컴퓨터 공학 기초 실험2 보고서

실험제목: Subtractor & Arithmetic Logic Unit

실험일자: 2014년 09월 22일 (목)

제출일자: 2014년 09월 29일 (목)

학 과: 컴퓨터공학과

담당교수: 이혁준 교수님

실습분반: 목요일 0, 1, 2

학 번: 2013722095

성 명: 최 재 은

1. 제목 및 목적
   1. 제목

Subtractor & Arithmetic Logic Unit

* 1. 목적

MUX의 응용방법을 알아보고 적용하여 프로그램을 구현한다.

이를 통해 ALU(산술 논리 장치)의 원리와 동작 방법을 이해하고 설계해본다.

Opcode를 설정하여 이를 MUX에 넣어주고 이에 따라 조건적으로 어떤 산술이 실행되는지를 확인하면서 프로그램의 조건적 구동에 대해 이해한다.

1. 원리(배경지식)

Multiplexer란 입력비트(opcode)에 따라 상응하는 값, 또는 논리 연산 값을 출력해주는 연산 모듈로써 이 프로그램에서는 먼저 add, sub, or, xor, xnor, ~a, ~b의 연산 결과 값을 먼저 계산하고 그 결과값을 MUX의 입력값으로 넣어주게 됩니다. 이때 MUX의 3bit

(8가지 경우의 수 선택 가능) opcode에 따라서 각 연산 결과값을 출력해주도록 프로그램이 설계되었습니다. Subtraction에서는 A-B의 형태로 표현이 되지만 실제 프로그램이

계산할 때는 덧셈으로 구동하게 됩니다. 즉 A+(-B)의 형태로 덧셈이 이루어집니다.

이때 -B는 2의 보수를 취해주게 됩니다.

flag에서 C(carry)는 두 수의 합으로 인한 carry가 부호비트(sign bit, 가장 왼편에 위치)보다 앞에 더해졌을 때와 두 수의 차로 인해 sign bit의 앞 비트에서 borrow가 발생할 때

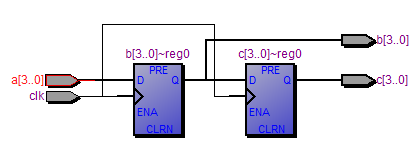
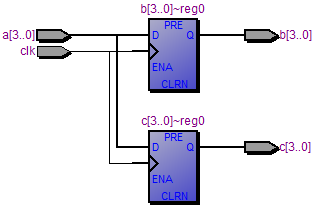
1이 됩니다. C는 마지막 모듈의 co로 확인할 수 있습니다.

V(overflow)는 두 수의 합으로 인해서 sign bit가 바뀌었을 때 (예를 들어 0100+0100)

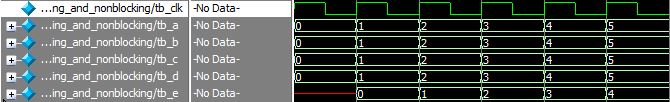
Overflow flag가 1이 됩니다. V는 마지막 모듈의 co와 그 이전 모듈이 co를 exclusive

Or의 input으로 넣었을 때 출력 값으로 확인할 수 있습니다.

Blocking은 reg변수 = 변수로 값을 대입하고 Nonblocking은 reg변수 <= 변수로 값을 대입한다는 차이를 보입니다. 이를 실제 module로 만들게 되면 아래 같은 형태가 나옵니다.

 <Blocking> <Nonblocking>

NonBlocking의 경우 output C가 flip-flop 두 개를 거쳐서 값이 들어가게 되므로 다음과 같은 형태의 waveform을 확인할 수 있습니다.

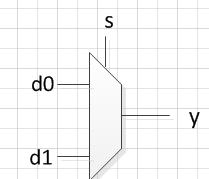


1. 설계 세부사항

**1) mux2**

- mux2는 1bit opcode는 2개의 경우의 수를 가짐으로써 입력값 두 개를 선택적으로 출력해주는 역할을 합니다.

