|  |
| --- |
| **P1) CarWarningSystem (자동차 경고 시스템)에 대한 Gate-Level 설계 및 시뮬레이션** |
| * 목적: 자동차 시동이 걸려있을 떄, 내부에서 안전벨트가 착용되지 않을 경우 혹은 문이 제대로 닫히지 않았을 경우를 확인하여 알람을 보내는 회로를 제작 * 수행방법: 회로 설계를 위한 block diagram을 그린 후 dut에 대한 input과 output, wire와 gate를 선언한다. 회로가 제대로 설계되었는지 확인을 위해 tb를 생성하고 input에 여러 값을 대입하며 결과를 확인한다. |
| **설계를 위한 Block Diagram (각 Node의 신호명 포함)** |
| 제일 왼쪽의 세 개의 항목(Door, Ignition, Seatbelt)은 input에 해당한다. 이 변수들의 값에 따라 gate를 거쳐 최종 output이 결정된다. 순서대로 문닫힘, 시동 상태, 안전벨트 착용을 가리킨다. Door는 문이 닫혔을 경우 1, 열렸을 경우 0를 가리킨다. Ignition은 시동이 걸렸을 경우 1, 꺼져있을 경우 0이다. 마지막으로 seatbelt는 제대로 채워졌을 경우 1, 채워지지 않았을 경우 0이다. 이 회로도는 not을 제외한 첫 gate가 and이므로 두 개의 and gate에 모두 들어가는 Ignition이 1일 때 즉, 시동이 걸렸을 때를 기준으로 판단한다. 시동이 걸려있지 않다면 문이 열려있거나 벨트가 채워지지 않았더라도 알람이 울리지 않지만, 시동이 걸려있다면 문이 닫히지 않았거나 안전벨트가 채워지지 않았거나 이 둘 중 하나의 상황에라도 해당된다면 or gate를 통해 1이 출력되고 알람이 울린다. |
| **Main Code** |
| * DUT  |  | | --- | | module P1\_DUT #(  )( // input, ouput -> 회로의 시작과 끝 (0/1 bit)  output Alarm ,  input Door ,  input Ignition ,  input SeatBelt  );  // gate를 거친 input의 값을 저장하는 line  wire DoorOpened;  wire NoSeatBelt;  wire Node1;  wire Node2;  // 각 게이트들. ouput을 먼저 입력한 후 input을 순차적으로 입력  not NOT\_U0(DoorOpened, Door);  not NOT\_U1(NoSeatBelt, SeatBelt);  and AND\_U0(Node1, DoorOpened, Ignition);  and AND\_U1(Node2, NoSeatBelt, Ignition);  or OR\_U0(Alarm, Node1, Node2);  endmodule |  * Test Bench  |  | | --- | | `timescale 1ns/1ps // 시간 설정, 추후에 나올 #~에 대해 ns 단위를 적용하고 wave에서는 ps 단위로 결과값을 나타냄  module P1\_TB;  // 테스트에서 활용할 input과 output 선언  reg Door;  reg Ignition;  reg SeatBelt;  wire Alarm;  // P1\_DUT.v 파일의 P1\_DUT를 호출한다. 이 때 각 input과 output에 대응하는 변수들을 대입  P1\_DUT dut(  .Alarm (Alarm ),  .Door (Door ),  .Ignition (Ignition ),  .SeatBelt (SeatBelt ));  initial begin  // Initial  Door = 1'b0;  Ignition = 1'b0;  SeatBelt = 1'b0;  // Ignition  #100; // 100ns 대기  Door = 1'b0; // 1bit binary에 0 대입  Ignition = 1'b1; // 1bit binary에 1 대입  SeatBelt = 1'b0;  // Ready to drive  #100  Door = 1'b1;  Ignition = 1'b1;  SeatBelt = 1'b1;  // Warning  #100  Door = 1'b1;  Ignition = 1'b1;  SeatBelt = 1'b0;  #100  $finish;  end  // debug or display  always @(posedge Alarm) begin  if(dut.DoorOpened == 1) begin  $display("====================\n");  $display("[Warning] Door is opened!