디지털 논리2 및

컴퓨터 공학 기초 실험2   
보고서

과제제목: Subtractor & Arithmetic Logic Unit

실험일자: 2017년 09월 15일 (금)

제출일자: 2017년 09월 21일 (목)

학 과: 컴퓨터공학과

담당교수: 이준환 교수님

실습분반: 금요일 5, 6, 7

학 번: 2012722028

성 명: 장 한 별

1. 제목 및 목적
   1. 제목

Subtractor & Arithmetic Logic Unit

* 1. 목적

지난 실습에서 설계했던 Carry Look-ahead Adder(CLA)를 이용하여 Subtractor 의 원리를 이해하고 설계한다. 또한 3-bit opcode에 따라 8 to 1 Multiplexer를 구현하여 산술 연산(덧셈, 뺄셈)과 논리 연산(AND, OR, XOR, XNOR, NOT)이 가능한 Arithmetic Logic Unit(ALU) 를 설계한다. 그 과정 중에서 비교 연산에 사용되는 Flag 들이 무엇으로 구성되어있고 어떻게 구현되는지 파악한다.

1. 원리(배경지식)
2. Subtractor

|  |
| --- |
| image2.jpeg |

<Subtractor>

Subtractor는 뺄셈 기능을 수행하는 감산기이다. 대부분의 digital system에서 뺄셈을 하기 위하여 binary number 체계에서는 2의 보수 (two’s complement)를 이용하여 뺄셈을 한다. Two’s complement를 구하는 방법은 해당 binary number를 invert한뒤에 1을 더하는 방법이다.

이를 이용하여 덧셈 기능을 수행하는 가산기에 뺄셈을 해야할 숫자를 invert시킨뒤 가산기에 넣어 가산하고 1을 더해주는 형식으로 감산기로서 가산기의 기능을 수행할 수 있다. 결과적으로 하나의 adder로 덧셈, 뺄셈을 모두 수행 가능하다. 이때 2의 보수를 이용하여 설계를 하기 때문에 항상 Signed Number 만을 사용한다.

1. Arithmetic Logic Unit

Arithmetic Logic Unit(ALU)는 산술 논리장치이다. 즉, 두 숫자 사이의 산술 연산과 논리 연산을 계산하는 회로이다. 이번 실습에서는 opcode에 따라 연산을 수행하였는데 8개의 산술&논리 연산을 수행하기 위해서는 8 = 23 즉, 3bit opcode가 필요하다.

따라서 이번 ALU를 구현하기 위하여, NOT, AND, OR, XOR, XNOR, Addition, Subtraction, 8-to-1 mux, Calculate flags를 instance하여 구현한다.

1. Flag

Arithmetic Logic Unit(ALU)는 상태에 따라서 Flag값을 update하게 된다. Flag에는 C, N, Z, V 총 4가지 종류가 있다.

C는 Carry를 의미하고, 최상위 bit에서 carry가 발생할 때, c의 값은 1이 된다.

N은 Negative를 의미하고, 최상위 bit의값이 1이되면 즉 연산결과가 음수가 나오면

이때의 상태를 나타내기 위해 N의 값이 1이 된다.

Z는 Zero를 의미하는데, 계산결과가 0이될 때 Z의값은 1이된다.

V는 Overflow를 의미하는데, Overflow는 연산결과가 잘못됐을 때 발생한다. 연산 과정에서 표현 가능한 범위를 초과할 때 인데, 음수가 나와야 할 계산결과가 양수가 나오거나 양수가 나와야 할 계산결과가 음수가 나올 때 발생된다. V가 1이되는 경우는 최상위 bit의 carry와 그 전 비트의 carry를 XOR한 값이 1일 때, V의 값이 1이된다.

1. Carry와 Overflow의 차이점

sign bit이 없을 때 계산 범위를 넘는 carry발생시 C가 올라가고 sign bit이 있을 때 연산의 결과가 최상위 bit는 부호를 결정짓기 때문에, sign bit의 바로 전 bit가 1올라가서 sign bit에 영향을 주어 예측한 결과와 다른 연산 결과가 나왔을 때 V가 올라간다.

