



HABIB UNIVERSITY
CS330L COMPUTER ARCHITECTURE LAB

Lab Project Report

Instructor: Maria Samad

Muhammad Hammad Maqdoom *mm05534*
Zoha Ovais Karim *zk05617*
Umema Zehra

Spring 2021

Contents

1	Introduction	1
2	Task 1	2
2.1	Modify single-cycle processor	2
3	Task 2	19
3.1	5 Stage Pipelining	19
4	Code	41
4.1	EDA Links	41

Chapter 1

Introduction

This project required us to build a 5-stage pipelined processor capable of executing a bubble sort program.

1. We modified the single-cycle processor to be able to run the bubble sort code on it.
2. We then modified the said processor to make it a pipelined one (5 stages). We then tested and run each instruction separately to verify that the pipelined version can at least execute one instruction correctly in isolation.
3. We then introduced circuitry to detect hazards (data, control, and structural) and tried to handle them in hardware i.e. by forwarding, stalling, and flushing the pipeline.

Chapter 2

Task 1

2.1 Modify single-cycle processor

The tasks that we attempted in lab 11 came in handy for this task. We made changes and alterations to the code in line with task 1 given to us.

We modified the single-cycle processor to be able to run the bubble sort code on it. We had worked on the bubble sort task in lab 4 (task 2). For the instructions, we made use of the venus simulator and then used the same instructions in verilog for instruction memory.

```
1 // Code your design here
2
3 module Adder(
4     input [63:0] a, b,
5     output reg [63:0] out
6 );
7 always@(*)
8     out = a + b;
9 endmodule
10
11 module registerFile(
12     input [63:0] WriteData,
13     input [4:0] RS1,
14     input [4:0] RS2,
15     input [4:0] RD,
16     input RegWrite, clk, reset,
17     output reg [63:0] ReadData1,
18     output reg [63:0] ReadData2
19 );
20 reg [63:0] Registers [31:0];
21 initial
22     begin
23         Registers[0] = 64'd 0;
24         Registers[1] = 64'd 0;
25         Registers[2] = 64'd 0;
26         Registers[3] = 64'd 0;
27         Registers[4] = 64'd 0;
28         Registers[5] = 64'd 0;
29         Registers[6] = 64'd 0;
```

```

30     Registers[7] = 64'd 0;
31     Registers[8] = 64'd 0;
32     Registers[9] = 64'd 0;
33     Registers[10] = 64'd 0;
34     Registers[11] = 64'd 0;
35     Registers[12] = 64'd 0;
36     Registers[13] = 64'd 0;
37     Registers[14] = 64'd 0;
38     Registers[15] = 64'd 0;
39     Registers[16] = 64'd 0;
40     Registers[17] = 64'd 0;
41     Registers[18] = 64'd 0;
42     Registers[19] = 64'd 0;
43     Registers[20] = 64'd 0;
44     Registers[21] = 64'd 0;
45     Registers[22] = 64'd 0;
46     Registers[23] = 64'd 0;
47     Registers[24] = 64'd 0;
48     Registers[25] = 64'd 0;
49     Registers[26] = 64'd 0;
50     Registers[27] = 64'd 0;
51     Registers[28] = 64'd 0;
52     Registers[29] = 64'd 0;
53     Registers[30] = 64'd 0;
54     Registers[31] = 64'd 0;
55     end
56     always @(posedge clk)
57         if(RegWrite)
58             begin
59                 Registers[RD] = WriteData;
60             end
61     always @(*)
62         if(reset)
63             begin
64                 ReadData1 = 64'b0;
65                 ReadData2 = 64'b0;
66             end
67         else
68             begin
69                 ReadData1 = Registers[RS1];
70                 ReadData2 = Registers[RS2];
71             end
72 endmodule
73
74 module Instruction_Parser(
75     input [31:0] instruction,
76     output [6:0] opcode, funct7,
77     output [4:0] rd , rs1 , rs2,
78     output [2:0] funct3
79 );
80 );
81
82 assign opcode = instruction[6:0];
83 assign rd = instruction[11:7];
84 assign funct3 = instruction[14:12];
85 assign rs1 = instruction[19:15];
86 assign rs2 = instruction[24:20];
87 assign funct7 = instruction[31:25];

```

```

88
89 endmodule
90
91 module mux2x1
92 (
93     input [63:0] a,b,
94     input s ,
95     output [63:0] data_out
96 );
97
98 assign data_out = s ? b : a;
99
100 endmodule
101
102 module data_generator
103 (
104     input [31:0] instruction,
105     output reg [63:0] imm_data
106
107 );
108
109 wire [6:0] opcode;
110 assign opcode = instruction[6:0];
111
112 always @(*)
113 begin
114     case (opcode)
115         7'b0000011: imm_data = {{52{instruction[31]}}}, instruction
[31:20]];
116         7'b0100011: imm_data = {{52{instruction[31]}}}, instruction [31:25],
instruction [11:7]];
117         7'b1100011: imm_data = {{52{instruction[31]}}}, instruction [31] ,
instruction [7], instruction [30:25], instruction [11:8]];
118         7'b0010011: imm_data = {{52{instruction[31]}}}, instruction[31:20]];
119         default : imm_data = 64'd0;
120     endcase
121 end
122
123 endmodule
124
125 // reg [63:0] immediate;
126 // wire [6:0] opcode;
127 // assign opcode = instruction[6:0];
128
129 // always @(instruction)
130 // begin
131 //     if(instruction[6]==0) //data transfer
132 //         if(instruction[5]==0) //load
133 //             immediate[11:0] = instruction [31:20];
134 //         else if(instruction[5]==1) //store
135 //             immediate[11:0]= {instruction[31:25],instruction[11:7]];
136 //     else if(instruction[6]==1) //conditional branches
137 //         immediate[11:0] = {instruction[31],instruction[7],instruction
[30:25] , instruction[11:8]];
138 // end
139
140 // assign imm_data[11:0]= immediate[11:0];
141 // assign imm_data[63:12] = {52{instruction[31]}};

```

```

142
143
144 // endmodule
145
146 module selector(
147     input branch, ZERO,
148     input [63:0] a, b,
149     input [2:0] funct3,
150     output reg sel
151 );
152
153 always@(*)
154 begin
155     if (branch == 1)
156         begin
157             case(funct3)
158             3'b001: //bne
159                 begin
160                     if(branch == 1 & ZERO == 0)
161                         sel = 1;
162                     else
163                         sel = 0;
164                 end
165             3'b000: //beq
166                 begin
167                     if(branch == 1 & ZERO == 1)
168                         sel = 1;
169                     else
170                         sel = 0;
171                 end
172             3'b101: //bge
173                 begin
174                     if (a >= b)
175                         sel = 1;
176                     else
177                         sel = 0;
178                 end
179             endcase
180         end
181     else
182         sel = 0;
183 end
184 endmodule
185
186
187
188
189 module Program_Counter
190 (
191     input clk, reset,
192     input [63:0] PC_In,
193     output reg [63:0] PC_Out
194 );
195
196 reg reset_force; // variable to force 0th value after reset
197
198 initial
199 PC_Out <= 64'd0;

```

```

200
201
202 always @(posedge clk or posedge reset) begin
203     if (reset || reset_force) begin
204         PC_Out = 64'd0;
205         reset_force <= 0;
206     end
207
208     // else if (!PCWrite) begin
209     //     PC_Out = PC_Out;
210     // end
211     else
212     PC_Out = PC_In;
213
214 end
215
216 always @(negedge reset) reset_force <= 1;
217
218 endmodule // Program_Counter
219
220 module Data_Memory(
221     input [63:0] mem_addr,
222     input [63:0] write_data,
223     input clk, mem_write, mem_read,
224     output reg [63:0] read_data,
225     output [63:0] element1,
226     output [63:0] element2 ,
227     output [63:0] element3,
228     output [63:0] element4,
229     output [63:0] element5,
230     output [63:0] element6,
231     output [63:0] element7,
232     output [63:0] element8);
233 reg [0:7] data_mem[63:0];
234 initial
235 begin
236     data_mem[0] = 64'd0;
237     data_mem[1] = 64'd0;
238     data_mem[2] = 64'd0;
239     data_mem[3] = 64'd0;
240     data_mem[4] = 64'd0;
241     data_mem[5] = 64'd0;
242     data_mem[6] = 64'd0;
243     data_mem[7] = 64'd0;
244     data_mem[8] = 64'd0;
245     data_mem[9] = 64'd0;
246     data_mem[10] = 64'd0;
247     data_mem[11] = 64'd0;
248     data_mem[12] = 64'd0;
249     data_mem[13] = 64'd0;
250     data_mem[14] = 64'd0;
251     data_mem[15] = 64'd0;
252     data_mem[16] = 64'd0;
253     data_mem[17] = 64'd0;
254     data_mem[18] = 64'd0;
255     data_mem[19] = 64'd0;
256     data_mem[20] = 64'd0;
257     data_mem[21] = 64'd0;