- 다음과 같은 형태로 표현됩니다



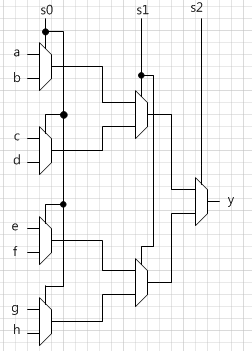
**<Truth Table>**

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | | | **Output** |
| **s** | **d0** | **d1** | **y** |
| **0** | **0** | **0** | **0** |
| **0** | **1** | **0** |
| **1** | **0** | **1** |
| **1** | **1** | **1** |
| **1** | **0** | **0** | **0** |
| **0** | **1** | **1** |
| **1** | **0** | **0** |
| **1** | **1** | **1** |

**2) MUX8\_4bit**

- 3bit opcode를 이용해 8가지 경우의 수를 선택적으로 출력해줍니다.

- MUX2 7개를 사용하여 설계하였고 다음과 같이 표현됩니다.



- 위와 같은 MUX8 모듈 4개를 사용하여 4bit의 연산에 대해 조건부 출력이 가능해집니다.

- 실제 opcode는 s2, s1, s0순으로 비트가 들어갑니다.

**<Truth table>**

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | | | **Output** |
| **S0** | **S1** | **S2** | **y** |
| **0** | **0** | **0** | **a** |
| **1** | **e** |
| **1** | **0** | **c** |
| **1** | **g** |
| **1** | **0** | **0** | **b** |
| **1** | **f** |
| **1** | **0** | **d** |
| **1** | **h** |

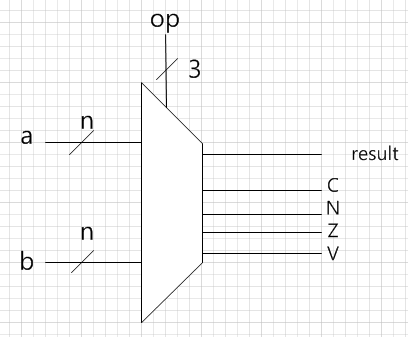
**3) cal\_flag4**

- mux8에서 출력된 결과값(result)과 두 입력 값의 합, 차를 받아 C(carry), N(Negative),

Z(zero), V(overflow)를 출력해주는 모듈로 C는 두 입력의 합과 차의 CO으로, N은 result의 마지막 비트(sign bit)로, Z는 result가 0인가로, V는 마지막 비트와 그 전 비트의 Xor 값으로 표현한다.

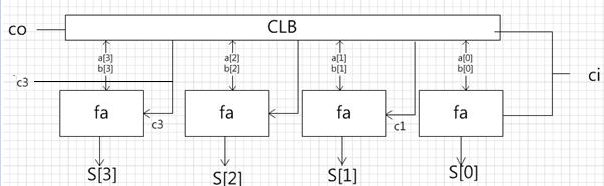
- 이러한 flag는 비교연산을 하는데 사용이 됩니다

- 다음과 같은 형태로 표현될 수 있습니다.



**4) CLA4**

**-** 다음과 같은 형태로 표현됩니다.

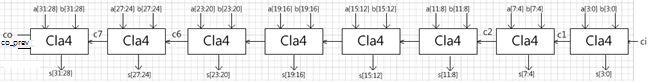


- flag에서 V(overflow)를 확인하기 위해서 c3(올림값)을 넘겨주게 됩니다.

- 그 이외의 동작 및 구동은 이전의 CLA4와 동일합니다.

**5) CLA32**

**-** 다음과 같은 형태로 표현됩니다.



- co\_prev는 최종적인 cla4 module에서 V(overflow)를 확인하기 위해서 사용됩니다.

- 그 이외의 동작 및 구동은 이전의 CLA32와 동일합니다.

**5) ALU4**

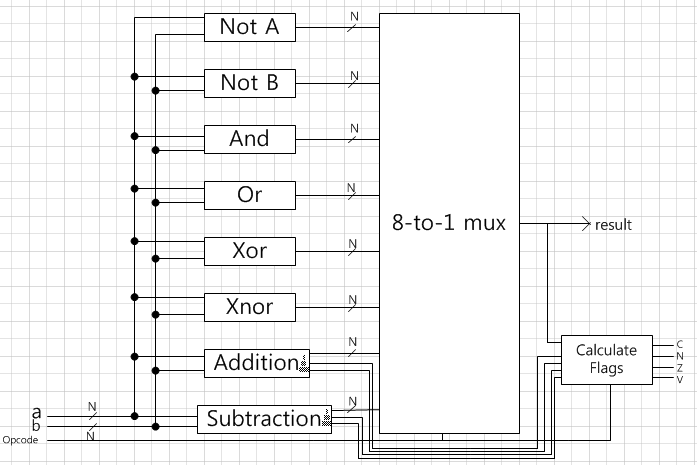
**-** 4bit 입력을 두 개 받아서 이에 대한 ~a, ~b, and, or, xor, xnor, addition, subtraction을 MUX8을 통해 조건부로 출력해 주도록 설계하였습니다.

