1. 10주차 결과보고서

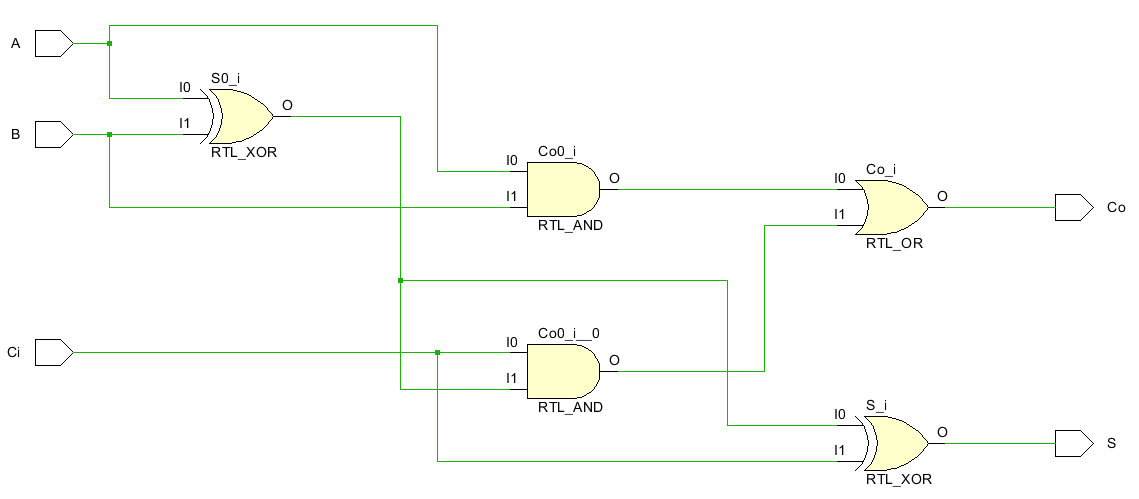
|  |
| --- |
| - 4 bit Binary Parallel Adder/Subtractor  - 4 bit Adder + Subtractor  - BCD Adder  - 결과 검토 및 논의 사항  - 추가 이론 |

**20141196 김성희**

1. 4 bit Binary parallel Adder/ Subtractor
   1. **가. 1 bit Full Adder**

|  |  |
| --- | --- |
| **-논리식** Sum = A⊕B⊕Cin Cout = AB+Cin(A⊕B) | **-코드**  A, B, Cin 🡨 input  S, Cout 🡨 output  Cin : carry in Cout : carry out S : sum |

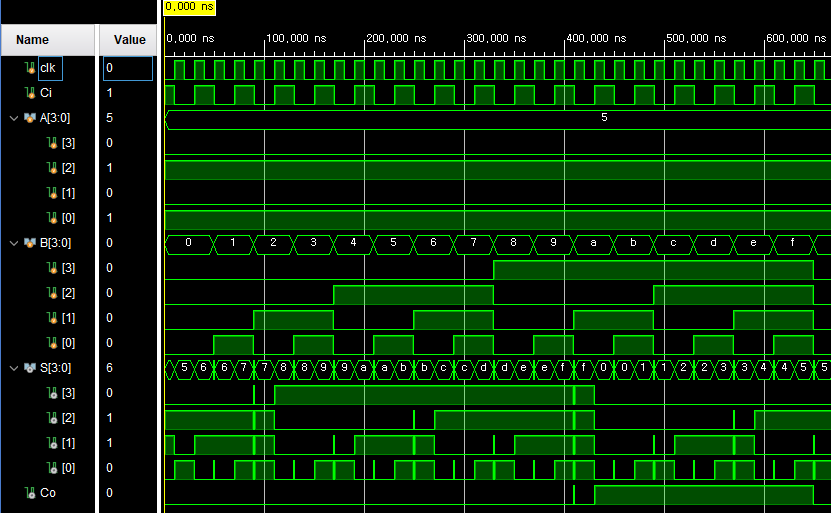
**-회로도(schematic)**

  
A⊕B 게이트를 재사용하는 것을 볼 수 있다.

