EEE 3544 System IC Design  
HW 2: Adder & Multiplier

2021142180 Kim Min Chan

Date: 2024.05.12

**[Contents]**

**[1. Introduction]**

**[2. Background]**

2.1 RCA

2.2 CSA

2.3 Combinational Multiplier

2.4 Design Compiler

**[3. Verilog Design]**

3.1 Top Module

3.2 32-Bit RCA

3.3 32-Bit CSA

3.4 32-bit Multiplier

**[4. Simulation & Analysis]**

4.1 Simulation Result

4.2 Result Analysis

**[5. Design Compiler]**

5.1 Synthesis Result

5.2 Timing Report Analysis

5.3 Area Report Analysis

**[6. Discussion]**

**[1. Introduction]**

본 과제에서는 Adder와 Multiplier을 직접 설계한 뒤, 이를 바탕으로 두 가지 Functional Unit을 갖는 Top Module을 설계할 것이다. 여기서 Adder은 **Ripple Carry Adder(RCA)**와 **Carry Select Adder(CSA)**을 사용할 것이며, Multiplier은 Adder을 이용한 **Simple Combinational Multiplier**을 이용할 것이다.

Verilog Design 이후, 해당 Logic Circuit이 올바르게 동작하는지 Behavioral Simulation을 거친 뒤, **Design Compiler**을 이용하여 **Synthesis** 과정까지 진행할 것이다. 이때 **Timing Report**와 **Area Report**을 분석하여 Design의 성능과 유효성 등을 검토할 것이다.

**[2. Background]**

**2.1 RCA**

텍스트, 도표, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

**Figure 2.1 32-bit RCA**

**RCA, Ripple Carry Adder**은 가장 간단한 형태의 Adder로, 1-bit Full Adder가 **serial 하게 연결**되어 구성된다. Figure 2.1은 32-bit RCA을 나타낸 것이다. 4-bit RCA 8개가 serial하게 연결되어 있는 것을 알 수 있다. 이전 단계의 RCA의 Carry Out 이 다음 단계의 Carry in 로 입력된다. 4-bit RCA의 latency을 라 했을 때, wire delay 등을 무시하면 **의 latency**을 갖는다.

도표, 기술 도면, 평면도, 라인이(가) 표시된 사진

자동 생성된 설명

**Figure 2.2 4-bit RCA**

Figure 2.2은 4-bit RCA을 나타낸 것이다. 1-bit Full Adder 4개가 serial 하게 연결되어 구성된다. 이전 단계 FA의 이 다음 단계의 로 입력된다.

RCA은 가장 간단한 형태의 Adder로, bit 수만큼의 1-bit Full Adder로 구성된다. 따라서 **작은 Fan-Out**을 갖는 반면, **높은 latency(delay)**을 갖는다.

**2.2 CSA**

텍스트, 도표, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

**Figure 2.3 32-bit CSA**

**CSA, Carry Select Adder**은 RCA의 단점을 보완하기 위해 고안된 Adder이다. RCA은 이전 단계의 Carry Out, 이 출력되기 전까지 다음 단계의 계산이 이루어질 수 없어 덧셈 연산이 serial하게 이루어지며, 이에 따라 긴 delay을 갖는다.

이를 보완하기 위해 CSA은 모든 가능한 Carry in, 에 대해 **미리 연산을 수행**한 뒤, 이전 단계의 carry에 따라 MUX을 이용해 **결과를 선택**하는 방식으로 연산을 수행한다. 예를 들어 n번째 단계의 4-bit CSA은 인 경우에 대해 모두 연산을 수행하여 을 계산해 놓은 뒤, n-1번째 단계의 이면, 을 선택하는 것이다.

32-bit CSA은 1개의 4-bit RCA와 7개의 4-bit CSA로 구성되며, 4-bit CSA은 2개의 RCA로 구성된다. 따라서 32-bit CSA은 총 **15개의 4-bit RCA와 35개의 2-to-1 MUX**로 구성된다. RCA와 비교해보았을 때 **높은 Fan-out**을 갖는 것을 알 수 있다.

