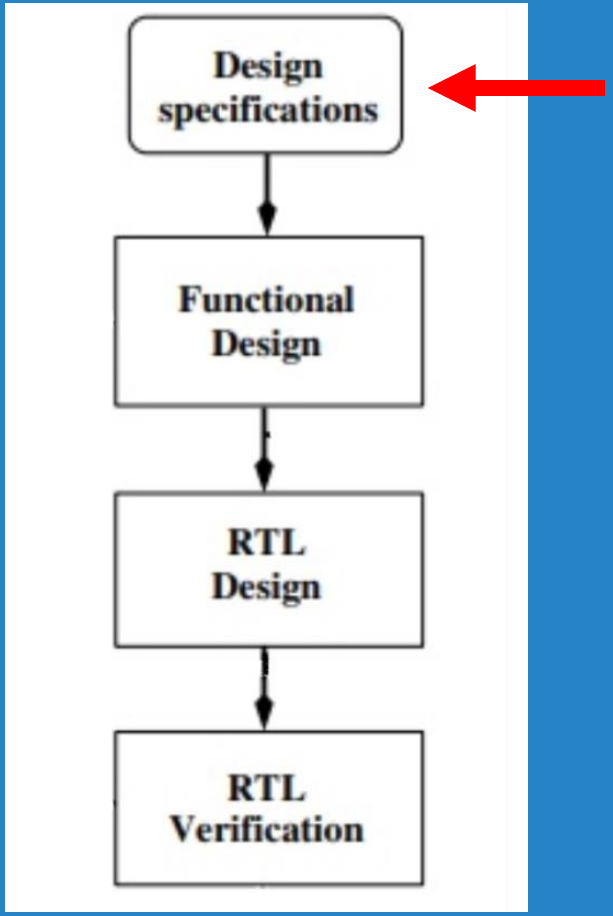
Determining Primality of a Number With Verilog

Taylor Barnes
1 October 2020

Problem Specification



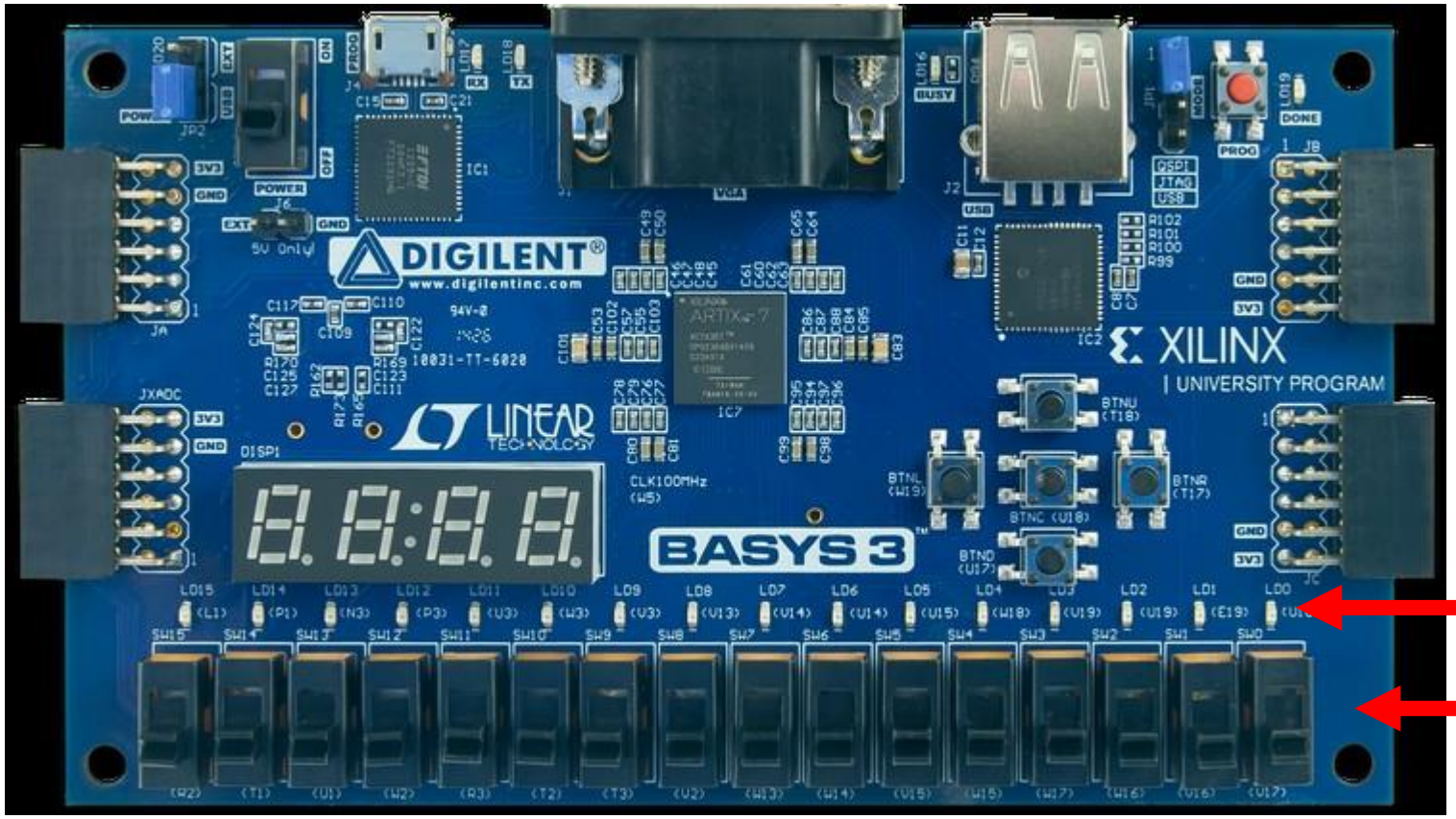
Design Objective:

- Given any nonnegative integer number, determine if it is prime or not prime.

Hardware/Software:

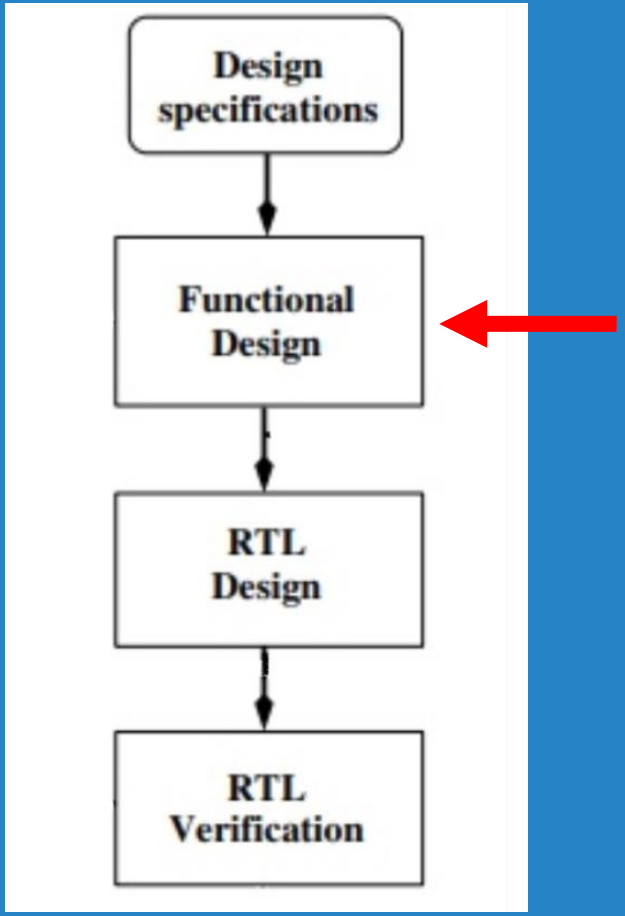
- To implement this design, I will be using Vivado 2017.4, which is used to program the Basys 3 Artix-7 FPGA Board.
- Basys 3 boards are freely available for use by computer engineering students at UCF, but unfortunately, due to COVID restrictions, I was not able to use one for this experiment.
- The Basys 3 board includes 16 on/off switches and 16 LEDs.
- For this design, I will be using the inputs and outputs of the board as follows:
  - 1 LED will be used to display output (1 for prime, 0 for non-prime).
  - 1 switch will be used to activate the system.
  - The remaining 15 switches will be used to input a number in binary.

Basys Board Layout



- Input: n-bit number using n switches (n <= 15)
- Output: 1 LED will display an on/off signal. It will turn on if the number is prime, and off if the number is not prime.

Algorithm



Simple Algorithm:

- The simplest algorithm to determine if a number is prime is to divide that number by every positive integer up to the square root of our number.
- If a divisor is found, the number is not prime. If no divisor is found, the number is prime.

Faster Algorithm:

- The simple algorithm above is very slow for large numbers because it tests many redundant numbers.
- To speed up the process, split all positive integers into groups of six consecutive numbers.
- For every six numbers, two will be divisible by 2, one will be divisible by 3, and one will be divisible by 2 and 3.
- When dividing, our faster algorithm will skip all multiples of 2 and 3 to avoid redundancy.