* 1. **나. 4 bit Binary Parallel Adder**

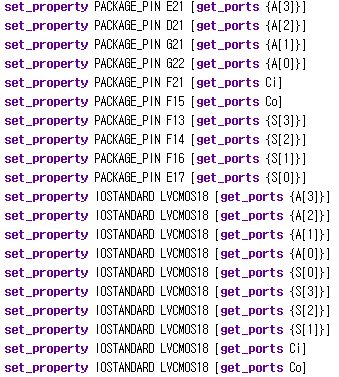
**-코드** (좌 : Design source, 우: Simulation source – 4bit input A의 값을 고정, B의 값을 변조)

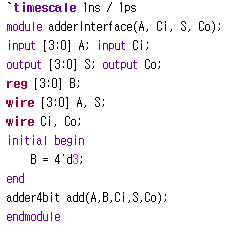
|  |  |
| --- | --- |
| A,B는 input bit를, Ci는 carry in을 의미한다. S는 sum을, Co는 carry out을 의미하며 위 가.에서의 1bit full adder module 4개를 사용했다. |  |

**-simulation**  
  
A는 0101로 고정되어 있는 모습을 볼 수 있다. (A, B 둘 다 바꾸면 512가지의 경우의 수가 나온다.) A =A[3][2][1][0], B =B[3][2][1][0], S =S[3][2][1][0], Ci는 첫 carry in, Co는 마지막 carry out을 의미한다.

**-fpga**

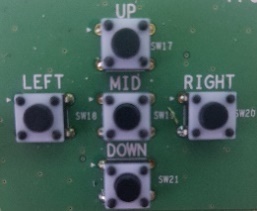
업로드:  
 Design Sources경로에 code 짜기 -> Run Synthesis & Run Implementation -> Constraints경로에 .xdc 파일 추가하기 -> Open Implemented Design 클릭 후 Constraints Wizard 클릭(Define Target, .xdc파일 타겟 설정) -> Window 탭에서 I/O ports 클릭 -> 원하는 pin 선택 및 LVCMOS18(I/O Std) 선택 -> 저장 후 .xdc reload(아래 그림처럼 코드가 자동으로 짜서 나온다.) -> Generate Bitstream 클릭(Synthesis, Implementation도 자동으로 실행) 후에 Open Hardware Manager 클릭 -> Open Target 클릭 후 Auto Connect 클릭 -> Program Device 클릭 -> FPGA에 업로드 끝

 <- input output port에 pin 배정(LED, SWITCH)

 <- fpga와 4bit adder모듈사이를 연결해 줄 모듈 코드

Fpga의 switch가 5개밖에 없어서 4bit input중 하나인 B는 0011로 고정해 놓았다.   
다음 표는 input값에 따른 output값이 fpga에서 어떻게 나타나는지에 대한 표이다.  
PUSH SWITCH에서 U: UP, L: LEFT, M: MID, R: RIGHT, D: DOWN을 의미한다.  
사진의 LD1~5는 각각 Co, S[3]~[0]을 의미한다. 즉 첫번째 사진의 결과는 00110을 의미한다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **사진** |  |  |  |  |
| **A[3] [2] [1] [0] Cin** | **0000 0** | **0001 0** | **0010**  **0** | **0011**  **0** |
| **PUSH**  **SWITCH** |  | **R** | **M** | **M R** |
| **사진** |  |  |  |  |
| **A[3] [2] [1] [0] Cin** | **0100**  **0** | **0101**  **0** | **0110**  **0** | **0111**  **0** |
| **PUSH**  **SWITCH** | **L** | **L R** | **L M** | **L M R** |
| **사진** |  |  |  |  |
| **A[3] [2] [1] [0] Cin** | **1000**  **0** | **1001**  **0** | **1010**  **0** | **1011**  **0** |
| **PUSH**  **SWITCH** | **U** | **U R** | **U M** | **U M R** |
| **사진** |  |  |  |  |
| **A[3] [2] [1] [0] Cin** | **1100**  **0** | **1101**  **0** | **1110**  **0** | **1111**  **0** |
| **PUSH**  **SWITCH** | **U L** | **U L R** | **U L M** | **U L M R** |
| **사진** |  |  |  |  |
| **A[3] [2] [1] [0] Cin** | **0000**  **1** | **0001**  **1** | **0010**  **1** | **0011**  **1** |
| **PUSH**  **SWITCH** | **D** | **R D** | **M D** | **M R D** |
| **사진** |  |  |  |  |
| **A[3] [2] [1] [0] Cin** | **0100**  **1** | **0101**  **1** | **0110**  **1** | **0111**  **1** |
| **PUSH**  **SWITCH** | **L D** | **L R D** | **L M D** | **L M R D** |
| **사진** |  |  |  |  |
| **A[3] [2] [1] [0] Cin** | **1000**  **1** | **1001**  **1** | **1010**  **1** | **1011**  **1** |
| **PUSH**  **SWITCH** | **U D** | **U R D** | **U M D** | **U M R D** |
| **사진** |  |  |  |  |
| **A[3] [2] [1] [0] Cin** | **1100**  **1** | **1101**  **1** | **1110**  **1** | **1111**  **1** |
| **PUSH**  **SWITCH** | **U L D** | **U L R D** | **U L M D** | **U L M R D** |