- 이 때 Addition, Subtraction에서의 co와 마지막 비트연산의 올림 값을 cal\_flag4에 입력 값으로 넣어주어 C, N, Z, V를 출력하도록 설계하였습니다.

- opcode에 따른 MUX8의 출력 값은 다음과 같습니다.

|  |  |
| --- | --- |
| Opcode | Operation |
| 3’b000 | Not a |
| 3’b001 | Not b |
| 3’b010 | And |
| 3’b011 | Or |
| 3’b100 | Exclusive or |
| 3’b101 | Exclusive nor |
| 3’b110 | Addition |
| 3’b111 | Subtraction |

- ALU4는 다음과 같이 표현될 수 있습니다.



**6) ALU32**

- ALU4동일한 방식으로 구현이 되며 구조 또한 동일합니다. 다만 input, output이 32 bit기 때문에 이전의 CLA32 module처럼 CLA4 module 8개를 직렬구조로 연결하여 계산에 사용됩니다.

- opcode에 따른 operation이나 구조의 표현이 ALU4와 동일하여 생략하였습니다.

- 사용된 인자의 구성은 다음과 같습니다.

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Name | Bandwidth(bit) | Description |
| Input | a | N | Operand A |
| b | N | Operand B |
| op | 3 | Opcode |
| Output | result | N | ALU 연산 결과 |
| c | 1 | Carry flag |
| n | 1 | Negative flag |
| z | 1 | Zero flag |
| v | 1 | Overflow flag |

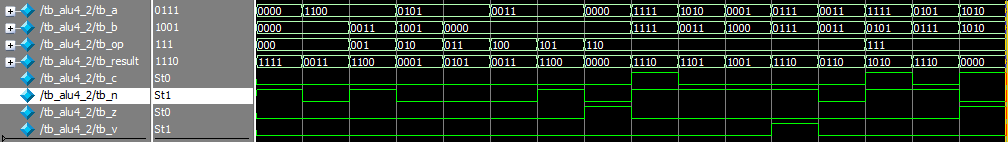
- 사용된 모듈의 구성은 다음과 같습니다.

|  |  |  |
| --- | --- | --- |
| Classification | Name | Description |
| Module | alu N | N-bit ALU |
| Instance | U0\_inv\_Nbits | N-bit inverter a |
| U1\_inv\_Nbits | N-bit inverter a |
| U2\_and2\_Nbits | N-bit a and b |
| U3\_or2\_Nbits | N-bit a or b |
| U4\_xor2\_Nbits | N-bit a xor b |
| U5\_xnor2\_Nbits | N-bit a xnor b |
| U6\_add | N-bit CLA(a add b) |
| U7\_sub | N-bit CLA(a sub b) |
| U8\_mx8\_Nbits | N-bit 8-to-1 multiplexer |
| U9\_cal\_flagsN | Calculate flags |

1. 설계 검증 및 실험 결과
   1. 시뮬레이션 결과

**1) ALU4**

- 다음과 같은 wave form 을 확인할 수 있었습니다.



- opcode가 000일 때 Not a의 값(0011)이, 001일 때 Not b의 값(1100)이, 010일 때 and의 값(0001)이, 011일 때 or의 값(0101)이, 100일 때 xor의 값(0011)이, 101일 때 xnor의 값(1100)이, 110일 때 add의 값이, 111일 때 sub의 값이 들어가있음을 확인할 수 있었습니다.

**2) ALU32**

- example.tv를 읽어서 test를 진행하는 testvector를 이용해 검증하였습니다.

- example에는 다음과 같은 값이 들어가있습니다.

