**Lab1.**

# **Homework1. booth algorithm**

**Verilog Code / 주석**

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 문서, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 문서, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 폰트, 스크린샷, 대수학이(가) 표시된 사진

자동 생성된 설명

<Booth Algorithm Multiplier>

**Synthesis Report**

**텍스트, 폰트, 라인, 번호이(가) 표시된 사진

자동 생성된 설명**

**<Timing>**

**텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명**

**<Utilization>**

LUT(Look-up Table)을 17600개 중에서 38개를 사용하여 0.22 %의 사용 비욜이 도출되었다. 이는 논리 회로 구현을 위해 사용되고, 매우 작은 규모임을 나타낸다.

FF는 상태저장을 위해 사용되고, clock 신호에 맞춰 값이 변하는 register로 사용된다. 이는 전체 35200개의 사용비율 중에서 33개를 사용하여 0.09%의 매우 적은 사용량임을 확인할 수 있다.

IO(Onput/Output)은 FPGA와 외부 장치와의 interface를 위해 사용되는 IO핀이다. IO핀은 100개 중에서 34개를 사용하여, 전체 IO 핀의 34%를 차지하였다. 상대적으로 IO 핀 사용량이 높은 것은 해당 설계가 FPGA 내부에서 복잡한 연산보다 외부 장치와의 데이터 교환에 더 중점을 둔 설계임을 보여준다.

**텍스트, 스크린샷, 소프트웨어, 번호이(가) 표시된 사진

자동 생성된 설명**

**<Power>**

Power report는 FPGA 설계의 전력 소비와 열 관리 측면에서 중요한 정보를 제공한다. 이를 통해 설계가 어떻게 전력을 소모하는지, 열 관리가 적절히 이루어지는지 등을 파악할 수 있다.

**도표, 평면도, 라인, 기술 도면이(가) 표시된 사진

자동 생성된 설명**

**<Synthesis Schematic>**

**도표, 라인, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명**

**<RTL(Register Transfer Logic) Level Schematic>**

**Test Bench Code**

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

**Simulation Result**

스크린샷, 텍스트, 전자제품, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

첫번째 경우에는 Mplier가 -90, Mcand가 102인 경우인데, 이때, Booth algorithm을 통해 product 결과는 -9180으로 계산되었다. 이는 15bit 범위 내에 결과가 들어가가기 때문에 오버플로우는 발생하지 않았다.

두번째 경우는 Mplier가 102, Mcand가 51인 경우로 product는 5202로 계산되었다. 이 경우도 15bit 범위 내에 결과가 들어가기 때문에 오버플로우는 따로 발생하지 않았다. 만약 product값이 ±16383을 넘으면 오버플로우가 발생할 것이다. 예를 들면, Mplier=-128, Mcand=-128을 입력으로 설정하여 simulation을 진행한다면 최대 크기의 절댓값인 16384를 초과하기에 오버플로우가 발생할 것이다.