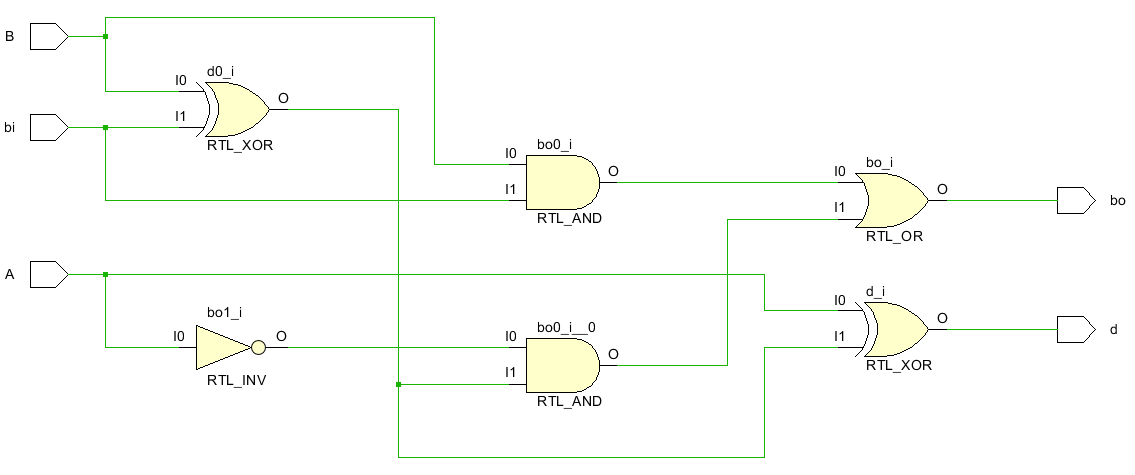


input A[3][2][1][0] = UP LEFT MID RIGHT, Ci = DOWN

* 1. **다. 1 bit Full Subtractor**

|  |  |
| --- | --- |
| **-논리식** d = A⊕B⊕bin bout=Bbin+A**’(**B+bin)  =Bbin+A**’**(B⊕bin)  d = difference  b = borrow | **-코드**  A, B, bin 🡨 input  d, bout 🡨 output  bin : borrow in bout : borrow out d : difference |

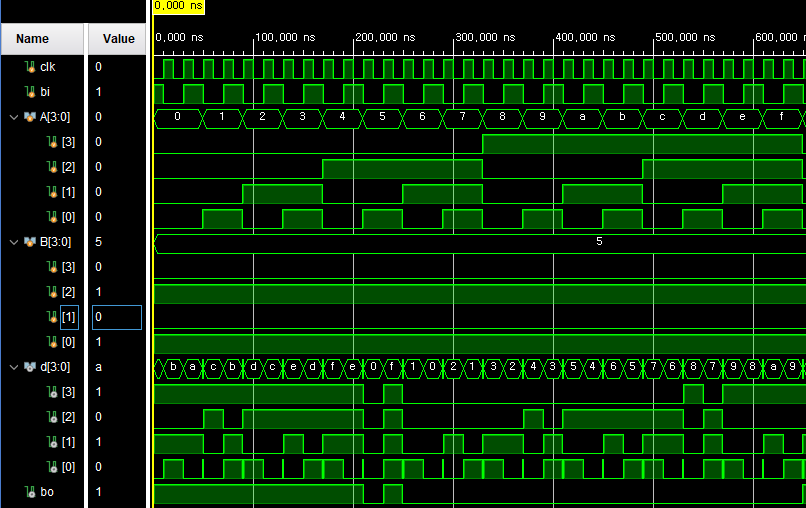
**-회로도(schematic)**

  
B⊕bin 게이트를 재사용하는 것을 볼 수 있다.

