Wencong Huang (Team Member A)

Jared Orange (Team Member B),

Marcos Buznego (Team Member C),

CS 2200: Theory of Computer Science

HW 2

September 20th 2018

Problem 1

And.cmp

a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

```
/**
 * And gate:
 * out = 1 if (a == 1 and b == 1)
 *       0 otherwise
 */

CHIP And {
    IN a, b;
    OUT out;

    PARTS:
    Nand(a=a,b=b,out=nout);
    Not(in=nout,out=out);
}
```

```
load And.hdl,
output-file And.out,
compare-to And.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

Not.cmp

in	out
0	1
1	0

```
/**
 * Not gate:
 * out = not in
 */

CHIP Not {
    IN in;
    OUT out;

    PARTS:
    // Put your code here:
    Nand(a=in, b=in, out=out);
}
```

```
load Not.hdl,
output-file Not.out,
compare-to Not.cmp,
output-list in%B3.1.3 out%B3.1.3;

set in 0,
eval,
output;

set in 1,
eval,
output;
```

Or.cmp

a	b	out
0	0	0
0	1	1
1	0	1
1	1	1

```
/**
 * Or gate:
 * out = 1 if (a == 1 or b == 1)
 *       0 otherwise
 */

CHIP Or {
    IN a, b;
    OUT out;

    PARTS:
    // Put your code here:
    Nand(a=a, b=a, out=notA);
    Nand(a=b, b=b, out=notB);
    Nand(a=notA, b=notB, out=out);
}
```

```
load Or.hdl,
output-file Or.out,
compare-to Or.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

Xor.cmp

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

```
/**
 * Exclusive-or gate:
 * out = not (a == b)
 */

CHIP Xor {
    IN a, b;
    OUT out;

    PARTS:
    // Put your code here:
    Not(in=a, out=aNot);
    Not(in=b, out=bNot);
    And(a=aNot, b=b, out=out1);
    And(a=a, b=bNot, out=out2);
    Or(a=out1, b=out2, out=out);
}
```

```
load Xor.hdl,
output-file Xor.out,
compare-to Xor.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

Mux.cmp

a	b	sel	out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

```
/**
 * Multiplexor:
 * out = a if sel == 0
 *       b otherwise
 */

CHIP Mux {
    IN a, b, sel;
    OUT out;

    PARTS:
    // Put your code here:
    Not(in=sel, out=nsel);
    And(a=sel, b=b, out=c1);
    And(a=nsel, b=a, out=c2);
    Or(a=c1, b=c2, out=out);
}
```

```
load Mux.hdl,
output-file Mux.out,
compare-to Mux.cmp,
output-list a%B3.1.3 b%B3.1.3 sel%B3.1.3 out%B3.1.3;

set a 0,
set b 0,
set sel 0,
eval,
output;

set sel 1,
eval,
output;

set a 0,
set b 1,
set sel 0,
eval,
output;

set sel 1,
eval,
output;

set a 1,
set b 0,
set sel 0,
eval,
output;

set sel 1,
eval,
```

```
output;

set a 1,
set b 1,
set sel 0,
eval,
output;

set sel 1,
eval,
output;
```

And16.cmp

a	b	out
0000000000000000	0000000000000000	0000000000000000
0000000000000000	1111111111111111	0000000000000000
1111111111111111	1111111111111111	1111111111111111
1010101010101010	0101010101010101	0000000000000000
0011110011000011	0000111111110000	0000110011000000
0001001000110100	1001100001110110	0001000000110100

```
/**
 * 16-bit bitwise And:
 * for i = 0..15: out[i] = (a[i] and b[i])
 */

CHIP And16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
    // Put your code here:
    And(a=a[0], b=b[0], out=out[0]);
    And(a=a[1], b=b[1], out=out[1]);
    And(a=a[2], b=b[2], out=out[2]);
    And(a=a[3], b=b[3], out=out[3]);
    And(a=a[4], b=b[4], out=out[4]);
    And(a=a[5], b=b[5], out=out[5]);
    And(a=a[6], b=b[6], out=out[6]);
    And(a=a[7], b=b[7], out=out[7]);
    And(a=a[8], b=b[8], out=out[8]);
    And(a=a[9], b=b[9], out=out[9]);
    And(a=a[10], b=b[10], out=out[10]);
    And(a=a[11], b=b[11], out=out[11]);
    And(a=a[12], b=b[12], out=out[12]);
    And(a=a[13], b=b[13], out=out[13]);
    And(a=a[14], b=b[14], out=out[14]);
    And(a=a[15], b=b[15], out=out[15]);
}
```