1. blocking과 non-blocking의 차이점

|  |
| --- |
|  |
| 위 그림은 blocking\_and\_nonblocking module 을 compile후 나타난 RTL viewer이다.  그림에서 위쪽의 module은 nonblocking module이고, 아래쪽 module은 blocking module이다. 그림에도 나와있듯이 clk자체는 nonblocking모듈의 2개의 filp flop이나 blocking모듈의 2개의 flip flop에 같은 clk이 들어간다. 하지만 위의 unblocking모듈에서 input a를 보면 첫번째 flip flop (b<=a) 에 들어가고 첫번째 flip flop의 Q가 그 다음 flip flop(c<=b)에 연결되어있다. 이때 c에 들어가는 b는 b의 이전 값이 들어간다. 즉 a의 b에 a의 값이 들어왔다고 해서 a값으로 바뀐 b의 값이 들어가는 것이 아니고 그전에 가지고있던 b의 값이 c로 들어가는 것 이다. 반대로 밑의 blocking모듈을 보면 두 개의 filp flop에 모두 input a가 들어간다. b=a, c=b로 나타나 있는데 순차적으로 input이 들어가므로 b에 a를넣고 바뀐 a의 값으로 바뀐b를 c에 넣는다. 이와 같이 바로바로 바뀐 값들을 입력해주는 특징이 있다. |

3. 설계 세부사항

1. 4-bit CLA to detect overflow

|  |
| --- |
|  |
| 위 그림은 4-bit CLA to detect overflow이다. 언뜻 봐서는 이전의 4bits CLA와 같아 보이지만 다른 점이 있다. output에서 co, s뿐만 아니라 c3도 output으로 출력한다.  그 이유는 overflow를 확인하기 위해서이다. Overflow를 확인하는 방법으로는 co와 c3를 XOR한 값이 1이 될 때 Overflow가 1이 set 되면서 연산 결과를 확인할 수 있다.  따라서 4bits ALU를 설계 할 때 위의 4-bit CLA to detect overflow 가 필요하다. |

1. Modification of 32bits CLA

|  |
| --- |
|  |
| 위 그림은 Modification of 32bits CLA이다. 언뜻 봐서는 이전의 32bits CLA와 같아 보이지만 다른 점이 있다. 8개의 cla4를 사용하였던 이전버전과는 다르게 마지막 8번째 CLA를 4-bit CLA to detect overflow로 바꾸었다. 그 이유는 32-bits CLA역시 Overflow를 확인하기 위해 마지막 co와 그전의 carry를 구해야 되기 때문이다. |

1. Opcode

|  |  |
| --- | --- |
| Opcode | Operation |
| 3’b000 | NOT A |
| 3’b001 | NOT B |
| 3’b010 | AND |
| 3’b011 | OR |
| 3’b100 | XOR |
| 3’b101 | XNOR |
| 3’b110 | ADD |
| 3’b111 | SUB |

이번 ALU는 위 표와 같이 Opcode에 따라서 해당하는 기능들을 수행하게 된다.

총8개의 명령어이므로 3bit로 구성하였다.

1. 4bits ALU

i) Structural description

|  |  |
| --- | --- |
|  | 옆의 그림은 4bits ALU 의 structural description 이다.  입력으로 4bits의 a, b 3bits의 opcode가 들어온다. 8개의 연산을 실행하고, 이후 그 결과가 8-to-1mux에 들어간다. 이때 그 8-to-1 mux의 신호가 opcode에 따라서 result(4bits) 값을 결정짓게 되며, 그 result값을 Calculate flags에서 받 게되는데 그 값에따라 flag값들을 update시킨다. |

ii) I/O Description

|  |  |  |  |
| --- | --- | --- | --- |
| **Port** | **Name** | **Bandwidth** | **Description** |
| Input | a | 4 bit | Operand A |
| b | 4 bit | Operand B |
| op | 3 bit | Opcode |
| Output | result | 4 bit | result |
| c | 1 bit | Carry flag |
| n | 1 bit | Negative flag |
| z | 1 bit | Zero flag |
| v | 1 bit | Overflow flag |

iii ) Module Description

|  |  |  |
| --- | --- | --- |
| **Classification** | **Name** | **Description** |
| Module | alu4 | 4-bits ALU |
| Instance | U0\_inv\_4bits | 4-bits Inverter a |
| U1\_inv\_4bits | 4-bits Inverter b |
| U2\_and2\_4bits | 4-bits a and b |
| U3\_or2\_4bits | 4-bits a or b |
| U4\_xor2\_4bits | 4-bits a xor b |
| U5\_xnor2\_4bits | 4-bits a xnor b |
| U6\_add | 4-bits CLA for add |
| U7\_sub | 4-bist CLA for sub |
| U8\_mx8\_4bits | 4-bits 8-to-1 Multiplexer |
| U9\_cal\_flags4 | Calculate flags |

