컴퓨터 공학 기초 실험2 보고서

실험제목 : Carry Look-Ahead Adder

& Compare with Ripple Carry Adder

(+ 7-segment decoder)

실험일자 : 2016년 09월 08일 (목)

제출일자 : 2016년 09월 21일 (수)

학 과: 컴퓨터공학과

담당교수: 이혁준 교수님

실습분반: 목요일 0, 1, 2

학 번: 2013722095

성 명: 최재은

1. 제목 및 목적
   1. 제목

Carry Look-Ahead Adder & Compare with Ripple Carry Adder

* 1. 목적

Carry Look-Ahead Adder의 기본개념을 이해하고, 구성을 확인하고 작성하여 최종적으로 32비트를 계산할 수 있는 Adder를 만들어본다. 기존에 만들었던 4비트 연산 Ripple Carry Adder 또한 32비트를 연산할 수 있도록 만들어본다. flip-flop을 추가하여 clock에 따라 작동하게 하여, 해당 모듈이 실제 작동할 때 한 클럭당 걸리는 시간을 비교해봄으로써 어떤 모듈이, 왜 더 효율적인지를 알아본다.

1. 원리(배경지식)

후위 모듈에서 발생한 올림값을 carry in으로 받아 연산에 적용하는 방법을 사용하는 Ripple Carry Adder는 후위 모듈에서의 연산이 끝나야 다음모듈을 실행할 수 있기 때문에 계산할 비트가 많아질수록 full adder 또한 많아진다. 이러한 이유로 하나의 full adder에 소요되는 시간을 T라고 하면 n개의 비트를 계산하는데 소요되는 시간은 n\*T가 된다. 즉, 소요시간이 매우 비효율적으로 길다고 볼 수 있다. 이에 비해 Carry look-ahead Adder(CLA)는 입력된 각 비트의 올림값을 따로 계산하는 block module을 따로 가지고 있어서, block module이 계산하는데 걸리는 소요시간만이 대부분의 소요시간이다. 즉 내부의 후위 비트 계산 모듈의 완료여하에 영향을 받지 않는다는 개념이다. 각 비트는 합을 계산할 때, block module에서 계산된 해당 carry in(올림값)을 받는다. 32 bit CLA는 4 bit CLA 8개를 Ripple 형식으로 연결하여 만든 것으로 이에 대한 소요 시간이 추가적으로 발생한다. CLB는 한번에 4개의 비트를 받아서 이에 대한 올림값을 계산해주는 모듈로써, 1비트 두 개의 합의 올림값, 2비트 두 개의 합의 올림값, 3비트 두 개의 합의 올림값, 4비트 두 개의 합의 올림값을 미리 계산해준다. 이러한 모듈이 소요하는 시간을 알아보기 위해 TimeQuest timing Analyzer를 사용하는데, 이때 발생한 slack을 통해 실제 실제로 clock을 얼마나 더 주어야 하는지를 알 수 있다. 이 때 해당 모듈의 frequency를 알 수 있는데 이를 통해 실제로 모듈이 한 클럭당 소요하는 시간을 계산할 수 있다(frequency를 역수로 취함으로써). 이를 통해서 어떤 방식의 Adder가 더 효율적인지를 확인할 수 있는 것이다.

1. 설계 세부사항

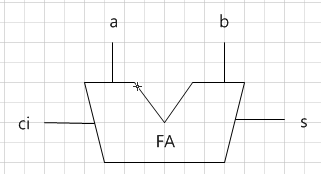
1) Full Adder

|  |  |  |  |
| --- | --- | --- | --- |
| Input | | | Output |
| Ci | a | b | s |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

- RCA에서와 다르게 CLB에서 올림값을 계산해 주기 때문에 Carry out이 필요 없습니다.

- half adder를 Instance하지 않고 두 개의 Exclusive Or gate를 이용하여 구현하였습니다.

- 다음과 같은 형태로 표현됩니다.



2) 4’b CLA

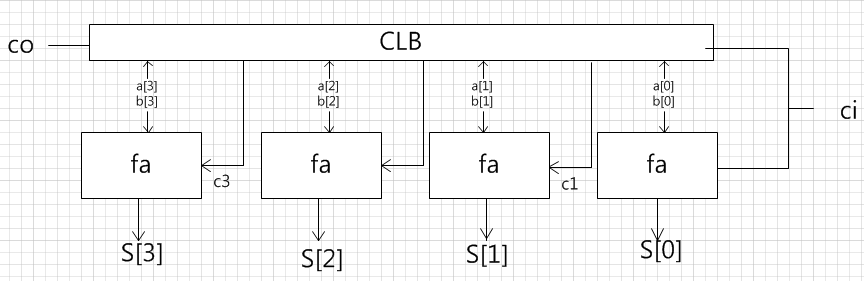
- RCA에서의 계산시간이 오래 걸리는 것을 보완한 Adder.

- 모든 올림 수를 계산하기 위해 Carry look-ahead block을 만들어 주어야 합니다.

- 4개의 full adder를 instance하여 구현되었으며, 각 full adder는 각 자릿수의 비트와 CLB에서 발생한

올림값을 받아 합을 연산하여 s에 저장해 줍니다.

- 다음과 같이 표현될 수 있습니다.



- 인자의 구성은 다음과 같습니다.