Simple Algorithm (Pseudocode)

```
if num = 0 or num = 1
      return false

i = 2

while i^2 <= num
      if num mod i = 0
            return false
      i++

return true
```

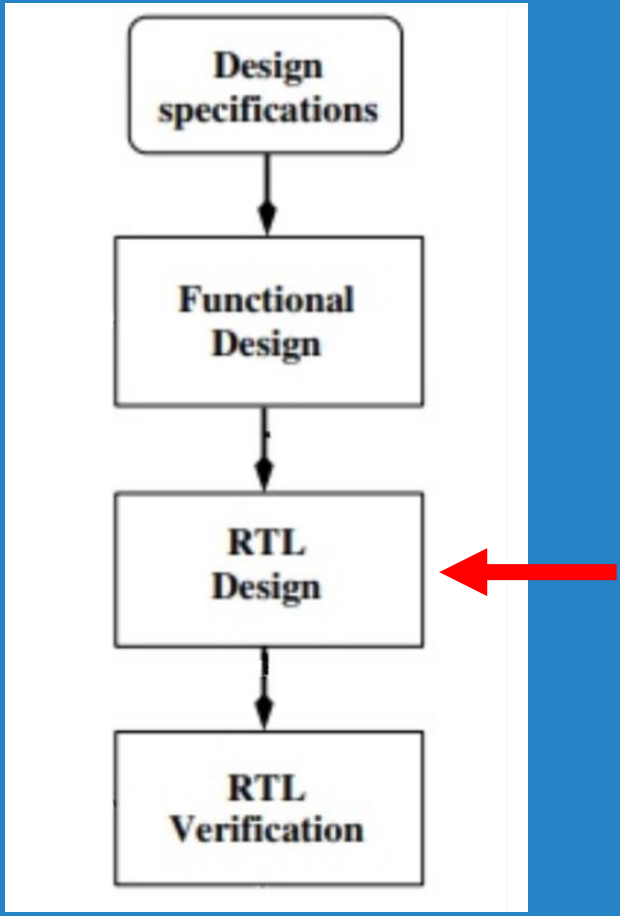
- Here, num is our input number that we are testing to see if it is prime.
- i is our incrementor value. We will be testing i to see if it is a divisor of num.
- In this case, “return true” means that num is prime. “return false” means that num is not prime.

Faster Algorithm (Pseudocode)

```
if num ≤ 3 then
        return num > 1
else if num mod 2 = 0 or num mod 3 = 0
        return false
i = 5
while i^2 <= num
        if num mod i = 0 or num mod (i + 2) = 0
                return false
        i = i + 6
return true
```

- In this case, i starts at 5 and increments by 6 on every iteration.
- This effectively splits all numbers into groups of 6, as was described previously.

Verilog Implementation



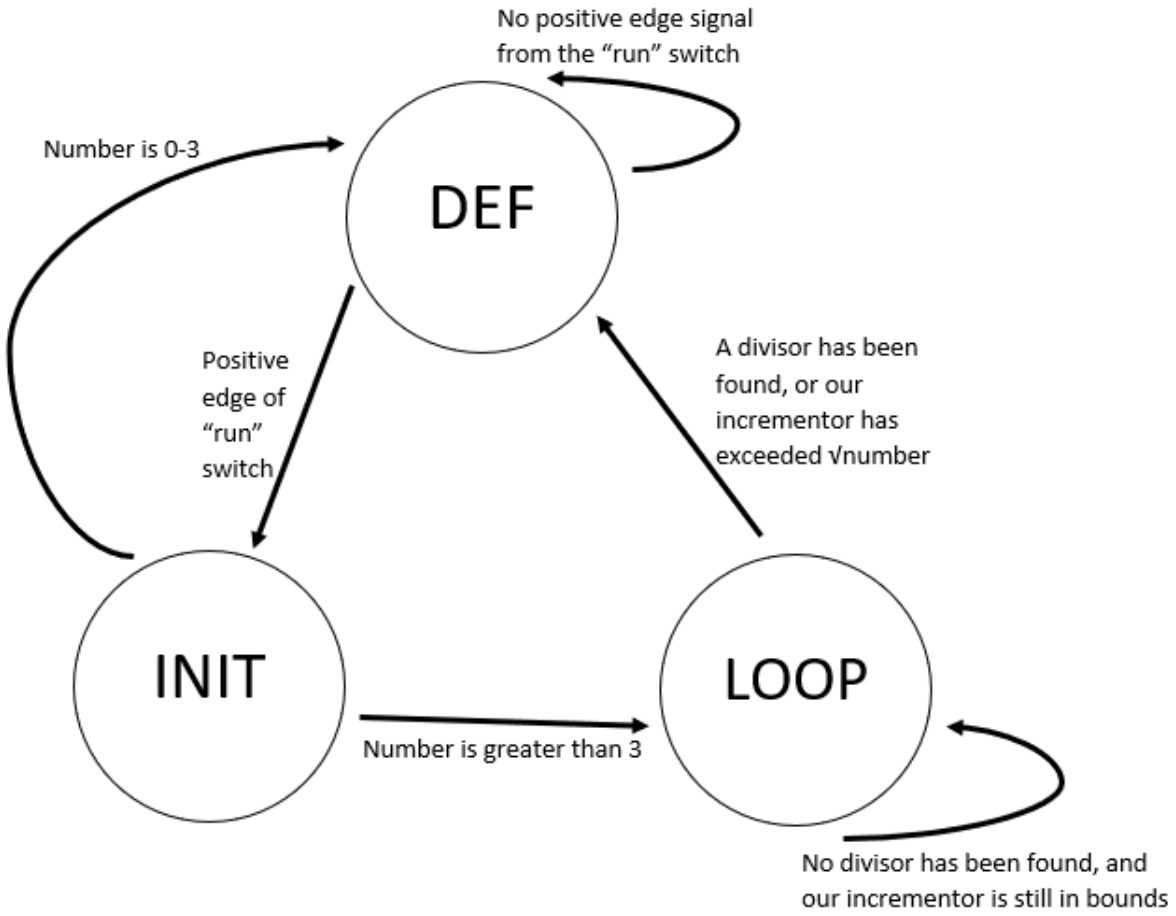
Finite State Machine:

- The algorithms are similar enough that they can both be modeled by nearly identical finite state machines.
- The FSM model was chosen because the process of repeated division is a loop that eventually will need to break.

Verilog Code:

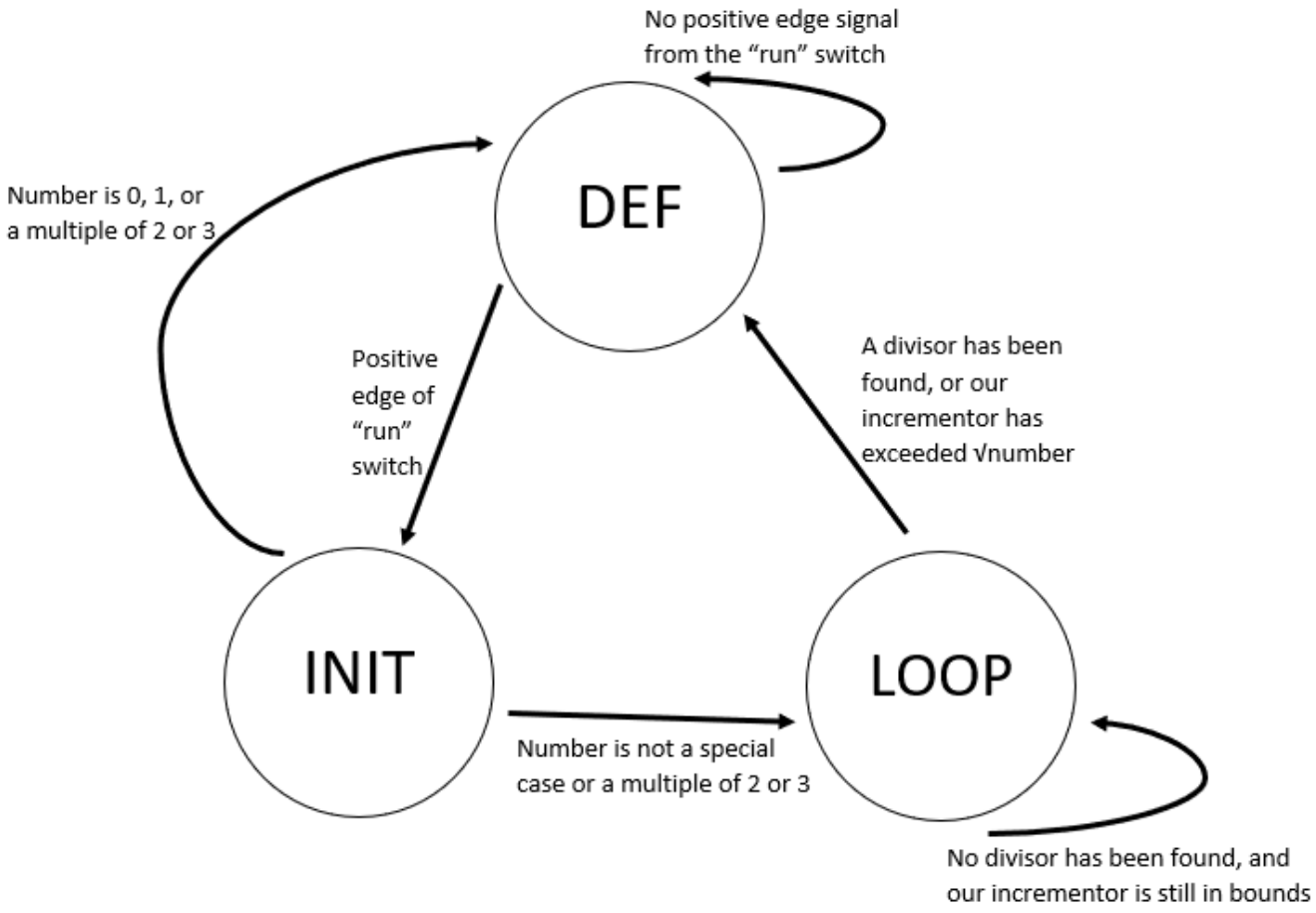
- The pseudocode from the previous slides can be translated fairly directly to Verilog.
- The main difference between the pseudocode and the Verilog code is simply implementing the FSM.