```



```

258     data_mem[22] = 64'd0;
259     data_mem[23] = 64'd0;
260     data_mem[24] = 64'd0;
261     data_mem[25] = 64'd0;
262     data_mem[26] = 64'd0;
263     data_mem[27] = 64'd0;
264     data_mem[28] = 64'd0;
265     data_mem[29] = 64'd0;
266     data_mem[30] = 64'd0;
267     data_mem[31] = 64'd0;
268     data_mem[32] = 64'd0;
269     data_mem[33] = 64'd0;
270     data_mem[34] = 64'd0;
271     data_mem[35] = 64'd0;
272     data_mem[36] = 64'd0;
273     data_mem[37] = 64'd0;
274     data_mem[38] = 64'd0;
275     data_mem[39] = 64'd0;
276     data_mem[40] = 64'd0;
277     data_mem[41] = 64'd0;
278     data_mem[42] = 64'd0;
279     data_mem[43] = 64'd0;
280     data_mem[44] = 64'd0;
281     data_mem[45] = 64'd0;
282     data_mem[46] = 64'd0;
283     data_mem[47] = 64'd0;
284     data_mem[48] = 64'd0;
285     data_mem[49] = 64'd0;
286     data_mem[50] = 64'd0;
287     data_mem[51] = 64'd0;
288     data_mem[52] = 64'd0;
289     data_mem[53] = 64'd0;
290     data_mem[54] = 64'd0;
291     data_mem[55] = 64'd0;
292     data_mem[56] = 64'd0;
293     data_mem[57] = 64'd0;
294     data_mem[58] = 64'd0;
295     data_mem[59] = 64'd0;
296     data_mem[60] = 64'd0;
297     data_mem[61] = 64'd0;
298     data_mem[62] = 64'd0;
299     data_mem[63] = 64'd0;
300 end
301 always @(negedge clk)
302 begin
303     if (mem_write)
304     begin
305         data_mem[mem_addr] = write_data[7:0];
306         data_mem[mem_addr+1] = write_data[15:8];
307         data_mem[mem_addr+2] = write_data[23:16];
308         data_mem[mem_addr+3] = write_data[31:24];
309         data_mem[mem_addr+4] = write_data[39:32];
310         data_mem[mem_addr+5] = write_data[47:40];
311         data_mem[mem_addr+6] = write_data[55:48];
312         data_mem[mem_addr+7] = write_data[63:56];
313     end
314 end
315 always @(*)

```

```

316     begin
317         if(mem_read)
318             begin
319                 read_data = {data_mem[mem_addr+7],data_mem[mem_addr+6],
data_mem[mem_addr+5], data_mem[mem_addr+4], data_mem[mem_addr+3],
data_mem[mem_addr+2], data_mem[mem_addr+1], data_mem[mem_addr]};
320             end
321         end
322     assign element1= {data_mem[7],data_mem[6],data_mem[5], data_mem[4],
data_mem[3], data_mem[2], data_mem[1], data_mem[0]};
323     assign element2= {data_mem[15],data_mem[14],data_mem[13], data_mem[12],
data_mem[11], data_mem[10], data_mem[9], data_mem[8]};
324     assign element3= {data_mem[23],data_mem[22],data_mem[21], data_mem[20],
data_mem[19], data_mem[18], data_mem[17], data_mem[16]};
325     assign element4= {data_mem[31],data_mem[30],data_mem[29], data_mem[28],
data_mem[27], data_mem[26], data_mem[25], data_mem[24]};
326     assign element5= {data_mem[39],data_mem[38],data_mem[37],data_mem[36],
data_mem[35], data_mem[34], data_mem[33], data_mem[32]};
327     assign element6= {data_mem[47], data_mem[46],data_mem[45],data_mem[44],
data_mem[43], data_mem[42], data_mem[41], data_mem[40]};
328     assign element7= {data_mem[55], data_mem[54],data_mem[53],data_mem[52],
data_mem[51], data_mem[50], data_mem[49], data_mem[48]};
329     assign element8= {data_mem[63], data_mem[62],data_mem[61],data_mem[60],
data_mem[59], data_mem[58], data_mem[57], data_mem[56]};
330 endmodule
331
332 module Instruction_Memory(
333     input [63:0] Inst_Address,
334     output reg [31:0] Instruction
335 );
336 reg [7:0] inst_memory [131:0];
337
338 initial
339 begin
340     inst_memory[0] = 8'b10010011;
341     inst_memory[1] = 8'b00000010;
342     inst_memory[2] = 8'b00110000;
343     inst_memory[3] = 8'b00000000;
344
345     inst_memory[4] = 8'b00100011;
346     inst_memory[5] = 8'b00110010;
347     inst_memory[6] = 8'b01010000;
348     inst_memory[7] = 8'b00000000;
349
350     inst_memory[8] = 8'b10010011;
351     inst_memory[9] = 8'b00000010;
352     inst_memory[10] = 8'b00100000;
353     inst_memory[11] = 8'b00000000;
354
355     inst_memory[12] = 8'b00100011;
356     inst_memory[13] = 8'b00110110;
357     inst_memory[14] = 8'b01010000;
358     inst_memory[15] = 8'b00000000;
359
360     inst_memory[16] = 8'b10010011;
361     inst_memory[17] = 8'b00000010;
362     inst_memory[18] = 8'b10100000;
363     inst_memory[19] = 8'b00000000;

```

```

364
365     inst_memory[20] = 8'b00100011;
366     inst_memory[21] = 8'b001111010;
367     inst_memory[22] = 8'b01010000;
368     inst_memory[23] = 8'b00000000;
369
370     inst_memory[24] = 8'b00010011;
371     inst_memory[25] = 8'b00000101;
372     inst_memory[26] = 8'b01000000;
373     inst_memory[27] = 8'b00000000;
374
375     inst_memory[28] = 8'b10010011;
376     inst_memory[29] = 8'b00000101;
377     inst_memory[30] = 8'b00110000;
378     inst_memory[31] = 8'b00000000;
379
380     inst_memory[32] = 8'b01100011;
381     inst_memory[33] = 8'b00010110;
382     inst_memory[34] = 8'b00000101;
383     inst_memory[35] = 8'b00000000;
384
385     //bne 101
386     inst_memory[36] = 8'b01100011;
387     inst_memory[37] = 8'b10010100;
388     inst_memory[38] = 8'b00000101;
389     inst_memory[39] = 8'b00000000;
390
391     //beq 011
392     inst_memory[40] = 8'b01100011;
393     inst_memory[41] = 8'b00001100;
394     inst_memory[42] = 8'b00000000;
395     inst_memory[43] = 8'b00000100;
396
397     inst_memory[44] = 8'b00010011;
398     inst_memory[45] = 8'b00001001;
399     inst_memory[46] = 8'b00000000;
400     inst_memory[47] = 8'b00000000;
401
402     inst_memory[48] = 8'b01100011;
403     inst_memory[49] = 8'b00000110;
404     inst_memory[50] = 8'b10111001;
405     inst_memory[51] = 8'b00000100;
406
407     inst_memory[52] = 8'b10110011;
408     inst_memory[53] = 8'b00001001;
409     inst_memory[54] = 8'b00100000;
410     inst_memory[55] = 8'b00000001;
411
412     inst_memory[56] = 8'b01100011;
413     inst_memory[57] = 8'b10001110;
414     inst_memory[58] = 8'b10111001;
415     inst_memory[59] = 8'b00000010;
416
417     inst_memory[60] = 8'b10010011;
418     inst_memory[61] = 8'b00010010;
419     inst_memory[62] = 8'b00111001;
420     inst_memory[63] = 8'b00000000;
421

```

```

422 inst_memory[64] = 8'b00010011;
423 inst_memory[65] = 8'b10010011;
424 inst_memory[66] = 8'b00111001;
425 inst_memory[67] = 8'b00000000;
426
427 inst_memory[68] = 8'b10110011;
428 inst_memory[69] = 8'b10000010;
429 inst_memory[70] = 8'b10100010;
430 inst_memory[71] = 8'b00000000;
431
432 inst_memory[72] = 8'b00110011;
433 inst_memory[73] = 8'b00000011;
434 inst_memory[74] = 8'b10100011;
435 inst_memory[75] = 8'b00000000;
436
437 inst_memory[76] = 8'b00000011;
438 inst_memory[77] = 8'b10111110;
439 inst_memory[78] = 8'b00000010;
440 inst_memory[79] = 8'b00000000;
441
442 inst_memory[80] = 8'b10000011;
443 inst_memory[81] = 8'b00111110;
444 inst_memory[82] = 8'b00000011;
445 inst_memory[83] = 8'b00000000;
446 //bge 111
447 inst_memory[84] = 8'b01100011;
448 inst_memory[85] = 8'b01011100;
449 inst_memory[86] = 8'b11011110;
450 inst_memory[87] = 8'b00000001;
451
452 inst_memory[88] = 8'b00110011;
453 inst_memory[89] = 8'b00001111;
454 inst_memory[90] = 8'b11000000;
455 inst_memory[91] = 8'b00000001;
456
457 inst_memory[92] = 8'b00110011;
458 inst_memory[93] = 8'b00001110;
459 inst_memory[94] = 8'b11010000;
460 inst_memory[95] = 8'b00000001;
461
462 inst_memory[96] = 8'b10110011;
463 inst_memory[97] = 8'b00001110;
464 inst_memory[98] = 8'b11100000;
465 inst_memory[99] = 8'b00000001;
466
467 inst_memory[100] =8'b00100011;
468 inst_memory[101] =8'b10110000;
469 inst_memory[102] =8'b11000010;
470 inst_memory[103] =8'b00000001;
471
472 inst_memory[104] =8'b00100011;
473 inst_memory[105] =8'b00110000;
474 inst_memory[106] =8'b11010011;
475 inst_memory[107] =8'b00000001;
476
477 inst_memory[108] =8'b10010011;
478 inst_memory[109] =8'b10001001;
479 inst_memory[110] =8'b00011001;