|  |
| --- |
| **Discussion** |
| . - **Verilog Coding을 시작하기 전 작성한 Block Diagram**    <Booth Multiplier block diagram>  위 그림은 Booth algorithm multilier의 하드웨어 구조를 나타낸 것이다. Add-Shift Controller에서는 Booth algorithm에 따라 B1, B0 값을 검사하여 다음 연산을 결정한다. St 신호가 활성화되면 곱셈 연산이 시작되고, Counter와 연결되어 있어 Booth algorithm의 반복 횟수를 카운팅받는다. 따라서, Counter가 7에 도달하면 곱셈 연산이 종료된다.  Multiplicand register C는 피승수를 저장하는 8bit register이다. 필요에 따라 8bit Complementer로 보내져 2의 보수 값으로 변환되어 뺄셈 연산에 사용된다. 8bit Complementer는 Multiplicand register C에서 나온 피승수 값을 2의 보수로 변환한다. Booth algorithm에서 B1B0 = 10일 때, 피승수의 2의 보수 값을 ACC에 더하여 뺄셈을 수행한다. 이로 인해 9bit Full Adder에서는 피승수 부호 반전에 의한 덧셈을 간단하게 수행할 수 있다. 9bit Full Adder는 ACC의 9bit와 8bit Complementer에서 나온 값을 받아서 덧셈 또는 뺄셈 연산을 수행한다. Carryout 신호를 통해 2의 보수를 취해야하는지 여부를 확인한다. 9bit ACC는 중간 덧셈 결과를 저장하고, 1bit의 부호 확장 비트를 포함하고 있다. RegA는 각 반복 연산 단계에서 Booth 연산 결과를 저장한다. 최종적으로 ACC는 product 출력에 필요한 값을 포함하게 되고, Booth algorithm의 shift 덧셈, 뺄셈을 반복하면서 결과를 도출한다.    <Booth Multiplier state diagram>  S0: Booth Algorithm의 초기 상태, St 신호가 1이 되면, 승수와 피승수를 로드하고, 상태를 S1으로 전이시킨다.  만약, St가 0이면 S0로 다시 돌아간다. 승수는 RegB에 로드되고, 피승수는 RegC에 로드된다.  S1: 덧셈 또는 뺄셈을 결정하는 단계이다. B[1]B[0]의 값에 따라 아래와 같은 조건이 선택된다.  B[1]B[0] == 01: 피승수(RegC)를 ACC에 더한다.  B[1]B[0] == 10: 피승수의 2를 2의 보수 취하고, ACC에 더한다.  B[1]B[0] == 00 or 11: 아무 연산도 수행하지 않고, 다음 단계로 넘어간다.  S2: shift 연산 단계이다. ACC와 RegB의 값을 오른쪽으로 1-bit씩 shift 동작한다. 이때, Counter를 증가시켜 Booth Algorithm이 8회 반복되도록한다.  - **작성한 Verilog Module에 대한 설명**  작성한 code는 Booth Algorithm을 사용하여 8-bit signed 정수 두 개를 곱하는 Booth\_Multiplier module이다. 이 module은 이상적인 FSM(Finite State Machine)을 사용하여 Booth Algorithm을 단계적으로 수행한다. Booth Algorithm은 승수의 하위 두 비트에 따라 덧셈 또는 뺄셈 연산을 선택하여 shift 연산을 통해 최종적으로 곱셈 연산 결과를 계산한다.  입력으로는 clk(clock), rst(reset), St(start), Mplier(8-bit 승수), Mcand(8-bit 피승수), 출력으로는 product(15-bit 결과값, 부호 비트 1bit+ 곱셈 결과 14bit)을 선언하였다. ACC는 accumulator 역할을 하는 9-bit register이고, 계산 중간 결과값을 저장한다. RegB는 승수를 저장하는 9-bit register로, 하위 두 비트인 B[0], B[1]을 통해 조건을 확인한다. 이때, 코드의 단순화와 가독성을 위해 RegB[1]을 B[1], RegB[0]을 B[0]으로 정의하였다. RegC는 피승수를 저장하는 8-bit register이다. Counter는 Booth Algorithm이 8회 반복되면 초기상태로 돌아가도록 카운팅하는 3-bit Counter이다. 코드를 작성하기 전 작성한 state diagram에서 정한 state S0,S1,S2를 각각 00,01,10으로 할당한다.  