```
load And16.hdl,
output-file And16.out,
compare-to And16.cmp,
output-list a%B1.16.1 b%B1.16.1 out%B1.16.1;

set a %B0000000000000000,
set b %B0000000000000000,
eval,
output;

set a %B0000000000000000,
set b %B1111111111111111,
eval,
output;

set a %B1111111111111111,
set b %B1111111111111111,
eval,
output;

set a %B1010101010101010,
set b %B0101010101010101,
eval,
output;

set a %B0011110011000011,
set b %B0000111111110000,
eval,
output;

set a %B0001001000110100,
set b %B1001100001110110,
eval,
output;
```

DMux.cmp

in	sel	a	b
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

```
/**
 * Demultiplexor:
 * {a, b} = {in, 0} if sel == 0
 *          {0, in} if sel == 1
 */

CHIP DMux {
    IN in, sel;
    OUT a, b;

    PARTS:
    // Put your code here:
    Not(in=sel, out=notsel);
    And(a=in, b=sel, out=b);
    And(a=in, b=notsel, out=a);
}
```

```
load DMux.hdl,
output-file DMux.out,
compare-to DMux.cmp,
output-list in%B3.1.3 sel%B3.1.3 a%B3.1.3 b%B3.1.3;

set in 0,
set sel 0,
eval,
output;

set sel 1,
eval,
output;

set in 1,
set sel 0,
eval,
output;

set sel 1,
eval,
output;
```

Not16.cmp

```
|        in        |       out        |
| 0000000000000000 | 1111111111111111 |
| 1111111111111111 | 0000000000000000 |
| 1010101010101010 | 0101010101010101 |
| 0011110011000011 | 1100001100111100 |
| 0001001000110100 | 1110110111001011 |
```

```
/**
 * 16-bit Not:
 * for i=0..15: out[i] = not in[i]
 */

CHIP Not16 {
    IN in[16];
    OUT out[16];

    PARTS:
    Nand(a=in[0], b=in[0], out=out[0]);
    Nand(a=in[1], b=in[1], out=out[1]);
    Nand(a=in[2], b=in[2], out=out[2]);
    Nand(a=in[3], b=in[3], out=out[3]);
    Nand(a=in[4], b=in[4], out=out[4]);
    Nand(a=in[5], b=in[5], out=out[5]);
    Nand(a=in[6], b=in[6], out=out[6]);
    Nand(a=in[7], b=in[7], out=out[7]);
    Nand(a=in[8], b=in[8], out=out[8]);
    Nand(a=in[9], b=in[9], out=out[9]);
    Nand(a=in[10], b=in[10], out=out[10]);
    Nand(a=in[11], b=in[11], out=out[11]);
    Nand(a=in[12], b=in[12], out=out[12]);
    Nand(a=in[13], b=in[13], out=out[13]);
    Nand(a=in[14], b=in[14], out=out[14]);
    Nand(a=in[15], b=in[15], out=out[15]);

}
```

```
load Not16.hdl,
output-file Not16.out,
compare-to Not16.cmp,
output-list in%B1.16.1 out%B1.16.1;

set in %B0000000000000000,
eval,
output;

set in %B1111111111111111,
eval,
output;

set in %B1010101010101010,
eval,
output;

set in %B0011110011000011,
eval,
output;

set in %B0001001000110100,
eval,
output;
```

Or16.cmp

a	b	out
0000000000000000	0000000000000000	0000000000000000
0000000000000000	1111111111111111	1111111111111111
1111111111111111	1111111111111111	1111111111111111
1010101010101010	0101010101010101	1111111111111111
0011110011000011	0000111111110000	0011111111110011
0001001000110100	1001100001110110	1001101001110110