* 1. **라. 4 bit Binary Parallel Subtractor**

**-code** (좌: Design source, 우: simulation source) A-B

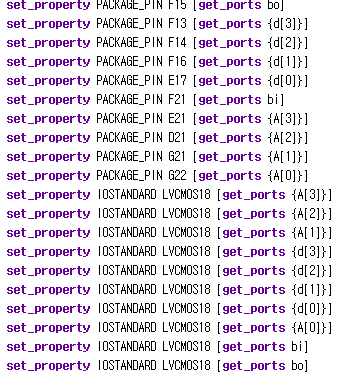
|  |  |
| --- | --- |
| A,B는 input bit를, bi는 borrow in을 의미한다. d는 difference를, bo는 borrow out을 의미하며 위 다.에서의 1bit full subtractor module 4개를 사용했다. |  |

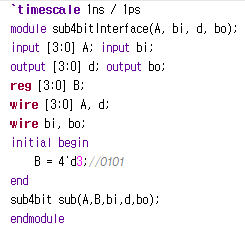
**-simulation A-B = d with bo  
**

B는 0101으로 고정해 놨다. (경우의 수를 줄이기 위해서) 그림의 Value는 다음과 같이 해석할 수 있다. 0000-0101 (with bi=1) =1010 (with bo=1)즉 10000+0000-0101-0001=1010을 의미한다.  
(10000\*bo + A-B-bi = d)

**-fpga**

업로드:  
 Design Sources경로에 code 짜기 -> Run Synthesis & Run Implementation -> Constraints경로에 .xdc 파일 추가하기 -> Open Implemented Design 클릭 후 Constraints Wizard 클릭(Define Target, .xdc파일 타겟 설정) -> Window 탭에서 I/O ports 클릭 -> 원하는 pin 선택 및 LVCMOS18(I/O Std) 선택 -> 저장 후 .xdc reload(아래 그림처럼 코드가 자동으로 짜서 나온다.) -> Generate Bitstream 클릭(Synthesis, Implementation도 자동으로 실행) 후에 Open Hardware Manager 클릭 -> Open Target 클릭 후 Auto Connect 클릭 -> Program Device 클릭 -> FPGA에 업로드 끝

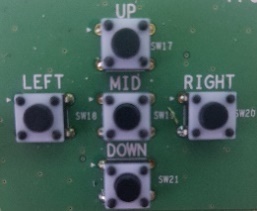
 <- input output port에 pin 배정(LED, SWITCH)

 <- fpga와 4bit subtractor모듈사이를 연결해 줄 모듈 코드

0011

Fpga의 switch가 5개밖에 없어서 4bit input중 하나인 B는 0011로 고정해 놓았다.  
(시뮬레이션에서는 0101이엇으나 fpga에서는 0011로)  
다음 표는 input값에 따른 output값이 fpga에서 어떻게 나타나는지에 대한 표이다.  
4 bit Binary Parallel Adder와 같은 방식이기에 4개만 표현하였다. 나머진 표로 대체  
PUSH SWITCH에서 U: UP, L: LEFT, M: MID, R: RIGHT, D: DOWN을 의미한다.  
사진의 LD1~5는 각각 bo, d[3]~[0]을 의미한다. 즉 첫번째 사진의 결과는 0,0110을 의미한다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **사진** |  |  |  |  |
| **A[3] [2] [1] [0] bin** | **0000 0** | **1011 0** | **0000**  **1** | **1011**  **1** |
| **PUSH**  **SWITCH** |  | **U M R** | **D** | **U M R D** |