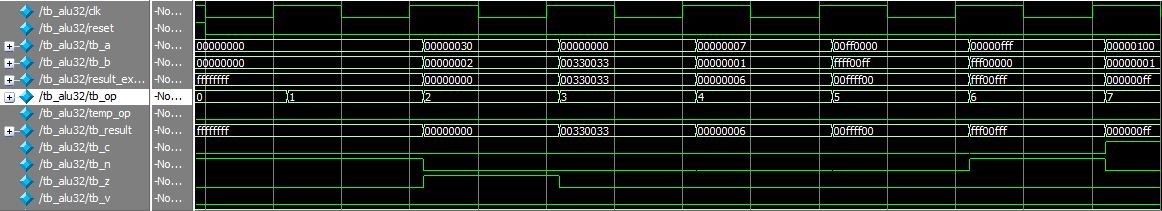
00000000\_00000000\_0\_ffffffff 00000000\_00000000\_1\_ffffffff

00000030\_00000002\_2\_00000000 00000000\_00330033\_3\_00330033

00000007\_00000001\_4\_00000006 00ff0000\_ffff00ff\_5\_00ffff00

00000fff\_fff00000\_6\_fff00fff 00000100\_00000001\_7\_000000ff

- 이에 대한 waveform은 다음과 같으며 예상과 같은 값이 나왔습니다.



* 1. 합성(synthesis) 결과

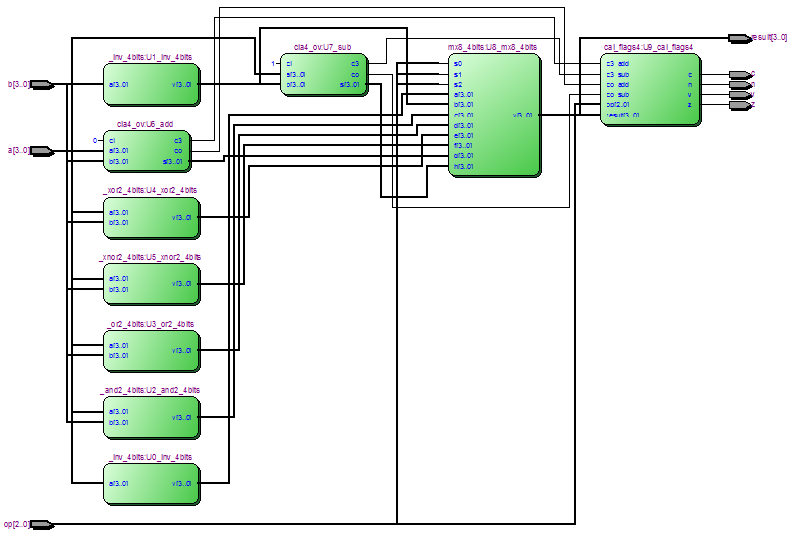
**1) ALU4**

**<RTL Viewer>**

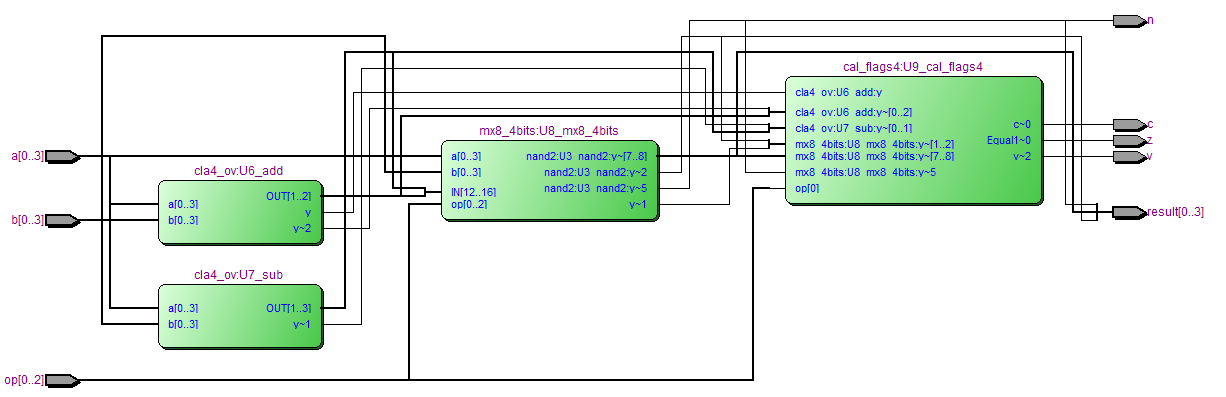
**-** 입력 값인 a와 b가 1차적으로 연산되고 이 결과들이 mux의 입력 값으로 들어가는 것을

확인할 수 있습니다.

- 또한 이 입력 값들은 opcode에 의해서 선택적으로 출력이 되고 그 결과와 1차 연산에서의 일부 출력 값들이 flag module에 들어가서 C, N, Z, V를 출력해 주도록 설계되었습니다.