1. 32-bits ALU

i) Structural description

|  |  |
| --- | --- |
|  | 옆의 그림은 32bits ALU 의 structural description 이다.  4bits ALU의 structural description과 동일하고 입출력 bit size만 다르다.  입력으로 32bits의 a, b 3bits의 opcode가 들어온다. 논리, 산술 연산도 32bit로 수행가능하도록 바꾸어준다. 이때 overflow연산을 하기 위하여 4bits CLA 7개와 마지막 1개의 modification of 4 bits CLA로 바꾸어준다. 8개의 연산을 실행하고, 이후 그 결과가 8-to-1mux에 들어간다. 이때 그 8-to-1 mux의 신호가 opcode에 따라서 result값을 결정짓게 되며, 그 result값을 Calculate flags에서 받게 되는데 그 값에 따라 flag값들을 update시킨다. |

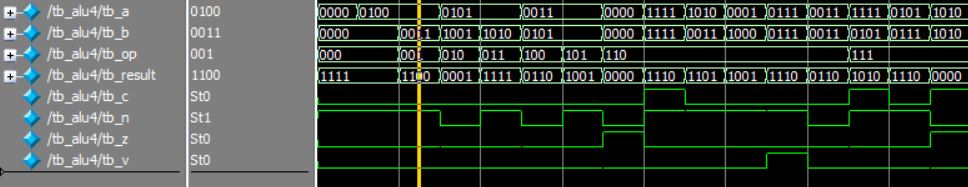
ii) I/O Description

|  |  |  |  |
| --- | --- | --- | --- |
| **Port** | **Name** | **Bandwidth** | **Description** |
| Input | a | 32 bit | Operand A |
| b | 32 bit | Operand B |
| op | 3 bit | Opcode |
| Output | result | 32 bit | result |
| c | 1 bit | Carry flag |
| n | 1 bit | Negative flag |
| z | 1 bit | Zero flag |
| v | 1 bit | Overflow flag |

iii ) Module Description

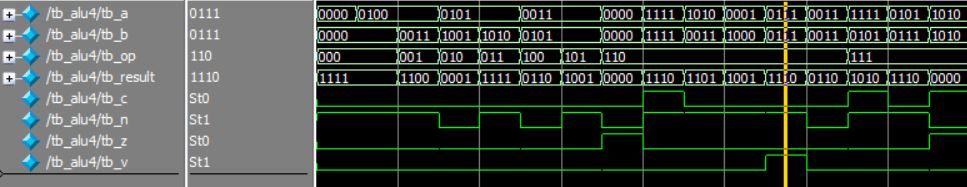
|  |  |  |
| --- | --- | --- |
| **Classification** | **Name** | **Description** |
| Module | alu32 | 32-bits ALU |
| Instance | U0\_inv\_32bits | 32-bits Inverter a |
| U1\_inv\_32bits | 32-bits Inverter b |
| U2\_and2\_32bits | 32-bits a and b |
| U3\_or2\_32bits | 32-bits a or b |
| U4\_xor2\_32bits | 32-bits a xor b |
| U5\_xnor2\_32bits | 32-bits a xnor b |
| U6\_add | 32-bits CLA for add |
| U7\_sub | 32-bist CLA for sub |
| U8\_mx8\_32bits | 32-bits 8-to-1 Multiplexer |
| U9\_cal\_flags32 | Calculate flags |