```
/**
 * 16-bit bitwise Or:
 * for i = 0..15 out[i] = (a[i] or b[i])
 */

CHIP Or16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
    // Put your code here:
	Or(a=a[0], b=b[0], out=out[0]);
	Or(a=a[1], b=b[1], out=out[1]);
	Or(a=a[2], b=b[2], out=out[2]);
	Or(a=a[3], b=b[3], out=out[3]);
	Or(a=a[4], b=b[4], out=out[4]);
	Or(a=a[5], b=b[5], out=out[5]);
	Or(a=a[6], b=b[6], out=out[6]);
	Or(a=a[7], b=b[7], out=out[7]);
	Or(a=a[8], b=b[8], out=out[8]);
	Or(a=a[9], b=b[9], out=out[9]);
	Or(a=a[10], b=b[10], out=out[10]);
	Or(a=a[11], b=b[11], out=out[11]);
	Or(a=a[12], b=b[12], out=out[12]);
	Or(a=a[13], b=b[13], out=out[13]);
	Or(a=a[14], b=b[14], out=out[14]);
	Or(a=a[15], b=b[15], out=out[15]);

}
```

```
load Or16.hdl,
output-file Or16.out,
compare-to Or16.cmp,
output-list a%B1.16.1 b%B1.16.1 out%B1.16.1;

set a %B0000000000000000,
set b %B0000000000000000,
eval,
output;

set a %B0000000000000000,
set b %B1111111111111111,
eval,
output;

set a %B1111111111111111,
set b %B1111111111111111,
eval,
output;

set a %B1010101010101010,
set b %B0101010101010101,
eval,
output;

set a %B0011110011000011,
set b %B0000111111110000,
eval,
output;

set a %B0001001000110100,
set b %B1001100001110110,
eval,
output;
```

Or8Way.cmp

```
| in | out |
| 00000000 | 0 |
| 11111111 | 1 |
| 00010000 | 1 |
| 00000001 | 1 |
| 00100110 | 1 |
```

```
/**
 * 8-way Or:
 * out = (in[0] or in[1] or ... or in[7])
 */

CHIP Or8Way {
    IN in[8];
    OUT out;

    PARTS:
    // Put your code here:
    Or(a=in[0], b=in[1], out=c1);
    Or(a=in[2], b=in[3], out=c2);
    Or(a=in[4], b=in[5], out=c3);
    Or(a=in[6], b=in[7], out=c4);
    Or(a=c1, b=c2, out=c5);
    Or(a=c3, b=c4, out=c6);
    Or(a=c5, b=c6, out=out);
}
```

```
load Or8Way.hdl,
output-file Or8Way.out,
compare-to Or8Way.cmp,
output-list in%B2.8.2 out%B2.1.2;

set in %B00000000,
eval,
output;

set in %B11111111,
eval,
output;

set in %B00010000,
eval,
output;

set in %B00000001,
eval,
output;

set in %B00100110,
eval,
output;
```

DMux4Way.cmp

in	sel	a	b	c	d
0	00	0	0	0	0
0	01	0	0	0	0
0	10	0	0	0	0
0	11	0	0	0	0
1	00	1	0	0	0
1	01	0	1	0	0
1	10	0	0	1	0
1	11	0	0	0	1

```
/**
 * 4-way demultiplexor:
 * {a, b, c, d} = {in, 0, 0, 0} if sel == 00
 *                {0, in, 0, 0} if sel == 01
 *                {0, 0, in, 0} if sel == 10
 *                {0, 0, 0, in} if sel == 11
 */

CHIP DMux4Way {
    IN in, sel[2];
    OUT a, b, c, d;

    PARTS:
    DMux(in=in, sel=sel[1], a=t1, b=t2);
    DMux(in=t1, sel=sel[0], a=a, b=b);
    DMux(in=t2, sel=sel[0], a=c, b=d);
}
```

