5 장

4-Bit Adder/Subtractor & Arithmetic Logic Unit(ALU)

1. 실험 목표

▣ 논리회로에서 구현되는 양수와 음수의 표현 방법과 처리 방식에 대해서 이해한다.

▣ 덧셈기/ 뺄셈기 (Adder/Subtractor)의 구성과 동작 원리를 이해하고 직접 구성해 본다.

▣ 덧셈과 뺄셈을 동시에 수행 가능한 단일 공유 Adder/ Subtractor 를 구성해 본다.

▣ CPU (Central Processing Unit)의 핵심 모듈인 산술 논리 장치 (Arithmetic Logic Unit: ALU) 구조를 이해한다.

▣ 기본적인 ALU 구조를 설계하고, 시뮬레이션을 수행하며, FPGA 를 이용하여 실제 회로를 구현해 본다.

2. 실험 이론

(1) 4-Bit Adder/Subtractor

디지털 시스템에서는 한번에 처리하는 데이터의 단위가 4 비트, 8 비트, 16 비트 32 비트 등으로 임의의 크기의 덧셈/뺄셈을 수행하는 경우, n-비트의 덧셈을 지원하는 Adder 는 1 비트 Adder 를 n 개를 연결하여 간단하게 설계할 수 있다. 아래의 [그림 4-1]과 같이 가장 낮은 비트 자리에서는 아래 단에서 올라오는 올림수가 없기 때문에 Half Adder 를 사용하고, 다음 비트 자리부터 모든 비트 위치에서는 Full Adder 를 연결하여 구성한다. [그림 4-1]은 4-비트 Adder 의 구성도를 보여주며, 비트 0 에서 발생하는 올림수 (C_1)는 다음 단의 입력으로 연결하여 최종단의 비트 위치에서는 전체 덧셈에 대한 최종 올림수 (C_4)를 외부로 제공하게 된다.

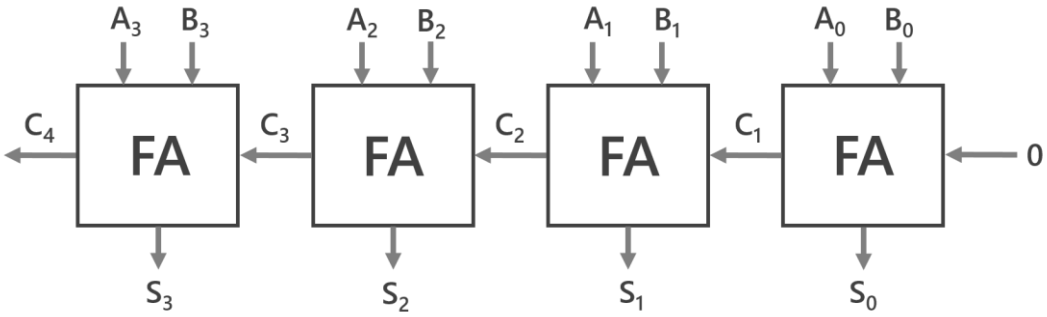

그림 5-1. 4-비트 Adder 의 구성도

뺄셈이 가능하도록 논리회로를 구성하기 위해서는 음수에 대한 표현을 고려해야 한다. 논리회로에서 음수를 표현하기 위한 방식으로 2 의 보수가 있다.

표 5-1. 4 비트의 2 의 보수 표현

숫자	2 의 보수	숫자	2 의 보수
0	0000	-8	1000
1	0001	-7	1001
2	0010	-6	1010
3	0011	-5	1011
4	0100	-4	1100
5	0101	-3	1101
6	0110	-2	1110
7	0111	-1	1111

양수의 경우에는 보통 쓰는 숫자를 이진수로 바꿔서 논리회로에서 사용할 수 있다. 그러나 음수의 경우에는 다른 방법으로 표현을 하게 된다. 즉 양수에서 음수로 음수에서 양수로 변환해야 할 때는 1 의 보수로 바꾼 다음에 1 을 더해주면 된다. 예를 들면, 0011 을 1 의 보수로 바꾸면 1100 이 되고 다시 1 을 더하면 1101 된다. 이 결과는 아래의 표 7-1 에서 -3 의 값과 같게 된다. 음수에서 양수로 바꾸는 것도 위와 동일한 방법으로 하면 된다.