**<Technology Map Viewer>**



- 3bit opcode에 의해서 mux의 출력이 선택됩니다.

**<Flow summary>**

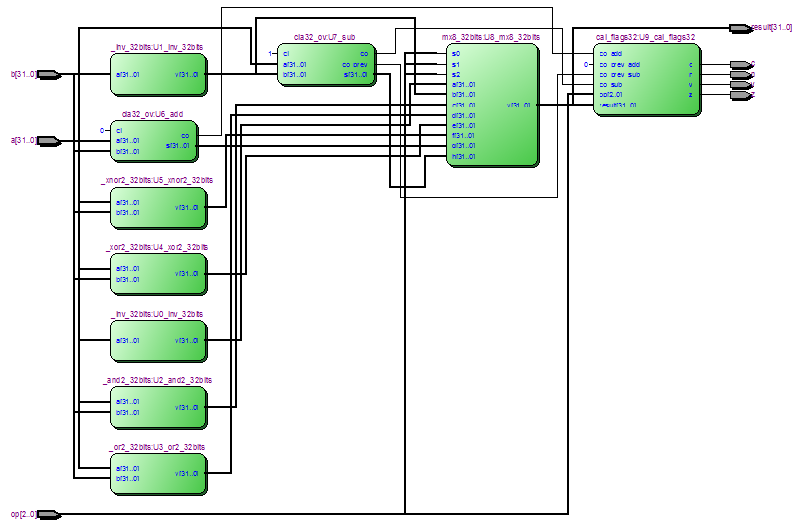


- 42개의 로직이 사용되었으며, 19개의 핀을 사용하고 있음을 확인할 수 있습니다.

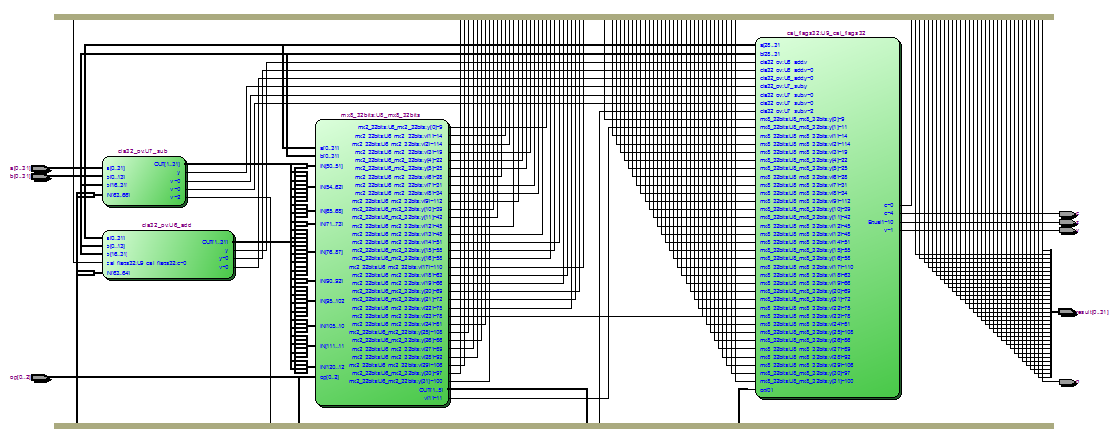
**2) ALU32**

**<RTL Viewer>**

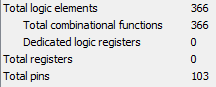
- 4bit와 동일한 구조로 설계되어있음을 확인할 수 있고, 입력 출력 값이 32bit임을 확인할 수 있습니다.



**<Technology Map Viewer>**



**<Flow Summary>**



- 366개의 로직이 사용되었으며 103개의 pin이 사용되었음을 확인할 수 있습니다.

- input으로 32bit a, b, 3bit op, output으로 32bit result, 1bit c, n, z, v 총 103개입니다.