\n");  $display("====================\n");  end else if (dut.NoSeatBelt == 1) begin  $display("====================\n");  $display("[Warning] Please fasten your seatbelt!\n");  $display("====================\n");  end  end  endmodule | |
| **Schematic & Waveform or Display** |
| * Schematic : 직접 그린 block diagram과 input의 위치만을 제외하고는 똑같이 나왔다. 이 schematic에서 dut으로 네모박스가 있는 부분이 우리가 설계한 내용이며, input과 output은 사용자가 tb파일에서 직접 지정한 값, 혹은 그로 인한 output값이다. 또한, 각 gate들도 입력에 따라 그 값이 바뀔 수 있다.      * Waveform : 가장 먼저, 예제와 같이 100ns씩 delay를 주었을 때의 waveform이다. 가장 먼저 모든 변수를 0으로 초기한다. SeatBelt와 Door는 not gate를 통하여 DoorOpened, NoSeatBelt wire가 1로 변한다. 하지만 and gate에서는 Ignition의 값이 0이기 때문에 아무런 변화가 없다. 100ns동안 대기한 후 Ignition의 값이 1로 변화한다. 따라서 AND\_U1, AND\_U0 모두 두 개의 input들이 모두 1이므로 OR\_U0은 1,1의 인풋으로 1을 반환한다. 다시 100ns 대기 후 이번엔 Door와 SeatBelt의 값을 모두 1로 바꾼다. Not gate를 거쳐 두 개의 wire는 0으로 변환되고 따라서 두 개의 and gate는 각가 1, 0의 input을 받아 0을 반환한다. 알람이 울리지 않는다. 다시 100ns를 기다린 후 이번엔 SeatBelt의 값을 0으로 바꾼다. 이는 안전벨트가 채워지지 않았다는 신호이다. 따라서 NOT\_U1을 통해 noSeatBelt wire가 1이 되고, AND\_U1 gate도 1이 되어 다시 알람이 울린다. 다시 100ns 대기 후 종료한다.     이번엔 #숫자 -> 대기 시간을 이해하기 위해 약간 수정 후 다시 실행하였다. 첫 번째 대기시간 #100은 그대로 유지된다. 그 후 시동은 걸렸으나 문도 열려 있고 안전벨트도 채워지지 않은 상태가 #200 (200ns) 유지된다. 그리고 문도 닫고 안전벨트도 채워진 상황이 300ns 유지되며 마지막으로 안전벨트만 채워지지 않은 상황을 만들고 100ns를 유지한 후 종료한다. |
| **Comments (실험에 대한 고찰)** |
| * 이 시뮬레이션은 회로에 대한 이해보다는 fpga 프로그램에 대한 적응이 훨씬 많이 필요했던 실습이었다. 처음에 강의만 듣고는 dut과 tb의 각 변수들의 활용을 정확히 이해하기 힘들었지만 직접 실습을 하면서 클래스와 함수의 관계로 이해하니 수월하게 할 수 있었다. 그리고 조금 더 간편하고 깔끔한 회로도를 위해 인버터를 활용하는 것이 인상깊었다. 그리고 tb 파일의 가장 위 timescale에 대해서도 쉽게 이해할 수 있었다. |

|  |
| --- |
| **A1) 2:1MUX Gate Level 설계 및 시뮬레이션** |
| * 목적: 2개의 input과 시그널을 이용하여 1개의 output를 출력하는 2:1MUX를 제작 * 수행방법: 진리표를 참고하여 맵과 회로도(block diagram)을 만든 후 이를 참고하여 dut, tb 파일을 만든다. |