1. 설계 검증 및 실험 결과
2. 4-bits ALU
   1. 시뮬레이션 결과



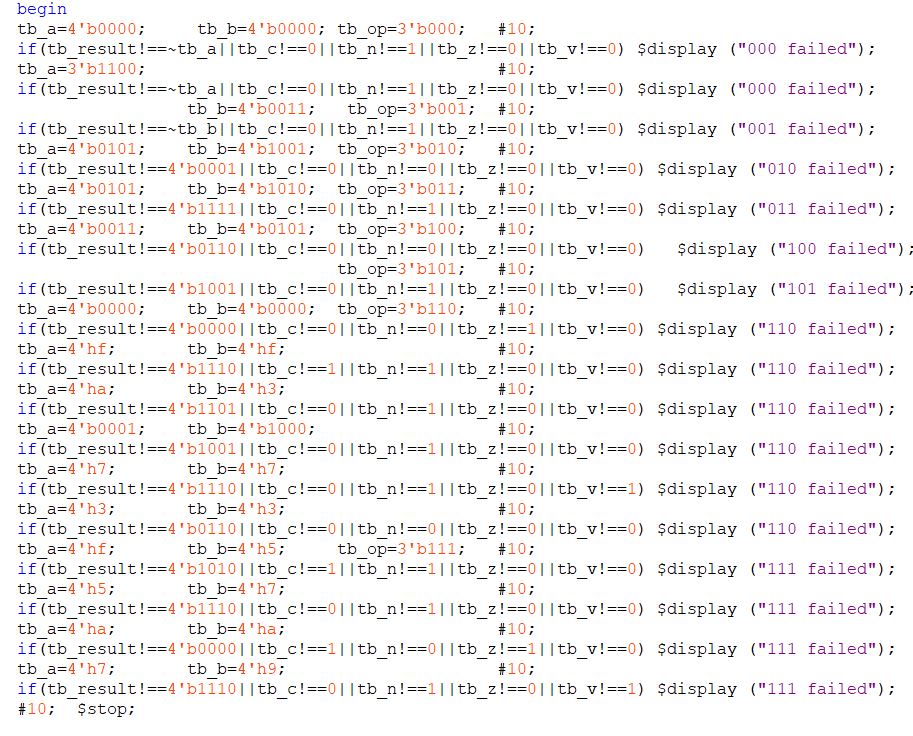
**<4-bits ALU 의 waveform 1>**

위 그림은 self-checking 방법을 사용한 testbench의 waveform (binary number 로 설정)이다. tb\_op 가 001 이므로 00112 인 B의 값을 반전 시킨 11002 인것을 보면 NOT B 논리 연산이 정확히 나온 것을 확인할 수 있다.



**<4-bits ALU 의 waveform 2>**

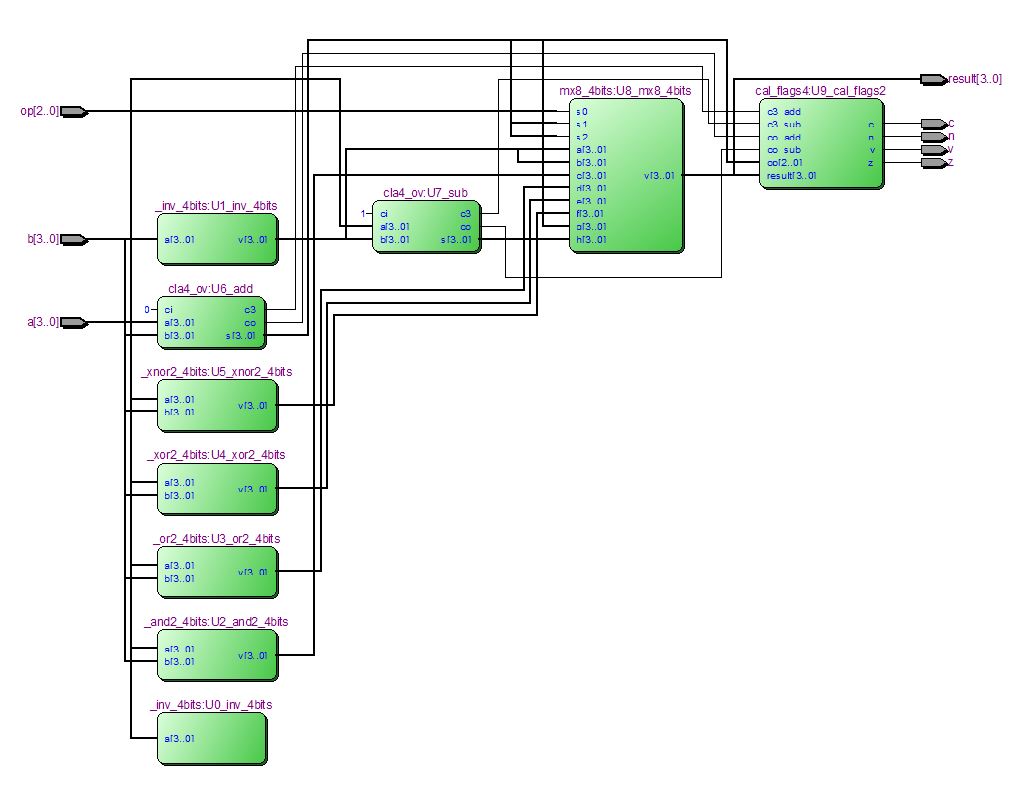
2번째 waveform은 tb\_op 가 1102 , 즉, add 산술 연산이다. A값은 01112 이고, B값도 01112 이므로 01112 + 01112 = 11102 으로 result 값이 정확히 나온 것을 확인할 수 있다. 이때, 최상위 비트를 sign bit로 인식해서 0에서 1로 변화되었기 때문에 V가 1로 set이 되었고, 또한, sign bit에서 1은 음수로 인식하기 때문에 N도 1로 set 된 것을 확인할 수 있다.