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Name | Bandwidth (bit) | Description |
| Input | a | 4 | Input data a |
| b | 4 | Input data b |
| ci | 1 | Carry in |
| Output | co | 1 | Carry out |
| s | 4 | Sum |
| Wire | c1 | 1 | Internal carry |
| c2 | 1 | Internal carry |
| c3 | 1 | Internal carry |

- 모듈의 구성은 다음과 같습니다.(full adder 4개를 instance하여 사용하였습니다.)

|  |  |  |
| --- | --- | --- |
| Classification | Name | Description |
| Module | cla4 | 4 bit Carry Look-ahead Adder |
| Instance | U0\_fa\_v2 | Full Adder |
| U1\_fa\_v2 |
| U2\_fa\_v2 |
| U3\_fa\_v2 |
| U4\_clb4 | Carry generation |

● 4’b CLB

- CLB로 carry out 값을 미리 계산하기 위해서는 generation signal(Gn), propagation signal(Pn)을 정의하여야 하는데, Gn = An \* Bn / Pn = An + Bn으로 정의할 수 있다.

- 이를 적용하여 4’bit CLA를 위한 carry를 계산하면 다음과 같다.

C1 = G0 + P0\*ci

= G[0] + ( P[1] \* ci )

C2 = G1 + P1\*G0 + P1\*P0\*ci

= G[1] + (P[1] \* G[0]) + (P[1] \* P[2] \* ci )

C3 = G2 + P2\*G1 + P2\*P1\*G0 + P2\*P1\*P0\*ci

= G[2] + (P[2] \* g[1]) + (P[2] \* P[1] \* G[0]) + (P[2] \* P[1] \* P[0] \* ci)

CO = G3 + P3\*G2 + P2\*P1\*G0 + P2\*P1\*P0\*Ci

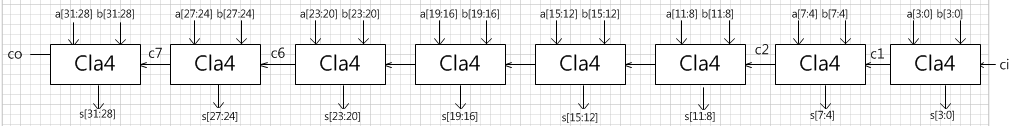
= G[3] + (P[3] \* G[2]) + (P[3] \* P[2] \* G[1]) + (P[3] \* P[2] \* P[1] \* G[0]) + (P[3] \* P[2] \* P[1] \* P[0] \* ci)

3) 32 bit Carry Look-ahead Adder

- CLB와 Full Adder를 바탕으로 만들어진 4bit CLA 8개를 Ripple식으로 연결하여 구현하였습니다.

- CLA 모듈이 직렬로 연결되어있기 때문에 이전 CLA모듈에서의 연산이 끝나야 다음 모듈로 넘어갈 수 있으며 이에 따른 시간이 소요됩니다.

- 다음과 같은 형태로 표현될 수 있습니다.



- 인자의 구성은 다음과 같습니다.

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Name | Bandwidth (bit) | Description |
| Input | a | 32 | Input data a |
| b | 32 | Input data b |
| ci | 1 | Carry in |
| Output | co | 1 | Carry out |
| s | 32 | Sum |
| Wire | c(1~7) | 1 | Internal carry |

- 모듈의 구성은 다음과 같습니다.

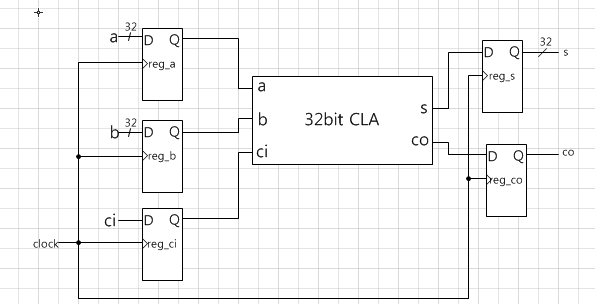
|  |  |  |
| --- | --- | --- |
| Port | Name | Description |
| Module | cla32 | 32 bit CLA |
| Instance | U0\_cla4 | 4 bit CLA |
| U1\_cla4 |
| U2\_cla4 |
| U3\_cla4 |
| U4\_cla4 |
| U5\_cla4 |
| U6\_cla4 |
| U7\_cla4 |

4) 32bit Carry Look-ahead Adder with clock

- Timing analysis를 통해서 delay를 분석하는데, 이를 통해 해당 모듈의 logic이 정확하게

동작할 수 있는 조건(Maximum clock frequency)을 찾을 수 있다.

- circuit module의 앞뒤로 flip-flop을 붙여줌으로써 구현한다.

 - 다음과 같은 형태로 표현할 수 있습니다.

- 인자의 구성은 다음과 같습니다.