```
load DMux4Way.hdl,
output-file DMux4Way.out,
compare-to DMux4Way.cmp,
output-list in%B2.1.2 sel%B2.2.2 a%B2.1.2 b%B2.1.2 c%B2.1.2 d%B2.1.2;

set in 0,
set sel %B00,
eval,
output;

set sel %B01,
eval,
output;

set sel %B10,
eval,
output;

set sel %B11,
eval,
output;

set in 1,
set sel %B00,
eval,
output;

set sel %B01,
eval,
output;

set sel %B10,
eval,
```

```
output;

set sel %B11,
eval,
output;
```

DMux8Way.cmp

in	sel	a	b	c	d	e	f	g	h
0	000	0	0	0	0	0	0	0	0
0	001	0	0	0	0	0	0	0	0
0	010	0	0	0	0	0	0	0	0
0	011	0	0	0	0	0	0	0	0
0	100	0	0	0	0	0	0	0	0
0	101	0	0	0	0	0	0	0	0
0	110	0	0	0	0	0	0	0	0
0	111	0	0	0	0	0	0	0	0
1	000	1	0	0	0	0	0	0	0
1	001	0	1	0	0	0	0	0	0
1	010	0	0	1	0	0	0	0	0
1	011	0	0	0	1	0	0	0	0
1	100	0	0	0	0	1	0	0	0
1	101	0	0	0	0	0	1	0	0
1	110	0	0	0	0	0	0	1	0
1	111	0	0	0	0	0	0	0	1

```
/**
 * 8-way demultiplexor:
 * {a, b, c, d, e, f, g, h} = {in, 0, 0, 0, 0, 0, 0, 0} if sel == 000
 *                            {0, in, 0, 0, 0, 0, 0, 0} if sel == 001
 *                            etc.
 *                            {0, 0, 0, 0, 0, 0, 0, in} if sel == 111
 */

CHIP DMux8Way {
    IN in, sel[3];
    OUT a, b, c, d, e, f, g, h;

    PARTS:
    // Put your code here:
    DMux(in=in, sel=sel[2], a=c1, b=c2);
    DMux4Way(in=c1, sel[0]=sel[0], sel[1]=sel[1], a=a, b=b, c=c, d=d);
    DMux4Way(in=c2, sel[0]=sel[0], sel[1]=sel[1], a=e, b=f, c=g, d=h);
}
```

```
load DMux8Way.hdl,
output-file DMux8Way.out,
compare-to DMux8Way.cmp,
output-list in%B2.1.2 sel%B2.3.2 a%B2.1.2 b%B2.1.2 c%B2.1.2 d%B2.1.2 e%B2.1.2 f%B2.1.2 g%B2.1.2 h%B2.1.2;

set in 0,
set sel %B000,
eval,
output;

set sel %B001,
eval,
output;

set sel %B010,
eval,
output;

set sel %B011,
eval,
output;

set sel %B100,
eval,
output;

set sel %B101,
eval,
output;

set sel %B110,
eval,
output;
```

```
39
40  set sel %B111,
41  eval,
42  output;
43
44  set in 1,
45  set sel %B000,
46  eval,
47  output;
48
49  set sel %B001,
50  eval,
51  output;
52
    set sel %B010,
    eval,
55  output;
56
57  set sel %B011,
58  eval,
59  output;
60
61  set sel %B100,
62  eval,
63  output;
64
65  set sel %B101,
66  eval,
67  output;
68
69  set sel %B110,
70  eval,
```

```
output;

set sel %B111,
eval,
output;

```

Mux16.cmp

a	b	sel	out
0000000000000000	0000000000000000	0	0000000000000000
0000000000000000	0000000000000000	1	0000000000000000
0000000000000000	0001001000110100	0	0000000000000000
0000000000000000	0001001000110100	1	0001001000110100
1001100001110110	0000000000000000	0	1001100001110110
1001100001110110	0000000000000000	1	0000000000000000
1010101010101010	0101010101010101	0	1010101010101010
1010101010101010	0101010101010101	1	0101010101010101

```
/**
 * 16-bit multiplexor:
 * for i = 0..15 out[i] = a[i] if sel == 0
 *                        b[i] if sel == 1
 */