**<4-bits ALU 의 self-checking testbench>**

위 그림은 4-bits ALU 의 self-checking 방식으로 짠 testbench 이다.

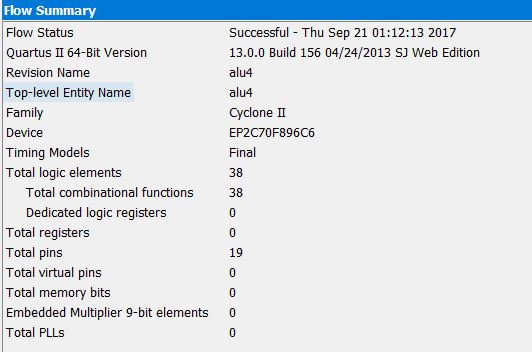
* 1. 합성(synthesis) 결과



**<4-bits ALU 의 RTP Map Viewer>**

위 그림은 4-bits ALU의 RTL Map Viewer 이다. 입력으로 3비트의 opcode와 4비트의 a, b가 입력됨을 확인할 수 있고, 논리 연산과 산술 연산 module이 있는데, sub를 살펴보면 a와 inverter된 b가 input으로 들어가고 ci로 11이 들어가는 것을 확인할 수 있다.

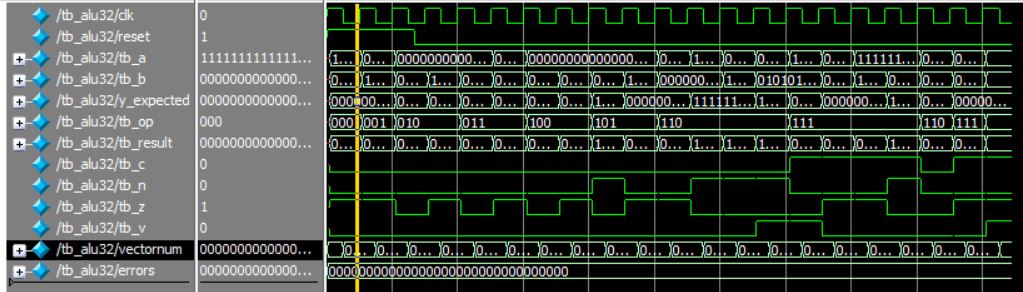
그 후 연산 결과가 mx8에 들어감을 확인할 수 있다. m8에 들어오는 opcode의 값에 따라서 선택적으로 하나의 결과를 result로 보내는 것을 확인할 수 있고, 이때의 결과는 cal\_flag 에도 들어가서 flag값인 C, N, Z, V 값을 update 하게 된다.



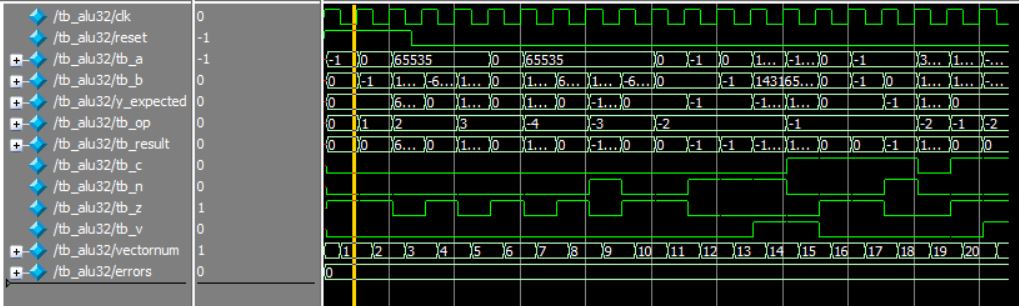
**<4-bits ALU 의 Flow Summary>**

위 그림은 4-bits ALU의 Flow Summary 이다. Total logic elements는 38 이고, Total pins는 19 임을 확인할 수 있다.

