

Design a 4-bit ALU

Group: 4 CSE460 Lab Section 9

ATHAR NOOR MOHAMMAD RAFEE

DEPT: CSE

ID: 20101396

Section: 9L

noor.mohammad.rafee@g.bracu.ac.bd

A.S.M MAHABUB SIDDIQUI

DEPT: CSE

ID: 20301040

Section: 9L

asm.mahabub.siddiqui@g.bracu.ac.bd

Ayon Das

DEPT: CSE

ID: 20301099

Section: 9L

ayon.das@g.bracu.ac.bd

MD. SAKIB

DEPT: CSE

ID: 20301180

Section: 9L

md.sakib1@g.bracu.ac.bd

MOHAMMED INZAM UL AZAM

DEPT: CSE

ID: 20101144

Section: 09L

mohammed.inzam.ul.azam@g.bracu.ac.bd

Abstract—This project presents the design and implementation of a 4-bit Arithmetic Logic Unit (ALU). The ALU performs arithmetic and logical operations on two 4-bit inputs and produces a 4-bit output. The design is implemented using Verilog hardware description language and simulated using timing function. The ALU supports basic arithmetic operations such as addition and subtraction, as well as logical operations such as ADD, NAND, and XNOR as per requirements of the project. Overall, this project demonstrates the design and implementation of a simple but functional sequential ALU using Verilog HDL.

Index Terms—ALU, Verilog HDL, opcode, timing diagram, SUB, XNOR, NAND, ADD, RESET

I. INTRODUCTION

This report presents the design and implementation of a 4-bit ALU using Verilog HDL and Quartus II software. The ALU was designed to perform various arithmetic and logical operations such as **ADDITION**, **SUBTRACTION**, bitwise **AND**, bitwise **NAND**, and bitwise **XNOR**. The design consists of various modules such as the Adder, Subtractor, and logic gates which were generated based on the verilog code. In this report, we provide a detailed description of the design and implementation process, including the Verilog code for each module and the timing diagram for verification. We also discuss the challenges encountered during the design process and how they were overcome. Finally, we present the results of the hardware testing, demonstrating that the ALU is capable of performing the desired operations accurately and efficiently. The design of a 4-bit ALU is an essential component in digital circuit design, and it is a fundamental building block in many larger circuits and VLSI design.

II. FINITE STATE MACHINE DESIGN AND IMPLEMENTATION

Finite State Machine (FSM) is a model for designing sequential logic circuits, where the circuit's behavior is determined by a finite number of states, inputs and outputs. In this case, the FSM is designed to implement *five* different

operations, namely RESET, XNOR, NAND, SUB, and ADD on two 4-bit inputs A and B. The way we coded the Verilog code represents the implementation of the FSM, which is designed to perform the above-mentioned arithmetic and logical operations on the given input values. From an high level perspective, The FSM has Four states, which are encoded as 2-bit values, as follows:

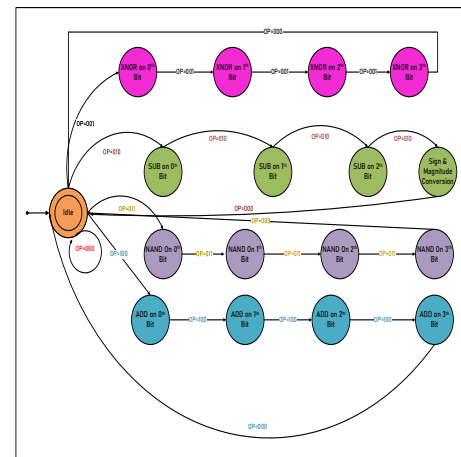


Fig. 1. FSM Diagram

- State 0 (2'b00): In this state, the circuit performs the selected operation on the first bit of the input values and transitions to the next state.
- State 1 (2'b01): In this state, the circuit performs the selected operation on the second bit of the input values and transitions to the next state.
- State 2 (2'b10): In this state, the circuit performs the selected operation on the third bit of the input values and transitions to the next state.
- State 3 (2'b11): In this state, the circuit performs the selected operation on the fourth and most significant bit

of the input values and transitions back to the initial state.

Before transition, it also sets the values of *zero* flag, *sign* flag and *carry* flag.

Things are checked and done slightly different based on the *opcode*. It can be observed from the above Fig 1 clearly. The four different operations are implemented using a *case* statement with *opcode* as the selector. Each operation *case* statement contains the logic required to perform the operation on the given input values, and update the output values of the circuit accordingly. For example, for the **ADD** operation, the code first calculates the SUM of the LSBs of the input values, adds the carry value to it (initially 0), and assigns the SUM and the carry value to the output register **C**. Then, it updates the *zero* flag, which is set to 1 if the output is 0, and transitions to the next state which is **IDLE** state that we can see from Figure 1. The outputs of the circuit include **C**, which stores the result of the operation, *carr*, which is the **carry** bit generated during addition or subtraction, *sign*, which is the sign bit of the output value, and *zero*, which is set to 1 if the output is 0000. Overall, the FSM implementation allows the circuit to perform different arithmetic and logical operations on the given input values, and update the output values based on the operation performed.