논리회로에서 2 의 보수를 사용하는 장점은 양수와 음수의 합을 쉽게 할 수 있는 점이다. 이점을 이용하면 디지털 회로에서 덧셈기와 뺄셈기는 쉽게 구현할 수 있다. 실제 뺄셈기는 올림수 대신에 내림수(borrow)라는 것을 써서 구현한다. 그러나 이런 경우 덧셈기와 뺄셈기를 각각 따로 만들어야 하기 때문에 두 배의 회로가 필요하다. 그러나 2 의 보수를 사용하면 덧셈기와 1 의 보수를 바꿔주는 것, 즉 invertor 만 있으면 가능하다.

2 의 보수 표현을 사용하면 이렇듯 장점이 있지만 한 가지 단점이 있다. 만약 7+7 을 할 경우 값은 14 가 되지만 실제 나오는 값은 14 가 아니라 -2 가 나온다.

$$\begin{array}{rr} 7_{10} & 0111_2 \\ +\ 7_{10} & +\ 0111_2 \\ \hline 14_{10} & 1110_2(=-2_{10}) \end{array}$$

2 의 보수체계의 4 비트에서는 7 이 양수에서 제일 큰 값이다. 그러나 더하기에서는 이보다 큰 값이 나올 수 있다. 음수에서도 가장 작은 값은 -8 이지만 이보다 작은 값이 나올 수 있다. 이러한 현상을 오버플로우 (overflow)라 하며, 이는 결과값이 주어진 표현 가능한 수의 범위를 초과하는 경우를 의미한다. 이러한 오버플로우 방지를 위해서는 최상위 비트의 2 개의 올림수를 조사하여 두 개의 값이 서로 틀리 경우에는 정상적으로 나오게 되며 값이 같은 경우에는 오버플로우가 된다. 따라서 이 두 개의 값을 조사하면 오버플로우를 방지할 수 있다.

다음 장의 [그림 5-2]는 입력이 A_i 와 B_i 의 4 비트인 Adder 와 Subtractor 를 Full Adder 를 이용하여 표현하였다. 기본적인 형태는 위의 Adder 와 같다. 그러나 입력 B_i 에서 멀티플렉서를 연결하여 2 의 보수 표현을 선택적으로 사용할 수 있다. 즉 빼기를 하는 경우에는 2 의 보수값을 대입할 수 있게 한 것이다. 그러나 2 의 보수는 부호를 바꿀 때 1 을 더해야 한다. 이러한 역할을 수행하는 것이 첫 번째 단의 Full Adder 이다. 이전의 [그림 7-1]은 최하위 자릿수에서 Half Adder 를 이용하여 아랫 단에서의 올림수를 0 으로 가정하였다. 즉 부호를 바꿀 때인 2 의 보수를 취할 경우에는 올림수의 입력을 가해 2 의 보수효과를 만드는 것이다.

[그림 5-2]에서 가산과 감산의 구분은 $\overline{\text{Add}}$/Subtract 입력의 값으로 바꿀 수 있다. 즉 이 입력이 1 이면 Subtractor 로 동작하며, 0 이면 Adder 로 작동하게 된다. 이 입력은 멀티플렉서의 셀렉트 입력으로도 사용된다. 또한 첫 번째 Full Adder 의 올림수로 입력된다. 이 입력이 0 이면 첫 번째 Full Adder 는 Half Adder 로 작동하며 B_i 의 입력은 그대로 Adder 로 들어가게 된다. 그러나 입력이 1 일 때는 첫 번째 Full Adder 에 올림수가 들어가고 멀티플렉서는 B_i 의 반대 값을 입력하게 된다. 이것은 2 의 보수로 만들어 A_i 에서 B_i 를 빼는 결과를 만들 수 있다. 또한 오버플로우를 체크하기 위하여 최상위 2 비트는 XOR 에 연결하였다.