```

```

480     inst_memory[111] =8'b00000000;
481
482     inst_memory[112] =8'b11100011;
483     inst_memory[113] =8'b00000100;
484     inst_memory[114] =8'b00000000;
485     inst_memory[115] =8'b11111100;
486
487     inst_memory[116] =8'b00010011;
488     inst_memory[117] =8'b00001001;
489     inst_memory[118] =8'b00011001;
490     inst_memory[119] =8'b00000000;
491
492     inst_memory[120] =8'b11100011;
493     inst_memory[121] =8'b00001100;
494     inst_memory[122] =8'b00000000;
495     inst_memory[123] =8'b11111010;
496
497     inst_memory[124] =8'b01100011;
498     inst_memory[125] =8'b00000010;
499     inst_memory[126] =8'b00000000;
500     inst_memory[127] =8'b00000000;
501
502     inst_memory[128] =8'b00010011;
503     inst_memory[129] =8'b00000000;
504     inst_memory[130] =8'b00000000;
505     inst_memory[131] =8'b00000000;
506
507 end
508 always@(Inst_Address)
509     Instruction = {inst_memory[Inst_Address+3],inst_memory[Inst_Address+2],
510                   inst_memory[Inst_Address+1], inst_memory[Inst_Address]};
511 endmodule
512
513 module Control_Unit
514 (
515     input [6:0] Opcode,
516     output reg [1:0] ALUOp,
517     output reg Branch, MemRead, MemtoReg, MemWrite, ALUSrc, Regwrite
518 );
519 always @(*)
520 begin
521     case (Opcode)
522     7'b0110011: // R-type (add/sub)
523     begin
524         ALUSrc = 1'b0;
525         MemtoReg = 1'b0;
526         Regwrite = 1'b1;
527         MemRead = 1'b0;
528         MemWrite = 1'b0;
529         Branch = 1'b0;
530         ALUOp = 2'b10;
531     end
532     7'b0000011: // I-type (ld)
533     begin
534         ALUSrc = 1'b1;
535         MemtoReg = 1'b1;
536         Regwrite = 1'b1;
537         MemRead = 1'b1;

```

```

537         MemWrite = 1'b0;
538         Branch = 1'b0;
539         ALUOp = 2'b00;
540     end
541     7'b0100011: // S-type(sd)
542     begin
543         ALUSrc = 1'b1;
544         MemtoReg = 1'bx;
545         Regwrite = 1'b0;
546         MemRead = 1'b0;
547         MemWrite = 1'b1;
548         Branch = 1'b0;
549         ALUOp = 2'b00;
550     end
551     7'b0010011: // I-type (addi)
552     begin
553         ALUSrc = 1'b1;
554         MemtoReg = 1'b0;
555         Regwrite = 1'b1;
556         MemRead = 1'b1;
557         MemWrite = 1'b0;
558         Branch = 1'b0;
559         ALUOp = 2'b00;
560     end
561     7'b1100011: // SB-type (beq/bne/bge)
562     begin
563         ALUSrc = 1'b0;
564         MemtoReg = 1'bx;
565         Regwrite = 1'b0;
566         MemRead = 1'b0;
567         MemWrite = 1'b0;
568         Branch = 1'b1;
569         ALUOp = 2'b01;
570     end
571     default: begin
572         ALUSrc = 1'b0;
573         MemtoReg = 1'b0;
574         Regwrite = 1'b0;
575         MemRead = 1'b0;
576         MemWrite = 1'b0;
577         Branch = 1'b0;
578         ALUOp = 2'b00;
579     end
580 endcase
581 end
582 endmodule
583
584 module ALU_Control
585 (
586     input [1:0] ALUOp,
587     input [3:0] Funct,
588     output reg [3:0] Operation
589 );
590 always@(*)
591 begin
592     case(ALUOp)
593         2'b00: //for both addi and slli
594         begin

```

```

595         case(Funct[2:0])
596         3'b000: //addi
597         begin
598             Operation = 4'b0010;
599         end
600         3'b001: //slli
601         begin
602             Operation = 4'b1000;
603         end
604         endcase
605     end
606     2'b01:
607     begin
608         Operation = 4'b0110;
609     end
610     2'b10:
611     begin
612         case(Funct)
613         4'b0000:
614         begin
615             Operation = 4'b0010;
616         end
617         4'b1000:
618         begin
619             Operation = 4'b0110;
620         end
621         4'b0111:
622         begin
623             Operation = 4'b0000;
624         end
625         4'b0110:
626         begin
627             Operation = 4'b0001;
628         end
629         endcase
630     end
631 endcase
632 end
633 endmodule
634
635 module alu_64(
636     input [63:0] a,
637     input [63:0] b,
638     input [3:0] ALUOp,
639     output reg [63:0] Result,
640     output reg ZERO
641 );
642 always @ (*)
643 begin
644     case(ALUOp)
645     4'b0000 :
646     begin
647         Result = a&b;
648     end
649     4'b0001 :
650     begin
651         Result = a|b;
652     end

```

```

653     4'b0010 :
654         begin
655             Result = a+b;
656         end
657     4'b0110:
658         begin
659             Result = a-b;
660         end
661     4'b1100:
662         begin
663             Result = ~(a|b);
664         end
665     4'b1000:
666         begin
667             Result = a << b;
668         end
669     default : Result = 0;
670 endcase
671
672 if (Result == 64'b0)
673     ZERO = 1'b1;
674 else
675     ZERO = 1'b0;
676
677 end
678 endmodule
679
680
681
682 module RISC_V_Processor(
683     input clk, reset
684 );
685
686 //Mux output
687 wire [63:0] PC_In_from_mux;
688 //Program counter output
689 wire [63:0] PC_Out;
690 //Adders outputs
691 wire [63:0] a1_out;
692 wire [63:0] a2_out;
693 //Input to Adder a1
694 wire [63:0] b_in = 64'd4;
695 //Output from IM
696 wire [31:0] Instruction;
697 //Output from IP
698 wire [4:0] rd;
699 wire [4:0] rs1;
700 wire [4:0] rs2;
701 wire [6:0] opcode;
702 wire [6:0] funct7;
703 wire [2:0] funct3;
704 //Outputs from RegisterFile
705 wire [63:0] ReadData1;
706 wire [63:0] ReadData2;
707 //Outputs from Control Unit
708 wire [1:0] ALUOp;
709 wire Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;
710 //Outputs from ALU Control

```



```

711 wire [3:0] Operation;
712 //Funct input to ALU_Control
713 wire [3:0] Funct;
714 assign Funct = {Instruction[30], Instruction[14:12]};
715 //Output from ALU
716 wire [63:0] Result_from_alu;
717 wire zero_output;
718 //Output from Data generator
719 wire [63:0] imm_data;
720 //Output from mux2
721 wire [63:0] out_from_mux2;
722 //sel for mux1
723 wire sel;
724 //Output from Data memory
725 wire [63:0] out_from_DM;
726 //Output from mux3
727 wire [63:0] out_from_mux3;
728 //Input to Adder a2
729 wire [63:0] b_adder2;
730 assign b_adder2 = imm_data << 1;
731
732 Program_Counter pc (.clk(clk), .reset(reset), .PC_In(PC_In_from_mux), .
    PC_Out(PC_Out));
733
734 Adder a1 (.a(PC_Out), .b(b_in), .out(a1_out));
735
736 Adder a2(.a(PC_Out), .b(b_adder2), .out(a2_out));
737
738 mux2x1 mux1(.a(a1_out), .b(a2_out), .s(sel), .data_out(PC_In_from_mux));
739
740 Instruction_Memory im(.Inst_Address(PC_Out), .Instruction(Instruction));
741
742 Instruction_Parser ip(.instruction(Instruction), .rd(rd), .rs1(rs1), .rs2(
    rs2), .funct3(funct3), .funct7(funct7), .opcode(opcode));
743
744 registerFile rf(.WriteData(out_from_mux3), .RS1(rs1), .RS2(rs2), .RD(rd), .
    clk(clk), .reset(reset), .RegWrite(RegWrite), .ReadData1(ReadData1), .
    ReadData2(ReadData2));
745
746 Control_Unit cu(
747     .Opcode(opcode),
748     .ALUOp(ALUOp),
749     .Branch(Branch),
750     .MemRead(MemRead),
751     .MemtoReg(MemtoReg),
752     .MemWrite(MemWrite),
753     .ALUSrc(ALUSrc),
754     .Regwrite(RegWrite));
755
756 ALU_Control aluc(
757     .ALUOp(ALUOp),
758     .Funct(Funct),
759     .Operation(Operation));
760
761 data_generator dg(
762     .instruction(Instruction),
763     .imm_data(imm_data));
764