FSM 구조에서 state register 부는 rst이 활성화되면 상태를 S0로 초기화하고, 모든 register를 0으로 초기화한다. Rst이 비활성화되면 nst(다음 상태)에 따라 다음 상태로 전이한다. 위의 state diagram에서 기술했듯이 St가 1이 되면 RegB에 승수, RegC에 피승수를 로드하고, RegB[0]를 0으로 초기화하고, B[1]B[0] 값에 따라 덧셈 또는 뺄셈 동작을 선택한다. S2 상태로 넘어오면 ACC와 RegB(승수)를 모두 오른쪽으로 1-bit씩 shift한다. 이는 Counter가 7이 될 때까지 반복된다.  두번째 FSM 구조 중에서 next state logic에서는 앞 코드와 비슷한 동작 원리로 S0에서 St가 1이면 S1으로 상태전이가 발생하고, S1에서는 B[1]B[0] 조건에 따라 S2로 이동하거나 Counter에 따라 S1 hold, S0로 이동한다. S2에서는 Counter가 7이 아니면 S1으로 이동하여 반복 수행하고, 7이면 S0로 초기화하며, 곱셈 연산이 종료된다.  마지막 FSM 구조는 output logic이다. Carryout은 B[1]B[0] == 10일 때, 1이 되어 피승수의 2의 보수가 Cmout에 저장된다. Addout은 ACC와 Cmout을 더한 값으로, 위 조건이 성립하면 피승수의 보수를 더하는 연산을 수행하는 구조이다. 최종적으로 곱셈 연산 결과인 poduct는 ACC와 RegB의 상위 비트를 결합하여 출력한다.  - **작성한 Code가 어떤 동작을 하는지? 왜 그런 동작을 하는지?**  작성한 코드는 Booth Algorithm을 이용하여 두 개의 8-bit 수를 곱하는 15-bit 곱셈 연산 결과를 생성한다. Booth Algorithm은 곱셈 수행 시에 곱셈 속도를 높이고, 덧셈 연산 횟수를 줄여주는 algorithm이다. 여기서 핵심은 승수의 비트 패턴을 분석하여 곱셈에서의 불필요한 덧셈과 뺄셈을 줄이는 것이다.  St 신호가 활성화되면 Mplier와 Mcand를 RegB와 RegC register에 로드하고, S0 상태에서 S1 상태로 전이된다. B0 ^ B1 ==1 조건이 성립하면 S2 상태로 전이되어 shift 연산을 준비한다. Counter가 7에 도달하면 연산이 완료된 것으로 인식하고 S0로 돌아간다. Counter가 7이 아니라면 다시 S2에서 S1 상태로 넘어가 반복적으로 덧셈, 뺄셈을 진행한다. B0 B1 = 10일 때, 피승수의 2의 보수를 더하여 뺄셈을 수행한다. B0 B1 = 01일 때, 피승수를 더하여 덧셈을 수행한다. B0 B1 = 00 또는 B0 B1 = 11일 때는 아무 연산도 하지 않고 shift만 수행한다.  Carryout은 B1=1이고 B0=0일 때 1이 되며, 피승수 RegC의 2의 보수를 Cmout에 저장하여 뺄셈을 수행할 수 있도록 준비한다. ACC와 Cmout의 덧셈을 수행하여 Addout에 저장한다. Cmout은 조건에 따라 피승수의 보수 또는 원래의 값이 선택된다. 최종적으로 2의 보수 연산이 필요한 경우에는 Carryout을 고려하여 ACC에 더한다. S2 상태에서는 오른쪽으로 shift 동작을 수행하여 RegB와 ACC가 1-bit 씩 오른쪽으로 이동하도록 한다. 결론적으로 Booth Algorithm의 partial product generation & accumulation이 수행된다.  product는 연산이 완료되었을 때 곱의 연산 결과를 저장한다. 해당 값은 RegB와 ACC의 성분을 합하여 출력한다. 상위 7-bit는 ACC의 상위 bit를 부호 확장하여 부호를 유지하도록 한다. 반면에 하위 7-bit는 ACC의 하위 7-bit를, 마지막 8-bit는 RegB의 상위 8-bit를 결합하여 최종 결과를 도출한다.  **- 작성한 Test Bench Code가 어떤 동작을 하는지?**  작성한 test bench code는 슈\_Booth\_Multiplier module로, 앞서 작성한 Booth\_Multiplier module을 테스트하고자 작성하였다. 이 testbench는 Booth\_Multiplier의 입력 신호를 설정하고, 출력된 곱셈 연산 결과를 통해 module이 정상적으로 동작하는지 확인하는 데 목적이 있다.  먼저, 입출력 신호를 선언하고, Booth\_Multiplier를 인스턴스화 하고, 이름에 의한 연결을 한다. 10ns 주기로 clock 신호를 생성하였다. 먼저, rst=0일 때, 입력 신호를 모두 0으로 초기화 한다. 그러고 10ns 후에 rst를 다시 활성화해주었다. 첫번째 test case는 승수가 8’b10100110(-90), 피승수가 8’b01100110(-102)인 case로, 각각의 값을 Mplier, Mcand에 할당해주었다. 