4-bit RCA가 , 2-to-1 MUX가 의 delay을 갖는다면, 32-bit CSA은 **의 delay**을 갖게 된다. 32-bit RCA와 비교해보았을 때, MUX의 delay 이 4-bit RCA의 delay 에 비해 무시할 수 있을 정도로 매우 작다면, **약 7배**의 성능향상을 달성할 수 있다.

하지만 CSA은 모든 Carry에 대해 병렬적으로 연산을 수행해야 하므로 복잡도가 증가하고, 높은 Fan-out을 갖는다는 단점이 존재한다.

**2.3 Combinational Multiplier**

도표, 라인, 스케치, 평면도이(가) 표시된 사진

자동 생성된 설명

**Figure 2.5 Simple Combinational Multiplier**

Figure 2.5은 Simple Combinational Multiplier을 나타낸 것이다. Multiplier의 구조에 대해 설명하기 이전에, binary의 multiplication과정에 대해 설명하도록 하겠다.

Multiplication은 기본적으로 덧셈으로 구현된다. Multiplier가 n-bit라 가정하면, n번의 덧셈과 마지막 1번의 덧셈으로 구현된다. 앞 n 단계에서는 이전 단계의 Carry와 Sum그리고 현재 단계의 Multiplicand의 덧셈이 이루어진다. 그리고 마지막 단계에서는 n번째 단계의 SUM과 carry을 더하는 과정을 수행한다.

설명을 위해 Multiplicand을 A, Multiplier을 B 라 하면, n 번째 단계에서는 다음의 연산이 이루어진다.

즉, 각 단계는 이전 단계의 결과에 대해 dependent하므로 serial 하게 연결된다. Multiplier가 N-bit이고, 각 단계의 latency을 , 마지막 단계에 사용되는 RCA의 latency을 라 하면 Multiplier은 의 delay을 갖는다.

**2.4 Design Compiler**

텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

자동 생성된 설명

**Figure 2.6 HW Design Flow**

Figure 2.6은 **Hardware Design Flow**을 나타낸 것이다. 여기서 Verilog을 통해 RTL Design을 구현한 후, Testbench을 작성하여 설계한 design이 올바르게 동작하는지 verify한다. Functional Verification이 완료되면, Logic Synthesis 과정을 통해 실제 회로를 구현하게 된다. 본 과제에서는 Design Compiler을 이용하여 Logical Synthesis 과정을 진행할 것이다.

Design Compiler은 합성 가능한 Verilog design을 standard cell library을 target으로 한 netlist로 compile하는 역할을 수행한다.

텍스트, 폰트, 스크린샷, 도표이(가) 표시된 사진

자동 생성된 설명

**Figure 2.7 Design Compile**

Figure 2.7은 Design Compile의 input와 output을 나타낸 것으로, Verilog Code와 Design Constraints, 그리고 Cell Library가 입력되어 Netlist을 비롯한 Timing Report, Area Report 등이 출력되는 것을 확인할 수 있다. 여기서 Design Constraint은 compile 과정 중에 사용되는 제약 조건을 의미하며, Design Rule Constraints, Timing Constraints 등이 있다.

텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

자동 생성된 설명

**Figure 2.8 Design Compile Procedure**

Figure 2.8은 design compile 과정을 나타낸 것이다. 먼저 target(ASIC, FPGA 등)의 standard cell library을 load한다. 이후, HDL file을 읽고, 분석한 뒤 elaborate한다. 여기서 elaborate 과정은 module instance들을 서로 연결하여 hierarchy을 구성하며, parameter 값을 계산하는 등 simulation을 위한 준비 과정이다.

이후 design environment parameter을 설정하고, design rule와 design constraint을 설정한다. 이후 compile을 진행하면 netlist와 report가 출력된다.

Design Compile 과정을 통해 HDL을 바탕으로 실제 netlist을 생성할 수 있으며, 실제 하드웨어 레벨의 area, timing와 같은 spec을 분석할 수 있다. 이를 바탕으로 구체적인 spec에 맞춰 설계를 진행하는 과정이 진행된다.

**[3. Verilog Design]**

**3.1 Top Module**

도표, 스크린샷, 라인, 평면도이(가) 표시된 사진

자동 생성된 설명

**Figure 3.1 Top Module Diagram**

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

**Figure 3.2 Top Module Verilog Code**

Figure 3.1은 Top Module을 Diagram으로 나타낸 것이다. 32-bit의 input A, B가 입력되어 register에 저장되고, 해당 값이 module A, B에 입력된 후, 각 module의 결과 값이 Sel input에 의해 결정되어 출력된다. Input와 Output 값을 register에 저장하는 것은 입력 및 출력을 clock signal에 synchronize하기 위함이다.

