5주차 결과보고서

전공: 컴퓨터공학과 학년: 2학년 학번: 20231523 이름: 김민정

**1. 실험 목적**

강의자료에 나온 드모르간의 정리를 이해하고 이에 더 나아가 드모르간 법칙뿐만아니라 다른 Boolean 함수에 대해 알아본다. 이 과정 중에서 논리식을 쓰면서 위 내용을 이해한다.

이해한 강의자료의 내용을 토대로 드모르간 정리와 Boolean 함수를 Verilog 언어를 통해 구현한다. 이 과정 중에서 Vivado와 FPGA를 사용한다. Vivado의 simulation을 통해 구현되는 결과를 확인하고, FPGA를 통해 드모르간 정리와 Boolean 함수의 동작을 확인한다.

**2. 드모르간 제 1, 2 법칙(with source code, testbench code and simulation)**

드모르간의 제 1법칙은 ~(A+B) = ~A\*~B가 성립함을 의미한다. 이 둘이 같은 진리값을 가지는지는 아래 vivado simulation에서 확인할 수 있다. 여기서 output c가 ~(A+B)를 의미하고 output d가 ~A\*~B를 의미한다. 이 둘은 같은 input일 때 같은 결과 진리값을 가짐을 확인할 수 있다. 이를 NOR과 비교해보자. NOR logic gate는 OR의 연산 이후 inversion을 하여 결과값을 얻는다. 이는 드모르간 제 1법칙의 적용과정과 동일함을 확인할 수 있다.

1) ~(A+B) / ~A\*~B source code, testbench code, and simulation.

|  |  |
| --- | --- |
| Source code | Testbench code |
| 텍스트, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명 | 텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진  자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명 |

|  |
| --- |
| simulation |
| 스크린샷, 사각형이(가) 표시된 사진  자동 생성된 설명 |

이제 드모르간 제 2법칙을 한 번 살펴보자. 드모르간의 제 2법칙은 ~(A\*B) = ~A+~B가 성립함을 의미한다. 이 둘이 같은 진리값을 가지는지는 아래 vivado simulation에서 확인할 수 있다. 여기서 output c가 ~(A\*B)를 의미하고 output d가 ~A+~B를 의미한다. 이 둘은 같은 input일 때 같은 결과 진리값을 가짐을 확인할 수 있다. 이를 NAND과 비교해보자. NAND logic gate는 AND의 연산 이후 inversion을 하여 결과값을 얻는다. 이는 드모르간 제 2법칙의 적용과정과 동일함을 확인할 수 있다.(이와 관련된 표는 다음페이지에 있다.)

Testbench 코드는 위의 드모르간 제 1법칙에서의 testbench code와 동일하다.

2) ~(A\*B) / ~A+~B source code, testbench code, and simulation.

|  |  |
| --- | --- |
| Source code | Testbench code |
| 텍스트, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명 | 텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진  자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명 |

|  |
| --- |
| simulation |
| 스크린샷, 사각형이(가) 표시된 사진  자동 생성된 설명 |

**3. 드모르간 법칙 예시 적용 - (A'+B')\*C'와** **((A\*B)+C)' / (A'\*B')+C'와 ((A+B)\*C)' 비교를 통해**

2번의 소제목에서 (A’+B’)\*C’와 ((A\*B)+C)' 그리고 (A'\*B')+C'와 ((A+B)\*C)'은 서로같은 input에 대해 같은 output 값을 갖는다. 즉 드모르간의 정리가 성립함을 확인할 수 있다는 것을 의미한다. 먼저 (A’+B’)\*C’와 ((A\*B)+C)'의 비교를 한 후에 (A'\*B')+C'와 ((A+B)\*C)'를 비교하겠다. 첫번째와 두번째 표는 (A’+B’)\*C’의 source code와 testbech code, 그리고 simulation 결과를 나타낸 것이고, 세번째와 네번쨰표는 ((A\*B)+C)'의 source code와 testbech code, 그리고 simulation 결과를 나타낸 것이다. 이 둘을 보았을 때 같은 input일 때, 같은 output값을 도출. 이는 즉 동일한 식임을 의미한다.

1) (A’+B’)\*C’ Source code, Testbench code, Simulation

|  |  |
| --- | --- |
| Source code | Testbench code |
| 텍스트, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명 | 텍스트, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명 |

|  |
| --- |
| simulation |
| 스크린샷, 사각형이(가) 표시된 사진  자동 생성된 설명 |

2) ((A\*B)+C)' Source code, Testbench code, Simulation

|  |  |
| --- | --- |
| Source code | Testbench code |
| 텍스트, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명 | 텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진  자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명 |

|  |
| --- |
| simulation |
| 스크린샷, 사각형, 도표이(가) 표시된 사진  자동 생성된 설명 |
|  |

이제 (A'\*B')+C'와 ((A+B)\*C)'를 비교를 하겠다. 이번에도 첫번째와 두번째 표는 (A'\*B')+C'의 source code와 testbech code, 그리고 simulation 결과를 나타낸 것이고, 세번째와 네번쨰표는 ((A+B)\*C)'의 source code와 testbech code, 그리고 simulation 결과를 나타낸 것이다. 이 둘을 보았을 때 같은 input일 때, 같은 output값을 도출. 이는 즉 동일한 식임을 의미한다.

