2018년 2학기 컴퓨터공학실험Ⅱ
CSE3016-05반 10주차 결과 보고서

학번: 20171665
이름: 이 선 호

2018. 11. 23

목 차

I 4 Bit Binary Parallel Adder and Subtractor

1. 2 to 4 Decoder (AND Gate)

1) Verilog 코딩

inv.v

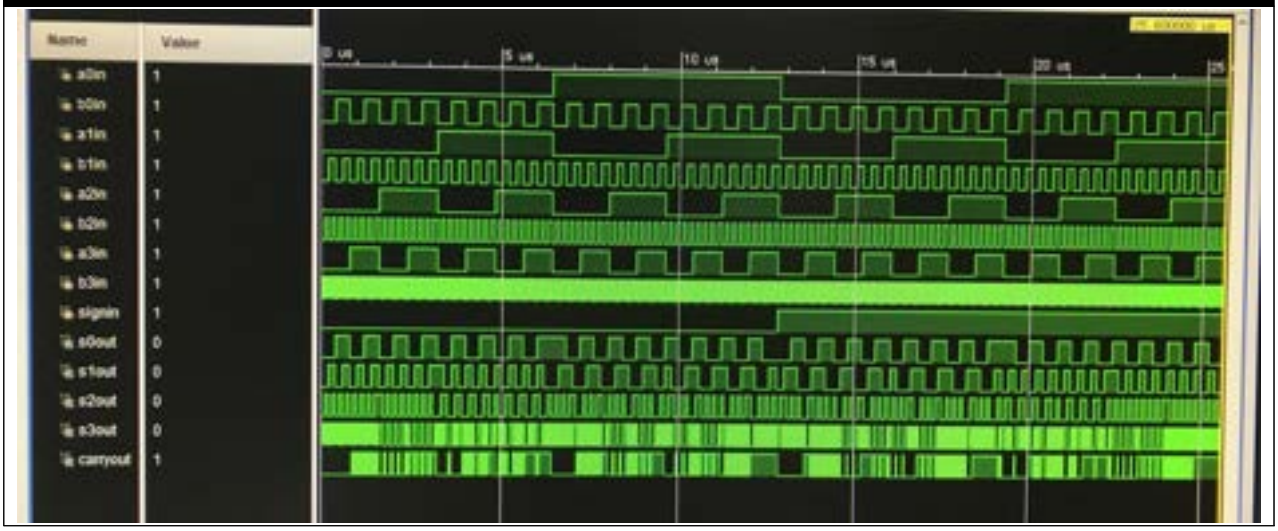
```
`timescale 1ns / 1ps

module inv(ina0, inb0, ina1, inb1, ina2, inb2, ina3, inb3, insign, outs0, outs1,
outs2, outs3, outcarry);

input ina0, inb0, ina1, inb1, ina2, inb2, ina3, inb3, insign;
output outs0, outs1, outs2, outs3, outcarry;
wire carry1, carry2, carry3;

assign carry1 = (ina0 && (inb0 && ~insign || ~inb0 && insign)) || insign &&
(~ina0 && (inb0 && ~insign || ~inb0 && insign) || ina0 && ~(inb0 && ~insign
|| ~inb0 && insign));
xor(outs0, ina0, (inb0 && ~insign || ~inb0 && insign), insign);

assign carry2 = (ina1 && (inb1 && ~insign || ~inb1 && insign)) || carry1 &&
(~ina1 && (inb1 && ~insign || ~inb1 && insign) || ina1 && ~(inb1 && ~insign |
|| ~inb1 && insign));
xor(outs1, ina1, (inb1 && ~insign || ~inb1 && insign), carry1);

assign carry3 = (ina2 && (inb2 && ~insign || ~inb2 && insign)) || carry2 &&
(~ina2 && (inb2 && ~insign || ~inb2 && insign) || ina2 && ~(inb2 && ~insign
|| ~inb2 && insign));
xor(outs2, ina2, (inb2 && ~insign || ~inb2 && insign), carry2);

assign outcarry = (ina3 && (inb3 && ~insign || ~inb3 && insign)) || carry3 &&
(~ina3 && (inb3 && ~insign || ~inb3 && insign) || ina3 && ~(inb3 && ~insign
|| ~inb3 && insign));
xor(outs3, ina3, (inb3 && ~insign || ~inb3 && insign), carry3);

endmodule
```