1. 고찰 및 결론
   1. 고찰

이번 실험을 진행하면서 mux의 작동에 대해 다시 공부를 했고, 가장 큰 어려움은 역시

Alu32의 testvector였다고 생각합니다. 처음엔 Example.tv파일을 어디에 넣어야 하는지도

모르고 그냥 project 폴더에 넣어버렸고, example 파일을 어떻게 써 넣어야 하는지도 몰라서 혼란이 많았습니다. 어찌해서 파일을 만들었더니 이번에는 modelsim에서 해당 파일을 읽을 수 없다는 오류를 내보내서 멘탈이 크게 흔들렸습니다. 그 문제는 공지사항을 보고 해결하였으며, 그 다음으로 생겼던 문제는 파일에서 한 줄을 읽어왔으나 공간이 부족하다는 warning이었습니다. 이는 처음에 입력을 binary로 풀어서 넣은 값들을

Hexadecimal로 넣음으로써 해결하였습니다. 그 다음으로 생긴 문제는 입력 값들이 한 칸 씩 당겨져서 들어가는 문제였는데 이는 아무런 값도 받지 않는 null이라는 reg를 만들어서 tb\_b와 tb\_op 사이에 넣음으로써 해결하였습니다.

* 1. 결론

두 개의 adder를 사용할 때는 아예 ~b를 다른 adder에 입력해 주었지만, b가 입력될 때 mux2를 사용하여 b와 ~b를 선택적으로 넣게 된다면 하나의 adder에서 a+b, a+(~b)를 받게 되므로 Adder와 Subtractor의 역할을 할 수 있게 됩니다.

이번 실험을 통해서 if를 사용해 조건적으로 코딩을 작성할 수 있다는 것을 배웠습니다.

또한 mux opcode의 bit식 표현을 통해 더 다양한 경우의 수를 선택 가능하게 됨을 배웠습니다. 이를 통해서 clock에 따라 입력을 다르게 들어가게 하고, 또 이에 추가적으로 MUX를 사용함으로써 신호등 같은 주기적으로 신호가 바뀌는 시스템을 구현할 수 있지 않을까 생각해봤습니다.

1. 참고문헌
2. 제목 및 목적
   1. 제목

Modified Carry Look-ahead Adder

* 1. 목적

기존의 CLA의 보완하여 만들어진 M-CLA를 구현해봄으로써 두 모듈의 차이점을 이해한다. 두 모듈의 비교를 통해 어떤 모듈이 더 생산적인 면으로 효율적인지를 비교해본다.

1. 원리(배경지식)

기존의 CLA는 CLB에서 모든 올림값을 계산하여 하위 FA에 ci으로 입력해주었지만

M-CLA는 Fa와 Fa\_v2를 모두 사용하여 하위 fa모듈이 다음 fa모듈에 올림값을 넘겨준다.

M-CLA의 CLB는 전체 4bit의 올림값을 하위 FA모듈로 넘겨주는 것이 아니라 4bit의 최종적인 올림값(CO)만을 계산하여 다음 CLA모듈로 넘겨준다.

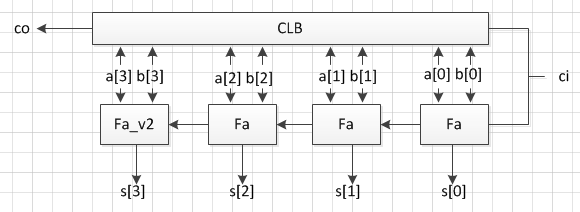
1. 설계 세부사항

합만을 계산하는 Fa\_v2와 carry out까지 계산하는 Fa모듈을 모두 사용하였습니다.

CLB는 모든 비트를 받아서 최종적인 올림값인 CO만들 계산하여 다음 CLA로 넘겨줍니다.

마지막 비트에서의 올림값은 CLB에서 계산되므로 CLA의 마지막 Fa는 Fa\_v2로 설정하였습니다.

- 다음과 같은 형태로 표현할 수 있습니다.



- CLB에서는 1,2,3번째 비트를 계산할 필요가 없기 때문에 마지막 비트의 올림값만 연산해 줍니다. 그 Boolean 식은 다음과 같습니다.

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)

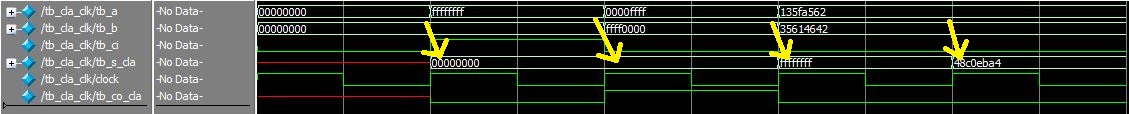
- 그 이외의 모듈 적용 방식이나 테스트 벤치는 기존의 CLA와 동일하므로 생략하였습니다.