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Name | Bandwidth (bit) | Description |
| Input | clock | 1 | Clock |
| a | 32 | Input data a |
| b | 32 | Input data b |
| ci | 1 | Carry in |
| Output | co\_cla | 1 | Carry out |
| s\_cla | 32 | Sum |
| Register | reg\_a | 32 | Register a |
| reg\_b | 32 | Register b |
| reg\_ci | 1 | Register carry in |
| reg\_s\_cla | 32 | Register sum |
| reg\_co\_cla | 1 | Register carry out |
| Wire | wire\_s\_cla | 32 | Wire sum |
| wire\_co\_cla | 1 | Sire carry out |

- 모듈의 구성은 다음과 같습니다.

|  |  |  |
| --- | --- | --- |
| Classification | Name | Description |
| Top\_module | cla\_clk | 32 bit CLA with clock |
| Instance | U0\_cla32 | 32 bit CLA |

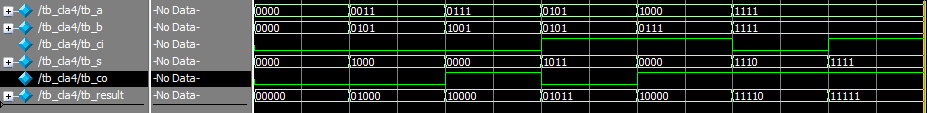
1. 설계 검증 및 실험 결과
   1. 시뮬레이션 결과

**\* Carry Look-Ahead Adder (4’bit)**

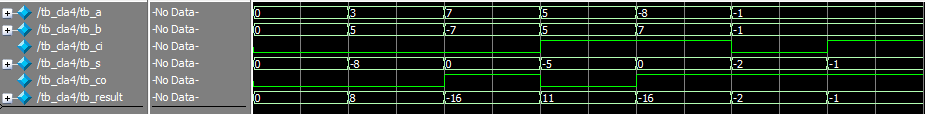
**-** CLA는 CLB를 통해서 올림값을 받아 연산이 됩니다.

- 예상과 다르지 않은 결과를 확인할 수 있었습니다.

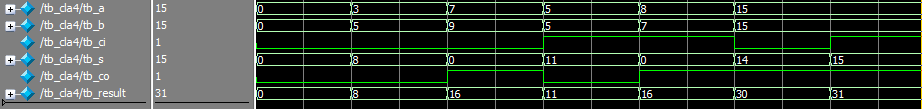
1) Binary



2) Decimal



3) Unsigned



\* **Carry Look-Ahead Adder with clock (32’bit)**

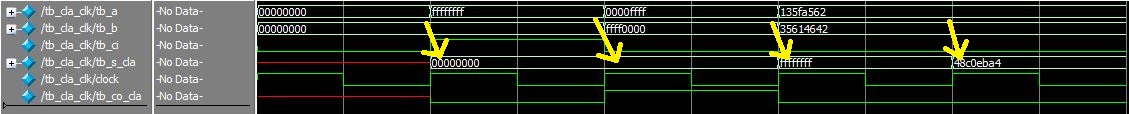
**-** 32’bit CLA는 4’bit CLA 8개를 Ripple carry Adder 처럼 연결하여 구현하였습니다.

- 이 때 Ripple되는 과정에서 시간이 소요됩니다.

- 플립플롭을 사용하여 클럭이 positive edge일때에 연산 결과가 나오게 되므로 한 주기 늦게 결과가 나오는 것처럼 보임을 확인할 수 있습니다.

- 모든 비트에 최대값이 들어갈 때, carry in값이 1인경우와 0인경우, 두 비트를 합쳤 을 때 모든 비트에 최대값이 들어가는 경우(carry in값은 0)를 확인했습니다.

1) hexadecimal

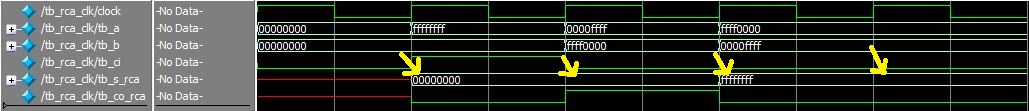


**\* Ripple Carry Adder with clock (32’bit)**

- 32’bit RCA 역시 4’bit RCA 8개를 Ripple식으로 연결하여 구현하였습니다.

- 1번째 비트부터의 계산이 차례대로 끝나야 32번째 비트 연산까지 계산을 할 수 있 습니다.

- 32bit CLA에서와 동일한 방식으로 검증하였습니다.

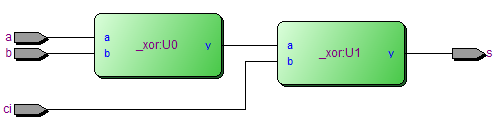
 1) hexadecimal

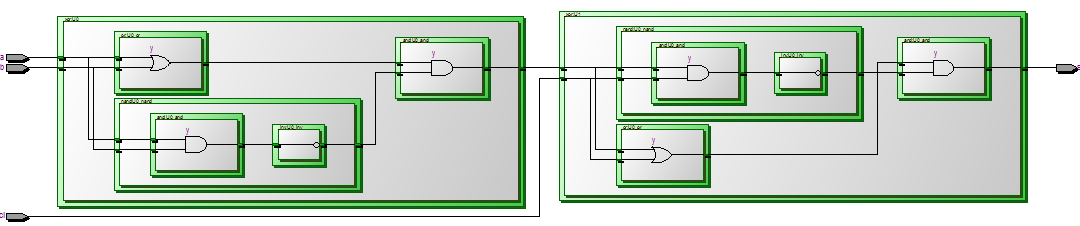
* 1. 합성(synthesis) 결과

**\* Full Adder**

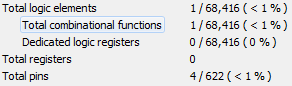
**<RTL viewer>**

- CLB에서 올림값을 계산해 주기 때문에 carry out 값이 없습니다.