각 모듈은 RCA, CSA, Multiplier 중 하나이다. 여기서 주의해야 할 점은 module 종류에 따라 input 및 output의 bit width가 다르다는 점이다. Module이 Adder일 경우 출력이 32-bit이므로, DFF\_Result에 값을 extension하여 저장해야 한다. 반면 module이 Multiplier일 경우 Multiplier가 8bit이므로 truncate하여 값을 입력해야 한다.

Figure 3.2은 위 사항을 바탕으로 Top Module을 작성한 것이다. Clock signal에 의해 synchronize하고, asynchronous reset을 구현하기 위하여 sensitivity list을 위와 같이 설정하였다. DFF\_Result에 값을 입력하는 부분(line #25) 을 보면, Sel bit에 따라 module A와 B의 결과 중 선택하도록 설계하였으며, RCA로부터 출력되는 Result\_A의 경우 extension을 진행한 뒤 값을 입력해주었다.

**3.2 32-Bit RCA**

텍스트, 도표, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

**Figure 3.3 32-bit RCA**

텍스트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명

**Figure 3.4 32-bit RCA Verilog Code**

Figure 3.3은 32-bit RCA의 block diagram을 나타낸 것이다. 32bit의 input A, B와 1bit의 Cin을 입력 받는다.

Figure 3.4은 32-bit RCA의 Verilog code을 나타낸 것이다. 앞서 설명한 것과 마찬가지로, 이전 단계의 carry out이 다음 단계의 carry in으로 입력되어 serial 하게 연결되도록 구성하였다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

**Figure 3.5 4-bit RCA Verilog Code**

4-bit RCA 역시 이전 설명과 마찬가지로, 이전 단계의 FA의 Cout이 다음 단계의 Cin으로 입력되도록 설계하였다.

**3.3 32-Bit CSA**

텍스트, 도표, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

**Figure 3.5 32-bit CSA Diagram**

Figure 3.6은 32-bit CSA의 Verilog code 일부를 나타낸 것이다. 첫 번째 단계는 4-bit RCA로 구성되어 있으며 이후 7개의 단계는 4-bit CSA로 구성된다. MUX Selection 부분을 보면 각 단계의 Sum와 Carry Out이 이전 단계의 Carry Out에 의해 선택되는 것을 확인할 수 있다.

Figure 3.7은 4-bit CSA의 Verilog Code을 나타낸 것으로, 2 개의 4-bit RCA로 구성된다. Cin에 따라 Sum와 Cout이 선택되어 출력된다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**Figure 3.6 32-bit CSA Verilog Code**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**Figure 3.7 4-bit CSA Verilog Code**

**3.4 32-bit Multiplier**

텍스트, 스크린샷, 소프트웨어, 번호이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명

**Figure 3.8 32-bit Multiplier Verilog Code**

Figure 3.8은 32-bit Multiplier의 Verilog을 나타낸 것이다. 8개의 단계(ROW\_MUL module)와 마지막 Sum을 위한 32-bit RCA로 이루어진다. 각 단계의 ROW\_MUL은 A와 B의 n번째 bit, 이전 단계의 Sout[31:1], 그리고 이전 단계의 Cout을 입력 받는다. 그리고 Sout와 Cout을 출력하여 다음 단계로 입력된다. 결과인 Y은 각 단계의 SUM의 LSB와 마지막 SUM, Sout을 Concatenation하여 출력한다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

**Figure 3.9 ROW\_MUL module Verilog Code**

Figure 3.9은 각 단계의 연산을 수행하는 ROW\_MUL module이다. 32bit의 A, Sin, Cin와 1bit의 B을 입력 받으며, 32bit의 Sout, Cout을 출력한다. 해당 모듈은 32개의 1-bit FA로 구성된다. 각 FA의 A에는 A&B가 입력되며, B에는 Sin, Cin에는 Cin이 입력된다. 결과는 Sout, Cout으로 출력된다.