Simple Algorithm (FSM)



- DEF state: this state constantly waits for a positive edge signal from the "on" switch.
- INIT state: this state carries out the first steps of the algorithm. It checks to see if the input is a special case (very small number).
- LOOP state: this state divides the input number by every subsequent possible divisor until a conclusion is reached.

Faster Algorithm (FSM)



- Same states as the previous FSM.
- The only difference is that the INIT state here does a few more checks than in the simple algorithm.

Simple Algorithm (Verilog code)

```
2   `timescale 1ns / 1ps
3
4
5
6   module simple(
7       input [11:0] num, //user input number to check primality
8       input clk, //clock
9       input sw, //switch that the user throws to run the algorithm
10      output reg led //indicator of primality. 1 = prime, 0 = non-prime
11      );
12
13      reg [12:0] i = 2; //this number is used to check for possible primes
14      parameter LOOP = 2'b01, DEF = 2'b00, INIT = 2'b11; //states for the FSM
15      reg [1:0] state = DEF; //start FSM in default state
16      reg swx = 0; //ensures that we only run algorithm on the posedge of switch
17
18      always @ (posedge clk)
19      begin
20
21          //reset swx, so that we can rerun the algorithm
22          if(!sw)
23          begin
24              swx = 0;
25          end
26
27          //FSM
28          case(state)
```

Simple Algorithm (Verilog code)

```
30          //first, check for easy numbers (numbers below 4)
31          INIT:
32              //numbers 3 and below behave unusually
33              if(num <= 3)
34              begin
35                  led <= num > 1;
36                  state <= DEF;
37              end
38              //if no divisor has been found yet, look for a larger divisor
39              else
40              begin
41                  i <= 2;
42                  state <= LOOP;
43              end
```

Simple Algorithm (Verilog code)

```
43              //the main algorithm runs here
44              LOOP:
45                  //our running divisor is i. if i is greater than sqrt of the number, then the algorithm is finished
46                  if(i * i <= num)
47                  begin
48                      //check new possible divisors
49                      if(num % i == 0)
50                      begin
51                          led <= 0;
52                          state <= DEF;
53                      end
54                      //if they don't work, increment i
55                      else
56                      begin
57                          i <= i + 1;
58                      end
59                  end
60                  //break from algorithm, number is prime
61                  else
62                  begin
63                      led <= 1;
64                      state <= DEF;
65                  end
66
67              //default state, constantly check for switch posedge
68              DEF:
69                  if(sw && !swx)
70                  begin
71                      swx <= 1;
72                      state <= INIT;
73                  end
74
75              //default case, unused
76              default:
77                  state <= DEF;
78          endcase
```

Faster Algorithm (Verilog Code)

```
3   `timescale 1ns / 1ps
4
5   module faster(
6       input [11:0] num, //user input number to check primality
7       input clk, //clock
8       input sw, //switch that the user throws to run the algorithm
9       output reg led //indicator of primality. 1 = prime, 0 = non-prime
10      );
11
12      reg [12:0] i = 5; //this number is used to check for possible primes
13      parameter LOOP = 2'b01, DEF = 2'b00, INIT = 2'b11; //states for the FSM
14      reg [1:0] state = DEF; //start FSM in default state
15      reg swx = 0; //ensures that we only run algorithm on the posedge of switch
16
17      always @ (posedge clk)
18      begin
19
20          //reset swx, so that we can rerun the algorithm
21          if(!sw)
22          begin
23              swx = 0;
24          end
25
26          //FSM
27          case(state)
```