**<Technology Map Viewer>**



**<Flow summary>**

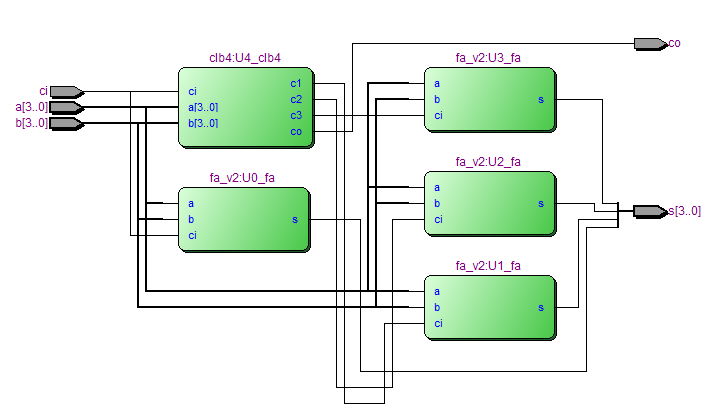
- Full Adder\_v2에서는 기존의 module을 instance 해서 사용하지 않고, xor논리 연산자를 이용해

assign하여 사용하였으므로 기본 논리 회로가 1개 사용되었습니다.

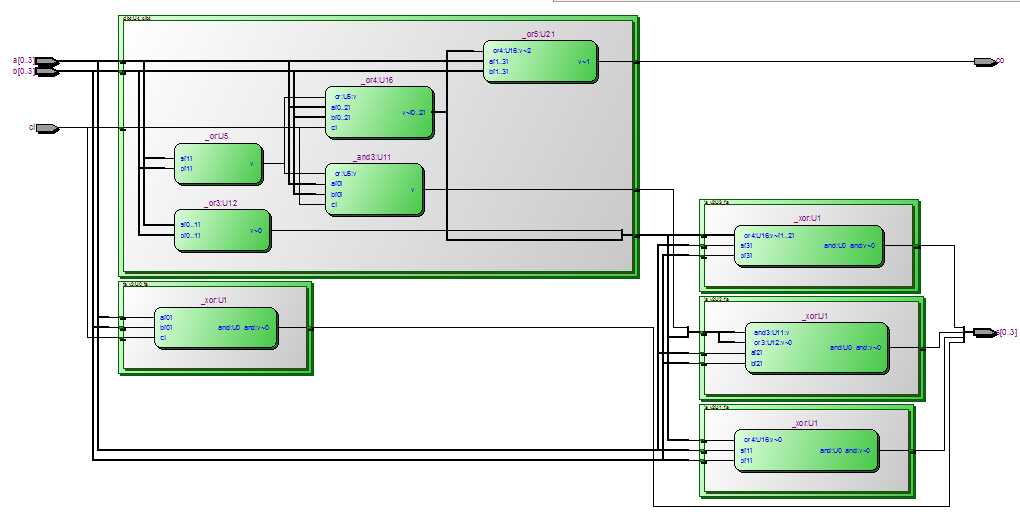
- input으로 a, b, ci, output 으로 s, 총 4개의 pin이 사용되었음을 확인할 수 있습니다.

**\* Carry Look-Ahead Adder (4’bit)**

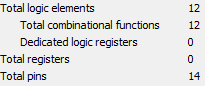
**<RLT viewer>**



**<Technology Map Viewer >**



- CLB module에서 발생한 올림 값이 각각의 Full adder로 들어가고 있음을 확인할 수 있습니다.

**<Flow Summary>**

- 12개의 기본 논리 회로가 사용되었고 4’bit 인자 a, b, s와

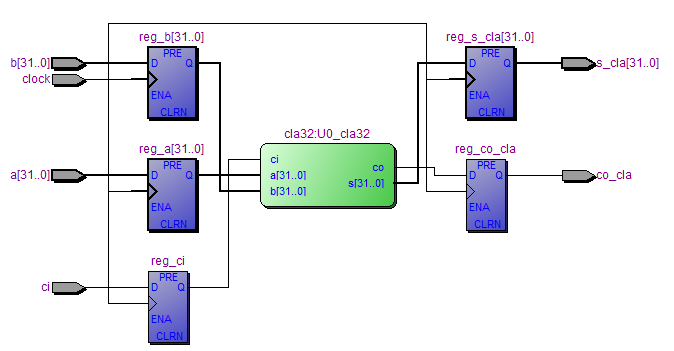
다음 cla4 module로 넘어갈 co, 합이 저장될 s, 총 14개의

Pin이 사용되었음을 확인할 수 있습니다.