inv_tb.v

```
inv adder_subtractor(
.ina0(a0in),
.inb0(b0in),
.ina1(a1in),
.inb1(b1in),
.ina2(a2in),
.inb2(b2in),
.ina3(a3in),
.inb3(b3in),
.insign(signin),
.outs0(s0out),
.outs1(s1out),
.outs2(s2out),
.outs3(s3out),
.outcarry(carryout)
);

initial begin
a0in = 1'b0;
a1in = 1'b0;
a2in = 1'b0;
a3in = 1'b0;
b0in = 1'b0;
b1in = 1'b0;
b2in = 1'b0;
b3in = 1'b0;
signin = 1'b0;
end

always@(a0in or a1in or a2in or a3in or b0in or b1in or b2in or b3in or signin) begin
signin <=#12800 ~signin;
a0in <=#6400 ~a0in;
a1in <=#3200 ~a1in;
a2in <=#1600 ~a2in;
a3in <=#800 ~a3in;
b0in <=#400 ~b0in;
b1in <=#200 ~b1in;
b2in <=#100 ~b2in;
b3in <=#50 ~b3in;
end

initial begin
#25600;
$finish;
end

endmodule
```

3) Simulation 결과

Simulation



4) 과정

예비 보고서에서 4비트 이진 가산기는 4비트 이진 감산기와 원리가 유사함을 미리 조사했다. 그 이유는 어떤 이진수 A에서 이진수 B를 뺀다고 할 때, A−B는 A+(−B)와 같기 때문이다. 그래서 B의 보수를 그대로 B 대신 이진 가산기의 입력으로 넣으면 된다. 그런데 실습 시간에서 요구하는 사항은 가산기와 감산기가 합쳐진 가감산기를 구현해야 했기 때문에 A와 B 두 이진수를 논리 회로에 넣기 전에 수행 연산 종류에 따라서 입력을 다르게 넣어야 한다.

$$1 \oplus x = 1'x + 1x' = 0 + x' = x'_{(tractor)}$$
$$0 \oplus x = 0'x + 0x' = 1x + 0 = x\,(Adder)$$

그래서 새로운 입력을 만들어서 그 값을 0 또는 1로 정하여 B와 XOR 연산을 사전에 연산하는 과정을 구현했다. 감산기로 사용할 때는 1을, 가산기 때는 0을 입력하면 된다. 새로운 입력을 insign, 입력 B의 이진수에서 LSB를 inb0으로 시작해서 MSB까지 in3으로 정했으며, 이를 아래 코드로 작성했다.

```
inb0 && ~insign || ~inb0 && insign
… 다른 비트 자릿수에서도 유사 …
```

이후 과정은 4비트 병렬 가산기에서 전 가산기를 구현할 때와 유사하다. 앞선 6주차 실습과 결과보고서에서 전 가산기를 구현했던 내용을 참고하면 전 가산기는 두 개의 1비트와 한 개의 이전 carry를 입력으로 받고, 입력으로 두 개 비트의 합인 sum과 현재

carry를 내보낸다. i를 현재 비트의 자릿수, 두 개 비트를 각각 A_i, B_i, 이들의 합을 S_i, 이전 carry와 현재 carry를 각각 C_i, C_{i+1}라 하면 이들의 boolean expression 관계식은 다음과 같다.