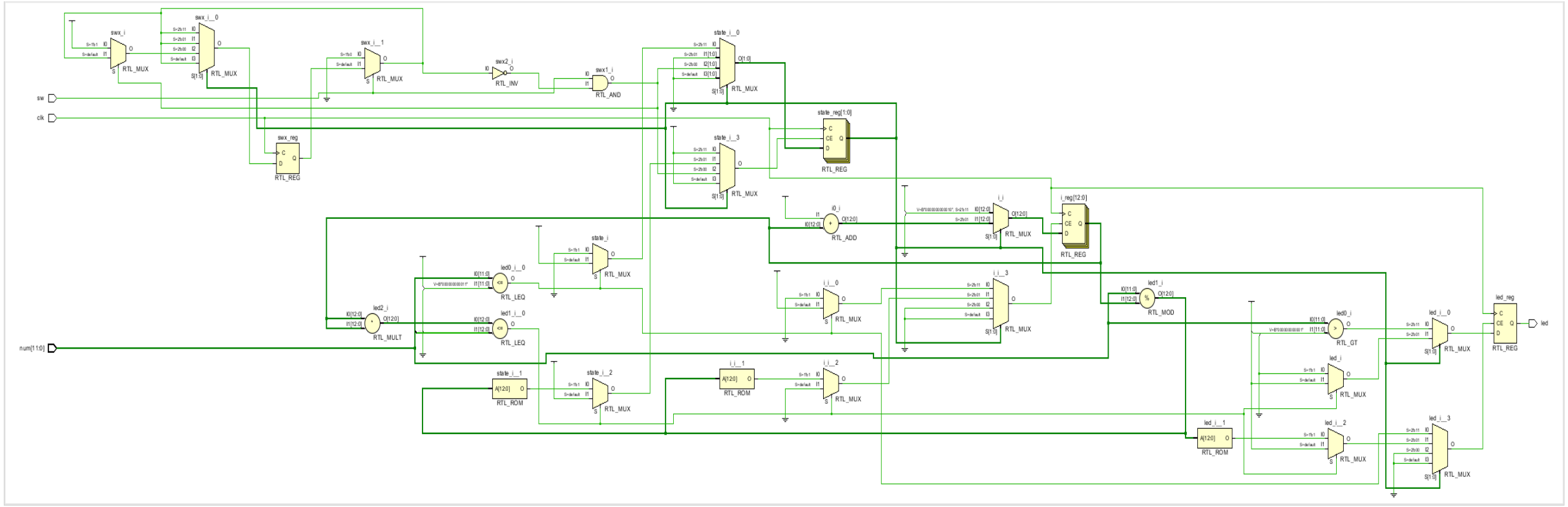
Faster Algorithm (Verilog Code)

```
29          //first, check for easy numbers (numbers below 25, or multiples of 2 and 3)
30          INIT:
31              //numbers 3 and below behave unusually
32              if(num <= 3)
33              begin
34                  led <= num > 1;
35                  state <= DEF;
36              end
37              //check for divisibility by 2 and 3
38              else if(num % 2 == 0 || num % 3 == 0)
39              begin
40                  led <= 0;
41                  state <= DEF;
42              end
43              //if no divisor has been found yet, look for a larger divisor
44              else
45              begin
46                  i <= 5;
47                  state <= LOOP;
48              end
```

Faster Algorithm (Verilog Code)

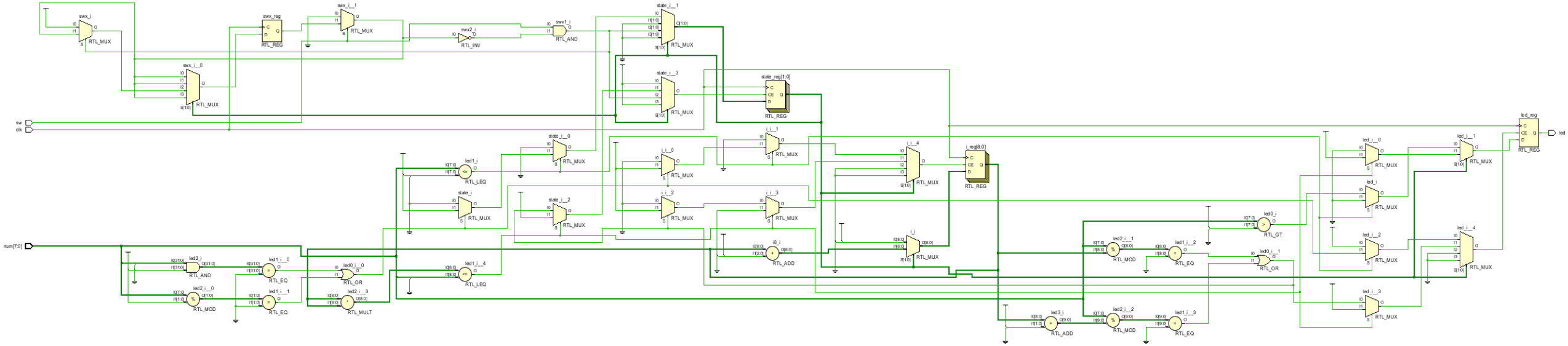
```
50          //the main algorithm runs here
51          LOOP:
52              //our running divisor is i. if i is greater than sqrt of the number, then the algorithm is finished
53              if(i * i <= num)
54              begin
55                  //check new possible divisors
56                  if(num % i == 0 || num % (i + 2) == 0)
57                  begin
58                      led <= 0;
59                      state <= DEF;
60                  end
61                  //if they don't work, increment i
62                  else
63                  begin
64                      i <= i + 6;
65                  end
66              end
67              //break from algorithm, number is prime
68              else
69              begin
70                  led <= 1;
71                  state <= DEF;
72              end
73
74          //default state, constantly check for switch posedge
75          DEF:
76              if(sw && !swx)
77              begin
78                  swx <= 1;
79                  state <= INIT;
80              end
81
82          //default case, unused
83          default:
84              state <= DEF;
85      endcase
```