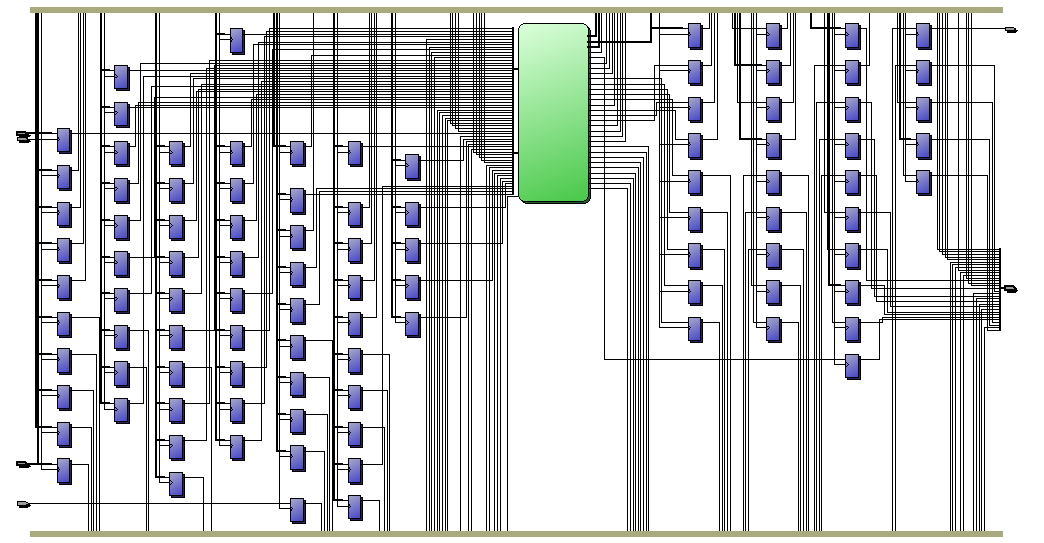
**\* Carry Look-Ahead Adder (32’bit)**

**<RLT viewer>**

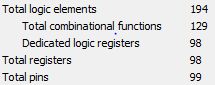
- 각 인자들이 filp-flop을 통해서 clock에 따라 입력됨을 확인할 수 있습니다.



**<Technology Map Viewer >**



**<Flow Summary>**

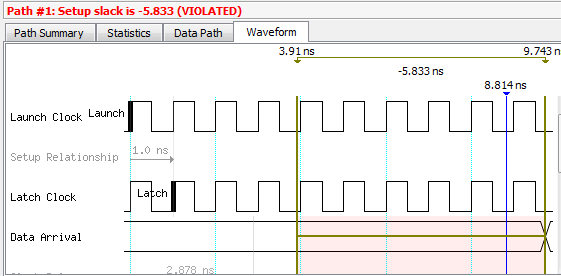
- 129개의 기본 논리 회로가 사용되었고, 98개의 레지스터

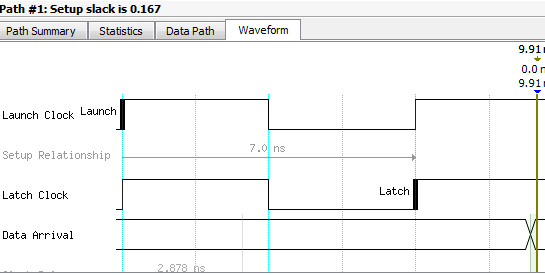
레지스터가 사용되었음을 확인할 수 있습니다.

- 32’bit짜리 a, b, s\_cla와 ci, clock, co\_cla, 총 99개의

Pin이 사용되었음을 확인할 수 있습니다.

**<Timing analysis>**

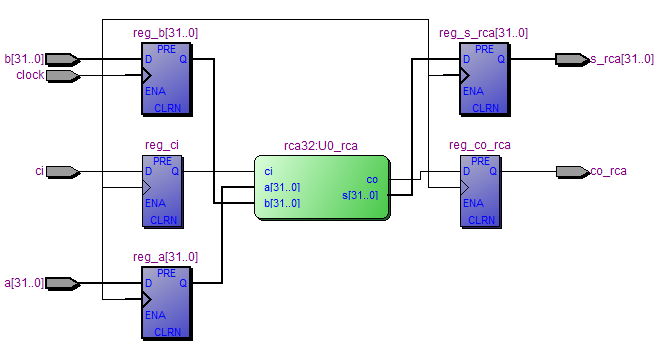
- 아래의 wave form을 보면 -5.833의 slack이 발생함을 확인할 수 있습니다.



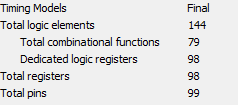
- clock을 7로 재설정 해준 후의 wave form입니다.

 - clock 재설정 이후 확인한 frequency는 다음과 같습니다.

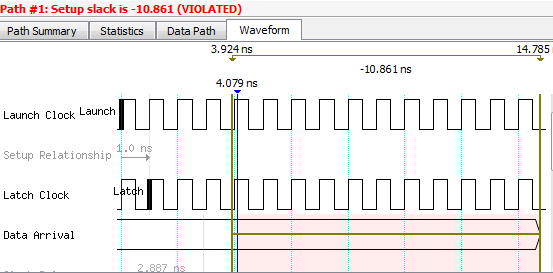
**\* Ripple Carry Adder with register (32’bit)**