$$C_{i+1} = A_iB_i + C_i(A_i \oplus B_i)$$
$$S_i = A_i \oplus B_i \oplus C_i$$

앞에서 B 대신 B를 insign과 XOR한 값을 입력으로 넣었기 때문에, 각 비트 자리의 carry를 carry1부터 carry3, 그리고 최종 carry를 outcarry라고 정하여 코드를 작성하면 아래와 같다.

```
assign carry2 = (ina1 && (inb1 && ~insign || ~inb1 && insign)) || carry1 && (~ina1 && (inb1 &&
~insign || ~inb1 && insign) || ina1 && ~(inb1 && ~insign || ~inb1 && insign));
… 다른 비트 자릿수에서도 유사 …
```

단, LSB 자리에서는 이전 carry가 존재하지 않으므로 이전 carry 대신 insign 입력 값으로 치환해서 사용했다. 뺄셈의 경우는 이전 carry가 1이 되어야 LSB 자리에서의 발림수가 생기기 때문에 정상적인 뺄셈 연산이 가능하므로 insign을 LSB 자리의 carry로 대신 입력하는 것이 바람직하다. 그리고 최종 carry인 outcarry외 다른 carry는 논리회로에서 굳이 출력할 필요가 없으므로 wire로 선언하여 각 자리의 전 가산기의 입력 또는 출력 사이에 연결했다.

현재 계산하는 비트 자리에서의 합을 구현하기 위해 위의 식을 참고하여 코드를 아래와 같이 구현했다.

```
xor(outs2, ina2, (inb2 && ~insign || ~inb2 && insign), carry2);
… 다른 비트 자릿수에서도 유사 …
```

Verilog에서 기본적으로 제공하는 xor 함수를 사용했고, 두 개 비트뿐만이 아니라 이전 자리에서의 carry도 같이 고려해야 하므로 이 세 개를 xor 함수의 입력으로 설정했다. 그리고 출력은 LSB 자리의 outs0부터 MSB 자리의 outs3까지 출력하도록 xor 함수의 가장 맨 앞의 parameter로 설정했다.

Simulation을 통해 가감산기가 정상적으로 구동되는지 확인하기 위해 각 입력들이 50ns, 100ns, 200ns, … , 12800ns 주기로 바뀌도록 always문을 사용해서 구현했다.