Simple Algorithm (Schematic)



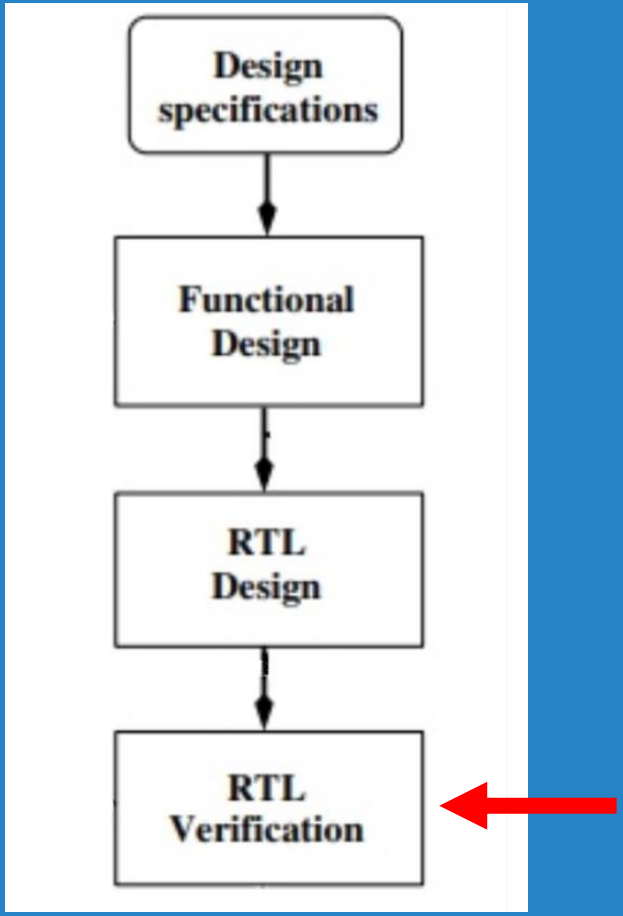
- 108 nets
- 45 leaf cells

Faster Algorithm (Schematic)



- 146 nets
- 53 leaf cells

Verification



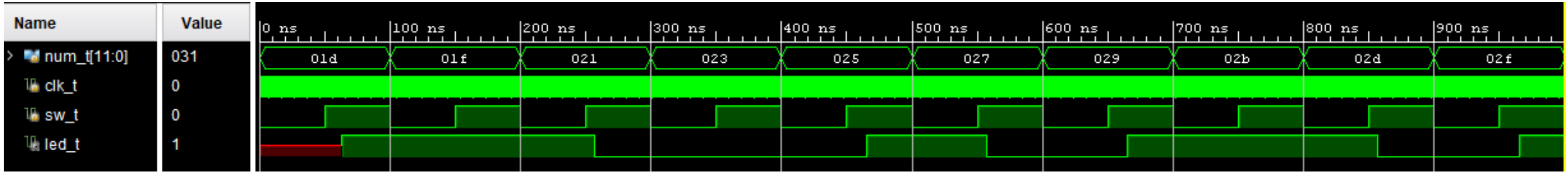
Simulation:

- Several tests were run with Vivado's simulation software to verify the correctness of the algorithm.
- The simulations also gave insight into just how much important the speed factor is in this design.