 **<RTL Viewer>**

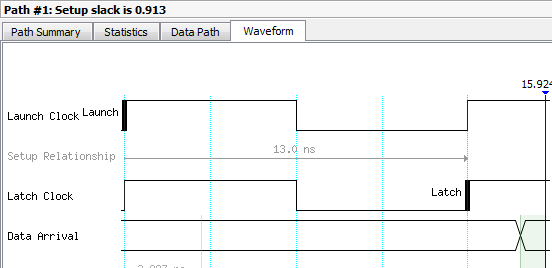
**<Flow summary>**

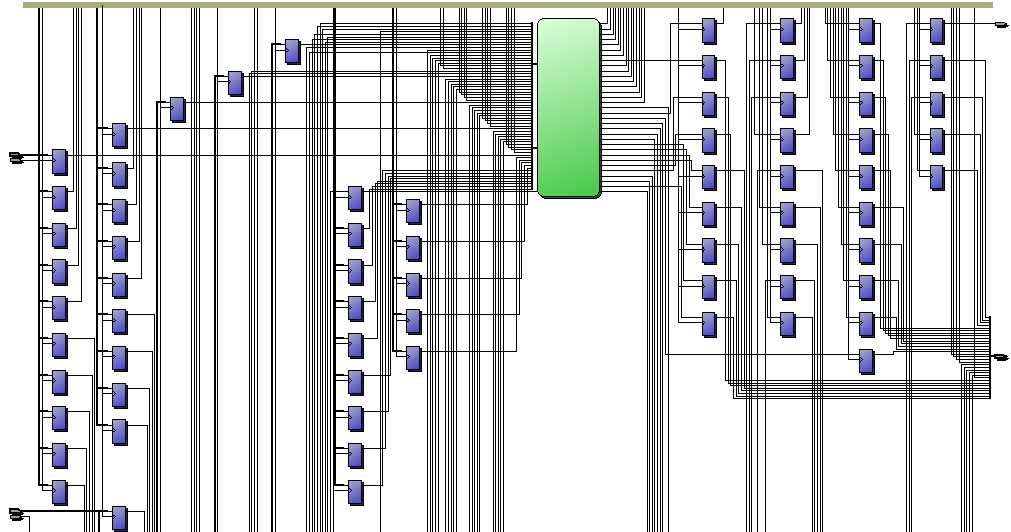
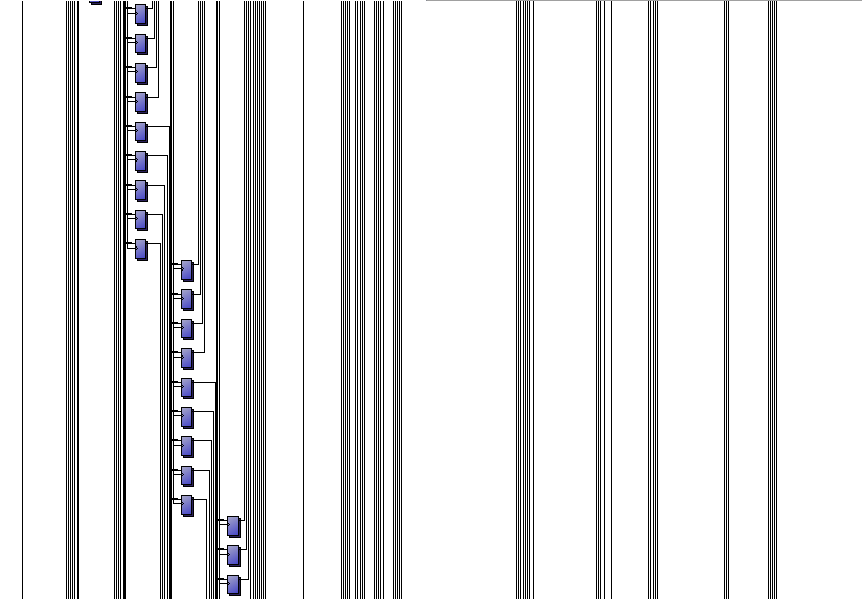
- 79개의 기본논리 회로와 98개의 레지스터가 사용되었음을 확인할 수 있습니다.

- 32’bit짜리 a, b, s\_rca와 ci, clock, co\_rca, 총 99개 의 Pin이 사용되었음을 확인할 수 있습니다.

**<Timing analysis>**

- 아래의 wave form을 보면 -10.861의 slack이 발생함을 확인할 수 있습니다.

- clock을 13으로 설정해 준 이후의 wave form입니다.

**<Technology Map Viewer>**



- clock 설정 이후 확인한 frequency는 다음과 같습니다.

1. 고찰 및 결론
   1. 고찰

이번 과제를 수행하면서 CLB module에서 애를 먹었는데 0을 o라고 적어서 이를 찾아내는데 시간을 많이 소모하였습니다. 코딩에 있어 한자한자 제대로 적어주는 것이 얼마나 중요한 지를 뼈저리게 느꼈습니다. 또한 CLA의 구성 module과 RCA의 구성 module을 비교하면서 알고리즘의 구성을 잘 짜는 것이 프로그램의 구동 시간에 큰 영향을 미친다는 것을 배웠습니다. 두 모듈의 소요 시간차를 보면서 알고리즘의 최적화와 이를 구현하는 것의 중요성을 보았습니다.

* 1. 결론

동일한 작업을 수행하는 두 모듈의 frequency를 비교하면 CLA 모듈의 frequency가 RCA 모듈의 frequency보다 훨씬 크다는 것을 확인할 수 있습니다(CLA : 146 Mhz / RCA : 83Mhz). frequency가 클수록 clock당 소요시간(주기)이 짧기 때문에 같은 횟수의 clock을 수행하게 되면 CLA module의 소요시간이 더 적다는 것을 확인할 수 있게 해줍니다.

CLA모듈이 CLB모듈을 추가적으로 사용하기 때문에 RCA모듈보다 logic을 많이 사용하므로

Size는 CLA모듈이 더 크다는 것을 확인할 수 있었습니다.

1. 참고문헌

이원석, 정길수 共著 / 실용적인 IC를 익히는 논리회로 실험 / 생능출판사 / 2013년

**< 7-segment Decoder >**

1. 목적

실제 숫자가 표기되는 7-segment display에 4’bit 입력값 두 개를 받아 합을 계산하는 RCA모듈을 이용하여 그 합과 올림값을 표시되도록 함. 또한 이를 통해 DE2-70 기판을 사용하는 방법을 익힌다.