여기서 32개의 FA module instance을 모두 따로 불러오는 대신 모듈 배열을 이용하여 32개의 module instance을 한 번에 불러왔다.

**[4. Simulation & Analysis]**

**4.1 Simulation Result**

**스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명**

**Figure 4.1 Simulation Results**

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 메뉴이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

**Figure 4.2 Testbench Verilog Code**

Figure 4.2은 Testbench Verilog code을 나타낸 것이다. Testcase은 각 연산의 기본적인 정확성을 확인할 수 있는 경우와 예외적인 corner case을 확인할 수 있는 경우로 구성하였다.

먼저 입력 A, B에 0이 포함되어 있는 경우, A, B가 모두 0이 아닌 경우, 그리고 A, B의 연산 결과가 출력 bit width을 초과하는 경우로 case을 구성하였다.

Testbench을 바탕으로 simulation을 진행한 결과 Figure 4.1와 같은 waveform을 얻을 수 있었다.

**4.2 Result Analysis**

해당 과정은 Functional Verification에 해당하는 과정으로, 설계한 design의 기능이 spec와 일치하는지 검증하는 단계이다. SPEC의 기본적인 functionality은 물론, 예외적인 corner case가 존재한다면, 해당 경우에 어떻게 동작하는지에 대해 검증하는 것이 중요하다.

먼저 clk, Sel은 0으로, nrst은 1로 초기화된다. 첫 cycle에 nrst가 입력되고, 두 번째 cycle에 값이 초기화된다. 다음 cycle부터 순차적으로 값이 입력되고, 그에 대한 결과가 한 cycle 뒤에 출력되는 것을 알 수 있다. Test case가 모두 끝난 뒤에 Sel을 1로 설정하여 module B의 결과를 확인하였다.

그 결과 각 test case에 대해 연산이 올바르게 출력되는 것을 알 수 있다. 5번째 test case의 경우 와 이 입력되는데, 이 경우 Adder 모듈에서는 bit overflow가 발생하여 0이 출력되는 것을 확인할 수 있다.

이를 통해 각 module이 모두 올바르게 동작함을 확인할 수 있으며, 앞서 설계한 HDL Design이 Functional Verification을 통과했다고 생각할 수 있다.

**[5. Design Compiler]**

다음으로 synopsis design compiler을 이용하여 Logical Synthesis을 진행하였다. 앞서 설명했듯, logical synthesis 과정을 통해 HDL 레벨의 디자인은 실제 physical circuit netlist로 변환된다. 이때 clock speed와 같은 timing constraint 등 design constraint와 rule constraint을 추가하여 area, timing와 관련된 report을 생성할 수 있다. 이를 바탕으로, 해당 디자인이 SPEC을 만족한다면 그 다음 과정을 진행하고, 만족하지 않으면 디자인을 수정하는 방식으로 진행된다.

**5.1 Synthesis Result**

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명

**Figure 5.1 Compile Scripts for Top\_1, option 1**

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

**Table 5.1 Compile Results**

Figure 5.1은 Top\_1을 Compile을 하는데 사용한 scripts이다. 먼저 디자인의 이름을 Top\_1으로 설정하고, report와 result가 저장될 directory을 설정한다. 다음으로 해당 디자인을 compile하는데 필요한 모든 모듈의 HDL file을 analyze한다. 이때 Verilog을 이용하여 설계를 진행했으므로, “-format Verilog”로 option을 준다. 다음으로 elaborate을 진행한다. 앞서 설명했듯, elaborate은 module instance을 서로 연결하여 Hierarchy을 구성하는 등, synthesis을 위한 준비 과정에 해당한다. Current design을 앞서 설정했던 design name로 설정해준 뒤, timing constraint을 추가한다. 해당 과정에서는 100MHz의 clock을 이용할 것이므로, “-period 10”을 option로 부여하였다. 또한 rising edge와 falling edge가 같은 비율로 존재하길 원하므로 “-waveform {0 5}”을 부여했다. 이렇게 compile을 위한 모든 준비를 맞췄으면, compile을 진행한다. 마지막으로 compile 결과를 파일로 저장한 뒤 exit한다.

Table 5.1은 Design Compile 결과를 나타낸 것이다. 이에 대한 분석은 **[5.2 Timing Report Analysis]** 및 **[5.3 Area Report Analysis]** 에서 자세하게 진행하였다.