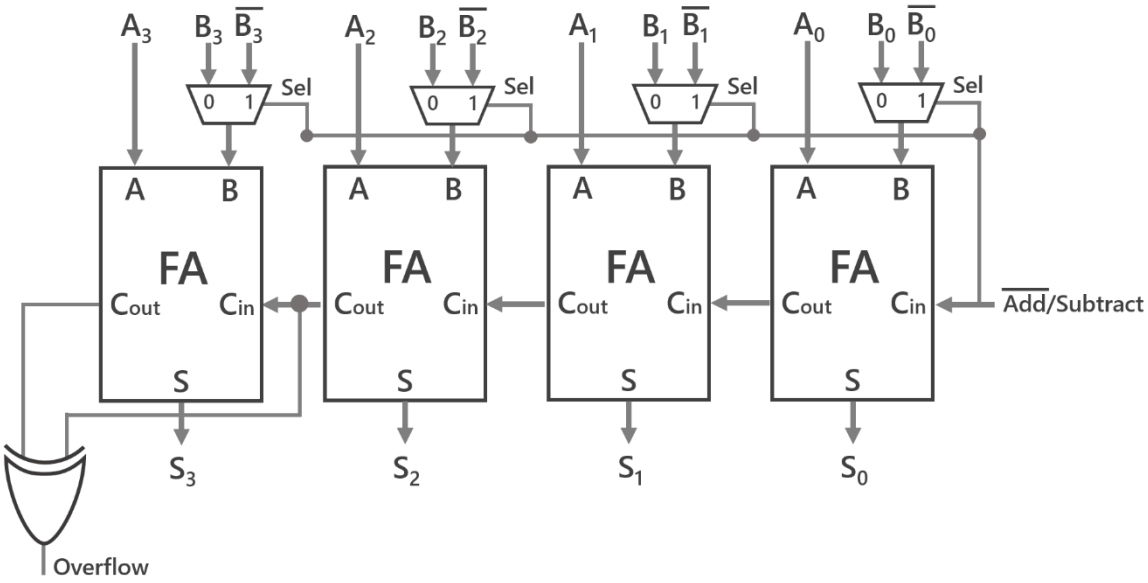

그림 5-2. 4-비트 Adder/Subtractor 의 구성도

(2) Arithmetic Logic Unit (ALU)

디지털 시스템에서 가장 핵심이 되는 모듈은 프로그램을 구성하는 명령어를 수행하는 CPU (Central Processor Unit)이다. 이러한 CPU 는 모든 디지털 시스템과 내장형 시스템의 기본 엔진으로 주요 구성은 제어장치 (Control Unit), 산술논리장치 (ALU), 레지스터 파일, 그리고 모든 모듈을 연결하는 버스 (Bus)로 구성된다. 이 모듈 중에서 핵심 모듈인 ALU 는 일반적인 산술연산 (Arithmetic)과 논리 연산 (Logic function)을 수행한다.

가) 논리연산

본 실험에서 가장 기본적인 4 비트 ALU 를 설계하여 모든 디지털시스템에 적용 가능한 범용 기기로써 기본 연산동작을 이해하기 위함이다. 아래 [그림 8-1]에 나타나 있는 대로 두 개의

논리 입력, A 와 B 에 대하여, 하나의 논리 출력, F 를 제공하는 논리회로를 설계해 보자. 두 개의 입력 A, B 에 대하여 NOT 연산을 수행한다면 A, A′, B, B′ 에 해당하는 네 가지 출력을 제공할 수 있다. 논리 연산의 가장 기본적인 연산인 AND, OR, 그리고 NOT 을 지원하고 또한 이를 조합하여 논리회로를 구성한다면 총 16 가지의 논리연산 결과 지원이 가능하다.

그림 5-3. Logic Unit 의 구성도

여기서 CPU 의 설계사양에 의해 ALU 의 기능에 대해 변화가 생긴다. 16 개의 논리연산을 수행하도록 설계한다면 ALU 는 고속으로 동작하도록 구현할 수 있겠지만, 많은 게이트를 포함하고 배선도 증가하므로 면적이 늘어나게 된다. 속도보다 최적화된 면적에 비중을 두어 설계한다면 사용빈도가 적거나 거의 없는 연산을 포함하지 않아도 된다. 예를 들어 가 구현되어있을 때 $\overline{A+B}$ 를 구현하지 않고, $\overline{A}$ 를 먼저 연산을 수행하여 저장하고, 이를 다시 입력으로 넣어서 의 연산기능을 수행하도록 한다. 물론 이러한 일은 ALU 의 동작을 여러 번 반복 수행하므로 연산 속도가 떨어지게 된다.

표 5-2: 2 입력 1 출력 논리회로의 16 개의 논리 연산

번호	논리연산	번호	논리연산
1	A	9	$A+B$
2	$\overline{A}$	10	$\overline{A+B}$
3	B	11	$\overline{A}B$
4	$\overline{B}$	12	$A\overline{B}$
5	$A\overline{A}=0$	13	$\overline{A}+B$
6	$A+\overline{A}=1$	14	$A+\overline{B}$
7	AB	15	$A\oplus B$
8	$\overline{AB}$	16	$\overline{A\oplus B}$

나) 산술연산

니블(Nibble, 바이트의 절반; 4bit)단위의 두 입력 A 와 B 를 더하거나 빼는 연산과 입력된 수의 보수(여기서는 1 의 보수를 구한다. 2 의 보수로 바꿔서 입력 값으로 사용하기 위해서는 1 의 보수를 취한 뒤 1 을 증가하는 연산(B increment)을 수행해야 한다)를 구하는 연산을 기본으로 한다. 이 연산만으로 반복수행을 통해 원하는 연산을 모두 수용할 수 있다. 하지만 많이 쓰이는 연산에 대해 회로를 추가해줌으로써 산술연산 기능을 좀더 강화 할 수 있다