2. 배경원리

DE2-70 기판에는 입력 값을 조작할 수 있는 스위치가 있고, quartus 프로그램에서의 pin설정을 통해 각 스위치를 입력으로 조정해줄 수 있습니다. 이를 조작하면 코드를 작성한 대로 값이 넘어가게 되는데 이 과제에서는 기판의 7-seg display에 그 값을 넘겨줌으로써 실제 조작하는 값들이 16진수로써 표기되도록 설정하였습니다.

3. 설계세부사항

- seg\_dec module은 입력 값을 4비트로 받게 하여 case를 통해 이를 7비트로 출력되도록

구현하였습니다.

- seg\_dec module을 instance하여 구현한 print\_RCA\_result\_to\_7segment module은 4bit

RCA module을 통해서 합과 올림값을 출력하는데 이때의 합은 case에서 해당 값에 설정한

7bit값으로 바뀌어서 13bit짜리 dec\_out인자의 [6:0] bit에 들어가게 되고, 올림값은

[13:7] bit에 들어가게 구현하였습니다.

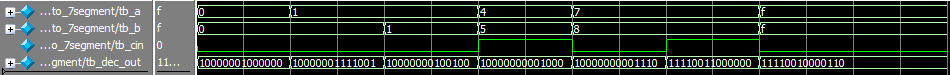
- 각 input case마다 output을 설정해줌으로써 display에 표시를 다르게 하였습니다.

**<7-segment module’s Truth table>**

|  |  |  |
| --- | --- | --- |
| Input[4bit] | Output[7bit] | 7-Segment Display Outputs |
| 4’h 0 | 7’b 1000000 | 0 |
| 4’h 1 | 7’b 1111001 | 1 |
| 4’h 2 | 7’b 0100100 | 2 |
| 4’h 3 | 7’b 0110000 | 3 |
| 4’h 4 | 7’b 0011001 | 4 |
| 4’h 5 | 7’b 0010010 | 5 |
| 4’h 6 | 7’b 0000010 | 6 |
| 4’h 7 | 7’b 1111000 | 7 |
| 4’h 8 | 7’b 0000000 | 8 |
| 4’h 9 | 7’b 0011000 | 9 |
| 4’h a | 7’b 0001000 | A |
| 4’h b | 7’b 0000011 | b |
| 4’h c | 7’b 1000110 | C |
| 4’h d | 7’b 0100001 | d |
| 4’h e | 7’b 0000110 | E |
| 4’h f | 7’b 0001110 | F |

4. 설계 검증 및 실험 결과

* 1. 시뮬레이션 결과

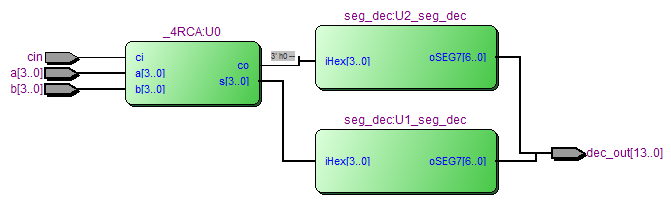
- tb\_dec\_out의 14자리중 앞의 7자리는 올림값이 7seg에 들어갈 때 case의 값이고, 뒤의

7자리는 합이 7seg에 들어갈 때 case의 값입니다.

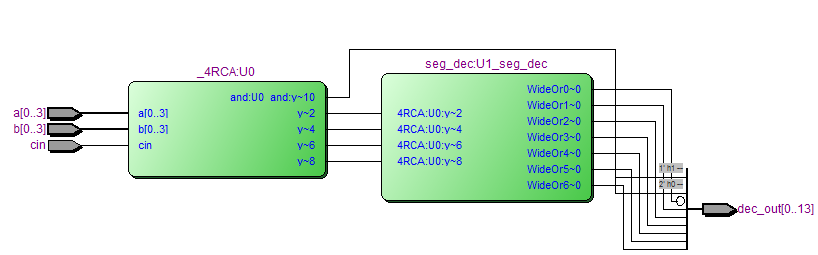
- 위의 Truth table을 보고 비교해 본 결과 정확한 값이 도출됨을 확인할 수 있었습니다.

* 1. 합성 결과

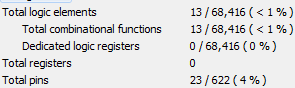
**<RTL Viewer>**



**<Technology Map Viewer>**



**<Flow summary>**

- 총 13개의 기본 논리 소자가 사용되었습니다.

- 4bit 인자 a, b, 14bit 인자 dec\_out, cin, 총 23개의

pin이 사용되었음을 확인할 수 있습니다.