**5.2 Timing Report Analysis**

Timing Report에서는 각 Path에서의 delay을 분석하여 data arrival time을 계산하고 최종적으로는 Slack 값을 계산한다. 분석을 위해 set up time, data arrival time, slack 등에 대해 간단하게 설명한다.

먼저 **Set Up Time**이란, register가 input data을 인식하기 위해 edge 이전에 필요한 최소 시간을 의미한다. Register은 clock에 synchronize되어 있으므로, 매 rising(falling) edge에 input을 인식하는데, 이를 위해서는 Set Up Time 만큼 data가 edge보다 먼저 도착해야 한다는 뜻이다.

다음으로 **Hold Up Time**이란, register가 Input data을 사용하기 위해 edge 이후 필요한 최소 시간을 의미한다. 즉, Edge 이후 Hold Up Time 만큼 data가 유지되어야 한다.

**Data Arrival Time**이란 말 그대로 data가 도착한 시간을 의미한다. 마지막으로, **Slack**이란 data가 요구되는 시간보다 얼마나 빠르게 도착했는지 나타내는 parameter이다. 따라서 Slack은 required time – data arrival time로 계산할 수 있으며, required time은 set up time을 뺀 시간이다. 따라서 Slack은 양의 값을 가져야 하며, 음의 값을 갖는 경우, data가 제한된 시간 내에 도착하지 못하였음을 나타낸다. 이 경우 결과가 input data가 도착한 바로 다음 clock에 출력되지 못하고 기다려야 함을 의미한다.

텍스트, 번호, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**Table 5.2 Slack**

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

**Table 5.3 Data Arrival/Required Time**

Figure 5.2은 각 Combination와 Clock Speed 별 slack 값을, Figure 5.3은 Data Arrival Time와 Data Required Time을 나타낸 것이다. Slack은 Required Time에서 Arrival Time을 뺀 것임을 확인할 수 있다.

Clock Speed가 100MHz인 경우, Slack은 모두 큰 양수 값을 갖는다. 이는 required time보다 해당 값만큼 빠르게 도달했음을 의미하며, 따라서 값이 입력된 바로 다음 clock cycle에 값이 출력됨을 알 수 있다. 반면 Clock Speed가 1GHz인 경우 COMB3, COMB5은 slack이 음수임을 확인할 수 있다. 이는 결과가 required time 내에 도달하지 못했음을 의미한다. 특히 COMB3의 경우 매우 큰 크기의 slack을 갖는 것을 알 수 있다. 해당 디자인의 Data Arrival Time을 보면 clock speed가 100MHz인 경우보다 큰 것을 확인할 수 있다. 이를 자세하게 분석하기 위해, 해당 timing reports을 분석하였다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

**Table 5.4 Path Delay – COMB3, 1GHz**

Figure 5.4은 COMB3, clock speed가 1GHz인 경우, timing report에서 가장 큰 path delay을 갖는 path에 대한 정보이다. module\_B/ROW2/ROW[19] /U8/Y에서 큰 딜레이가 발생했음을 알 수 있다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

**Figure 5.2 IBUFX32 SPEC**

Figure 5.2은 IBUFX32 cell library의 spec을 나타낸 것이다. 해당 표를 통해 IBUFX32의 operating frequency은 300MHz임을 알 수 있다. Operating Frequency보다 훨씬 높은 1GHz의 Clock Speed을 입력하였기 때문에 문제가 발생한 것으로 생각할 수 있다. 이를 해결하기 위해 먼저 “compile\_ultra” command을 이용해 optimization을 진행하였다. 그 결과 required time은 0.98, arrival time은 0.98로 slack은 0임을 확인할 수 있었다.

또한 slack이 양수가 되도록 하는 clock frequency을 찾기 위해, operating frequency 이내인 300MHz의 clock speed을 사용하여 compile해보았다. 그 결과 data required time은 4.98, data arrival time은 4.35, slack은 0.63로 timing violation이 발생하지 않음을 확인할 수 있었다.