여기서 구현된 쉬프트 연산은 Logical Shift Right(LSR), Logical Shift Left(LSL), Arithmetic Shift Right(ASR)이다. 쉬프트 연산을 이용하여 손쉽게 비트 수준에서 곱셈이나 나눗셈이 가능한 장점 등이 있다. Arithmetic Shift 는 부호 비트를 유지하는 쉬프트 연산이다.

다) 4 비트 ALU 의 구조

이번 실험에서 구현해야 할 4 비트 ALU 의 입출력을 살펴보면 [그림 8-2]와 같다. 그림에서 보듯이 두 개의 4 비트 입력 A(A3, A2, A1, A0), B(B3, B2, B1, B0)와 4 비트 출력 F(F3, F2, F1, F0)가 있고, 여기에 입력 값에 따른 연산 동작을 결정하는 4 비트 제어신호 S, 캐리 입력 C_{IN}, 캐리 출력 C_{OUT} 이 있다.

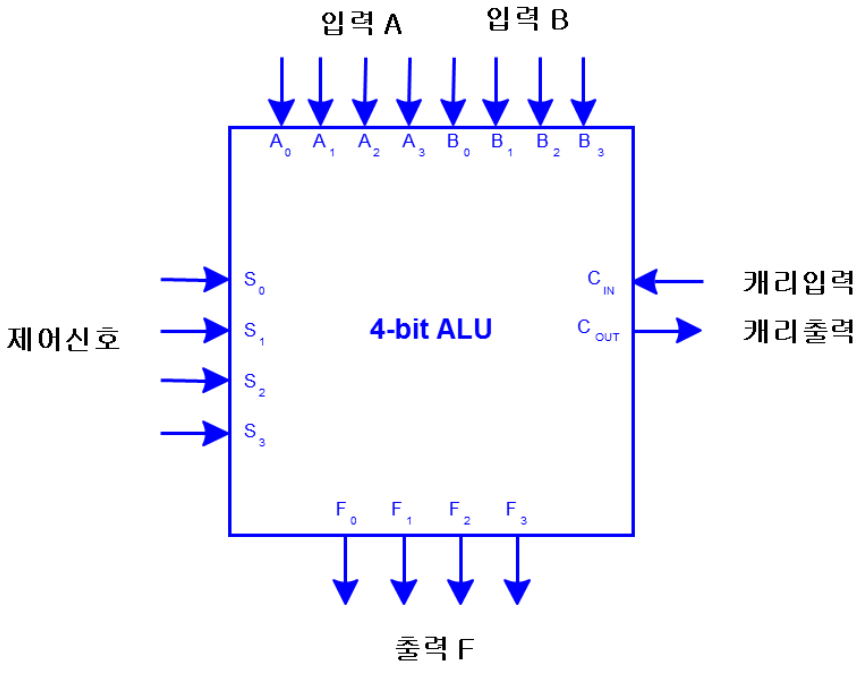

그림 5-4: 4 비트 ALU 의 구조

4 개의 제어신호에 따라 수행해야 할 연산을 나타내면 [표 8-2]와 같다.

[표 5-3]에서는 4 비트의 버스를 사용한다고 가정할 때 동일한 버스에서 제어신호를 전송하여 ALU 의 연산을 제어할 수 있도록 설계되었다. 또한 4 비트의 ALU 는 CIN 과 COUT 을 처리할 수 있기 때문에 8 비트, 16 비트, 32 비트 등으로 확장하여 사용할 수 있다.

표 5-3: 동작 제어 신호에 따른 연산

제어 신호				연산
S3	S2	S1	S0	
0	0	0	0	Logical 0
0	0	0	1	Logical 1
0	0	1	0	B
0	0	1	1	~B
0	1	0	0	A and B
0	1	0	1	A or B
0	1	1	0	A xor B
0	1	1	1	B Logical Shift Right
1	0	0	0	B Logical Shift Left
1	0	0	1	B Arithmetic Shift Right
1	0	1	0	A plus B
1	0	1	1	B increment
1	1	0	0	Reserved
1	1	0	1	Reserved
1	1	1	0	Reserved
1	1	1	1	Reserved

3. 예비보고서 작성

▣ 오버플로우 검사 조건이 왜 최상위 2 비트 올림수가 다를 때인지 조사하시오.