III. VERIFICATION

After implementing the verilog code. We used the timing function to verify the working of the code. Below, we are attaching the screenshots from the timing diagram

A. ADD Operation:

Below figure 2 shows the **ADD** operation between two binary $A = 1111$ and $B = 1111$ numbers where the result is stored in register **C** sequentially in each clock cycles.

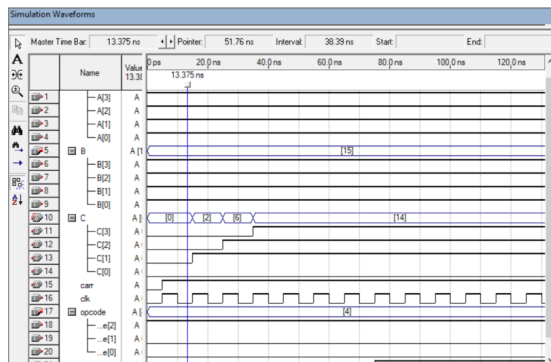


Fig. 2. Timing Diagram for ADD Operation

B. SUB Operation

Below figure 3 shows the **SUB** operation between two binary $A = 0111$ and $B = 0111$ where the output $C = 0000$. As we are working with signed number, the numbers are represented as **2s complement**.

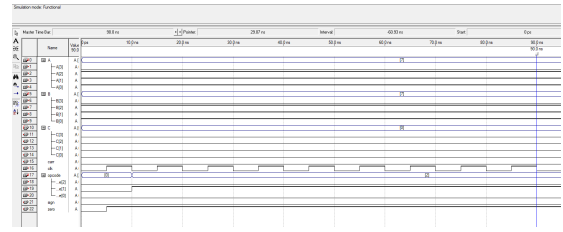


Fig. 3. Timing Diagram for Sub Operation

NAND and **XNOR** operation are pretty much straight forward. All we had to do is put the equation in the verilog and then the operation performed as expected. Below timing diagram, those operations are attached.

C. NAND Operation

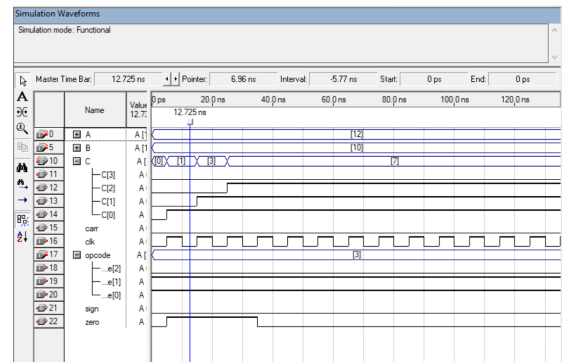


Fig. 4. Timing Diagram for NAND Operation

D. XNOR Operation

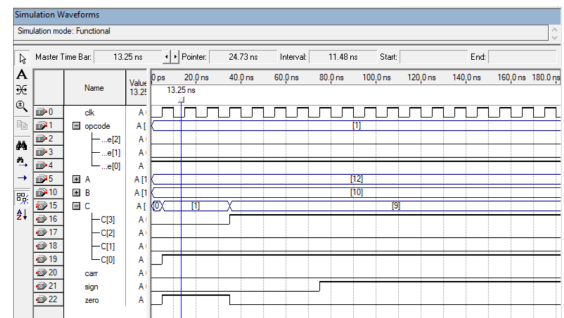


Fig. 5. Timing Diagram for XNOR Operation

E. Multiple Operation With Reset

The below attached Figure 6 shows multiple Operations with **SET** and **RESET** functionality. At first the **ADD** operation is performed between binary number 0001 and 0111. After that, the **opcode** is changed to **RESET** which is 000 during time 50ns to 60ns. Next the **opcode** was changed to 010 and performed **SUB** operation during the next 8 clock cycles or two **BUS** cycles. After that the mandatory **RESET**

and then **NAND** operation for another 4 cycles as the *opcode* was changed to *011*.

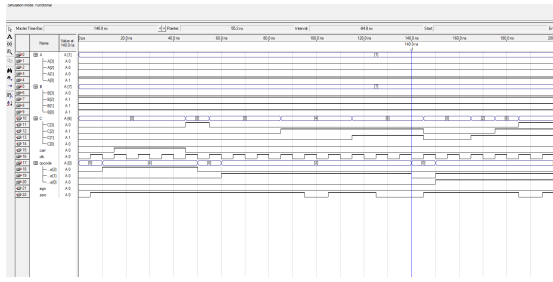


Fig. 6. Timing Daigram with reset

IV. CONCLUSION

In conclusion, we have successfully designed and implemented a 4-bit ALU using finite state machines and *Verilog HDL*. We started by defining the project requirements and selecting the appropriate operations to be implemented. Then, we designed the FSM with four states and carefully defined the transitions and outputs for each state. We also implemented the FSM using Verilog HDL and simulated the design using timing diagram to verify its functionality. The simulation results show that our design works correctly for all the selected operations and input combinations. Finally, this project not only provided hands-on experience with Verilog HDL programming and digital circuit design, but also reinforced the importance of a systematic approach to problem-solving and the importance of utilizing efficient design strategies.