Clock Speed을 바꾸지 않고 해당 문제를 해결하기 위해 design을 수정하였다. 분석 결과 해당 path에서 delay가 길게 나오는 것은 asynchronous reset이 입력되었을 때 DFF\_Result에 40’bZ을 입력하기 때문이다. 따라서 이를 없애고 다시 compile 한 결과 아래와 같은 결과를 얻을 수 있었다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

**Table 5.5 Slack & Total Area – Revised.**

다음으로 Design 별 Slack을 비교해보았다. COMB1은 module A, B가 모두 32-bit RCA, COMB2은 module A, B가 모두 32-bit CSA이다. 두 design의 slack을 비교해보면, clock speed가 1GHz일 때와 100MHz일 때 모두 COMB2, 즉 CSA가 근소하게 빠른 것을 확인할 수 있다. 이를 통해 CSA의 delay가 RCA의 delay보다 빠르다는 것을 확인할 수 있었다. 하지만 이론적으로 계산한 정도의 delay 차이는 아니었다. 이는 이론적으로 계산한 delay의 경우 wire delay 등 다른 요소를 배제하고 계산한 것이기 때문이다. 실제로 CSA은 RCA에 비해 매우 높은 Fan-Out을 가지며, Routing 등 추가적인 과정이 존재하므로 이론적인 속도 증가에 비해 작은 속도 증가를 보임을 알 수 있다.

이번에는 COMB1, COMB2, COMB4의 timing을 분석해보았다. 보다 자세한 분석을 위해 critical path을 확인해보았다. COMB1와 COMB2은 같은 모듈을 사용하므로 의미가 없고, COMB4의 critical path은 모듈 A, 즉 RCA임을 확인할 수 있었다. 이를 통해 delay가 긴 module이 critical path에 포함된다는 것을 다시 한번 확인할 수 있었다.

COMB5와 COMB6을 비교해보면, clock speed가 1GHz인 경우 COMB5은 timing constraint을 만족하지 못하는 반면, COMB6은 timing constraint을 만족하는 것을 알 수 있다. 이를 분석하기 위해 timing report에서 critical path을 분석하였다. 그 결과 COMB5와 COMB6 모두 critical path에 module B, 즉 32-bit Multiplier가 포함됨을 알 수 있었다. 하지만 compile 결과 생성된 netlist을 비교해본 결과, COMB5와 COMB6의 multiplier 모듈은 서로 다른 netlist을 가짐을 알 수 있었다. 때문에 서로 다른 critical path을 갖고, COMB5가 COMB6에 비해 0.03ns 긴 delay을 갖는 것을 알 수 있었다. 또한, COMB6은 library set up time이 0.02 ns인데 반해, COMB5은 library set up time이 0.05ns인 것을 확인할 수 있었다. Clock speed가 비교적 낮은 경우 이러한 Set Up Time의 차이는 실제 결과에 거의 영향을 주지 않지만, clock speed가 점차 높아짐에 따라 Set Up Time은 timing constraint을 만족시키는데 매우 큰 영향을 주기 때문에, 이를 적절하게 고려해야 함을 알 수 있었다.

또한 같은 모듈을 사용하더라도, 전체 design에 따라 netlist가 다르게 구성될 수 있음 역시 확인할 수 있었다.

**5.3 Area Report Analysis**

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명 텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

**Table 5.6 Area – 100MHz/1GHz**

Table 5.6은 앞서 compile을 진행하여 얻은 Area Report을 나타낸 것이다. Area Report에서는 크게 두 가지를 비교할 수 있다. 첫 번째, 같은 Clock Speed에서 서로 다른 Design 간의 Area을 비교할 수 있다. 두 번째, 같은 Design에서 Clock Speed에 따른 Area을 비교할 수 있다.

먼저 같은 Clock Speed 내에서 각 design 별 Area을 비교해보았다. COMB1은 RCA 두 개로 구성되어 있고, COMB2은 CSA 두 개로 구성되어 있다. 앞서 설명했듯, CSA가 RCA에 비해 높은 Fan-out을 가지므로 더 높은 Combination Area을 갖는 것을 알 수 있다. 표를 통해 확인해보면 약 4배의 combinational area을 갖는 것을 알 수 있다. 이는 RCA와 비교했을 때, 2배에 달하는 4-bit RCA와 35개의 추가적인 MUX, 그리고 이렇게 증가한 combinational block을 간섭없이 배치하기 위함으로 생각해볼 수 있다.