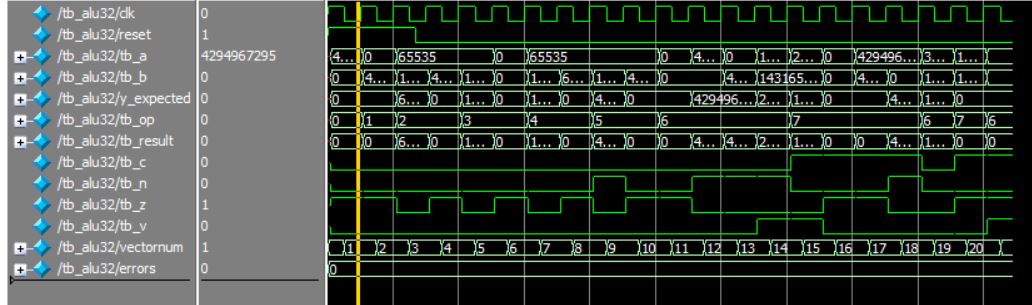
1. 32-bits ALU
2. 시뮬레이션 결과



**<32-bits ALU 의 waveform 1(binary)>**



**<32-bits ALU 의 waveform 2(decimal)>**

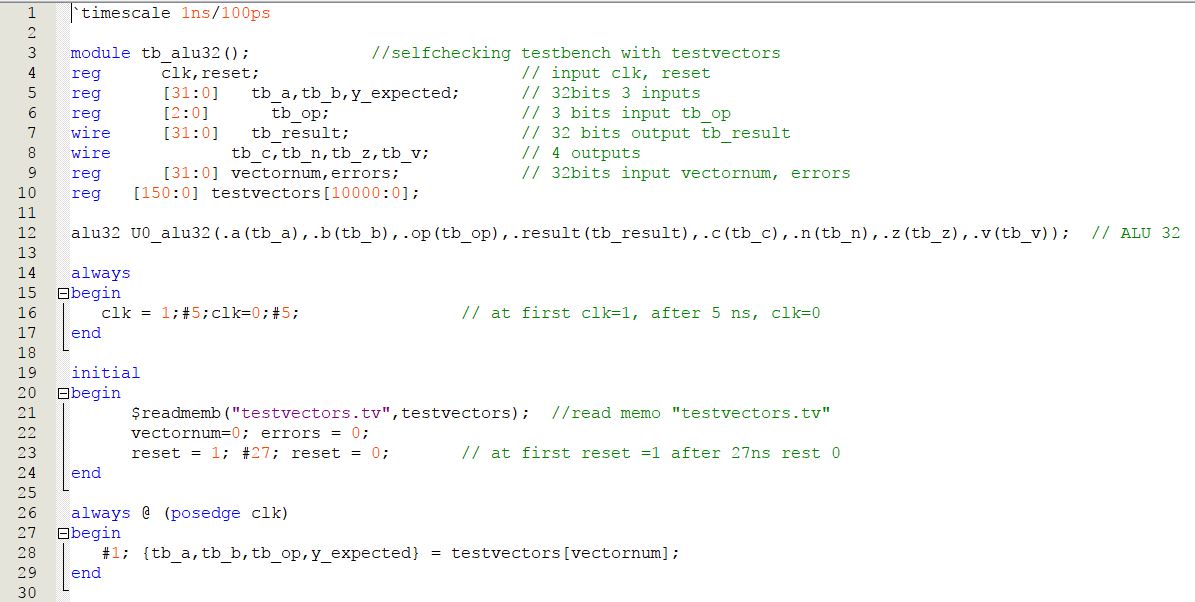


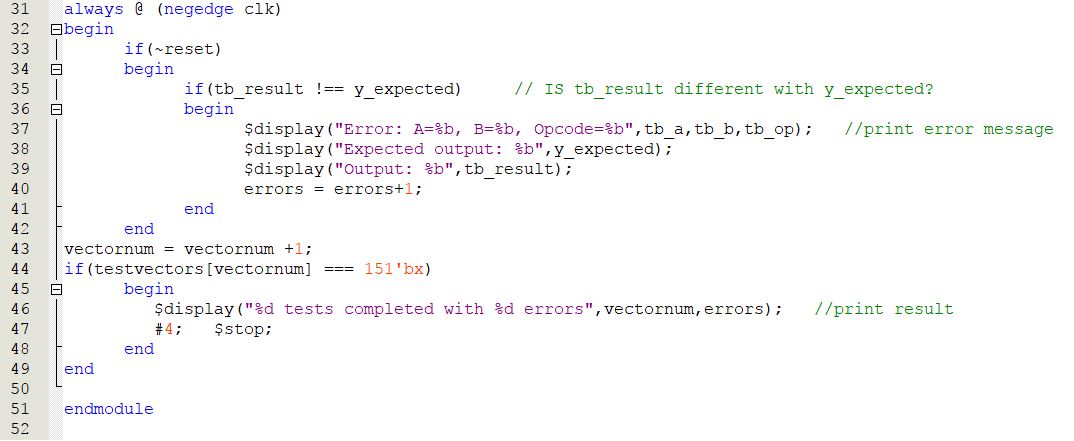
**<32-bits ALU 의 waveform 3(unsigned)>**