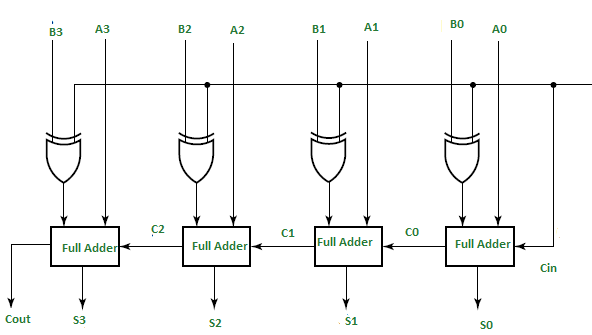


input A[3][2][1][0] = UP LEFT MID RIGHT, bi = DOWN

B = 0011로 고정; 10000\*bo+A-B-0001\*bi=d;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A[3] | A[2] | A[1] | A[0] | bi | D | bo |
| 0 | 0 | 0 | 0 | 0 | 1101 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1100 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1110 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1101 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1111 | 1 |
| 0 | 0 | 1 | 0 | 1 | 1110 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0000 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1111 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0001 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0000 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0010 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0001 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0011 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0010 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0100 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0011 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0101 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0100 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0110 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0101 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0111 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0110 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1000 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0111 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1001 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1000 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1010 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1001 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1011 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1010 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1100 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1011 | 0 |

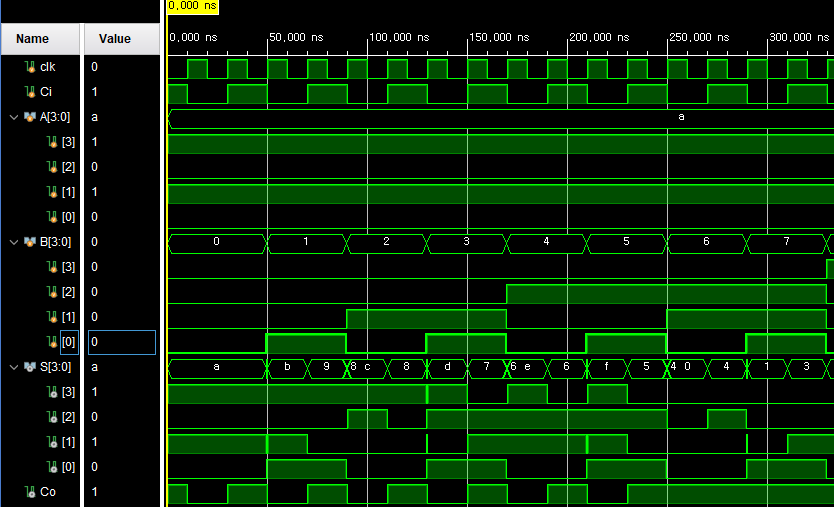
4bit Adder + Subtractor

위 나. 의 Adder에 어떤 부분을 추가하여 subtractor 기능을 추가할 것이다.  
그러나 이때 원래 Adder의 input 값인 Ci, 즉 첫 carry in은 carry in으로서의 기능을 하지 못하고 Adder로서 쓰일 것인지 Subtractor로서 쓰일 것인지 선택하는 일종의 switch 역할을 하게 된다. (Ci값이 0이면 Adder로, Ci값이 1이면 Subtractor로서 작용한다. Ci = Cin)  
  
A-B == A + (-B) == A + (!B +1) == A + (B⊕1111(2) + 1)

**-code**

|  |  |
| --- | --- |
| FA에 A[i]와 B[i]^Ci를 넣었기 때문에 Ci가 0일 때는 원래의 adder와 같이 A[i], B[i]가 입력으로 들어가지만 Ci가 1이라면 A[i]와 !B[i]가 입력으로 들어가서 subtractor 역할을 하게 된다. |  |

**-simulation**



A는 1010으로 고정해 놨다. B가 0001일 때를 살펴보자. Ci가 0인 경우 S는 1011이고 1인 경우 1001이다. 즉 Ci=0이면 1010+0001=1011이, Ci=1이면 1010-0001=1001이 적용되는 것을 볼 수 있다.