**5. 고찰 및 결론**

**A. 고찰**

1학기 때 traffic signal을 만들면서 de2-70 기판을 사용했었는데 그 때 만든

print segment에서는 제 임의대로 알파벳을 출력했다가(알파벳을 되도록 소문자로 출력)

이번에 가져다 쓸 때 이상하게 나와서 급하게 수정했었습니다. Pin을 설정하는 것을

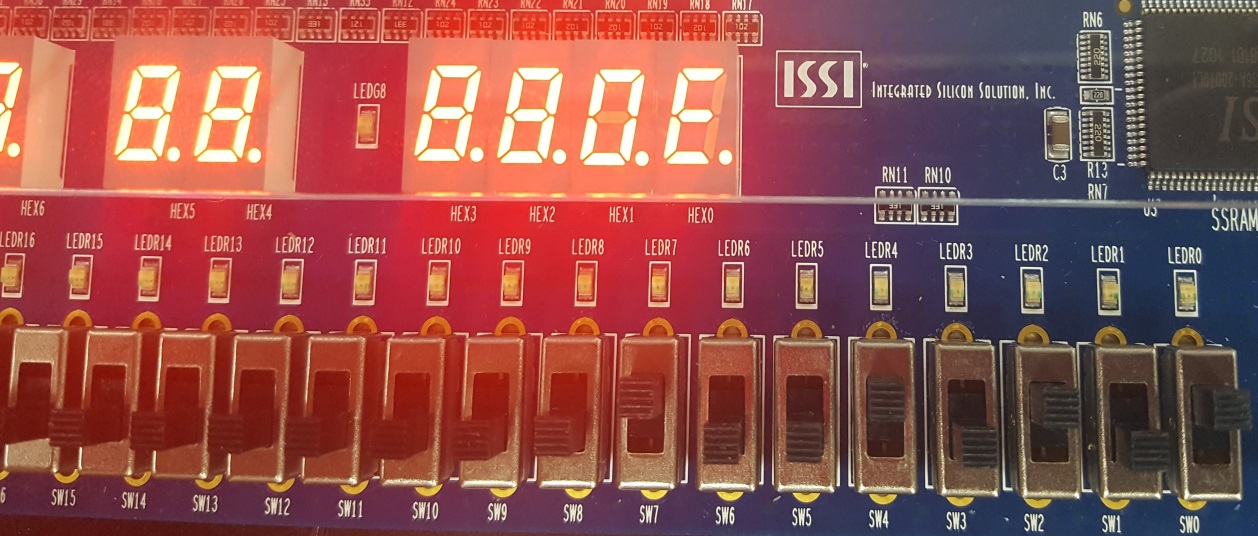
조교님께서 올려주신 파일을 가져다 그대로 사용하였는데 나중에 프로젝트를 하면서

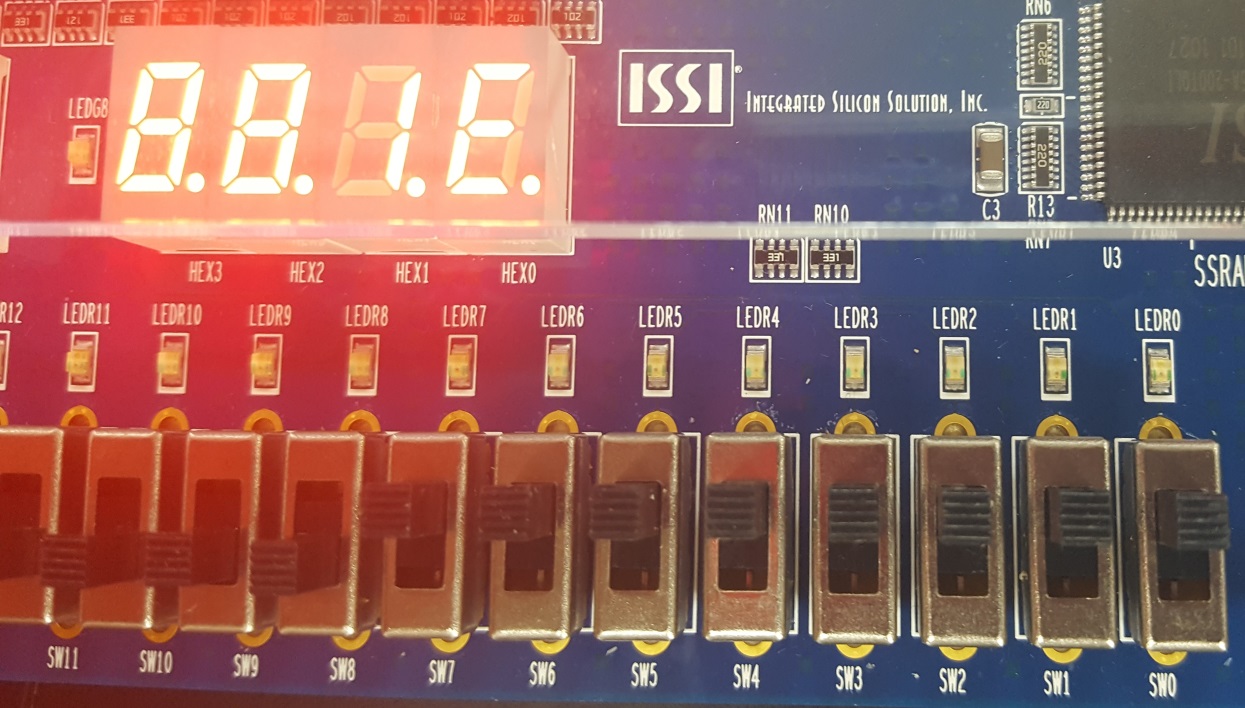
이걸 스스로 설정이나 할 수는 있을까 하는 의문이 생깁니다. 이 이외에는 그다지 큰

어려움은 없었습니다.

**B. 결론 (결과 사진)**

**** 1. ci = 0 / a = f / b = 0

 2. ci = 0 / a = 5 / b = 9

 3. ci = 0 / a = f / b = f

 4. ci = 1 / a = f / b = f