APPENDIX

A. Verilog HDL Code

```

1 module project(input clk, input [3:0] A,
2   input [3:0] B,
3   input [2:0] opcode, output reg [3:0] C,
4   output reg carr, output reg sign, output
5   reg zero);
6 // Will be using state to indicate state of
7 // the machine.
8 reg [1:0] state = 0;
9 // Negative numbers will be represented in 2
10 // s complement.
11 reg signed [3:0] sub_result;
12 reg signed [3:0] sub_a;
13 reg signed [3:0] sub_b;
14 always @ (posedge clk) begin
15   case (state)
16     2'b00: begin
17       case (opcode)
18         3'b000: begin
19           C <= 4'b0000; //RESET
20           operation
21           carr <= 1'b0;
22           sign <= 1'b0;
23           zero <= 1'b1;
24         end
25         3'b001: begin //XNOR
26           operation on LSBs
27           C[0] <= ~(A[0] ^ B[0]);
28         end
29       endcase
30     end
31     2'b01: begin
32       case (opcode)
33         3'b010: begin //SUB
34           operation on next bit
35           sub_a <= A;
36           sub_b <= B;
37           sub_result <= sub_b -
38             sub_a;
39           C[1] <= sub_result[1];
40           zero <= C[1] == 1'b0;
41         end
42         3'b011: begin //NAND
43           operation on next bit
44           C[1] <= ~(A[1] & B[1]);
45           zero <= zero & (C[1] ==
46             1'b0);
47         end
48         3'b100: begin //ADD
49           operation on next bit
50           {carr, C[1]} <= A[1] + B
51             [1] + carr;
52           zero <= zero & (C[1] ==
53             1'b0);
54         end
55       endcase
56     end
57     2'b10: begin
58       case (opcode)
59         3'b101: begin //SUB
60           operation on next bit
61           sub_a <= A;
62           sub_b <= B;
63           sub_result <= sub_b -
64             sub_a;
65           C[1] <= sub_result[1];
66           zero <= C[1] == 1'b0;
67         end
68         3'b110: begin //NAND
69           operation on next bit
70           C[1] <= ~(A[1] & B[1]);
71           zero <= zero & (C[1] ==
72             1'b0);
73         end
74         3'b111: begin //ADD
75           operation on next bit
76           {carr, C[1]} <= A[1] + B
77             [1] + carr;
78           zero <= zero & (C[1] ==
79             1'b0);
80         end
81       endcase
82     end
83   endcase
84   state <= state + 1;
85 end