**-fpga**

Adder와 Subtractor를 합치기 전의 Adder와 동일한 방식으로 진행된다. Port와 pin도 똑같이 배정했으며 단지 B와 Ci사이에 exiclusive or를 적용했을 뿐이다. 따라서 간략하게 같은 A값에 대해서 Ci가 0일 때와 1일 때의 사진만을 비교하겠다. (B는 1010으로 고정, LD1: Co (Cout), LD2~5: S[3]~[0], switch는 UP LEFT MID RIGHT DOWN : A[3]~[0], Ci (Cin))

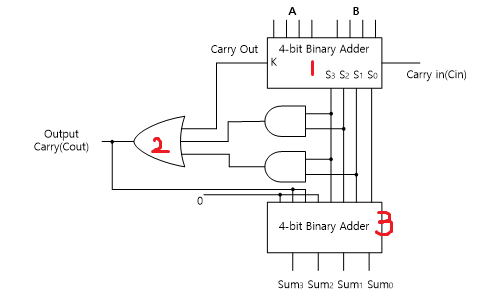
(A=1101, 좌: Ci=0, 우: Ci=1)

|  |  |
| --- | --- |
|  |  |

좌: 1101+1010 = 10111 우: 1101-1010 = 10011

BCD Adder

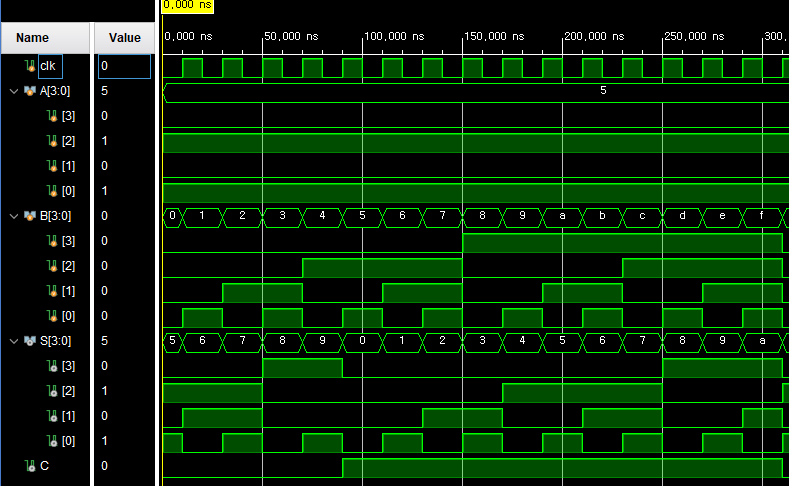
**-회로**



**-code**

|  |  |
| --- | --- |
| - add1 인스턴스는 위 회로의 1번 adder를 의미한다. - 따라서 B2는 1번 adder의 결과값(sum)이다.  - C는 output(Cout)을 의미하며 2번 or 게이트에 대한 연산 결과 값이다.  - A2=4’b0110\*C; 같은 경우는 1번 adder에서의 결과값이 10이상일 때 일의 자리수를 10미만의 수로 낮추기 위한 input값이다. (만약 10미만의 결과가 나왔다면 A2=0이 되어서 1번의 결과가 그대로 3번의 결과로 이어진다.)  - add2 인스턴스는 3번 adder를 의미한다. 이어서 S는 3번 adder의 결과값이다. | Input A값을 5로 고정 후 B의 값을 변조했다. |

**-simulation**



A+B = C S를 의미한다. 즉 한 자리 십진수 A, B에 대해서 그 덧셈의 값의 십의 자리수가 C, 일의 자리수가 S를 의미한다. 예를 들어 A=5, B=1001(9)일 때 C=1, S=0100임을 확인할 수 있는데 이는 5+9=CS=14를 의미한다. B가 10이상일 때에도 결과 값이 십의 자리와 일의 자리 두개로 잘 쪼개어지지만 실제로는 의미 없는 값이다. BCD Adder가 각 자리수마다 따로 계산하기 위해 만들어졌으므로 실제 사용할 때는 input값이 0~9이기 때문이다. 물론 A=5, B=15같은 같이 십의 자리수가 2이상이 되는 경우는 C=1, S=1010 처럼 이상하게 쪼개진다.