| **Block Diagram** |
| 먼저 세개의 input과 한 개의 output이 있는 진리표를 참고하여 맵을 그린다. 인접한 항목들을 묶어 간소화한 식을 도출하고 이를 참고하여 회로도를 그린다.  Input은 2개의 input과 1개의 signal로 이루어져 있다. 따라서 총 세개의 input이 존재하며 signal의 경우 온전한 signal을 가리키는 s input과, not s를 가리키는 sWn wire가 있다. 다른 두 개의 input은 D0과 D1을 각각 d0, d1으로 나타낸다. 이어 d input들은 s input과 and gate를 거쳐 wA0, wA1 wire로 이어진다. 그리고 이 값들을 OR\_U0 gate를 이용하여 결과값 y로 출력한다. |
| **Main Code** |
| * DUT  |  | | --- | | // 2:1 MUX  module A1\_DUT #(  )(  output y ,  input s ,  input d1 ,  input d0  );  wire wSn; // s의 인버터 적용 후  wire wA0; // AND\_U0 gate 거친 후  wire wA1; // AND\_U1 gate 거친 후  // A1\_DUT를 활용하여 test해 보기  not NOT\_U0(wSn, s);  and AND\_U0(wA0, d0, wSn);  and AND\_U1(wA1, s, d1);  or OR\_U0(y, wA0, wA1);  endmodule |  * Test Bench  |  | | --- | | ``timescale 1ns/1ps  // A1\_DUT를 활용하여 test해 보기  module A1\_TB;    reg s;  reg d0;  reg d1;  wire y;  // A1\_DUT를 호출하여 test bench 진행  A1\_DUT dut(  .y (y ),  .s (s ),  .d1 (d1 ),  .d0 (d0 ));  initial begin  // initial, 아래로 진리표 순서대로 적용하기  s = 1'b0;  d1 = 1'b0;  d0 = 1'b0;  #100  s = 1'b0;  d1 = 1'b0;  d0 = 1'b1;  #100  s = 1'b0;  d1 = 1'b1;  d0 = 1'b0;  #100  s = 1'b0;  d1 = 1'b1;  d0 = 1'b1;  #100  s = 1'b1;  d1 = 1'b0;  d0 = 1'b0;  #100  s = 1'b1;  d1 = 1'b0;  d0 = 1'b1;  #100  s = 1'b1;  d1 = 1'b1;  d0 = 1'b0;  #100  s = 1'b1;  d1 = 1'b1;  d0 = 1'b1;  #100  $finish;  end  endmodule | |
| **Waveform or Display** |
| * Schematic : 직접 그린 block diagram과 input의 위치만을 제외하고 똑같이 나왔다. 이 schematic에서 dut으로 네모박스가 있는 부분이 우리가 설계한 내용이며, input과 output은 사용자가 tb파일에서 직접 지정한 값, 혹은 그로 인한 output값이다. 또한, 각 gate들도 입력에 따라 그 값이 바뀔 수 있다.      * Waveform : 이 회로는 두 개의 input과 한 개의 signal로 총 input은 3개이다. 따라서 모든 조합을 확인하더라도 8번의 테스트만 거치면 된다. 따라서 진리표에 나온 모든 조합을 직접 테스트해 본다. 가장 먼저 초기화는 진리표의 가장 첫 번째 (0,0,0)과 값이 일치한다. y값 역시 0이 나온다. 그 후 (0,0,1), (0,1,1), (1,1,0), (1,1,1) 이렇게 네 가지 경우에만 waveform의 첫번째 줄 y의 값이 1로 출력되는 것을 확인할 수 있다. |
| **Comments** |