```

```

765 mux2x1 mux2(
766     .a(ReadData2),
767     .b(imm_data),
768     .s(ALUSrc),
769     .data_out(out_from_mux2));
770
771 alu_64 alu(
772     .a(ReadData1),
773     .b(out_from_mux2),
774     .ALUOp(Operation),
775     .Result(Result_from_alu),
776     .ZERO(zero_output));
777
778 selector s(
779     .branch(Branch),
780     .ZERO(zero_output),
781     .a(ReadData1),
782     .b(out_from_mux2),
783     .funct3(funct3),
784     .sel(sel));
785
786 Data_Memory dm(.mem_addr(Result_from_alu), .write_data(ReadData2), .clk(
    clk), .mem_write(MemWrite), .mem_read(MemRead), .read_data(out_from_DM),
    .element1(e11), .element2(e12), .element3(e13), .element4(e14), .
    element5(e15), .element6(e16), .element7(e17), .element8(e18));
787
788 mux2x1 mux3(
789     .b(out_from_DM),
790     .a(Result_from_alu),
791     .s(MemtoReg),
792     .data_out(out_from_mux3));
793
794 always @(posedge clk)
795     begin
796         $monitor(
797             "PC_In = ", PC_In_from_mux,
798             ", PC_Out = ", PC_Out,
799             ", Instruction = %b", Instruction,
800             ", Opcode = %b", opcode,
801             ", Funct3 = %b", funct3,
802             ", Zero = %b", zero_output,
803             ", Branch = %d", Branch,
804             ", sel = %d", sel,
805             ", rs1 = %d", rs1,
806             ", rs2 = %d", rs2,
807             ", rd = %d", rd,
808             ", funct7 = %b", funct7,
809             ", ALUOp = %b", ALUOp,
810             ", imm_data = %d", imm_data,
811             ", Operation = %b", Operation
812         );
813     end
814 endmodule

```

Test Bench:

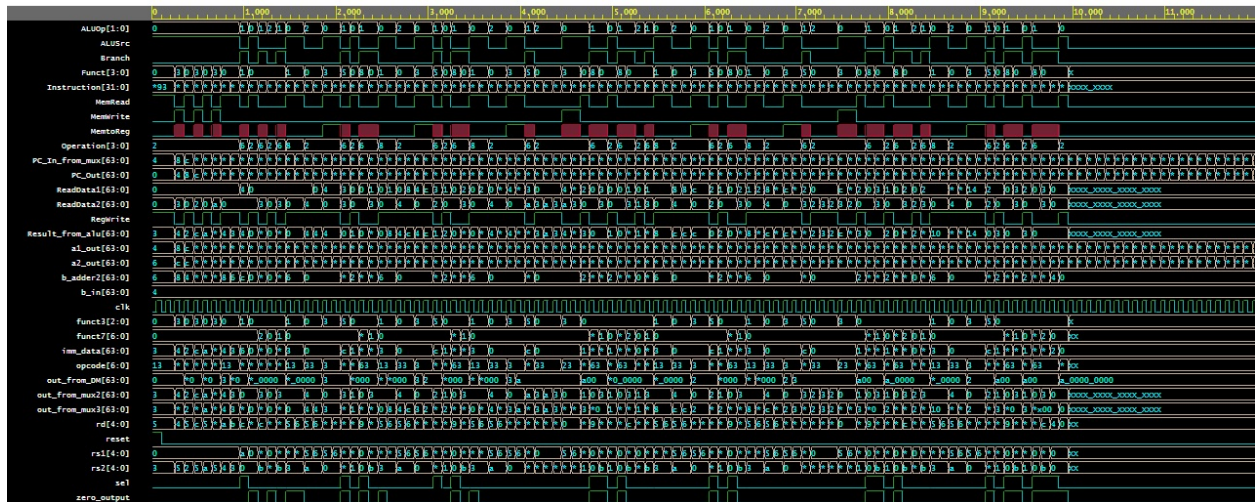
```

1 module tb();
2

```

```
3 reg clk, reset;
4
5 RISC_V_Processor processor(.clk(clk), .reset(reset));
6
7 initial
8 begin
9     $dumpfile("dump.vcd");
10    $dumpvars();
11    clk = 1'd0;
12    reset = 1'd1;
13    #10
14    reset = 1'd0;
15    #1200
16    $finish;
17 end
18
19 always #5 clk=~clk;
20
21 endmodule
```

Result from Test Bench: Wave Diagram



Chapter 3

Task 2

3.1 5 Stage Pipelining

We now modified the said processor to make it a pipelined one (5 stages). We then tested and ran each instruction separately to verify that the pipelined version can at least execute one instruction correctly in isolation.

```
1
2 module Adder(
3     input [63:0] a, b,
4     output reg [63:0] out
5 );
6 always@(*)
7     out = a + b;
8 endmodule
9
10
11 module alu_64(
12     input [63:0] a,
13     input [63:0] b,
14     input [3:0] ALUOp,
15     output reg ZERO,
16     output reg [63:0] Result
17 );
18 always @ (*)
19 begin
20     case(ALUOp)
21     4'b0000 :
22         begin
23             Result = a&b;
24         end
25     4'b0001 :
26         begin
27             Result = a|b;
28         end
29     4'b0010 :
30         begin
31             Result = a+b;
32         end
33     4'b0110 :
```

```

34         begin
35             Result = a-b;
36         end
37     4'b1100:
38         begin
39             Result = ~(a|b);
40         end
41     4'b1000:
42         begin
43             Result = a << b;
44         end
45     default : Result = 0;
46 endcase
47
48 if (Result == 64'd0)
49     ZERO = 1'b0;
50 else
51     ZERO = 1'b0;
52
53 end
54 endmodule
55
56
57 module ALU_Control
58 (
59     input [1:0] ALUOp,
60     input [3:0] Funct,
61     output reg [3:0] Operation
62 );
63 always@(*)
64 begin
65     case(ALUOp)
66         2'b00: //for both addi and slli
67             begin
68                 case(Funct[2:0])
69                     3'b000: //addi
70                     begin
71                         Operation = 4'b0010;
72                     end
73                     3'b001: //slli
74                     begin
75                         Operation = 4'b1000;
76                     end
77                 endcase
78             end
79         2'b01:
80             begin
81                 Operation = 4'b0110;
82             end
83         2'b10:
84             begin
85                 case(Funct)
86                     4'b0000:
87                     begin
88                         Operation = 4'b0010;
89                     end
90                     4'b1000:
91                     begin

```

```

92         Operation = 4'b0110;
93     end
94     4'b0111:
95         begin
96             Operation = 4'b0000;
97         end
98     4'b0110:
99         begin
100             Operation = 4'b0001;
101         end
102     endcase
103 end
104 endcase
105 end
106 endmodule
107
108
109
110
111 module Control_Unit
112 (
113     input [6:0] Opcode,
114     output reg [1:0] ALUOp,
115     output reg Branch, MemRead, MemtoReg, MemWrite, ALUSrc, Regwrite
116 );
117 always @(*)
118 begin
119     case (Opcode)
120     7'b0110011: // R-type (add/sub)
121         begin
122             ALUSrc = 1'b0;
123             MemtoReg = 1'b0;
124             Regwrite = 1'b1;
125             MemRead = 1'b0;
126             MemWrite = 1'b0;
127             Branch = 1'b0;
128             ALUOp = 2'b10;
129         end
130     7'b0000011: // I-type (ld)
131         begin
132             ALUSrc = 1'b1;
133             MemtoReg = 1'b1;
134             Regwrite = 1'b1;
135             MemRead = 1'b1;
136             MemWrite = 1'b0;
137             Branch = 1'b0;
138             ALUOp = 2'b00;
139         end
140     7'b0100011: // S-type(sd)
141         begin
142             ALUSrc = 1'b1;
143             MemtoReg = 1'bx;
144             Regwrite = 1'b0;
145             MemRead = 1'b0;
146             MemWrite = 1'b1;
147             Branch = 1'b0;
148             ALUOp = 2'b00;
149         end

```

```

150 7'b0010011: // I-type (addi)
151     begin
152         ALUSrc = 1'b1;
153         MemtoReg = 1'b0;
154         Regwrite = 1'b1;
155         MemRead = 1'b1;
156         MemWrite = 1'b0;
157         Branch = 1'b0;
158         ALUOp = 2'b00;
159     end
160 7'b1100011: // SB-type (beq/bne/bge)
161     begin
162         ALUSrc = 1'b0;
163         MemtoReg = 1'bx;
164         Regwrite = 1'b0;
165         MemRead = 1'b0;
166         MemWrite = 1'b0;
167         Branch = 1'b1;
168         ALUOp = 2'b01;
169     end
170 default: begin
171     ALUSrc = 1'b0;
172     MemtoReg = 1'b0;
173     Regwrite = 1'b0;
174     MemRead = 1'b0;
175     MemWrite = 1'b0;
176     Branch = 1'b0;
177     ALUOp = 2'b00;
178 end
179 endcase
180 end
181 endmodule
182
183 module Forwarding_Unit
184 (
185     input EXMEM_ReadData2, MEMWB_read_data,
186     input rs1, rs2,
187     input EXMEM_Regwrite,
188     input MEMWB_RegWrite,
189
190     output reg [1:0] fwd_A,
191     output reg [1:0] fwd_B
192 );
193
194
195
196 always @(*) begin
197
198     if (EXMEM_ReadData2 == rs1 && EXMEM_Regwrite && EXMEM_ReadData2 != 0)
199         begin
200             fwd_A = 2'b10;
201         end
202
203     else if (((MEMWB_read_data == rs1) && MEMWB_RegWrite && (
204         MEMWB_read_data != 0))
205         &&
206         !(EXMEM_Regwrite && (EXMEM_ReadData2 != 0) && (EXMEM_ReadData2
207         == rs1)))