tb\_a = 6666\_6666, tb\_b=6666\_6666 일 때 Subtractor :

a=32’h66666666, b=32’h66666666, opcode=3’b111 일 때, 일반적인 두 수의 뺄셈이 됨을 확인할 수 있다. Signed Number에서 32’h66666666에서 32’h66666666을 뺐을 때, 같은 값을 뺐으므로 결과값이 0이 나오는 것을 확인할 수 있다.

Unsigned Number에서는 0이 아닌 두 수의 합이 나와야 하지만 두 수의 합에서 Carry out이 발생하여, Carry out을 제외한 32-bits 만 읽어 값이 0이 된다. 하지만 뺄셈에서는 Carry Flag를 set 하지 않으므로 Carry Flag는 0이고, 모든 bit이 0이기 때문에 Zero Flag가 1이다.

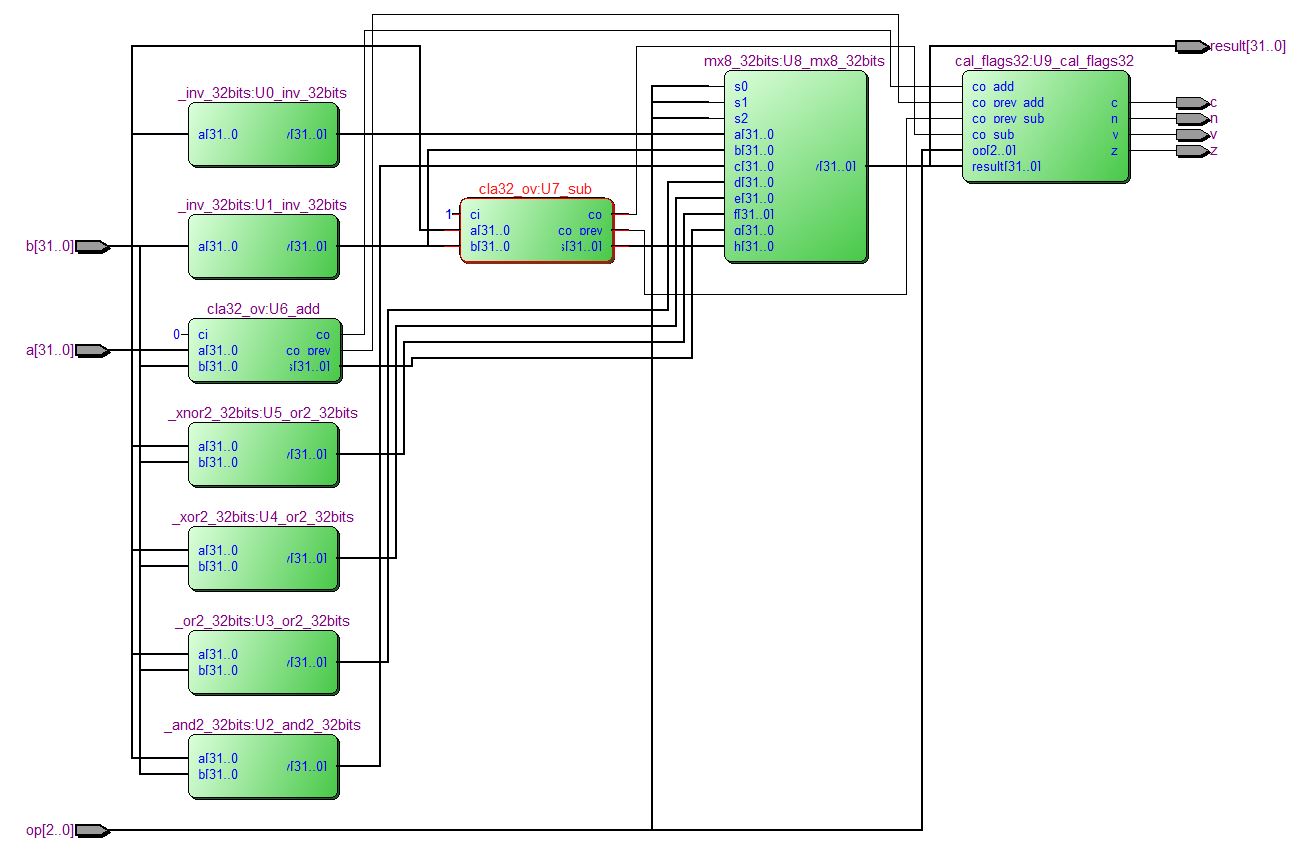




**<32-bits ALU 의 testbench>**

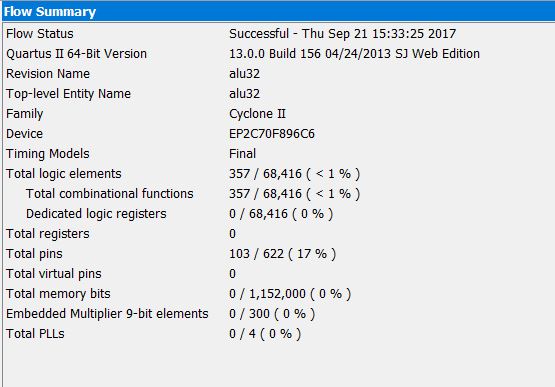
위 그림은 32-bits ALU 의 Self-checking with testvectors 방식의 testbench 이다.

1. 합성(synthesis) 결과



**<32-bits ALU 의 RTL Map Viewer>**

위 그림은 32-bits ALU의 RTL Map Viewer이다. input a, b 그리고 opcode를 통해 8개의 연산들이 수행되고, 그 결과들이 mx8의 입력으로 들어가 선택되어 result 값이 나오는 것을 확인할 수 있다. 그리고 그 결과가 C, N, Z, V의 결과로 출력되는 것을 확인할 수 있다. 이때, Addition 과 Subtraction 을 CLA 하나로 동작하기 위해 32-bits MUX를 사용한 것을 확인할 수 있다.



**<32-bits ALU 의 Flow Summary>**

위 그림은 32-bits ALU의 Flow Summary 이다. Total logic elements 가 357, Total pins가 103 임을 확인할 수 있다.