| * 이번에는 직접 회로도를 그린 후에 이를 바탕으로 직접 코드를 짜보는 실습이었다. 회로 수업을 작년 2학기에 들었던 터라 오랜만에 맵을 그리고 간소화를 하는 것이 처음에는 어색했지만 비교적 간단한 2:1 mux로 진행하여 재밌게 진행할 수 있었다. 시뮬레이션을 돌리고 Schematic을 확인하며 직접 그린 회로도와 일치하는 것이 실험을 더 재밌게 할 수 있었던 방법인 것 같다. 그리고 모든 진리표를 직접 입력하거나 각 input과 signal을 하나씩 선언하는 과정에서, 배열이나 반복문 등을 통해 조금 더 간소화 할 수 있는 방법이 있지 않을까 생각하게 되었다. |

|  |
| --- |
| **A3) 2:1MUX Gate Level 설계 및 시뮬레이션** |
| * 목적: 4개의 input과 2개의 시그널을 이용하여 1개의 output를 출력하는 4:1MUX를 제작 * 수행방법: 회로도를 참고하여 간소화된 식을 만들고 이를 참고하여 dut, tb 파일을 만든다 |
| **Block Diagram** |
| A2 과제는 진리표가 아닌 MUX 표기와 회로도만 나왔다. 따라서 진리표를 그리거나 맵을 그리나 식을 도출하거나, 이 세 가지 방법을 직접 해야 한다. 이 과정에서, 마지막 gate가 or인 것을 보아 그 직전의 and gate들을 각 a, b, c, d로 지정하여 a+b+c+d식으로 변환했다. 그리고 회로도를 참고하여 해당 알파벳들을 실제 식으로 변환하였다. Or게이트의 특성을 참고하여 결과에 영향을 미치지 않는 부분을 don’t care zone으로 설정하여 진리표를 4줄로 줄였다. 만약 모든 경우의 수를 직접 확인한다면 총 64개의 case를 직접 확인해야 했을 것이다. 직접 만든 진리표와 식을 참고하여 다시 회로도를 그렸다. 우선 네 개의 input과 두 개의 signal 비트를 input으로 지정하여 각자 d0~d3, s0~s1로 지정했다. Signal 비트는 인버터 값이 필요하기에 NOT\_U0, NOT\_U1 두 개의 not gate를 활용하여 wS1n, wS0n 두 개의 wire로 나타냈다. 그 후 각 조건에 맞게 AND gate를 연결하고 그 wire들을 마지막으로 or gate의 입력값으로 사용하여 y를 출력하도록 했다. |
| **Main Code** |
| * DUT  |  | | --- | | // 4:1 mux  module A2\_DUT #(  )(  output y ,  input s1 ,  input s0 ,  input d0 ,  input d1 ,  input d2 ,  input d3  );  wire wS1n; // s1의 인버터 적용 후  wire wS0n; // s0의 인버터 적용 후  wire wA0; // AND\_U0 gate 거친 후  wire wA1; // AND\_U1 gate 거친 후  wire wA2; // AND\_U2 gate 거친 후  wire wA3; // AND\_U3 gate 거친 후  not NOT\_U0(wS1n, s1);  not NOT\_U1(wS0n, s0);  and AND\_U0(wA0, d0, wS1n, wS0n);  and AND\_U1(wA1, d1, wS1n, s0);  and AND\_U2(wA2, d2, s1, wS0n);  and AND\_U3(wA3, d3, s1, s0);  or OR\_U0(y, wA0, wA1, wA2, wA3);  endmodule |  * Test Bench : 64개가 아닌 각 d input에 일치하는 회로를 만들고, don’t care와 wrong을 체크하는 방법으로 테스트해 보았다.  |  | | --- | | `timescale 1ns/1ps  module A2\_TB;  reg s1;  reg s0;  reg d0;  reg d1;  reg d2;  reg d3;  wire y;  A2\_DUT dut(  .y (y ),  .s1 (s1 ),  .s0 (s0 ),  .d0 (d0 ),  .d1 (d1 ),  .d2 (d2 ),  .d3 (d3 ));  initial begin  // initial  s1 = 1'b0;  s0 = 1'b0;  d0 = 1'b0;  d1 = 1'b0;  d2 = 1'b0;  d3 = 1'b0;    // D0 wrong  #100  s1 = 1'b1;  s0 = 1'b0;  d0 = 1'b1;  d1 = 1'b0;  d2 = 1'b0;  d3 = 1'b0;    // D0 wrong  #100  s1 = 1'b0;  s0 = 1'b1;  d0 = 1'b1;  d1 = 1'b0;  d2 = 1'b0;  d3 = 1'b0;    // D0 wrong  #100  s1 = 1'b1;  s0 = 1'b1;  d0 = 1'b1;  d1 = 1'b0;  d2 = 1'b0;  d3 = 1'b0;    // D0 right  #100  s1 = 1'b0;  s0 = 1'b0;  d0 = 1'b1;  d1 = 1'b0;  d2 = 1'b0;  d3 = 1'b0;    // D0 don't care  #100  s1 = 1'b0;  s0 = 1'b0;  d0 = 1'b1;  d1 = 1'b1;  d2 = 1'b0;  d3 = 1'b0;  // D0 don't care  #100  s1 = 1'b0;  s0 = 1'b0;  d0 = 1'b1;  d1 = 1'b1;  d2 = 1'b0;  d3 = 1'b1;    // D0 wrong  #100  s1 = 1'b1;  s0 = 1'b0;  d0 = 1'b1;  d1 = 1'b0;  d2 = 1'b0;  d3 = 1'b0;    // D1 right  #100  s1 = 1'b0;  s0 = 1'b1;  d0 = 1'b0;  d1 = 1'b1;  d2 = 1'b0;  d3 = 1'b0;    // D1 wrong  #100  s1 = 1'b1;  s0 = 1'b1;  d0 = 1'b0;  d1 = 1'b1;  d2 = 1'b0;  d3 = 1'b0;    // D1 wrong  #100  s1 = 1'b0;  s0 = 1'b0;  d0 = 1'b0;  d1 = 1'b1;  d2 = 1'b0;  d3 = 1'b0;    // D1 wrong  #100  s1 = 1'b1;  s0 = 1'b0;  d0 = 1'b0;  d1 = 1'b1;  d2 = 1'b0;  d3 = 1'b0;    // D2 right  #100  s1 = 1'b1;  s0 = 1'b0;  d0 = 1'b0;  d1 = 1'b0;  d2 = 1'b1;  d3 = 1'b0;    // D2 wrong  #100  s1 = 1'b0;  s0 = 1'b0;  d0 = 1'b0;  d1 = 1'b0;  d2 = 1'b1;  d3 = 1'b0;    // D2 wrong  #100  s1 = 1'b1;  s0 = 1'b1;  d0 = 1'b0;  d1 = 1'b0;  d2 = 1'b1;  d3 = 1'b0;    // D2 wrong  #100  s1 = 1'b0;  s0 = 1'b1;  d0 = 1'b0;  d1 = 1'b0;  d2 = 1'b1;  d3 = 1'b0;    // D3 right  #100  s1 = 1'b1;  s0 = 1'b1;  d0 = 1'b0;  d1 = 1'b0;  d2 = 1'b0;  d3 = 1'b1;    // D3 wrong  #100  s1 = 1'b1;  s0 = 1'b0;  d0 = 1'b0;  d1 = 1'b0;  d2 = 1'b0;  d3 = 1'b1;    // D3 wrong  #100  s1 = 1'b0;  s0 = 1'b1;  d0 = 1'b0;  d1 = 1'b0;  d2 = 1'b0;  d3 = 1'b1;    // D3 wrong  #100  s1 = 1'b0;  s0 = 1'b0;  d0 = 1'b0;  d1 = 1'b0;  d2 = 1'b0;  d3 = 1'b1;  #100  $finish;  end  endmodule | |
| **Waveform or Display** |