```



```

206         begin
207             fwd_A = 2'b01;
208         end
209
210     else
211         begin
212             fwd_A = 2'b00;
213         end
214
215
216     if ((EXMEM_ReadData2 == rs2) && (EXMEM_Regwrite) && (EXMEM_ReadData2 !=
217         0))
218         begin
219             fwd_B = 2'b10;
220         end
221
222     else if (((MEMWB_read_data == rs2) && (MEMWB_RegWrite == 1) && (
223     MEMWB_read_data != 0))
224         &&
225         !(EXMEM_Regwrite && (EXMEM_ReadData2 != 0) && (EXMEM_ReadData2
226     == rs2)
227         ))
228         begin
229             fwd_B = 2'b01;
230         end
231
232     else
233         begin
234             fwd_B = 2'b00;
235         end
236     end
237 endmodule // Forwarding Unit
238
239 module MUX_Triple
240 (
241     input [63:0] a, b, c,
242     input [1:0] sel,
243     output reg [63:0] Res
244 );
245
246 always@(*)
247 begin
248     case (sel)
249         2'b00: Res = a;
250         2'b01: Res = b;
251         2'b10: Res = c;
252         default: Res = 2'bX;
253     endcase
254 end
255
256 endmodule // Triple MUX
257
258
259 module data_generator

```

```

261 (
262 input [31:0] instruction,
263 output reg [63:0] imm_data
264
265 );
266
267 wire [6:0] opcode;
268 assign opcode = instruction[6:0];
269
270 always @(*)
271 begin
272     case (opcode)
273         7'b0000011: imm_data = {{52{instruction[31]}}}, instruction
[31:20]];
274         7'b0100011: imm_data = {{52{instruction[31]}}}, instruction [31:25],
instruction [11:7]];
275         7'b1100011: imm_data = {{52{instruction[31]}}}, instruction [31] ,
instruction [7], instruction [30:25], instruction [11:8]];
276         7'b0010011: imm_data = {{52{instruction[31]}}}, instruction[31:20]];
277         default : imm_data = 64'd0;
278     endcase
279 end
280
281 endmodule
282
283 module Data_Memory(
284     input [63:0] mem_addr,
285     input [63:0] write_data,
286     input clk, mem_write, mem_read,
287     output reg [63:0] read_data
288 );
289 reg [0:7] data_mem[63:0];
290 initial
291 begin
292     data_mem[0] = 64'd0;
293     data_mem[1] = 64'd0;
294     data_mem[2] = 64'd0;
295     data_mem[3] = 64'd0;
296     data_mem[4] = 64'd0;
297     data_mem[5] = 64'd0;
298     data_mem[6] = 64'd0;
299     data_mem[7] = 64'd0;
300     data_mem[8] = 64'h8;
301     data_mem[9] = 64'd0;
302     data_mem[10] = 64'h10;
303     data_mem[11] = 64'd0;
304     data_mem[12] = 64'd0;
305     data_mem[13] = 64'd0;
306     data_mem[14] = 64'd0;
307     data_mem[15] = 64'd0;
308     data_mem[16] = 64'd0;
309     data_mem[17] = 64'd0;
310     data_mem[18] = 64'd0;
311     data_mem[19] = 64'd0;
312     data_mem[20] = 64'd0;
313     data_mem[21] = 64'd0;
314     data_mem[22] = 64'd0;
315     data_mem[23] = 64'd0;

```

```

316     data_mem[24] = 64'd0;
317     data_mem[25] = 64'd0;
318     data_mem[26] = 64'd0;
319     data_mem[27] = 64'd0;
320     data_mem[28] = 64'd0;
321     data_mem[29] = 64'd0;
322     data_mem[30] = 64'd0;
323     data_mem[31] = 64'd0;
324     data_mem[32] = 64'd0;
325     data_mem[33] = 64'd0;
326     data_mem[34] = 64'd0;
327     data_mem[35] = 64'd0;
328     data_mem[36] = 64'd0;
329     data_mem[37] = 64'd0;
330     data_mem[38] = 64'd0;
331     data_mem[39] = 64'd0;
332     data_mem[40] = 64'd8;
333     data_mem[41] = 64'd0;
334     data_mem[42] = 64'd0;
335     data_mem[43] = 64'd0;
336     data_mem[44] = 64'd0;
337     data_mem[45] = 64'd0;
338     data_mem[46] = 64'd0;
339     data_mem[47] = 64'd0;
340     data_mem[48] = 64'd0;
341     data_mem[49] = 64'd0;
342     data_mem[50] = 64'd0;
343     data_mem[51] = 64'd0;
344     data_mem[52] = 64'd0;
345     data_mem[53] = 64'd0;
346     data_mem[54] = 64'd0;
347     data_mem[55] = 64'd0;
348     data_mem[56] = 64'd0;
349     data_mem[57] = 64'd0;
350     data_mem[58] = 64'd0;
351     data_mem[59] = 64'd0;
352     data_mem[60] = 64'd0;
353     data_mem[61] = 64'd0;
354     data_mem[62] = 64'd0;
355     data_mem[63] = 64'd0;
356 end
357 always @(negedge clk)
358 begin
359     if (mem_write)
360     begin
361         data_mem[mem_addr] = write_data[7:0];
362         data_mem[mem_addr+1] = write_data[15:8];
363         data_mem[mem_addr+2] = write_data[23:16];
364         data_mem[mem_addr+3] = write_data[31:24];
365         data_mem[mem_addr+4] = write_data[39:32];
366         data_mem[mem_addr+5] = write_data[47:40];
367         data_mem[mem_addr+6] = write_data[55:48];
368         data_mem[mem_addr+7] = write_data[63:56];
369     end
370 end
371 always @(*)
372 begin
373     if(mem_read)

```

```

374         begin
375             read_data = {data_mem[mem_addr+7], data_mem[mem_addr+6],
                           data_mem[mem_addr+5], data_mem[mem_addr+4], data_mem[mem_addr+3],
                           data_mem[mem_addr+2], data_mem[mem_addr+1], data_mem[mem_addr]};
376         end
377     end
378 endmodule
379
380
381
382 module EX_MEM(
383     input clk, reset, ZERO,
384     input [63:0] out, Result, IDEX_ReadData2,
385     input [4:0] IDEX_inst2,
386     input IDEX_Branch, IDEX_MemRead, IDEX_MemtoReg, IDEX_MemWrite,
    IDEX_Regwrite,
387     output reg EXMEM_ZERO,
388     output reg [4:0] EXMEM_inst2,
389     output reg [63:0] EXMEM_out, EXMEM_Result, EXMEM_ReadData2,
390     output reg EXMEM_Branch, EXMEM_MemRead, EXMEM_MemtoReg, EXMEM_MemWrite,
    EXMEM_Regwrite
391 );
392
393     always @(posedge clk or reset)
394     begin
395         if(clk)
396             begin
397                 EXMEM_out = out;
398                 EXMEM_ZERO = ZERO;
399                 EXMEM_Result = Result;
400                 EXMEM_ReadData2 = IDEX_ReadData2;
401                 EXMEM_inst2 = IDEX_inst2;
402                 EXMEM_Branch = IDEX_Branch;
403                 EXMEM_MemRead = IDEX_MemRead;
404                 EXMEM_MemtoReg= IDEX_MemtoReg;
405                 EXMEM_MemWrite= IDEX_MemWrite;
406                 EXMEM_Regwrite= IDEX_Regwrite;
407             end
408         else
409             begin
410                 EXMEM_out = 0;
411                 EXMEM_ZERO = 0;
412                 EXMEM_Result = 0;
413                 EXMEM_ReadData2 = 0;
414                 EXMEM_inst2 = 0;
415                 EXMEM_Branch = 0;
416                 EXMEM_MemRead = 0;
417                 EXMEM_MemtoReg= 0;
418                 EXMEM_MemWrite= 0;
419                 EXMEM_Regwrite=0;
420             end
421         end
422     end
423 endmodule
424
425
426
427