▣ ALU 의 연산 기능을 개선하기 위한, 추가적인 연산을 Reserved 자리에 구현해보시오.

▣ 아래 코드를 참조하여 ALU 를 Behavioral Model 로 구현하시오.

```
module ALU(en, ctrl_s, in_a, in_b, out_f);
  input       en;
  input [3:0] ctrl_s;
  input [3:0] in_a;
  input [3:0] in_b;

  output  [3:0] out_f;

  reg [3:0] reg_f;

  assign out_f = (en == 1)? reg_f: 4'bz;

  always @ (ctrl_s or in_a or in_b)
  begin
    case(ctrl_s)
      4'b0010:  reg_f = in_b;
      4'b1010:  reg_f = in_a + in_b;
      4'b1011:  reg_f = in_b + 4'b0001;
      default:  reg_f = 4'b0;
    endcase
  end
endmodule
```

▣ 논리 연산에 속하는 LSR, LSL, ASR 에 대해서 조사하시오. (Bitwise Instructions)

4. 실험에 필요한 기기

▣ 실험용 PC

▣ Quartus II-Modelsim 소프트웨어

5. 실험 내용과 과정

① Adder/Subtractor 의 Verilog HDL 코드 작성하여, 시뮬레이션을 진행한다. (반드시 ModelSim-Altera 툴을 사용하시오.) 그리고 표 5-4 에 해당되는 입력을 통해 올바른 결과가 나오는지 검증을 진행한다. [결과 보고서 1~2 번 수행에 해당 됨]

② 4-bit Adder/Subtractor 의 최대 지연 경로를 파악해본다. (4-bit Adder/Subtractor 에 대한 gate diagram 을 통해 유추가 가능할 것이다.) 그리고 이 최대 지연을 발생시키는 입력/입력 변화의 경우를 시뮬레이션을 통해 검증해본다. [결과 보고서 3 번 수행에 해당 됨]

③ 좀 더 빠른 클럭에 동기화하여 작동시킬 수 있는 Adder 를 어떻게 제작해야 할지 고민해본다. 그리고 이러한 고속 Adder 들에 대해서 조사를 진행한다. [결과 보고서 4 번 수행에 해당 됨]

④ 위에서 제시된 ALU 에 대한 Verilog 코드를 기반으로, 완성된 ALU 를 설계하여 본다. 그리고 여러 입력 값들을 통해 동작을 검증한다. [결과 보고서 5 번 항목 수행에 해당 됨]

6. 실험 결과 보고서

① 그림 5-2 에서 제시된 4-bit Adder/Subtractor 를 VerilogHDL 로 구현하시오.

② 예비 보고서에서 작성하였던 Adder/Subtractor 의 Verilog HDL 코드를 활용하여, 시뮬레이션을 진행하시오. (반드시, Modelsim-Altera 툴을 사용하고, Testbench 를 통해 Waveform 을 생성하시오.) 그리고 아래의 입력에 따른 결과를 진리표에 채워 넣으시오.

표 5-4. 4-bit Adder/Subtractor 의 진리표

입 력			출 력				
Ai	Bi	Add/Subtract	S3	S2	S1	S0	overflow
3	2	+					
1	-5	-					
-6	2	+					
5	4	+					
7	7	-					
-2	-6	+					
-4	-1	-					
-2	7	-					

③ 4-bit Adder/Subtractor 의 최대 지연 경로를 찾아서, 클럭에 의해 동작할 경우의 최대 주파수를 계산하시오.

④ 자리 올림 예견 법(Carry Look Ahead)으로 구현한 n-bit Adder 와, 이번 실험에서의 Ripple Carry 방식으로 구현된 n-bit Adder 를 비교하고 장단점을 논하시오.

⑤ 예비보고서 항목 작성을 통해 설계한 ALU 를 시뮬레이션을 통해 검증하시오.

⑥ 구현된 ALU 를 바탕으로, 곱셈과 나눗셈을 연산하는 과정에 대해서 생각해 보시오. (대략적인 동작 방식에 대한 다이어그램이나, 설명을 서술하시오.)

⑦ 4-bit X 4-bit 곱셈 연산을 수행하여 최대 8-bit 결과를 도출할 수 있는 곱셈 하드웨어 장치를 설계하고, 구조와 동작을 알아볼 수 있는 다이어그램으로 묘사해보시오. (Verilog HDL 로 구현할 필요는 없으며, 개인이 생각하는 설계방식을 충분히 잘 전달할 수 있는 그림과 글로 서술하면 됩니다.)