Ⅱ BCD Adder

1. BCD Adder

1) Verilog 코딩

inv.v

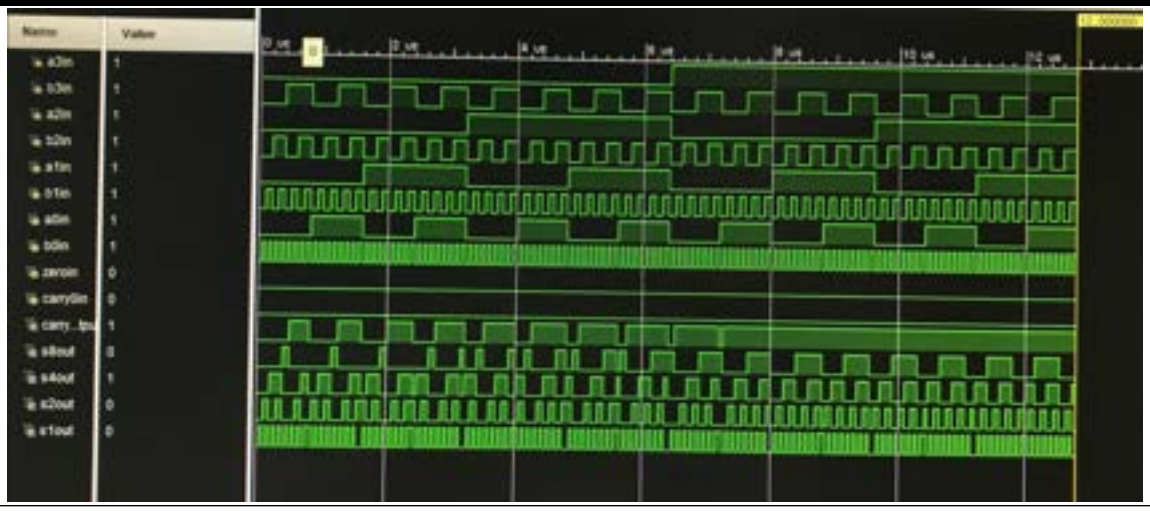
```
module inv(ina3, inb3, ina2, inb2, ina1, inb1, ina0, inb0, inzero, carry0,
outputcarry, outs8, outs4, outs2, outs1);

input ina3, inb3, ina2, inb2, ina1, inb1, ina0, inb0, inzero, carry0;
output outputcarry, outs8, outs4, outs2, outs1;

wire carry1, carry2, carry3, outcarry, outz1, outz2, outz4, outz8, carrys1,
carrys2, carrys4, temp1, temp2, temp3;

assign carry1 = (ina0 && inb0) || carry0 && (~ina0 && inb0 || ina0 && ~inb0);
xor(outz1, ina0, inb0, carry0);

assign carry2 = (ina1 && inb1) || carry1 && (~ina1 && inb1 || ina1 && ~inb1);
xor(outz2, ina1, inb1, carry1);

assign carry3 = (ina2 && inb2) || carry2 && (~ina2 && inb2 || ina2 && ~inb2);
xor(outz4, ina2, inb2, carry2);

assign outcarry = (ina3 && inb3) || carry3 && (~ina3 && inb3 || ina3 &&
~inb3);
xor(outz8, ina3, inb3, carry3);

assign temp1 = outz8 && outz4;
assign temp2 = outz8 && outz2;

assign temp3 = outcarry || temp1 || temp2;
assign outputcarry = temp3;

assign carrys1 = (outz1 && inzero) || inzero && (~outz1 && inzero || outz1 &&
~inzero);
xor(outs1, outz1, inzero, inzero);

assign carrys2 = (outz2 && temp3) || carrys1 && (~outz2 && temp3 || outz2
&& ~temp3);
xor(outs2, outz2, temp3, carrys1);

assign carrys4 = (outz4 && temp3) || carrys2 && (~outz4 && temp3 || outz4
&& ~temp3);
xor(outs4, outz4, temp3, carrys2);

xor(outs8, outz8, inzero, carrys4);

endmodule
```

inv_tb.v

```
`timescale 1ns / 1ps

module inv_tb;

reg a3in, b3in, a2in, b2in, a1in, b1in, a0in, b0in, zeroin, carry0in;
wire carryoutput, s8out, s4out, s2out, s1out;

inv bcd_adder(
.ina3(a3in),
.inb3(b3in),
.ina2(a2in),
.inb2(b2in),
.ina1(a1in),
.inb1(b1in),
.ina0(a0in),
.inb0(b0in),
.inzero(zeroin),
.carry0(carry0in),
.outs8(s8out),
.outs4(s4out),
.outs2(s2out),
.outs1(s1out),
.outputcarry(carryoutput)
);

initial begin
a0in = 1'b0;
a1in = 1'b0;
a2in = 1'b0;
a3in = 1'b0;
b0in = 1'b0;
b1in = 1'b0;
b2in = 1'b0;
b3in = 1'b0;
zeroin = 1'b0;
carry0in = 1'b0;
end

always@(a0in or a1in or a2in or a3in or b0in or b1in or b2in or b3in) begin
a3in <=#6400 ~a3in;
a2in <=#3200 ~a2in;
a1in <=#1600 ~a1in;
a0in <=#800 ~a0in;
b3in <=#400 ~b3in;
b2in <=#200 ~b2in;
b1in <=#100 ~b1in;
b0in <=#50 ~b0in;
end

initial begin
#12800;
$finish;
end

endmodule
```