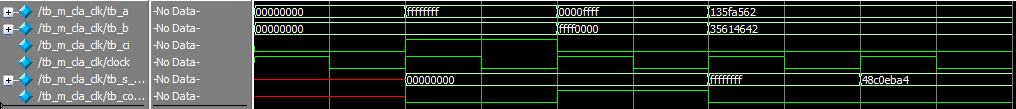
- 모듈의 구성은 다음과 같습니다.

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

1. 설계 검증 및 실험 결과
   1. 시뮬레이션 결과

- 기존 CLA의 wave form

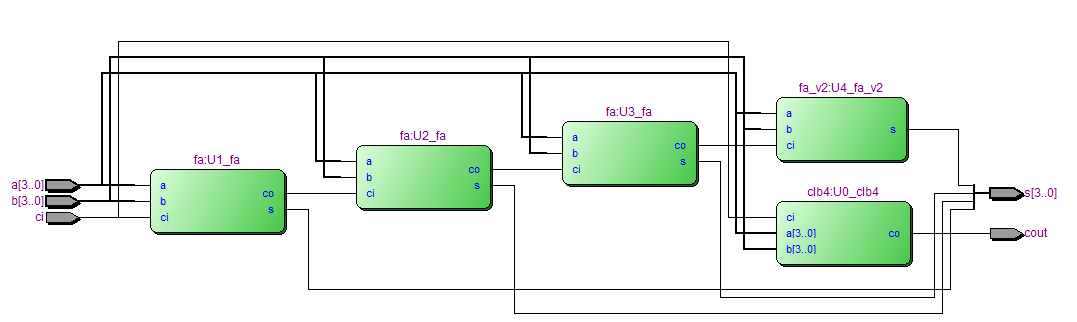
- Modified CLA



- 동일한 입력을 주었을 때 결과가 동일함을 확인할 수 있었습니다.

* 1. 합성(synthesis) 결과

**1) CLA4**



- 위의 사진처럼 마지막 비트의 연산만 fa\_v2로 연산되며, CLB4 모듈에서는 최종 올림값

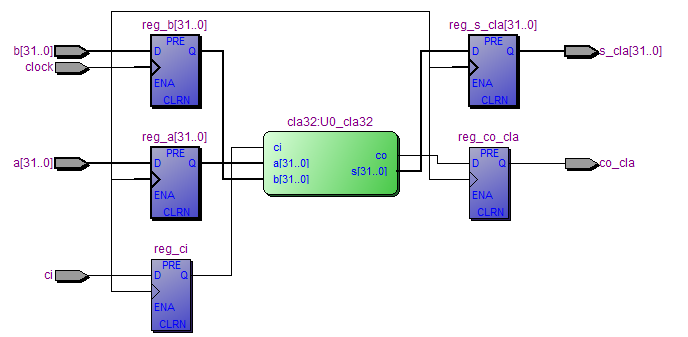
만을 연산하고 있음을 확인할 수 있습니다.

**2) m\_cla\_clk**

**<RTL Viewer>**

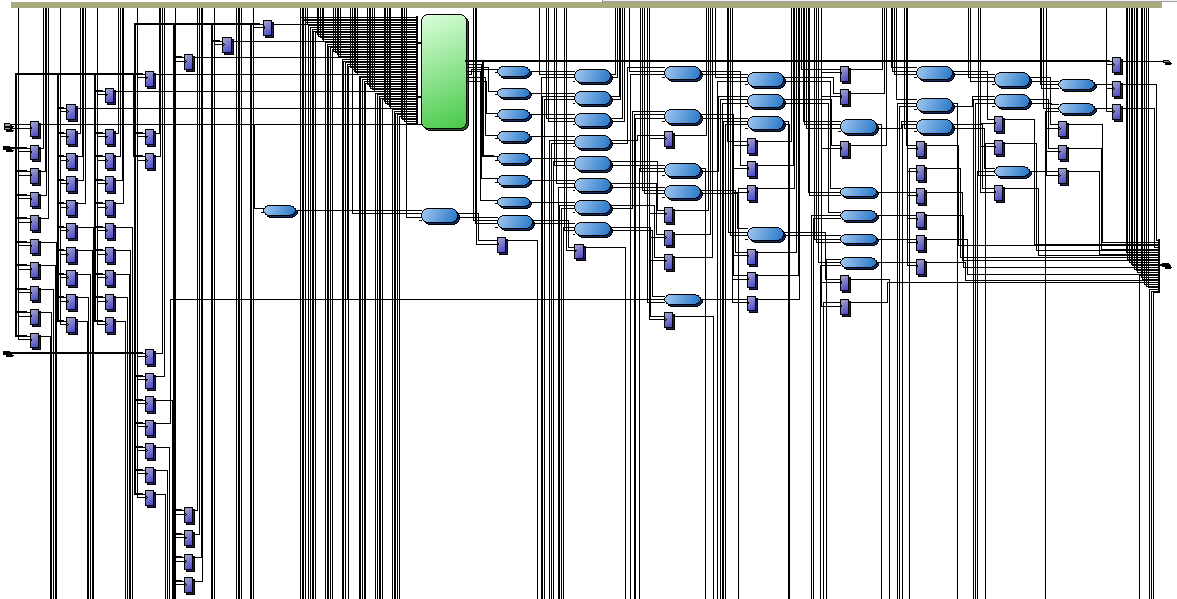
**-** 입력 값들은 flip-flop을 통해 cla32 module로 들어가고 출력 값들은 다시 flip-flop을 거쳐서 출력되는 구조임을 볼 수 있습니다. 이때 각 입출력은 clock에 의해 주기적으로