Comparison:

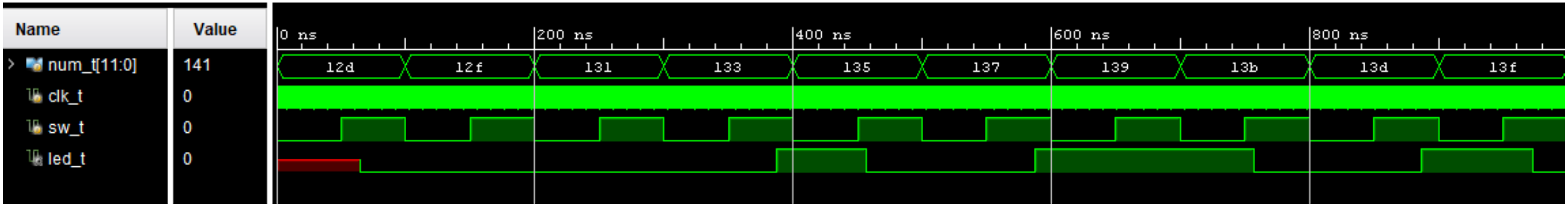
- Hardware cost vs. Speed.
- Is there a way to improve upon the faster algorithm?

Simple Algorithm Simulation 1



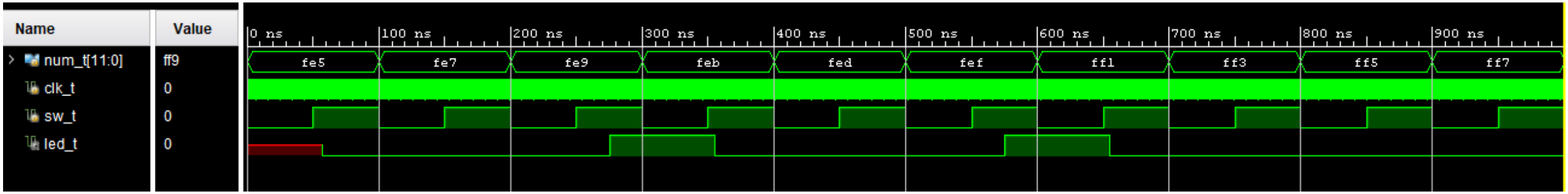
- The simulation tests 10 different numbers consecutively (each differing by 2).
- These results are reasonable. LED turns on when num is 29, 31, 37, 41, 43, and 47.
- The response time for prime numbers is < 20 clock cycles.

Simple Algorithm Simulation 2



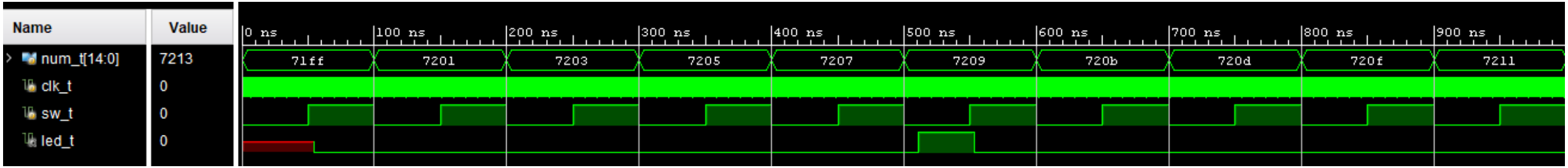
- The design correctly flags 307, 311, 313, and 317 as primes. However, it does so only after a delay of nearly 50 clock cycles.
- It is clear that, for large numbers, this algorithm will become unacceptably slow.

Faster Algorithm Simulation 1



- With the faster algorithm, the design is able to compute primality of 12-bit numbers like the ones above within a reasonable time frame.
- Once again, the algorithm is correct in all cases. 4073 and 4079 are both correctly identified as primes.

Faster Algorithm Simulation 2



- For the stated maximum of a 15-bit number, neither algorithm is sufficient.
- Here, the faster algorithm correctly identifies the 15-bit number 29191 as prime, but the computation takes more than 50 clock cycles.

Even Faster?

- The faster algorithm can be modified to avoid even more redundancy.

```
if num ≤ 3 then
    return num > 1
else if num mod 2 = 0 or num mod 3 = 0 or num mod 5 = 0
    return false
i = 7
while i^2 <= num
    if num mod i + 0,4,6,10,12,16,22 = 0
        return false
    i = i + 30
return true
```

- This can be modified infinitely until it approaches an infinitely large lookup table.

Hardware Cost vs. Speed

Simple algorithm: minimal pre-checks, maximum redundancy.

Faster algorithm

Modified faster algorithm

Large lookup table: 2^n entries in the table, n = size of input. No redundancy, potentially infinite size.

Minimum Hardware Cost

Maximum Speed

Conclusion

- The design answers the problem specification correctly in all cases.
- For 12-bit numbers and smaller, the faster-algorithm design runs relatively quickly.
- The increased potential size of the input from using the faster algorithm is worth the increased hardware cost.