```

```

22   zero <= C[0] == 1'b0;
23 end
24 3'b010: begin //SUB
25   operation on LSBs
26   sub_a <= A;
27   sub_b <= B;
28   sub_result <= sub_b -
29     sub_a;
30   C[0] <= sub_result[0];
31   zero <= C[0] == 1'b0;
32 end
33 3'b011: begin //NAND
34   operation on LSBs
35   C[0] <= ~(A[0] & B[0]);
36   zero <= C[0] == 1'b0;
37 end
38 3'b100: begin //ADD
39   operation on LSBs
40   {carr, C[0]} <= A[0] + B
41     [0];
42   zero <= C[0] == 1'b0;
43 end
44 endcase
45 if (opcode != 3'b000) begin
46   state <= 2'b01;
47 end
48 2'b01: begin
49   case (opcode)
50     3'b001: begin //XNOR
51       operation on next bit
52       C[1] <= ~(A[1] ^ B[1]);
53       zero <= zero & (C[1] ==
54         1'b0);
55     end
56     3'b010: begin //SUB
57       operation on next bit
58       sub_a <= A;
59       sub_b <= B;
60       sub_result <= sub_b -
61         sub_a;
62       C[1] <= sub_result[1];
63       zero <= C[1] == 1'b0;
64     end
65     3'b011: begin //NAND
66       operation on next bit
67       C[1] <= ~(A[1] & B[1]);
68       zero <= zero & (C[1] ==
69         1'b0);
70     end
71     3'b100: begin //ADD
72       operation on next bit
73       {carr, C[1]} <= A[1] + B
74         [1] + carr;
75       zero <= zero & (C[1] ==
76         1'b0);
77     end
78   endcase
79   state <= 2'b10;
80 end
81 2'b10: begin
82   case (opcode)
83     3'b101: begin //SUB
84       operation on next bit
85       sub_a <= A;
86       sub_b <= B;
87       sub_result <= sub_b -
88         sub_a;
89       C[1] <= sub_result[1];
90       zero <= C[1] == 1'b0;
91     end
92     3'b110: begin //NAND
93       operation on next bit
94       C[1] <= ~(A[1] & B[1]);
95       zero <= zero & (C[1] ==
96         1'b0);
97     end
98     3'b111: begin //ADD
99       operation on next bit
100      {carr, C[1]} <= A[1] + B
101        [1] + carr;
102      zero <= zero & (C[1] ==
103        1'b0);
104    end
105  endcase
106  state <= 2'b10;
107 end
108 2'b10: begin
109   case (opcode)
110     3'b101: begin //SUB
111       operation on next bit
112       sub_a <= A;
113       sub_b <= B;
114       sub_result <= sub_b -
115         sub_a;
116       C[1] <= sub_result[1];
117       zero <= C[1] == 1'b0;
118     end
119     3'b110: begin //NAND
120       operation on next bit
121       C[1] <= ~(A[1] & B[1]);
122       zero <= zero & (C[1] ==
123         1'b0);
124     end
125     3'b111: begin //ADD
126       operation on next bit
127       {carr, C[1]} <= A[1] + B
128         [1] + carr;
129       zero <= zero & (C[1] ==
130         1'b0);
131     end
132  endcase
133  state <= 2'b10;
134 end
135 2'b10: begin
136   case (opcode)
137     3'b101: begin //SUB
138       operation on next bit
139       sub_a <= A;
140       sub_b <= B;
141       sub_result <= sub_b -
142         sub_a;
143       C[1] <= sub_result[1];
144       zero <= C[1] == 1'b0;
145     end
146     3'b110: begin //NAND
147       operation on next bit
148       C[1] <= ~(A[1] & B[1]);
149       zero <= zero & (C[1] ==
150         1'b0);
151     end
152     3'b111: begin //ADD
153       operation on next bit
154       {carr, C[1]} <= A[1] + B
155         [1] + carr;
156       zero <= zero & (C[1] ==
157         1'b0);
158     end
159  endcase
160  state <= 2'b10;
161 end
162 2'b10: begin
163   case (opcode)
164     3'b101: begin //SUB
165       operation on next bit
166       sub_a <= A;
167       sub_b <= B;
168       sub_result <= sub_b -
169         sub_a;
170       C[1] <= sub_result[1];
171       zero <= C[1] == 1'b0;
172     end
173     3'b110: begin //NAND
174       operation on next bit
175       C[1] <= ~(A[1] & B[1]);
176       zero <= zero & (C[1] ==
177         1'b0);
178     end
179     3'b111: begin //ADD
180       operation on next bit
181       {carr, C[1]} <= A[1] + B
182         [1] + carr;
183       zero <= zero & (C[1] ==
184         1'b0);
185     end
186  endcase
187  state <= 2'b10;
188 end
189 2'b10: begin
190   case (opcode)
191     3'b101: begin //SUB
192       operation on next bit
193       sub_a <= A;
194       sub_b <= B;
195       sub_result <= sub_b -
196         sub_a;
197       C[1] <= sub_result[1];
198       zero <= C[1] == 1'b0;
199     end
200     3'b110: begin //NAND
201       operation on next bit
202       C[1] <= ~(A[1] & B[1]);
203       zero <= zero & (C[1] ==
204         1'b0);
205     end
206     3'b111: begin //ADD
207       operation on next bit
208       {carr, C[1]} <= A[1] + B
209         [1] + carr;
210       zero <= zero & (C[1] ==
211         1'b0);
212     end