CHIP Mux16 {
    IN a[16], b[16], sel;
    OUT out[16];

    PARTS:
    // Put your code here:
    Mux(a=a[0], b=b[0], sel=sel, out=out[0]);
    Mux(a=a[1], b=b[1], sel=sel, out=out[1]);
    Mux(a=a[2], b=b[2], sel=sel, out=out[2]);
    Mux(a=a[3], b=b[3], sel=sel, out=out[3]);
    Mux(a=a[4], b=b[4], sel=sel, out=out[4]);
    Mux(a=a[5], b=b[5], sel=sel, out=out[5]);
    Mux(a=a[6], b=b[6], sel=sel, out=out[6]);
    Mux(a=a[7], b=b[7], sel=sel, out=out[7]);
    Mux(a=a[8], b=b[8], sel=sel, out=out[8]);
    Mux(a=a[9], b=b[9], sel=sel, out=out[9]);
    Mux(a=a[10], b=b[10], sel=sel, out=out[10]);
    Mux(a=a[11], b=b[11], sel=sel, out=out[11]);
    Mux(a=a[12], b=b[12], sel=sel, out=out[12]);
    Mux(a=a[13], b=b[13], sel=sel, out=out[13]);
    Mux(a=a[14], b=b[14], sel=sel, out=out[14]);
    Mux(a=a[15], b=b[15], sel=sel, out=out[15]);
}
```

```
load Mux16.hdl,
output-file Mux16.out,
compare-to Mux16.cmp,
output-list a%B1.16.1 b%B1.16.1 sel%D2.1.2 out%B1.16.1;

set a 0,
set b 0,
set sel 0,
eval,
output;

set sel 1,
eval,
output;

set a %B0000000000000000,
set b %B0001001000110100,
set sel 0,
eval,
output;

set sel 1,
eval,
output;

set a %B1001100001110110,
set b %B0000000000000000,
set sel 0,
eval,
output;
```

```
set sel 1,
eval,
output;

set a %B1010101010101010,
set b %B0101010101010101,
set sel 0,
eval,
output;

set sel 1,
eval,
output;
```

Mux4Way16.cmp

a	b	c	d	sel	out
0000000000000000	0000000000000000	0000000000000000	0000000000000000	00	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000	01	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000	10	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000	11	0000000000000000
0001001000110100	1001110000111110	1010101010101010	0101010101010101	00	0001001000110100
0001001000110100	1001110000111110	1010101010101010	0101010101010101	01	1001110000111110
0001001000110100	1001110000111110	1010101010101010	0101010101010101	10	1010101010101010
0001001000110100	1001110000111110	1010101010101010	0101010101010101	11	0101010101010101

```
/**
 * 4-way 16-bit multiplexor:
 * out = a if sel == 00
 *       b if sel == 01
 *       c if sel == 10
 *       d if sel == 11
 */

CHIP Mux4Way16 {
    IN a[16], b[16], c[16], d[16], sel[2];
    OUT out[16];

    PARTS:
    Mux16(a=a, b=b, sel=sel[0], out=aORb);
    Mux16(a=c, b=d, sel=sel[0], out=cORd);
    Mux16(a=aORb, b=cORd, sel=sel[1], out=out);
}
```

```
load Mux4Way16.hdl,
output-file Mux4Way16.out,
compare-to Mux4Way16.cmp,
output-list a%B1.16.1 b%B1.16.1 c%B1.16.1 d%B1.16.1 sel%B2.2.2 out%B1.16.1;

set a 0,
set b 0,
set c 0,
set d 0,
set sel 0,
eval,
output;

set sel 1,
eval,
output;

set sel 2,
eval,
output;

set sel 3,
eval,
output;

set a %B0001001000110100,
set b %B1001100001110110,
set c %B1010101010101010,
set d %B0101010101010101,
```