RCA와 CSA을 하나씩 사용한 COMB4의 경우 COMB1와 2의 평균에 가까운 AREA을 갖는 것을 알 수 있다.

Multiplier의 경우 매우 큰 combinational area을 갖는 것을 알 수 있다. 이는 Multiplier가 총 의 매우 많은 수의 1-bit FA로 구성되기 때문이다. COMB5, COMB6의 경우 각각 Multiplier와 RCA, CSA 평균정도의 AREA을 갖는 것을 알 수 있다.

다음으로 Non-Combinational Area을 비교해보면 ADDER가 포함된 design의 해당 면적이 크다는 것을 확인할 수 있다. 이는 Multiplier은 Result의 bit width가 정확히 일치하므로 추가적인 과정이 필요 없으나, Adder은 Result의 bit width을 맞추기 위한 extension 과정이 필요하기 때문으로 생각할 수 있다.

마지막으로 Net Interconnection Area을 비교해볼 수 있다. COMB1와 COMB2을 비교해보면, COMB2가 COMB1에 비해 2배 가량 큰 Net Interconnection Area을 갖는다. 이는 CSA의 결과를 올바르게 Routing하는 과정에서 필요한 추가적인 Interconnection로 생각할 수 있다. COMB3 역시 Multiplier은 각 stage의 결과를 다음 Stage로 Routing하는데 있어서 많은 Interconnection을 필요로 하므로 큰 Net Interconnection Area을 갖는 것을 확인할 수 있다.

두 번째로, 같은 Design에서 Clock Speed에 따른 Area을 비교해보았다. 가장 눈에 띄는 변화는 바로 Buf/Inv Area이다. 해당 table에서 Clock Speed가 증가함에 따라 Buf/Inv area가 크게 증가한 것을 확인할 수 있다. Buf/Inv area에 포함된 component에는 Standard Buffer, Inversion Buffer, Clock Buffer, High drive Buffer, Low drive Buffer 등이 포함된다. 여기서 주목해야 할 component은 Clock Buffer이다. Clock Buffer은 module에 최소의 skew와 jitter로 clock signal을 전달하기 위해 존재한다. Clock Speed가 증가함에 따라, Clock skew와 jitter을 줄이고, integrity을 유지하기 위해 많은 수의 Clock Buffer가 추가적으로 사용된다. 때문에 Clock Speed의 증가는 큰 Buf/Inv Area로 이어지고, Total Area가 증가하게 된다.

**[6. Discussion]**

지금까지 과제에서 주어진 SPEC의 Hardware을 HDL level에서 디자인한 뒤, Functional Verification을 진행하고, Design Compiler을 이용하여 Logical Synthesis까지 진행해보았다. Logical Synthesis 과정에서 Timing reports와 Area reports을 분석하고, Delay가 한 cycle 이내에 위치하도록 코드를 수정하는 과정까지 진행해보았다.

본 과제에서는 Adder와 Multiplier가 모두 Clock Speed가 최대 1GHz 1 cycle 이내에 동작하도록 설계하였다. 하지만 현대의 processor은 4~5GHz라는 매우 높은 clock을 갖는다. Adder의 경우 각종 Optimization을 통해 해당 Clock speed에서도 Delay가 1 cycle 이내가 되도록 설계할 수 있다. 하지만 Multiplier의 경우, 한 cycle 이내에 동작하기 어렵다. 더 나아가 Floating Point 연산의 경우 비교적 매우 긴 delay을 갖는다. 따라서 연산을 1 Cycle 이내에 하기 보다, pipelining을 통해 throughput은 유지하면서 Clock Cycle을 보다 빠르게 할 수 있을 것이다.

본 과제에서 디자인한 Multiplier을 예로 들어 설명하면, 각 Multiplier은 총 9개의 stage로 구성된다고 생각할 수 있다. 따라서 각 단계의 연산은 1 cycle 내에 완성하도록 설계한 뒤에, 매 cycle마다 input을 넣어주면, 각 stage마다 다른 연산을 수행할 수 있다. Pipeline Stage가 모두 채워진 경우에는 매 clock마다 결과를 출력할 수 있으므로, IPC은 유지하고, clock speed은 높임으로써 performance 향상을 달성할 수 있을 것이라 생각한다.