10ns 후에 St 신호를 활성화하여 Booth\_Multiplier module이 곱셈을 시작하도록 하였다. 그 후에 St가 계속 1로 유지되는 것을 방지하기 위해 10ns 후에 St를 0으로 비활성화해주었다. 즉, 1 clock cycle 동안만 활성화되도록 설정하였다. 연산 완료 후에는 Mplier, Mcand를 모두 0으로 설정하여 이후 두번째 test case에서 영향을 주지 않도록 초기화하였다. 그리고 곱셈 연산 완료 후에 결과가 안정화될 시간을 위해 200ns의 지연을 추가하였다.  두번째 test case는 승수가 8’b01100110(102), 피승수가 8’b00110011(51)인 case로 각각의 값을 Mplier, Mcand에 할당하였다. Case1과 동일하게 St를 활성화하고 1 clock cycle 후에 비활성화하여 곱셈 연산을 시작해주었다. 그 후에 연산이 종료되면 Mplier, Mcand를 각각 0으로 초기화하였고, 200ns의 지연을 추가하여 결과가 안정화되도록 설계하였다.  **- Simulation 파형이 왜 그렇게 나온것인지?**  이 simulation 파형은 Booth algorithm을 이용한 Booth\_Multiplier module이 두 개의 8-bit 정수를 곱하여 15-bit 결과를 출력하는 과정을 보여준다. 이 파형에서는 두 가지 case의 test가 순서대로 실행되었다. 첫 번째는 승수 -90과 피승수 102의 곱이고, 두 번째 case는 승수 102와 피승수 51의 곱이다. 각각의 case는 Booth algorithm의 동작에 따라 입력값을 처리하고, 이를 통해 곱셈 연산 결과를 출력한다.  처음 0~10ns구간에는 rst 가 활성화되어 module이 초기화된다. 그 후 Mplier에는 -90 (2의 보수: 10100110), Mcand에는 102 (01100110)가 입력된다. 이 상태에서 St가 활성화되며, Booth\_Multiplier module이 연산을 시작한다. 초기화가 완료되면 RegB는 {Mplier, 1'b0}로 설정되어 9-bit로 확장된 승수를 저장하고, RegC는 피승수를 저장한다. ACC와 Counter는 Booth algorithm이 진행될 동안에 중간 계산 결과와 반복 횟수를 카운팅하고, 저장한다.  Booth algorithm은 RegB의 하위 두 비트를 검사해 덧셈, 뺄셈, shift 연산을 결정한다. 첫 번째 반복에서 B0=0, B1=1이므로, ACC에 RegC 값을 더해 01100110 (102)이 된다. 이후 ACC와 RegB는 오른쪽으로 shoft되고, Counter는 1로 증가하게된다. 두 번째 반복에서는 B0=1, B1=0이고, ACC에 RegC의 2의 보수를 더해 -36이 된다. 이러한 연산이 8회 반복되며 Booth algorithm이 요구하는 연산을 수행한다. 해당 과정을 통해 첫 번째 곱셈의 최종 결과로 product에는 -9180이 출력되는 것을 simulation 파형을 통해 확인할 수 있다.  두 번째 case는 첫 번째 case의 연산 종료로부터200ns 시점에서 시작한다. 이번에는 승수 102 (01100110), 피승수 51 (00110011)을 입력하고, 10ns 후 rst가 다시 활성화되어 module이 초기화된 후 곱셈 연산이 시작된다. Booth algorithm의 초기 반복에서 B0=0, B1=0이므로 아무 연산도 수행하지 않고 shift만 수행한다. 이후 조건에 맞는 덧셈 또는 뺄셈이 반복되며 ACC와 RegB가 업데이트된다. Booth algorithm의 반복이 완료되면 최종 곱셈 결과로 product에 5202가 출력되는 것을 simulation 파형을 통해 확인할 수 있다.  이 simulation 파형은 Booth\_Multiplier module이 Booth algorithm을 정확하게 구현했음을 보여준다. 각 연산이 예상한대로 수행되었고, 두 test case의 결과 -9180과 5202는 수학적 계산 결과와 일치하는 것을 확인할 수 있다. 이를 통해 Booth algorithm에 따라 조건부 덧셈, 뺄셈, shift가 성공적으로 수행되었음을 확인할 수 있다.  **- 작성한 Code가 잘 동작하지 않는다면 어디서 문제가 발생한 것인지?**  작성한 code는 정상적으로 동작하였다.  **- 오류를 해결했다면, 어떤 부분에서 해결을 했고, 왜 오류가 발생하였는지?** |

작성한 code는 정상적으로 동작하였고, 발생한 오류는 없다.