213  endcase
214  state <= 2'b10;
215 end
216 2'b10: begin
217   case (opcode)
218     3'b101: begin //SUB
219       operation on next bit
220       sub_a <= A;
221       sub_b <= B;
222       sub_result <= sub_b -
223         sub_a;
224       C[1] <= sub_result[1];
225       zero <= C[1] == 1'b0;
226     end
227     3'b110: begin //NAND
228       operation on next bit
229       C[1] <= ~(A[1] & B[1]);
230       zero <= zero & (C[1] ==
231         1'b0);
232     end
233     3'b111: begin //ADD
234       operation on next bit
235       {carr, C[1]} <= A[1] + B
236         [1] + carr;
237       zero <= zero & (C[1] ==
238         1'b0);
239     end
240  endcase
241  state <= 2'b10;
242 end
243 2'b10: begin
244   case (opcode)
245     3'b101: begin //SUB
246       operation on next bit
247       sub_a <= A;
248       sub_b <= B;
249       sub_result <= sub_b -
250         sub_a;
251       C[1] <= sub_result[1];
252       zero <= C[1] == 1'b0;
253     end
254     3'b110: begin //NAND
255       operation on next bit
256       C[1] <= ~(A[1] & B[1]);
257       zero <= zero & (C[1] ==
258         1'b0);
259     end
260     3'b111: begin //ADD
261       operation on next bit
262       {carr, C[1]} <= A[1] + B
263         [1] + carr;
264       zero <= zero & (C[1] ==
265         1'b0);
266     end
267  endcase
268  state <= 2'b10;
269 end
270 2'b10: begin
271   case (opcode)
272     3'b101: begin //SUB
273       operation on next bit
274       sub_a <= A;
275       sub_b <= B;
276       sub_result <= sub_b -
277         sub_a;
278       C[1] <= sub_result[1];
279       zero <= C[1] == 1'b0;
280     end
281     3'b110: begin //NAND
282       operation on next bit
283       C[1] <= ~(A[1] & B[1]);
284       zero <= zero & (C[1] ==
285         1'b0);
286     end
287     3'b111: begin //ADD
288       operation on next bit
289       {carr, C[1]} <= A[1] + B
290         [1] + carr;
291       zero <= zero & (C[1] ==
292         1'b0);
293     end
294  endcase
295  state <= 2'b10;
296 end
297 2'b10: begin
298   case (opcode)
299     3'b101: begin //SUB
300       operation on next bit
301       sub_a <= A;
302       sub_b <= B;
303       sub_result <= sub_b -
304         sub_a;
305       C[1] <= sub_result[1];
306       zero <= C[1] == 1'b0;
307     end
308     3'b110: begin //NAND
309       operation on next bit
310       C[1] <= ~(A[1] & B[1]);
311       zero <= zero & (C[1] ==
312         1'b0);
313     end
314     3'b111: begin //ADD
315       operation on next bit
316       {carr, C[1]} <= A[1] + B
317         [1] + carr;
318       zero <= zero & (C[1] ==
319         1'b0);
320     end
321  endcase
322  state <= 2'b10;
323 end
324 2'b10: begin
325   case (opcode)
326     3'b101: begin //SUB
327       operation on next bit
328       sub_a <= A;
329       sub_b <= B;
330       sub_result <= sub_b -
331         sub_a;
332       C[1] <= sub_result[1];
333       zero <= C[1] == 1'b0;
334     end
335     3'b110: begin //NAND
336       operation on next bit
337       C[1] <= ~(A[1] & B[1]);
338       zero <= zero & (C[1] ==
339         1'b0);
340     end
341     3'b111: begin //ADD
342       operation on next bit
343       {carr, C[1]} <= A[1] + B
344         [1] + carr;
345       zero <= zero & (C[1] ==
346         1'b0);
347     end
348  endcase
349  state <= 2'b10;
350 end
351 2'b10: begin
352   case (opcode)
353     3'b101: begin //SUB
354       operation on next bit
355       sub_a <= A;
356       sub_b <= B;
357       sub_result <= sub_b -
358         sub_a;
359       C[1] <= sub_result[1];
360       zero <= C[1] == 1'b0;
361     end
362     3'b110: begin //NAND
363       operation on next bit
364       C[1] <= ~(A[1] & B[1]);
365       zero <= zero & (C[1] ==
366         1'b0);
367     end
368     3'b111: begin //ADD
369       operation on next bit
370       {carr, C[1]} <= A[1] + B
371         [1] + carr;
372       zero <= zero & (C[1] ==
373         1'b0);
374     end
375  endcase
376  state <= 2'b10;
377 end
378 2'b10: begin
379   case (opcode)
380     3'b101: begin //SUB
381       operation on next bit
382       sub_a <= A;
383       sub_b <= B;
384       sub_result <= sub_b -
385         sub_a;
386       C[1] <= sub_result[1];
387       zero <= C[1] == 1'b0;
388     end
389     3'b110: begin //NAND
390       operation on next bit
391       C[1] <= ~(A[1] & B[1]);
392       zero <= zero & (C[1] ==
393         1'b0);
394     end
395     3'b111: begin //ADD
396       operation on next bit
397       {carr, C[1]} <= A[1] + B
398         [1] + carr;