```

```

428 module ID_EX(
429     input clk, reset,
430     input [3:0] inst1,
431     input [4:0] inst2,
432     input [63:0] ReadData1, ReadData2, PC_Out, imm_data,
433     input [1:0] ALUOp,
434     input Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite,
435     output reg [3:0] IDEX_inst1,
436     output reg [4:0] IDEX_inst2,
437     output reg [63:0] IDEX_PC_Out, IDEX_ReadData1, IDEX_ReadData2,
438     IDEX_imm_data,
439     output reg [1:0] IDEX_ALUOp,
440     output reg IDEX_Branch, IDEX_MemRead, IDEX_MemtoReg, IDEX_MemWrite,
441     IDEX_ALUSrc, IDEX_Regwrite
442 );
443 always @(posedge clk or reset)
444     begin
445         if(clk)
446             begin
447                 IDEX_PC_Out = PC_Out;
448                 IDEX_ReadData1 = ReadData1;
449                 IDEX_ReadData2 = ReadData2;
450                 IDEX_imm_data = imm_data;
451                 IDEX_inst1 = inst1;
452                 IDEX_inst2 = inst2;
453                 IDEX_Branch = Branch;
454                 IDEX_MemRead = MemRead;
455                 IDEX_MemWrite = MemWrite;
456                 IDEX_ALUSrc = ALUSrc;
457                 IDEX_Regwrite = RegWrite;
458                 IDEX_ALUOp = ALUOp;
459                 IDEX_MemtoReg = MemtoReg;
460             end
461         else
462             begin
463                 IDEX_PC_Out = 0;
464                 IDEX_ReadData1 = 0;
465                 IDEX_ReadData2 = 0;
466                 IDEX_imm_data = 0;
467                 IDEX_inst1 = 0;
468                 IDEX_inst2 = 0;
469                 IDEX_Branch = 0;
470                 IDEX_MemRead = 0;
471                 IDEX_MemWrite = 0;
472                 IDEX_ALUSrc = 0;
473                 IDEX_Regwrite = 0;
474                 IDEX_ALUOp = 0;
475                 IDEX_MemtoReg = 0;
476             end
477         end
478     end
479 endmodule
480
481
482
483 module IF_ID(

```

```

484     input clk, reset,
485     input [31:0] instruction,
486     input [63:0] PC_Out,
487     output reg [31:0] IFID_instruction,
488     output reg [63:0] IFID_PC_Out
489 );
490
491
492
493 always @(posedge clk or reset)
494 begin
495     if(clk)
496     begin
497         IFID_instruction = instruction;
498         IFID_PC_Out = PC_Out;
499     end
500     else
501     begin
502         IFID_instruction = 0;
503         IFID_PC_Out = 0;
504     end
505 end
506 endmodule
507
508
509 module Instruction_Memory(
510     input [63:0] Inst_Address,
511     output reg [31:0] Instruction
512 );
513
514 reg [7:0] inst_memory [131:0];
515
516 initial
517 begin
518
519     inst_memory[0] = 8'b10000011;
520     inst_memory[1] = 8'b00110100;
521     inst_memory[2] = 8'b10000101;
522     inst_memory[3] = 8'b00000010;
523
524
525     {inst_memory[7],inst_memory[6],inst_memory[5],inst_memory[4]}=32'h00548513;
526     // inst_memory[0] = 8'b10010011;
527     // inst_memory[1] = 8'b00000010;
528     // inst_memory[2] = 8'b00110000;
529     // inst_memory[3] = 8'b00000000;
530
531     // inst_memory[4] = 8'b00100011;
532     // inst_memory[5] = 8'b00110010;
533     // inst_memory[6] = 8'b01010000;
534     // inst_memory[7] = 8'b00000000;
535
536     // inst_memory[8] = 8'b10010011;
537     // inst_memory[9] = 8'b00000010;
538     // inst_memory[10] = 8'b00100000;
539     // inst_memory[11] = 8'b00000000;
540

```

```

541 // inst_memory[12] = 8'b00100011;
542 // inst_memory[13] = 8'b00110110;
543 // inst_memory[14] = 8'b01010000;
544 // inst_memory[15] = 8'b00000000;
545
546 // inst_memory[16] = 8'b10010011;
547 // inst_memory[17] = 8'b00000010;
548 // inst_memory[18] = 8'b10100000;
549 // inst_memory[19] = 8'b00000000;
550
551 // inst_memory[20] = 8'b00100011;
552 // inst_memory[21] = 8'b00111010;
553 // inst_memory[22] = 8'b01010000;
554 // inst_memory[23] = 8'b00000000;
555
556 // inst_memory[24] = 8'b00010011;
557 // inst_memory[25] = 8'b00000101;
558 // inst_memory[26] = 8'b01000000;
559 // inst_memory[27] = 8'b00000000;
560
561 // inst_memory[28] = 8'b10010011;
562 // inst_memory[29] = 8'b00000101;
563 // inst_memory[30] = 8'b00110000;
564 // inst_memory[31] = 8'b00000000;
565
566 // inst_memory[32] = 8'b01100011;
567 // inst_memory[33] = 8'b00010110;
568 // inst_memory[34] = 8'b00000101;
569 // inst_memory[35] = 8'b00000000;
570
571 // //bne 101
572 // inst_memory[36] = 8'b01100011;
573 // inst_memory[37] = 8'b10010100;
574 // inst_memory[38] = 8'b00000101;
575 // inst_memory[39] = 8'b00000000;
576
577 // //beq 011
578 // inst_memory[40] = 8'b01100011;
579 // inst_memory[41] = 8'b00001100;
580 // inst_memory[42] = 8'b00000000;
581 // inst_memory[43] = 8'b00000100;
582
583 // inst_memory[44] = 8'b00010011;
584 // inst_memory[45] = 8'b00001001;
585 // inst_memory[46] = 8'b00000000;
586 // inst_memory[47] = 8'b00000000;
587
588 // inst_memory[48] = 8'b01100011;
589 // inst_memory[49] = 8'b00000110;
590 // inst_memory[50] = 8'b10111001;
591 // inst_memory[51] = 8'b00000100;
592
593 // inst_memory[52] = 8'b10110011;
594 // inst_memory[53] = 8'b00001001;
595 // inst_memory[54] = 8'b00100000;
596 // inst_memory[55] = 8'b00000001;
597
598 // inst_memory[56] = 8'b01100011;

```

```

599 // inst_memory[57] = 8'b10001110;
600 // inst_memory[58] = 8'b10111001;
601 // inst_memory[59] = 8'b00000010;
602
603 // inst_memory[60] = 8'b10010011;
604 // inst_memory[61] = 8'b00010010;
605 // inst_memory[62] = 8'b00111001;
606 // inst_memory[63] = 8'b00000000;
607
608 // inst_memory[64] = 8'b00010011;
609 // inst_memory[65] = 8'b10010011;
610 // inst_memory[66] = 8'b00111001;
611 // inst_memory[67] = 8'b00000000;
612
613 // inst_memory[68] = 8'b10110011;
614 // inst_memory[69] = 8'b10000010;
615 // inst_memory[70] = 8'b10100010;
616 // inst_memory[71] = 8'b00000000;
617
618 // inst_memory[72] = 8'b00110011;
619 // inst_memory[73] = 8'b00000011;
620 // inst_memory[74] = 8'b10100011;
621 // inst_memory[75] = 8'b00000000;
622
623 // inst_memory[76] = 8'b00000011;
624 // inst_memory[77] = 8'b10111110;
625 // inst_memory[78] = 8'b00000010;
626 // inst_memory[79] = 8'b00000000;
627
628 // inst_memory[80] = 8'b10000011;
629 // inst_memory[81] = 8'b00111110;
630 // inst_memory[82] = 8'b00000011;
631 // inst_memory[83] = 8'b00000000;
632 // //bge 111
633 // inst_memory[84] = 8'b01100011;
634 // inst_memory[85] = 8'b01011100;
635 // inst_memory[86] = 8'b11011110;
636 // inst_memory[87] = 8'b00000001;
637
638 // inst_memory[88] = 8'b00110011;
639 // inst_memory[89] = 8'b00001111;
640 // inst_memory[90] = 8'b11000000;
641 // inst_memory[91] = 8'b00000001;
642
643 // inst_memory[92] = 8'b00110011;
644 // inst_memory[93] = 8'b00001110;
645 // inst_memory[94] = 8'b11010000;
646 // inst_memory[95] = 8'b00000001;
647
648 // inst_memory[96] = 8'b10110011;
649 // inst_memory[97] = 8'b00001110;
650 // inst_memory[98] = 8'b11100000;
651 // inst_memory[99] = 8'b00000001;
652
653 // inst_memory[100] = 8'b00100011;
654 // inst_memory[101] = 8'b10110000;
655 // inst_memory[102] = 8'b11000010;
656 // inst_memory[103] = 8'b00000001;

```



```

657
658 // inst_memory[104] =8'b00100011;
659 // inst_memory[105] =8'b00110000;
660 // inst_memory[106] =8'b11010011;
661 // inst_memory[107] =8'b00000001;
662
663 // inst_memory[108] =8'b10010011;
664 // inst_memory[109] =8'b10001001;
665 // inst_memory[110] =8'b00011001;
666 // inst_memory[111] =8'b00000000;
667
668 // inst_memory[112] =8'b11100011;
669 // inst_memory[113] =8'b00000100;
670 // inst_memory[114] =8'b00000000;
671 // inst_memory[115] =8'b11111100;
672
673 // inst_memory[116] =8'b00010011;
674 // inst_memory[117] =8'b00001001;
675 // inst_memory[118] =8'b00011001;
676 // inst_memory[119] =8'b00000000;
677
678 // inst_memory[120] =8'b11100011;
679 // inst_memory[121] =8'b00001100;
680 // inst_memory[122] =8'b00000000;
681 // inst_memory[123] =8'b11111010;
682
683 // inst_memory[124] =8'b01100011;
684 // inst_memory[125] =8'b00000010;
685 // inst_memory[126] =8'b00000000;
686 // inst_memory[127] =8'b00000000;
687
688 // inst_memory[128] =8'b00010011;
689 // inst_memory[129] =8'b00000000;
690 // inst_memory[130] =8'b00000000;
691 // inst_memory[131] =8'b00000000;
692
693 end
694 always@(Inst_Address)
695     Instruction = {inst_memory[Inst_Address+3],inst_memory[Inst_Address+2],
696     inst_memory[Inst_Address+1], inst_memory[Inst_Address]};
697
698
699
700
701 module Instruction_Parser(
702     input [31:0] instruction,
703     output [6:0] opcode, funct7,
704     output [4:0] rd , rs1 , rs2,
705     output [2:0] funct3
706 );
707 );
708
709 assign opcode = instruction[6:0];
710 assign rd = instruction[11:7];
711 assign funct3 = instruction[14:12];
712 assign rs1 = instruction[19:15];
713 assign rs2 = instruction[24:20];