```
set sel 0,
eval,
output;

set sel 1,
eval,
output;

set sel 2,
eval,
output;

set sel 3,
eval,
output;

```

Mux8Way16.cmp

a	b	c	d	e	f	g	h
0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0001001000110100	0010001101000101	0011010001010110	0100010101100111	0101011001111000	0110011110001001	0111100010011010	1000100110101011
0001001000110100	0010001101000101	0011010001010110	0100010101100111	0101011001111000	0110011110001001	0111100010011010	1000100110101011
0001001000110100	0010001101000101	0011010001010110	0100010101100111	0101011001111000	0110011110001001	0111100010011010	1000100110101011
0001001000110100	0010001101000101	0011010001010110	0100010101100111	0101011001111000	0110011110001001	0111100010011010	1000100110101011
0001001000110100	0010001101000101	0011010001010110	0100010101100111	0101011001111000	0110011110001001	0111100010011010	1000100110101011
0001001000110100	0010001101000101	0011010001010110	0100010101100111	0101011001111000	0110011110001001	0111100010011010	1000100110101011
0001001000110100	0010001101000101	0011010001010110	0100010101100111	0101011001111000	0110011110001001	0111100010011010	1000100110101011
0001001000110100	0010001101000101	0011010001010110	0100010101100111	0101011001111000	0110011110001001	0111100010011010	1000100110101011

```
/**
 * 8-way 16-bit multiplexor:
 * out = a if sel == 000
 *       b if sel == 001
 *       etc.
 *       h if sel == 111
 */

CHIP Mux8Way16 {
    IN a[16], b[16], c[16], d[16],
       e[16], f[16], g[16], h[16],
       sel[3];
    OUT out[16];

    PARTS:
    // Put your code here:
    Mux4Way16(a=a, b=b, c=c, d=d, sel=sel[0..1], out=abcORd);
    Mux4Way16(a=e, b=f, c=g, d=h, sel=sel[0..1], out=efgORh);
    Mux16(a=abcORd, b=efgORh,     sel=sel[2], out=out);
}
```

```
load Mux8Way16.hdl,
output-file Mux8Way16.out,
compare-to Mux8Way16.cmp,
output-list a%B1.16.1 b%B1.16.1 c%B1.16.1 d%B1.16.1 e%B1.16.1 f%B1.16.1 g%B1.16.1 h%B1.16.1 sel%B2.3.2 out%B1.16.1;

set a 0,
set b 0,
set c 0,
set d 0,
set e 0,
set f 0,
set g 0,
set h 0,
set sel 0,
eval,
output;

set sel 1,
eval,
output;

set sel 2,
eval,
output;

set sel 3,
eval,
output;

set sel 4,
eval,
output;
```

```
38
39  set sel 5,
40  eval,
41  output;
42
43  set sel 6,
44  eval,
45  output;
46
47  set sel 7,
48  eval,
49  output;
50
    set a %B0001001000110100,
    set b %B0010001101000101,
53  set c %B0011010001010110,
54  set d %B0100010101100111,
55  set e %B0101011001111000,
56  set f %B0110011110001001,
57  set g %B0111100010011010,
58  set h %B1000100110101011,
59  set sel 0,
60  eval,
61  output;
62
63  set sel 1,
64  eval,
65  output;
66
67  set sel 2,
68  eval,
69  output;
```

```
70
71  set sel 3,
72  eval,
73  output;
74
75  set sel 4,
76  eval,
77  output;
78
79  set sel 5,
80  eval,
81  output;
82
83  set sel 6,
84  eval,
85  output;
86
87  set sel 7,
88  eval,
89  output;
```

Problem 2

HalfAdder.cmp

a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

```
/**
 * Computes the sum of two bits.
 */

CHIP HalfAdder {
    IN a, b;    // 1-bit inputs
    OUT sum,    // Right bit of a + b
        carry;  // Left bit of a + b

    PARTS:
    // Put you code here:
    Xor(a=a, b=b, out=sum);
    And(a=a, b=b, out=carry);
}
```

```
load HalfAdder.hdl,
output-file HalfAdder.out,
compare-to HalfAdder.cmp,
output-list a%B3.1.3 b%B3.1.3 sum%B3.1.3 carry%B3.1.3;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