399       zero <= zero & (C[1] ==
400         1'b0);
401     end
402  endcase
403  state <= 2'b10;
404 end
405 2'b10: begin
406   case (opcode)
407     3'b101: begin //SUB
408       operation on next bit
409       sub_a <= A;
410       sub_b <= B;
411       sub_result <= sub_b -
412         sub_a;
413       C[1] <= sub_result[1];
414       zero <= C[1] == 1'b0;
415     end
416     3'b110: begin //NAND
417       operation on next bit
418       C[1] <= ~(A[1] & B[1]);
419       zero <= zero & (C[1] ==
420         1'b0);
421     end
422     3'b111: begin //ADD
423       operation on next bit
424       {carr, C[1]} <= A[1] + B
425         [1] + carr;
426       zero <= zero & (C[1] ==
427         1'b0);
428     end
429  endcase
430  state <= 2'b10;
431 end
432 2'b10: begin
433   case (opcode)
434     3'b101: begin //SUB
435       operation on next bit
436       sub_a <= A;
437       sub_b <= B;
438       sub_result <= sub_b -
439         sub_a;
440       C[1] <= sub_result[1];
441       zero <= C[1] == 1'b0;
442     end
443     3'b110: begin //NAND
444       operation on next bit
445       C[1] <= ~(A[1] & B[1]);
446       zero <= zero & (C[1] ==
447         1'b0);
448     end
449     3'b111: begin //ADD
450       operation on next bit
451       {carr, C[1]} <= A[1] + B
452         [1] + carr;
453       zero <= zero & (C[1] ==
454         1'b0);
455     end
456  endcase
457  state <= 2'b10;
458 end
459 2'b10: begin
460   case (opcode)
461     3'b101: begin //SUB
462       operation on next bit
463       sub_a <= A;
464       sub_b <= B;
465       sub_result <= sub_b -
466         sub_a;
467       C[1] <= sub_result[1];
468       zero <= C[1] == 1'b0;
469     end
470     3'b110: begin //NAND
471       operation on next bit
472       C[1] <= ~(A[1] & B[1]);
473       zero <= zero & (C[1] ==
474         1'b0);
475     end
476     3'b111: begin //ADD
477       operation on next bit
478       {carr, C[1]} <= A[1] + B
479         [1] + carr;
480       zero <= zero & (C[1] ==
481         1'b0);
482     end
483  endcase
484  state <= 2'b10;
485 end
486 2'b10: begin
487   case (opcode)
488     3'b101: begin //SUB
489       operation on next bit
490       sub_a <= A;
491       sub_b <= B;
492       sub_result <= sub_b -
493         sub_a;
494       C[1] <= sub_result[1];
495       zero <= C[1] == 1'b0;
496     end
497     3'b110: begin //NAND
498       operation on next bit
499       C[1] <= ~(A[1] & B[1]);
500       zero <= zero & (C[1] ==
501         1'b0);
502     end
503     3'b111: begin //ADD
504       operation on next bit
505       {carr, C[1]} <= A[1] + B
506         [1] + carr;
507       zero <= zero & (C[1] ==
508         1'b0);
509     end
510  endcase
511  state <= 2'b10;
512 end
513 2'b10: begin
514   case (opcode)
515     3'b101: begin //SUB
516       operation on next bit
517       sub_a <= A;
518       sub_b <= B;
519       sub_result <= sub_b -
520         sub_a;
521       C[1] <= sub_result[1];
522       zero <= C[1] == 1'b0;
523     end
524     3'b110: begin //NAND
525       operation on next bit
526       C[1] <= ~(A[1] & B[1]);
527       zero <= zero & (C[1] ==
528         1'b0);
529     end
530     3'b111: begin //ADD
531       operation on next bit
532       {carr, C[1]} <= A[1] + B
533         [1] + carr;
534       zero <= zero & (C[1] ==
535         1'b0);
536     end
537  endcase
538  state <= 2'b10;
539 end
540 2'b10: begin
541   case (opcode)
542     3'b101: begin //SUB
543       operation on next bit
544       sub_a <= A;
545       sub_b <= B;
546       sub_result <= sub_b -
547         sub_a;
548       C[1] <= sub_result[1];
549       zero <= C[1] == 1'b0;
550     end
551     3'b110: begin //NAND
552       operation on next bit
553       C[1] <= ~(A[1] & B[1]);
554       zero <= zero & (C[1] ==
555         1'b0);
556     end
557     3'b111: begin //ADD
558       operation on next bit
559       {carr, C[1]} <= A[1] + B
560         [1] + carr;
561       zero <= zero & (C[1] ==
562         1'b0);
563     end
564  endcase
565  state <= 2'b10;
566 end
567 2'b10: begin
568   case (opcode)
569     3'b101: begin //SUB
570       operation on next bit
571       sub_a <= A;
572       sub_b <= B;
573       sub_result <= sub_b -
574         sub_a;
575       C[1] <= sub_result[1];
576       zero <= C[1] == 1'b0;
577     end
578     3'b110: begin //NAND
579       operation on next bit
580       C[1] <= ~(A[1] & B[1]);
581       zero <= zero & (C[1] ==
582         1'b0);
583     end
584     3'b111: begin //ADD
585       operation on next bit