```

```

714 assign funct7 = instruction[31:25];
715
716 endmodule
717
718
719
720
721 module MEM_WB(
722     input clk, reset,
723     input [63:0] read_data, Result,
724     input EXMEM_MemtoReg, EXMEM_RegWrite,
725     input [4:0] EXMEM_inst2,
726     output reg [63:0] MEMWB_read_data, MEMWB_Result,
727     output reg MEMWB_MemtoReg, MEMWB_RegWrite,
728     output reg [4:0] MEMWB_inst2
729 );
730     always @(posedge clk or reset)
731     begin
732         if(clk)
733             begin
734                 MEMWB_read_data = read_data;
735                 MEMWB_Result = Result;
736                 MEMWB_MemtoReg = EXMEM_MemtoReg;
737                 MEMWB_RegWrite = EXMEM_RegWrite;
738                 MEMWB_inst2 = EXMEM_inst2;
739
740             end
741         else
742             begin
743                 MEMWB_read_data = 0;
744                 MEMWB_Result = 0;
745                 MEMWB_MemtoReg = 0;
746                 MEMWB_RegWrite = 0;
747                 MEMWB_inst2 = 0;
748             end
749         end
750     endmodule
751
752
753 module mux2x1
754 (
755     input [63:0] a,b,
756     input s ,
757     output [63:0] data_out
758 );
759
760 assign data_out = s ? b : a;
761
762 endmodule
763
764
765 module Program_Counter
766 (
767     input clk, reset,
768     input [63:0] PC_In,
769     output reg [63:0] PC_Out
770 );
771

```

```

772 reg reset_force; // variable to force 0th value after reset
773
774 initial
775 PC_Out <= 64'd0;
776
777
778 always @(posedge clk or posedge reset) begin
779     if (reset || reset_force) begin
780         PC_Out = 64'd0;
781         reset_force <= 0;
782     end
783
784     // else if (!PCWrite) begin
785     //     PC_Out = PC_Out;
786     // end
787     else
788         PC_Out = PC_In;
789
790 end
791
792 always @(negedge reset) reset_force <= 1;
793
794 endmodule // Program_Counter
795
796
797
798 module registerFile(
799     input [63:0] WriteData,
800     input [4:0] RS1,
801     input [4:0] RS2,
802     input [4:0] RD,
803     input RegWrite, clk, reset,
804     output reg [63:0] ReadData1,
805     output reg [63:0] ReadData2
806 );
807 reg [63:0] Registers [31:0];
808 initial
809     begin
810         Registers[0] = 64'd 0;
811         Registers[1] = 64'd 0;
812         Registers[2] = 64'd 0;
813         Registers[3] = 64'd 0;
814         Registers[4] = 64'd 0;
815         Registers[5] = 64'd 0;
816         Registers[6] = 64'd 0;
817         Registers[7] = 64'd 0;
818         Registers[8] = 64'd 0;
819         Registers[9] = 64'd 0;
820         Registers[10] = 64'd 0;
821         Registers[11] = 64'd 0;
822         Registers[12] = 64'd 0;
823         Registers[13] = 64'd 0;
824         Registers[14] = 64'd 0;
825         Registers[15] = 64'd 0;
826         Registers[16] = 64'd 0;
827         Registers[17] = 64'd 0;
828         Registers[18] = 64'd 0;
829         Registers[19] = 64'd 0;

```

```

830     Registers[20] = 64'd 0;
831     Registers[21] = 64'd 0;
832     Registers[22] = 64'd 0;
833     Registers[23] = 64'd 0;
834     Registers[24] = 64'd 0;
835     Registers[25] = 64'd 0;
836     Registers[26] = 64'd 0;
837     Registers[27] = 64'd 0;
838     Registers[28] = 64'd 0;
839     Registers[29] = 64'd 0;
840     Registers[30] = 64'd 0;
841     Registers[31] = 64'd 0;
842     end
843 always @(negedge clk)
844     if(RegWrite)
845         begin
846             Registers[RD] = WriteData;
847         end
848     always @(*)
849         if(reset)
850             begin
851                 ReadData1 = 64'b0;
852                 ReadData2 = 64'b0;
853             end
854         else
855             begin
856                 ReadData1 = Registers[RS1];
857                 ReadData2 = Registers[RS2];
858             end
859 endmodule
860
861 module selector(
862     input branch, ZERO,
863     input [63:0] a, b,
864     input [2:0] funct3,
865     output reg sel
866 );
867
868 always@(*)
869 begin
870     if (branch == 1)
871         begin
872             case(funct3)
873                 3'b101: //bne
874                 begin
875                     if(branch == 1 & ZERO == 0)
876                         sel = 1;
877                 end
878                 3'b011: //beq
879                 begin
880                     if(branch == 1 & ZERO == 1)
881                         sel = 1;
882                 end
883                 3'b111: //bge
884                 begin
885                     if (a >= b)
886                         sel = 1;
887                 end

```

```

888         endcase
889     end
890     else
891         sel <= 0;
892 end
893 endmodule
894
895
896 module RISC_V_Processor(
897     input clk, reset
898 );
899
900 //Mux output
901 wire [63:0] PC_In_from_mux;
902 //Program counter output
903 wire [63:0] PC_Out;
904 //Adders outputs
905 wire [63:0] a1_out;
906 wire [63:0] a2_out;
907 //Input to Adder a1
908 wire [63:0] b_in = 64'd4;
909 //Output from IM
910 wire [31:0] Instruction;
911
912 //Outputs from RegisterFile
913 wire [63:0] ReadData1;
914 wire [63:0] ReadData2;
915 //Outputs from Control Unit
916 wire [1:0] ALUOp;
917 wire Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;
918 //Outputs from ALU Control
919 wire [3:0] Operation;
920 //Funct input to ALU_Control
921 wire [3:0] Funct;
922 assign Funct = {Instruction[30], Instruction[14:12]};
923 //Output from ALU
924 wire [63:0] Result_from_alu;
925 wire zero_output;
926 //Output from Data generator
927 wire [63:0] imm_data;
928 //Output from mux2
929 wire [63:0] out_from_mux2;
930 //Output from Data memory
931 wire [63:0] out_from_DM;
932 //Output from mux3
933 wire [63:0] out_from_mux3;
934 //Outputs from IF_ID
935 wire [31:0] IFID_instruction;
936 wire [63:0] IFID_PC_Out;
937 wire [4:0] rd;
938 wire [4:0] rs1;
939 wire [4:0] rs2;
940 wire [6:0] opcode;
941 wire [6:0] funct7;
942 wire [2:0] funct3;
943 //Outputs from ID_EX
944 wire [63:0] IDEX_PC_Out;
945 wire [63:0] IDEX_ReadData1;

```

```

946 wire [63:0] IDEX_ReadData2;
947 wire [63:0] IDEX_imm_data;
948 wire [3:0] IDEX_inst1;
949 wire [4:0] IDEX_inst2;
950 wire [1:0] IDEX_ALUOp;
951 wire IDEX_Branch, IDEX_MemRead, IDEX_MemtoReg, IDEX_MemWrite, IDEX_ALUSrc,
    IDEX_Regwrite;
952 //Input to Adder a2
953 wire [63:0] b_adder2;
954 assign b_adder2 = IDEX_imm_data << 1;
955 //Outputs from EX_MEM
956 wire [63:0] EXMEM_out;
957 wire EXMEM_ZERO;
958 wire [63:0] EXMEM_Result;
959 wire [63:0] EXMEM_ReadData2;
960 wire [4:0] EXMEM_inst2;
961 wire EXMEM_Branch, EXMEM_MemRead, EXMEM_MemtoReg, EXMEM_MemWrite,
    EXMEM_RegWrite;
962 //Outputs from MEM_WB
963 wire [63:0] MEMWB_read_data;
964 wire [63:0] MEMWB_Result;
965 wire MEMWB_MemtoReg, MEMWB_RegWrite;
966 wire [4:0] MEMWB_inst2;
967 //sel to mux1
968 wire PC_src;
969 // Outputs from Forwarding Unit
970     wire [1:0] FU_fwdA;
971     wire [1:0] FU_fwdB;
972 //Outputs from Triple MUX
973 wire [63:0] Res;
974 // wire [63:0] ResB;
975     wire [63:0] Resa;
976     wire [63:0] Resb;
977 //addi x5, x0, 3
978 IF_ID ifid
979     (.clk(clk),
980     .reset(reset),
981     .instruction(Instruction),
982     .PC_Out(PC_Out),
983     .IFID_instruction(IFID_instruction),
984     .IFID_PC_Out(IFID_PC_Out));
985
986 ID_EX idex
987     (.clk(clk),
988     .reset(reset),
989     .ALUOp(ALUOp),
990     .Branch(Branch),
991     .MemRead(MemRead),
992     .MemtoReg(MemtoReg),
993     .MemWrite(MemWrite),
994     .ALUSrc(ALUSrc),
995     .RegWrite(RegWrite),
996     .ReadData1(ReadData1),
997     .ReadData2(ReadData2),
998     .PC_Out(IFID_PC_Out),
999     .imm_data(imm_data),
1000     .inst1({IFID_instruction[30], IFID_instruction[14:12]}),
1001     .inst2(rd),