3) Simulation 결과

Simulation



4) 과정

BCD 가산기는 각 자리의 비트 연산은 전 가산기와 유사하다. 그러나 두 이진수를 더했을 때 그 결과 값이 9를 초과한다면 그 값이 그대로 출력되는 것이 아니라 출력 carry를 1로 출력하고 십진수로 나타낸 값에서 10을 뺀 나머지를 다시 이진수로 출력해야 한다. 두 이진수의 합이 십진수로 나타냈을 때 값이 9를 초과하는지 확인하려면 MSB 자리 비트와 그 오른쪽 비트, 그리고 그의 또 오른쪽 비트를 확인하면 된다. 9를 이진수로 나타냈을 때 $1001_{(2)}$이므로 9를 초과하면 LSB의 왼쪽 자리 비트(z_2)부터 1이 출력되기 때문이다. 또한 두 이진수의 MSB 자리의 합을 계산하는 과정에서 carry가 발생하면 마찬가지로 9를 초과했다는 의미이므로 출력 carry를 1로 설정해야 한다. 각 자리 비트 합의 출력을 MSB부터 LSB까지를 차례대로 z_8, z_4, z_2, z_1로 고려하면 두 이진수의 합이 9를 초과하는지 알 수 있는 carry를 구하는 논리식은 $OutputCarry = z_8z_4 + z_8z_2$이다. temp1과 temp2, 그리고 temp3를 임의의 wire라 하고, outz8, outz4, outz2를 각각 z_8, z_4, z_2라고 하여 코드를 작성하면 아래와 같다.

```
assign temp1 = outz8 && outz4;
assign temp2 = outz8 && outz2;
assign temp3 = outcarry || temp1 || temp2;
assign outputcarry = temp3;
```

이 출력 carry가 1이 되면 앞에서 합한 이진수의 값을 6과 다시 더한 값이 최종 BCD

가산기의 출력 값이 되어야 하므로 출력 carry를 필요한 입력과 4비트 병렬 가산기에 연결해야 한다. 십진수 6을 이진수로 나타내면 $0110_{(2)}$이므로 출력 carry를 앞 과정에서의 출력 s_4, s_2와 각각 4비트 병렬 가산기의 입력으로 넣어주고, 나머지 출력인 z_8과 z_1를 0과 같은 자리 비트의 입력으로 넣어주면 최종 BCD 결과가 출력된다. 4비트 병렬 가산기 구현 과정에서의 코드 작성은 앞의 내용 'Ⅰ. 4 Bit Binary Parallel Adder'과 동일하다.

```
assign carrys2 = (outz1 && temp3) || carrys1 && (~outz2 && temp3 || outz2 && ~temp3);
… 다른 비트 자릿수에서도 유사 …
```

※temp3은 outputcarry 값, carrys1은 이전 carry를 의미함.
s0과 s8 자리에서는 temp3를 inzero(0)로 치환하고 이전 carry를 적절히 설정하면 위의 식의 형태와 동일함.

Simulation을 통해 가감산기가 정상적으로 구동되는지 확인하기 위해 각 입력들이 50ns, 100ns, 200ns, … , 6400ns 주기로 바뀌도록 always문을 사용해서 구현했다.

Ⅲ 논의

1. 결과 검토 및 논의사항

Simulation 결과를 확인했을 때 정상적으로 4비트 병렬 가감산기와 BCD 가산기를 구현했다는 사실을 알 수 있다. 4비트 병렬 가감산기 구현 실습에서 과연 가산기에서 일부 입력만 변화했을 때 똑같이 감산기를 구현할 수 있는지를 검토하기 위해 예비 보고서와 결과 보고서의 내용을 종합하여 앞에서 다루지 못한 내용을 정리하고자 한다.

$$C_{i+1} = A_iB_i + C_i(A_i \oplus B_i)\,(Adder)$$
$$C_{i+1} = A_iB_i' + C_i(A_i \oplus B_i)'\,(_{tractor})$$
$$S_i = A_i \oplus B_i \oplus C_i$$

덧셈과 뺄셈 연산에서 각각 C_i를 carry와 borrow(빌림수)로 고려했을 때 위와 같은 식이 성립한다는 사실을 할 수 있다. 감산기의 borrow 구하는 식의 형태를 변형하면 다음과 같다.