| * Schematic : 직접 그린 block diagram과 각 input, gate의 위치 부분에서 조금씩 차이가 나지만 내용은 일치한다. 이 schematic에서 dut으로 네모박스가 있는 부분이 우리가 설계한 내용이며, input과 output은 사용자가 tb파일에서 직접 지정한 값, 혹은 그로 인한 output값이다. 또한, 각 gate들도 입력에 따라 그 값이 바뀔 수 있다.      * Waveform : 이번 회로는 input이 총 6개로 모든 조합을 생각한다면 총 64가지 경우의 수가 나온다. 따라서 tb파일에서는 회로의 정확성을 판단할 수 있는 몇 개의 경우의 수를 꼽아 테스트를 진행했다. 가장 먼저 모든 값을 0으로 선언한다. 그 후 d0을 기준으로 테스트를 진행한다. D0이 true일 때, 나머지 d값들은 관계가 없으므로 s1, s0 두 개의 비트를 조합하여 wrong 세 가지와 right 한 가지를 순차적으로 확인한다. 이어 d0의 don’t care도 두가지 확인한다. 이후 d1~d3을 모두 right 한 가지, wrong 세 가지로 확인해본다. |
| **Comments** |
| * 처음에 회로도만 있는 모습을 보고 순간적으로 당황했지만 천천히 뒤에서부터 확인하여 식을 도출하였다. 그 후 진리표를 그릴 때 처음엔 모든 경우의 수를 확인하려 하였으나 마지막 gate가 or인 점을 고려하여 don’t care zone을 확인하고 그 전의 부분이 and gate인 점을 활용해서 한 개의 d input에 알맞는 s input 조합은 한 가지밖에 없음을 확인하였다. 따라서 진리표를 그리는 시간을 줄이고 더 간편하게 테스트를 진행할 수 있었다. |

|  |
| --- |
| **A3) Visual Studio Code 관련 학습 내용 정리** |
| * 목적: verilog를 사용하는 과정에서 소스 코드 제작 및 편집의 편리함을 위해 코드 작성은 visual studio code ide를 활용한다. 이를 더 효율적으로 사용하기 위해 단축키나 유용한 사용법을 알아보고 정리한다. * 수행방법: 구글링, 교재 찾아보기 * 기준 : 맥북 (커맨드와 컨트롤, 컨트롤과 알트 등의 차이가 있을 수 있음) |
| **단어 전체 선택 : 단어 더블 클릭 후 command + shift + L** |
| 변수명을 바꿀 때나 자료형을 바꿀 때 등 파일 내의 같은 단어를 모두 선택한다. (교수님께서 알려주신 command + control + 방향키는 다중 커서이며 이 명령어는 다중 선택에 해당한다.)  command + d를 사용한다면 선택한 단어를 아래 방향으로 가장 가까운 위치부터 하나씩 추가로 선택할 수 있다. |
| **코드 위치 변경 : option + 상하 방향키** |
| 선언문이나 코드의 위치를 바꾸고 싶다면 해당 라인에 커서를 대고 option을 누른 상태로 방향키를 움직이면 된다. Multi line도 가능하며 연속된 줄이 아닌 사용자가 원하는 라인들을 동시에 같은 라인 수만큼 옮기기도 가능하다. |
| **원하는 위치 다중 선택 : option + click, option + shift + I** |
| option키와 클릭을 함께 누르면 여러 곳에 커서가 생긴다. 이는 특정 단어에 해당하는 것이 아닌 사용자가 원하는 위치라면 어디든 가능해서 자유롭게 소스코드 추가가 가능하다. 작업 후 다른 곳을 클릭하거나 esc 버튼을 통해 빠져나올 수 있다. 원하는 영역을 드래그 한 후 option + shift + I를 활용하면 연속된 영역에서의 다중 커서를 더 편리하게 사용할 수 있다. |
| **라인 복사 : option + shift + 상하 방향키** |
| 사용자가 커서를 이동하여 라인의 내용을 모두 드래그할 필요 없이 커서가 위치한 라인에 있는 모든 소스코드를 위 혹은 아래로 복사 및 붙여넣기 해준다. |
| **커서 이동 : command + 좌우 방향기, option + 좌우 방향키** |
| 원하는 위치로의 커서 이동을 위해 좌우 방향키를 오래 누를 필요 없이 option과 방향키를 통해 한 단어 씩 이동이 가능하다. 띄어쓰기, 특수기호 등을 하나의 묶음으로 이동한다. Command + 방향키를 이동한다면 코드의 가장 앞이나 뒤로 이동할 수 있다. 이 단축키들을 shift와 함께 사용한다면 드래그를 더 편리하게 할 수 있다. |