**-fpga**

업로드:  
 Design Sources경로에 code 짜기 -> Run Synthesis & Run Implementation -> Constraints경로에 .xdc 파일 추가하기 -> Open Implemented Design 클릭 후 Constraints Wizard 클릭(Define Target, .xdc파일 타겟 설정) -> Window 탭에서 I/O ports 클릭 -> 원하는 pin 선택 및 LVCMOS18(I/O Std) 선택 -> 저장 후 .xdc reload(아래 그림처럼 코드가 자동으로 짜서 나온다.) -> Generate Bitstream 클릭(Synthesis, Implementation도 자동으로 실행) 후에 Open Hardware Manager 클릭 -> Open Target 클릭 후 Auto Connect 클릭 -> Program Device 클릭 -> FPGA에 업로드 끝

|  |  |
| --- | --- |
| port와 pin 배정 | fpga와 BCD adder모듈을 연결해주는 모듈이다. |

A의 값을 십진수 5로 고정했다. 위의 시뮬레이션과 동일한 조건이다. 결과값 십의 자리수 C는 LD1과, 일의 자리수 S[3]~[0]은 LD2~5와 연결했다. 입력값 B[3]~[0]는 각각 UP LEFT MID RIGHT switch와 연결했다.

Ex) B = 0111일 때 (LEFT MID RIGHT switch를 눌렀을 때), 5+7=12이므로 C=1 S=0010이 된다. 즉 LD1, 4에 불이 들어온다. 다음 표를 통해 어떤 스위치를 눌렀을 때 어떤 LD에 불이 들어오는지 알 수 있다.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **A+B** | **B[3] UP** | **B[2] LEFT** | **B[1] MID** | **B[0] RIGHT** | **결과**  **십진수** | **C LD1** | **S[3] LD2** | **S[2] LD3** | **S[1] LD4** | **S[0]**  **LD5** |
| **5+0** | **0** | **0** | **0** | **0** | **5** | **0** | **0** | **1** | **0** | **1** |
| **5+1** | **0** | **0** | **0** | **1** | **6** | **0** | **0** | **1** | **1** | **0** |
| **5+2** | **0** | **0** | **1** | **0** | **7** | **0** | **0** | **1** | **1** | **1** |
| **5+3** | **0** | **0** | **1** | **1** | **8** | **0** | **1** | **0** | **0** | **0** |
| **5+4** | **0** | **1** | **0** | **0** | **9** | **0** | **1** | **0** | **0** | **1** |
| **5+5** | **0** | **1** | **0** | **1** | **10** | **1** | **0** | **0** | **0** | **0** |
| **5+6** | **0** | **1** | **1** | **0** | **11** | **1** | **0** | **0** | **0** | **1** |
| **5+7** | **0** | **1** | **1** | **1** | **12** | **1** | **0** | **0** | **1** | **0** |
| **5+8** | **1** | **0** | **0** | **0** | **13** | **1** | **0** | **0** | **1** | **1** |
| **5+9** | **1** | **0** | **0** | **1** | **14** | **1** | **0** | **1** | **0** | **0** |



\*fpga의 LD1~16과 Switch UP LEFT MID RIGHT DOWN

1. 기타 논의 및 검토 사항 + 추가 이론

- 4bit binary parallel adder/subtractor에서 A, B값 중 하나를 고정할 때 simulation과 fpga에서 서로 다른 값을 고정한 점이 아쉽다. 실험을 마치고 정리할 때 발견했는데 만약 동일한 값을 고정했다면 좀 더 매끄럽게 정리가 됐을 것이다.

- Adder와 Subtractor를 합친 회로에 대한 실험에서 이상한 현상이 있다. 1101-1010이 0011이 아닌 10011이 된다. 이는 1101-1010을 1101+NOT(1010)+1로 바꾸어 계산하면서 자리수가 증가해서 나타나는 현상이다. 따라서 -연산을 할 때는 carry out을 빼고 결과를 계산할 필요가 있다. 또한 0000 – 1010의 경우 -10의 값을 표현할 수가 없어서 overflow가 발생한다. 이경우 16-10과 같이 인식되어 6의 결과가 발생한다. - 연산에서Overflow는 Carry out과 최상위 bit이 서로 같을 때 발생한다.