1. 고찰 및 결론
   1. 고찰

이번 실습 시간에서는 Subtractor & Arithmetic Logic Unit(ALU)에 대해 실습을 진행했다. 디지털 논리 회로 수업 시간에 ALU 에 대한 내용을 미리 배우고 실습했기 때문에 원리를 이해하는 것은 크게 어렵지않았으나 코드를 작성하는 것은 별개의 문제였다. 특히, 새로 배운 Self checking with testvertors 부분 이었다. 4-bits ALU 의 경우 testbench를 Self checking 방식으로 구현 하는 것은 어렵지 않았으나 testvectors 를 이용해 32-bits ALU를 구현하는 부분에선 조금 어려움이 있었다. 처음엔 완벽하게 구현했다고 생각했는데 WaveForm 에서는 빨간 줄들이 나와 출력 결과를 확인할 수 없었지만 testvectors.tv 파일을 만들어 입력값을 주어야 한다는 점을 깨닫고, 수정 후 올바른 결과를 확인할 수 있었다.

보고서를 작성하면서 그 전에 헷갈렸었던 Carry 와 OverFlow 가 설정되는 조건과 차이점도 명확하게 이해할 수 있었다.

* 1. 결론

Arithmetic Logic Unit(ALU) 는 두 숫자의 논리 연산, 산술 연산을 해주는 디지털 회로이다. 이러한 연산을 해주기 위해서는 해당 opcode를 Multiplexer에서 입력받아 그에 따른 result 값과 flag 값들을 set 한다. 이때 나오는 flag 가 다양한 조건문으로 사용된다.

Subtractor 에서 감산을 하기 위해, 예를 들어 A – B 의 경우, B 의 two’s complement 를 이용해 A + (-B) 형식으로 뺄셈을 덧셈으로 표현이 가능하다는 것을 깨닳았다. 따라서 하나의 Adder를 이용하여 덧셈과 뺄셈 연산 모두 수행이 가능하다는 점을 알 수 있다.

1. 참고문헌

공진흥 / 컴퓨터공학기초실험2 / 새빛관303호(광운대학교) / 2017년.

이준환 / 디지털논리회로2 / 참빛관B101호(광운대학교) / 2017년.