586       {carr, C[1]} <= A[1] + B
587         [1] + carr;
588       zero <= zero & (C[1] ==
589         1'b0);
590     end
591  endcase
592  state <= 2'b10;
593 end
594 2'b10: begin
595   case (opcode)
596     3'b101: begin //SUB
597       operation on next bit
598       sub_a <= A;
599       sub_b <= B;
600       sub_result <= sub_b -
601         sub_a;
602       C[1] <= sub_result[1];
603       zero <= C[1] == 1'b0;
604     end
605     3'b110: begin //NAND
606       operation on next bit
607       C[1] <= ~(A[1] & B[1]);
608       zero <= zero & (C[1] ==
609         1'b0);
610     end
611     3'b111: begin //ADD
612       operation on next bit
613       {carr, C[1]} <= A[1] + B
614         [1] + carr;
615       zero <= zero & (C[1] ==
616         1'b0);
617     end
618  endcase
619  state <= 2'b10;
620 end
621 2'b10: begin
622   case (opcode)
623     3'b101: begin //SUB
624       operation on next bit
625       sub_a <= A;
626       sub_b <= B;
627       sub_result <= sub_b -
628         sub_a;
629       C[1] <= sub_result[1];
630       zero <= C[1] == 1'b0;
631     end
632     3'b110: begin //NAND
633       operation on next bit
634       C[1] <= ~(A[1] & B[1]);
635       zero <= zero & (C[1] ==
636         1'b0);
637     end
638     3'b111: begin //ADD
639       operation on next bit
640       {carr, C[1]} <= A[1] + B
641         [1] + carr;
642       zero <= zero & (C[1] ==
643         1'b0);
644     end
645  endcase
646  state <= 2'b10;
647 end
648 2'b10: begin
649   case (opcode)
650     3'b101: begin //SUB
651       operation on next bit
652       sub_a <= A;
653       sub_b <= B;
654       sub_result <= sub_b -
655         sub_a;
656       C[1] <= sub_result[1];
657       zero <= C[1] == 1'b0;
658     end
659     3'b110: begin //NAND
660       operation on next bit
661       C[1] <= ~(A[1] & B[1]);
662       zero <= zero & (C[1] ==
663         1'b0);
664     end
665     3'b111: begin //ADD
666       operation on next bit
667       {carr, C[1]} <= A[1] + B
668         [1] + carr;
669       zero <= zero & (C[1] ==
670         1'b0);
671     end
672  endcase
673  state <= 2'b10;
674 end
675 2'b10: begin
676   case (opcode)
677     3'b101: begin //SUB
678       operation on next bit
679       sub_a <= A;
680       sub_b <= B;
681       sub_result <= sub_b -
682         sub_a;
683       C[1] <= sub_result[1];
684       zero <= C[1] == 1'b0;
685     end
686     3'b110: begin //NAND
687       operation on next bit
688       C[1] <= ~(A[1] & B[1]);
689       zero <= zero & (C[1] ==
690         1'b0);
691     end
692     3'b111: begin //ADD
693       operation on next bit
694       {carr, C[1]} <= A[1] + B
695         [1] + carr;
696       zero <= zero & (C[1] ==
697         1'b0);
698     end
699  endcase
700  state <= 2'b10;
701 end
702 2'b10: begin
703   case (opcode)
704     3'b101: begin //SUB
705       operation on next bit
706       sub_a <= A;
707       sub_b <= B;
708       sub_result <= sub_b -
709         sub_a;
710       C[1] <= sub_result[1];
711       zero <= C[1] == 1'b0;
712     end
713     3'b110: begin //NAND
714       operation on next bit
715       C[1] <= ~(A[1] & B[1]);
716       zero <= zero & (C[1] ==
717         1'b0);
718     end
719     3'b111: begin //ADD
720       operation on next bit
721       {carr, C[1]} <= A[1] + B
722         [1] + carr;
723       zero <= zero & (C[1] ==
724         1'b0);
725     end
726  endcase
727  state <= 2'b10;
728 end
729 2'b10: begin
730   case (opcode)
731     3'b101: begin //SUB
732       operation on next bit
733       sub_a <= A;
734       sub_b <= B;
735       sub_result <= sub_b -
736         sub_a;
737       C[1] <= sub_result[1];
738       zero <= C[1] == 1'b0;
739     end
740     3'b110: begin //NAND
741       operation on next bit
742       C[1] <= ~(A[1] & B[1]);
743       zero <= zero & (C[1] ==
744         1'b0);
745     end
746     3'b111: begin //ADD
747       operation on next bit
748       {carr, C[1]} <= A[1] + B
749         [1] + carr;
750       zero <= zero & (C[1] ==
751         1'b0);
752     end
753  endcase
754  state <= 2'b10;
755 end
756 2'b10: begin
757   case (opcode)
758     3'b101: begin //SUB
759       operation on next bit
760       sub_a <= A;
761       sub_b <= B;
762       sub_result <= sub_b -
763         sub_a;
764       C[1] <= sub_result[1];
765       zero <= C[1] == 1'b0;
766     end
767     3'b110: begin //NAND
768       operation on next bit
769       C[1] <= ~(A[1] & B[1]);
770       zero <= zero & (C[1] ==
771         1'b0);
772     end