```

```

1002     . IDEX_Branch (IDEX_Branch),
1003     . IDEX_MemRead (IDEX_MemRead),
1004     . IDEX_MemtoReg (IDEX_MemtoReg),
1005     . IDEX_MemWrite (IDEX_MemWrite),
1006     . IDEX_ALUSrc (IDEX_ALUSrc),
1007     . IDEX_Regwrite (IDEX_Regwrite),
1008     . IDEX_ALUOp (IDEX_ALUOp),
1009     . IDEX_ReadData1 (IDEX_ReadData1),
1010     . IDEX_ReadData2 (IDEX_ReadData2),
1011     . IDEX_PC_Out (IDEX_PC_Out),
1012     . IDEX_imm_data (IDEX_imm_data),
1013     . IDEX_inst1 (IDEX_inst1),
1014     . IDEX_inst2 (IDEX_inst2));
1015
1016
1017
1018
1019 EX_MEM  exmem (.clk (clk),
1020              .reset (reset),
1021              . IDEX_Branch (IDEX_Branch),
1022              . IDEX_MemRead (IDEX_MemRead),
1023              . IDEX_MemtoReg (IDEX_MemtoReg),
1024              . IDEX_MemWrite (IDEX_MemWrite),
1025              . IDEX_Regwrite (IDEX_Regwrite),
1026              .out (a2_out),
1027              .ZERO (zero_output),
1028              .Result (Result_from_alu),
1029              . IDEX_ReadData2 (IDEX_ReadData2),
1030              . IDEX_inst2 (IDEX_inst2),
1031
1032              . EXMEM_out (EXMEM_out),
1033              . EXMEM_Result (EXMEM_Result),
1034              . EXMEM_ReadData2 (EXMEM_ReadData2),
1035              . EXMEM_inst2 (EXMEM_inst2),
1036              . EXMEM_ZERO (EXMEM_ZERO),
1037              . EXMEM_Branch (EXMEM_Branch),
1038              . EXMEM_MemRead (EXMEM_MemRead),
1039              . EXMEM_MemtoReg (EXMEM_MemtoReg),
1040              . EXMEM_MemWrite (EXMEM_MemWrite),
1041              . EXMEM_Regwrite (EXMEM_RegWrite));
1042
1043
1044 MEM_WB  memwb (.clk (clk),
1045              .reset (reset),
1046              . EXMEM_MemtoReg (EXMEM_MemtoReg),
1047              . EXMEM_RegWrite (EXMEM_RegWrite),
1048              .read_data (out_from_DM),
1049              .Result (EXMEM_Result),
1050              . EXMEM_inst2 (EXMEM_inst2),
1051
1052              . MEMWB_read_data (MEMWB_read_data),
1053              . MEMWB_Result (MEMWB_Result),
1054              . MEMWB_MemtoReg (MEMWB_MemtoReg),
1055              . MEMWB_RegWrite (MEMWB_RegWrite),
1056              . MEMWB_inst2 (MEMWB_inst2));
1057
1058 Instruction_Parser ip
1059     (.instruction (IFID_instruction),

```

```

1060         .rd(rd),
1061         .rs1(rs1),
1062         .rs2(rs2),
1063         .funct3(funct3),
1064         .funct7(funct7),
1065         .opcode(opcode));
1066
1067 Program_Counter pc
1068     (.clk(clk),
1069     .reset(reset),
1070     .PC_In(PC_In_from_mux),
1071     .PC_Out(PC_Out));
1072
1073 Adder a1
1074     (.a(PC_Out),
1075     .b(b_in),
1076     .out(a1_out));
1077
1078 Adder a2
1079     (.a(IDEX_PC_Out),
1080     .b(b_adder2),
1081     .out(a2_out));
1082
1083 mux2x1 mux1
1084     (.a(a1_out),
1085     .b(EXMEM_out),
1086     .s(PC_src),
1087     .data_out(PC_In_from_mux));
1088
1089 mux2x1 mux2
1090     (.a(IDEX_ReadData2),
1091     .b(IDEX_imm_data),
1092     .s(IDEX_ALUSrc),
1093     .data_out(out_from_mux2));
1094
1095 mux2x1 mux3
1096     (.b(MEMWB_read_data),
1097     .a(MEMWB_Result),
1098     .s(MEMWB_MemtoReg),
1099     .data_out(out_from_mux3));
1100
1101 Instruction_Memory im
1102     (.Inst_Address(PC_Out),
1103     .Instruction(Instruction));
1104
1105 registerFile rf
1106     (.WriteData(out_from_mux3),
1107     .RS1(rs1),
1108     .RS2(rs2),
1109     .RD(MEMWB_inst2),
1110     .clk(clk),
1111     .reset(reset),
1112     .RegWrite(MEMWB_RegWrite),
1113     .ReadData1(ReadData1),
1114     .ReadData2(ReadData2));
1115
1116 Control_Unit cu
1117     (.Opcode(opcode),

```



```

1118         .ALUOp(ALUOp),
1119         .Branch(Branch),
1120         .MemRead(MemRead),
1121         .MemtoReg(MemtoReg),
1122         .MemWrite(MemWrite),
1123         .ALUSrc(ALUSrc),
1124         .Regwrite(RegWrite));
1125
1126 ALU_Control aluc
1127     (.ALUOp(IDEX_ALUOp),
1128     .Funct(IDEX_inst1),
1129     .Operation(Operation));
1130
1131 data_generator dg
1132     (.instruction(IFID_instruction),
1133     .imm_data(imm_data));
1134
1135 alu_64 alu
1136     (.a(Resa),
1137     .b(Resb),
1138     .ALUOp(Operation),
1139     .Result(Result_from_alu),
1140     .ZERO(zero_output));
1141
1142 selector s
1143     (.branch(EXMEM_Branch),
1144     .ZERO(EXMEM_ZERO),
1145     .a(IDEX_ReadData1),
1146     .b(out_from_mux2),
1147     .funct3(funct3),
1148     .sel(PC_src)
1149     );
1150
1151 Data_Memory dm
1152     (.mem_addr(EXMEM_Result),
1153     .write_data(EXMEM_ReadData2),
1154     .clk(clk),
1155     .mem_write(EXMEM_MemWrite),
1156     .mem_read(EXMEM_MemRead),
1157     .read_data(out_from_DM));
1158
1159 Forwarding_Unit fu
1160     (.EXMEM_ReadData2(FU_EXMEM_ReadData2),
1161     .MEMWB_read_data(FU_MEMWB_read_data),
1162     .rs1(FU_IDEX_inst1),
1163     .rs2(FU_IDEX_inst2),
1164     .EXMEM_Regwrite(FU_EXMEM_Regwrite),
1165     .MEMWB_RegWrite(FU_MEMWB_RegWrite),
1166     .fwd_A(FU_fwdA),
1167     .fwd_B(FU_fwdB));
1168
1169
1170 MUX_Triple mux_for_a
1171 (
1172     .a(IDEX_ReadData1), //00
1173     .b(out_from_mux3), //01
1174     .c(EXMEM_Result),   //10
1175     .sel(FU_fwdA),

```

```

1176     .Res(Resa)
1177 );
1178
1179 MUX_Triple mux_for_b
1180 (
1181     .a(IDEX_ReadData2), //00
1182     .b(out_from_mux3), //01
1183     .c(EXMEM_Result),  //10
1184     .sel(FU_fwdB),
1185     .Res(Resb)
1186 );
1187
1188
1189
1190 always @(posedge clk)
1191     begin
1192         $monitor(
1193             "PC_In = ", PC_In_from_mux,
1194             ", PC_Out = ", PC_Out,
1195             ", Instruction = %b", Instruction,
1196             ", Opcode = %b", opcode,
1197             ", Funct3 = %b", funct3,
1198             ", rs1 = %d", rs1,
1199             ", rs2 = %d", rs2,
1200             ", rd = %d", rd,
1201             ", funct7 = %b", IFID_instruction[31:25],
1202             ", ALUOp = %b", IDEX_ALUOp,
1203             ", imm_data = %d", imm_data,
1204             ", Operation = %b", Operation
1205         );
1206     end
1207 endmodule

```

Test Bench:

```

1
2 module tb();
3
4 reg clk, reset;
5
6 RISC_V_Processor processor(.clk(clk), .reset(reset));
7
8 initial
9 begin
10     $dumpfile("dump.vcd");
11     $dumpvars();
12     clk = 1'd0;
13     reset = 1'd1;
14     #10
15     reset = 1'd0;
16     #500
17     $finish;
18 end
19
20 always #5 clk=~clk;
21
22 endmodule

```

Chapter 4

Code

4.1 EDA Links

[Task 1 EDA Link](#)

[Task 2 EDA Link](#)