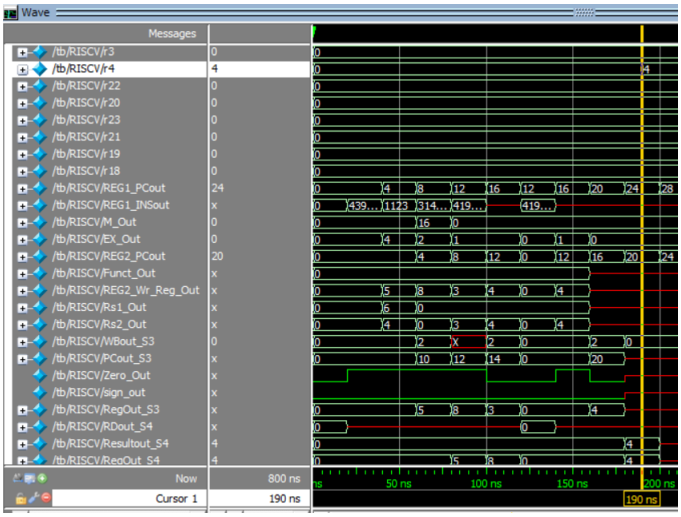


Figure 4.3: Tests for Flushing condition

4.4 Testing the Sorting Algorithm

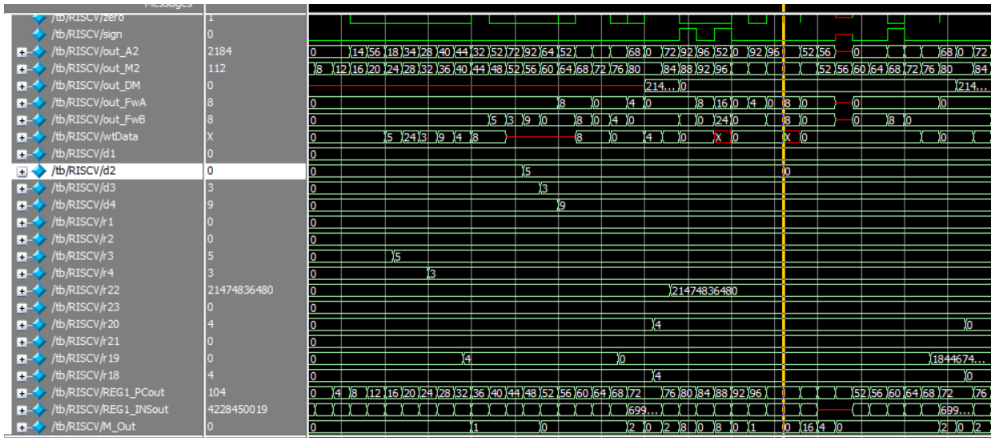


Figure 4.4: Result for sorting using Pipelined Architecture

4.5 Results

The sorting Algorithm was not able to execute properly. There have been numerous challenges faced while designing the pipelined architecture. The image above shows

Simulation results can also be viewed at: `https://www.edaplayground.com/x/sBdk`

4.6 Challenges faced

1. It was quiet difficult to manage the code and due to the differences in choice of naming connections and wires.

2. Majority of Time was spent on task 3 as when we integrated all components together, the results started to change. We were testing isolated conditions for each implementation but as insertion sort contains many different hazard, the modules are not able to work together properly.

3. There was also different implementations present for wiring the data hazard. In some flushing conditions, branch was detected within the decode stage hence reducing the instructions that will be lost. Both method yielded unfavourable results.

Bibliography

D. A. Patterson and J. L. Hennessy, Computer Organization and Design: The hardware/software interface. Morgan Kaufmann, 2021.