3) (A'\*B')+C' Source code, Testbench code, Simulation

|  |  |
| --- | --- |
| Source code | Testbench code |
| 텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진  자동 생성된 설명 | 텍스트, 스크린샷, 폰트, 영수증이(가) 표시된 사진  자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진  자동 생성된 설명 |

|  |
| --- |
| simulation |
| 스크린샷, 도표, 라인이(가) 표시된 사진  자동 생성된 설명 |

4) ((A+B)\*C)' Source code, Testbench code, Simulation

|  |  |
| --- | --- |
| Source code | Testbench code |
| 텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진  자동 생성된 설명 | 텍스트, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명 |

|  |
| --- |
| simulation |
| 스크린샷, 도표, 라인이(가) 표시된 사진  자동 생성된 설명 |

**4. 1Bit 비교기(with source code, testbench code and simulation)**

1 Bit 비교기는 한자리수의 서로 다른 비트를 비교하는 것이다. 이 비교기에서 bit operation은 equal, not equal, left bigger, right bigger이다 . input A와 B에 따른 oupt은 아래 진리표와 같다.

1) 1 bit 비교기 진리표

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | A=B  (equal) | A!=B  (not equal) | A>B  (left bigger) | A<B  (right bigger) |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

1 bit 비교기는 >,<,= operation 없이 not, or, and 등과 같은 논리 연산자를 통해 구현할 수 있다. Equal과 not equal은 xor연산자를 통해, 그리고 이러한 not equal 연산을 확장하여 left/right bigger을 제작할 수 있다. (1과 0의 특성을 이용해 and 연산자 이용). 아래는 Verilog 언어로 작성한 1bit 비교기의 source code와 testbench code 그리고 simulation 결과이다.

2) source code, testbench code, and simulation

|  |  |
| --- | --- |
| Source code | Testbench code |
| 텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진  자동 생성된 설명 | 텍스트, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진  자동 생성된 설명 |

|  |
| --- |
| simulation |
| 스크린샷, 사각형이(가) 표시된 사진  자동 생성된 설명 |

**5. 결과 검토 및 논의사항**

드모르간 정리를 Vivado에 구현을 하면서 연산자 and, or이 바뀜을 확인할 수 있었으며, literal은 서로 complement가 됨을 확인할 수 있었다.

1 bit 비교기 실습을 통해 논리연산자만을 이용하여, 두 비트가 같은지, 다른지, 아니면 어디가 더 큰지 판단할 수 있다는 점을 확인할 수 있었다.

이후 1 bit 비교기에서 더 나아가 2 bit 비교기 구현에 대해 이후 6번을 통해 논의해보겠다.

**6. 추가 이론 조사 및 작성**

2 bit 비교기도 1bit 비교기와 동일하게 equal, not equal, right bigger, left bigger을 수행하고, 이 또한 논리 연산자로 구현할 수 있다. 2 bit 비교기의 진리표는 아래와 같다.

1) 2 bit 비교기 진리표

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input | | Output | | | |
| A | B | A=B | A!=B | A>B | A<B |
| A1, A2 | B1, B2 | F1 | F2 | F3 | F4 |
| 00 | 00 | 1 | 0 | 0 | 0 |
| 01 | 0 | 1 | 0 | 1 |
| 10 | 0 | 1 | 0 | 1 |
| 11 | 0 | 1 | 0 | 1 |
| 01 | 00 | 0 | 1 | 1 | 0 |
| 01 | 1 | 0 | 0 | 0 |
| 10 | 0 | 1 | 0 | 1 |
| 11 | 0 | 1 | 0 | 1 |
| 10 | 00 | 0 | 1 | 1 | 0 |
| 01 | 0 | 1 | 1 | 0 |
| 10 | 1 | 0 | 0 | 0 |
| 11 | 0 | 1 | 0 | 1 |
| 11 | 00 | 0 | 1 | 1 | 0 |
| 01 | 0 | 1 | 1 | 0 |
| 10 | 0 | 1 | 1 | 0 |
| 11 | 1 | 0 | 0 | 0 |

logic gate로 표현하면 아래와 같다.

2) 2 bit 비교기 logic gate 표현