773     3'b111: begin //ADD
774       operation on next bit
775       {carr, C[1]} <= A[1] + B
776         [1] + carr;
777       zero <= zero & (C[1] ==
778         1'b0);
779     end
780  endcase
781  state <= 2'b10;
782 end
783 2'b10: begin
784   case (opcode)
785     3'b101: begin //SUB
786       operation on next bit
787       sub_a <= A;
788       sub_b <= B;
789       sub_result <= sub_b -
790         sub_a;
791       C[1] <= sub_result[1];
792       zero <= C[1] == 1'b0;
793     end
794     3'b110: begin //NAND
795       operation on next bit
796       C[1] <= ~(A[1] & B[1]);
797       zero <= zero & (C[1] ==
798         1'b0);
799     end
800     3'b111: begin //ADD
801       operation on next bit
802       {carr, C[1]} <= A[1] + B
803         [1] + carr;
804       zero <= zero & (C[1] ==
805         1'b0);
806     end
807  endcase
808  state <= 2'b10;
809 end
810 2'b10: begin
811   case (opcode)
812     3'b101: begin //SUB
813       operation on next bit
814       sub_a <= A;
815       sub_b <= B;
816       sub_result <= sub_b -
817         sub_a;
818       C[1] <= sub_result[1];
819       zero <= C[1] == 1'b0;
820     end
821     3'b110: begin //NAND
822       operation on next bit
823       C[1] <= ~(A[1] & B[1]);
824       zero <= zero & (C[1] ==
825         1'b0);
826     end
827     3'b111: begin //ADD
828       operation on next bit
829       {carr, C[1]} <= A[1] + B
830         [1] + carr;
831       zero <= zero & (C[1] ==
832         1'b0);
833     end
834  endcase
835  state <= 2'b10;
836 end
837 2'b10: begin
838   case (opcode)
839     3'b101: begin //SUB
840       operation on next bit
841       sub_a <= A;
842       sub_b <= B;
843       sub_result <= sub_b -
844         sub_a;
845       C[1] <= sub_result[1];
846       zero <= C[1] == 1'b0;
847     end
848     3'b110: begin //NAND
849       operation on next bit
850       C[1] <= ~(A[1] & B[1]);
851       zero <= zero & (C[1] ==
852         1'b0);
853     end
854     3'b111: begin //ADD
855       operation on next bit
856       {carr, C[1]} <= A[1] + B
857         [1] + carr;
858       zero <= zero & (C[1] ==
859         1'b0);
860     end
861  endcase
862  state <= 2'b10;
863 end
864 2'b10: begin
865   case (opcode)
866     3'b101: begin //SUB
867       operation on next bit
868       sub_a <= A;
869       sub_b <= B;
870       sub_result <= sub_b -
871         sub_a;
872       C[1] <= sub_result[1];
873       zero <= C[1] == 1'b0;
874     end
875
```

```

74      3'b001: begin //XNOR
              operation on next bit
75              C[2] <= ~(A[2] ^ B[2]);
76              zero <= zero & (C[2] ==
                  1'b0);
77      end
78      3'b010: begin //SUB
              operation on next bit
79              sub_a <= A;
80              sub_b <= B;
81              sub_result <= sub_b -
                  sub_a;
82              C[2] <= sub_result[2];
83              zero <= C[2] == 1'b0;
84      end
85      3'b011: begin //NAND
              operation on next bit
86              C[2] <= ~(A[2] & B[2]);
87              zero <= zero & (C[2] ==
                  1'b0);
88      end
89      3'b100: begin //ADD
              operation on next bit
90              {carr, C[2]} <= A[2] + B
                  [2] + carr;
91              zero <= zero & (C[2] ==
                  1'b0);
92      end
93
94      endcase
95      state <= 2'b11;
96  end
97  2'b11: begin
98      case (opcode)
99          3'b001: begin //XNOR
                  operation on MSBs
100                 C[3] <= ~(A[3] ^ B[3]);
101                 sign <= C[3];
102                 zero <= C == 4'b0000;
103             end
104          3'b010: begin //SUB
                  operation on MSBs
105                 sub_a <= A;
106                 sub_b <= B;
107                 {carr, sub_result} <=
                    sub_b - sub_a;
108                 C[3] <= sub_result[3];
109                 zero <= C == 4'b0000;
110             end
111          3'b011: begin //NAND
                  operation on MSBs
112                 C[3] <= ~(A[3] & B[3]);
113                 sign <= C[3];
114                 zero <= C == 4'b0000;
115             end
116          3'b100: begin //ADD
                  operation on MSBs
117                 {carr, C[3]} <= A[3] + B
                    [3] + carr;
118                 sign <= C[3];
119                 zero <= C == 4'b0000;
120             end
121      endcase
122      state <= 2'b00;
123  end
124 endcase
125

```

```

126 end
127 endmodule

```

Listing 1. Verilog code for 4-bit ALU