$$C_{i+1} = A_iB_i' + C_i(A_i \oplus B_i)'$$
$$= A_iB_i' + C_i(A_iB_i' + A_i'B_i)'$$
$$= A_iB_i' + C_i((A_i' + B_i)(A_i + B_i'))$$
$$= A_iB_i' + C_i(A_i'B_i' + A_iB_i)$$

도출된 식의 형태를 살펴보면 앞에서 가산기 식에서의 B를 B의 보수로 바꿔서 식을 작성했을 때와 형태가 동일함을 알 수 있다. 따라서 가감산기에서는 입력 B를 보수로 바꿔서 입력하면 가산기 대신 감산기를 수행한다는 사실을 알 수 있으며, 이는 이번 실습에서 XOR 논리 연산을 사용하여 B를 B의 보수로 바꿔서 4비트 병렬 가산기와 동일한 회로의 입력으로 넣는 것은 적절한 수행 과정임을 알 수 있다.

BCD 가산기에서는 0부터 9까지의 숫자만을 사용해야 하므로 두 이진수의 합이 9를 초과했을 때 6을 더하여 0부터 15까지 나타낼 수 있는 4비트의 이진수를 적절한 십진수 숫자로 바꿔야 한다. 이를 다시 생각해보면 만일 십진수로 나타냈을 때 십의 자리 수가 1이 되면 이를 제외하고 나머지 자리의 숫자를 출력하는 것과 동일하다. 따라서 두 이진수의 합이 9를 초과했을 때 발생하는 출력 carry를 이용하여($1010_{(2)}$부터 해당하는 이진수 결과 값이 나왔을 때) 6에 해당하는 이진수를 결과 값과 더해주는 작업을 진행하는 것은 적절한 수행 과정임을 다시 한 번 확인할 수 있다.

이번 실습을 통해 가산기를 사용하는 회로에서 과정에서의 필요에 따라 carry 등 출력이나 XOR와 같은 논리 연산 등을 사용하여 보상회로를 더 추가 구현하여 원하는 가산기 회로를 만들 수 있음을 알 수 있었다.

2. 추가 이론 조사 및 작성

여기서 더 나아가 4비트 이진 가감산기와 BCD 가산기의 내용을 종합하면 BCD 감산기도 충분히 구현할 수 있음을 예상할 수 있다. 가산기와 감산기의 형태가 입력만 다를 뿐 동일하고, BCD 감산기도 BCD 가산기와 마찬가지로 계산 원리만 익히면 가능하다. 예를 들어, 8 − 2 = 6에서 2를 9의 보수 표현으로 바꿔서 8 + 7 = 15로 변형한 다음에 앞에 발생한 carry를 5에 더하면 6이 출력됨을 알 수 있다. (9의 보수 표현으로 바꾸는 방법은 가장 아래 자리부터 현재 자리까지를 9로 갖는 수에서 바꾸고자 하는 수를 빼고 부호를 바꾸면 되며, 2의 보수 표현에서 다루는 방법과 유사하다.) 마찬가지로 28 − 13 = 15에서 13을 9의 보수 표현으로 바꿔서 28 + 86 = 114로 변형한 다음에 앞에 발생한 carry를 14에 더하면 15가 출력됨을 알 수 있다.

이를 논리회로로 구현하면, 우선 4비트 병렬 회로를 사용하여 9의 보수표현으로 바꾸고자 하는 수를 XOR 논리 게이트를 사용하여 보수로 바꾼 다음에 10의 값을 갖는 이

진수와 더하는 과정을 수행한다. 그리고 1을 더하는 과정을 마찬가지로 병렬 가산기를 사용하여 수행한 다음, 출력 carry가 1이면 이를 더하는 과정을, carry가 0이면 앞에서 시행했던 9의 보수 표현으로 바꾸는 과정을 다시 수행하면 원하는 결과가 나온다는 것을 알 수 있다. (출처: http://www.eeeguide.com/decimal-bcd-subtractor/)