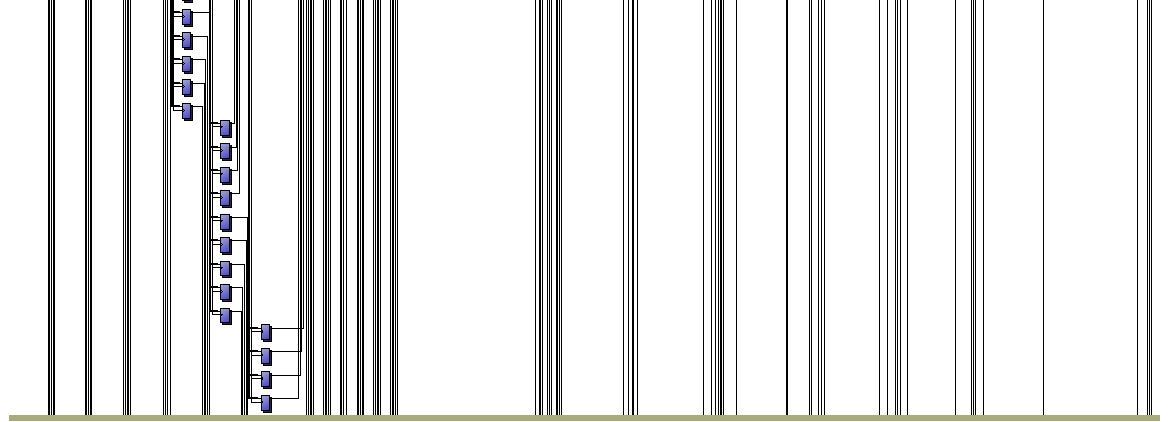
들어가고 나오게 됩니다.



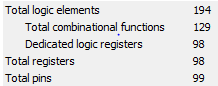
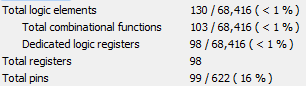
- RTL Viewer로 확인한 전체 구조는 기존의 CLA와 동일합니다.

**<Technology Map Viewer>**





**<Flow Summary>**

(기존의 CLA) (Modified CLA)

- 위에서 M-CLA가 더 적은 수의 Logic을 사용하는 것을 볼 수 있습니다.

**<Timing analysis>**

- 기존의 CLA



- M-CLA



- 두 모듈의 Fmax를 비교해보면 M-CLA의 모듈의 속도가 더 빠르다는 것을 알 수

있습니다.

1. 고찰 및 결론
   1. 고찰

해당 과제를 진행하면서 큰 어려움은 없었으나 PDF 수업자료를 확인하면서 알고리즘을 이해하고 Fa를 추가하고 CLB를 수정했습니다.

* 1. 결론

Modified CLA가 로직이 더 적고 속도 또한 약간 더 빠르다는 것을 확인할 수 있었습니다. 이를 보았을 때 로직이 적을수록 생산할 때 코스트가 더 적게 들게 되므로 생산면에 있어서는 m-cla가 더 효율적이라는 것을 알 수 있습니다.

1. 참고문헌

- 두 모듈의 속도에 대해서 13 김민우 의 의견을 참고하였습니다.