FullAdder.cmp

a	b	c	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

```
/**
 * Computes the sum of three bits.
 */

CHIP FullAdder {
    IN a, b, c;  // 1-bit inputs
    OUT sum,     // Right bit of a + b + c
        carry;   // Left bit of a + b + c

    PARTS:
    // Put you code here:
    HalfAdder(a=a, b=b, sum=w1, carry=c1);
    HalfAdder(a=w1, b=c, sum=sum, carry=c2);
    Or(a=c1, b=c2, out=carry);
}
```

```
load FullAdder.hdl,
output-file FullAdder.out,
compare-to FullAdder.cmp,
output-list a%B3.1.3 b%B3.1.3 c%B3.1.3 sum%B3.1.3 carry%B3.1.3;

set a 0,
set b 0,
set c 0,
eval,
output;

set c 1,
eval,
output;

set b 1,
set c 0,
eval,
output;

set c 1,
eval,
output;

set a 1,
set b 0,
set c 0,
eval,
output;

set c 1,
eval,
```

```
output;
set b 1,
set c 0,
eval,
output;
set c 1,
eval,
output;
```

Add16.cmp

a	b	out
0000000000000000	0000000000000000	0000000000000000
0000000000000000	1111111111111111	1111111111111111
1111111111111111	1111111111111111	1111111111111110
1010101010101010	0101010101010101	1111111111111111
0011110011000011	0000111111110000	0100110010110011
0001001000110100	1001100001110110	1010101010101010

```
/**
 * Adds two 16-bit values.
 * The most significant carry bit is ignored.
 */

CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
   // Put you code here:
   FullAdder(a=false, b=a[0],  c=b[0],  sum=out[0],  carry=c0);
   FullAdder(a=c0,    b=a[1],  c=b[1],  sum=out[1],  carry=c1);
   FullAdder(a=c1,    b=a[2],  c=b[2],  sum=out[2],  carry=c2);
   FullAdder(a=c2,    b=a[3],  c=b[3],  sum=out[3],  carry=c3);
   FullAdder(a=c3,    b=a[4],  c=b[4],  sum=out[4],  carry=c4);
   FullAdder(a=c4,    b=a[5],  c=b[5],  sum=out[5],  carry=c5);
   FullAdder(a=c5,    b=a[6],  c=b[6],  sum=out[6],  carry=c6);
   FullAdder(a=c6,    b=a[7],  c=b[7],  sum=out[7],  carry=c7);
   FullAdder(a=c7,    b=a[8],  c=b[8],  sum=out[8],  carry=c8);
   FullAdder(a=c8,    b=a[9],  c=b[9],  sum=out[9],  carry=c9);
   FullAdder(a=c9,    b=a[10], c=b[10], sum=out[10], carry=c10);
   FullAdder(a=c10,   b=a[11], c=b[11], sum=out[11], carry=c11);
   FullAdder(a=c11,   b=a[12], c=b[12], sum=out[12], carry=c12);
   FullAdder(a=c12,   b=a[13], c=b[13], sum=out[13], carry=c13);
   FullAdder(a=c13,   b=a[14], c=b[14], sum=out[14], carry=c14);
   FullAdder(a=c14,   b=a[15], c=b[15], sum=out[15], carry=c15);
}
```

```
load Add16.hdl,
output-file Add16.out,
compare-to Add16.cmp,
output-list a%B1.16.1 b%B1.16.1 out%B1.16.1;

set a %B0000000000000000,
set b %B0000000000000000,
eval,
output;

set a %B0000000000000000,
set b %B1111111111111111,
eval,
output;

set a %B1111111111111111,
set b %B1111111111111111,
eval,
output;

set a %B1010101010101010,
set b %B0101010101010101,
eval,
output;

set a %B0011110011000011,
set b %B0000111111110000,
eval,
output;

set a %B0001001000110100,
set b %B1001100001110110,
```

```
eval,
output;
```

Inc16.cmp

in	out
0000000000000000	0000000000000001
1111111111111111	0000000000000000
0000000000000101	0000000000000110
1111111111111011	1111111111111100

```
/**
 * 16-bit incrementer:
 * out = in + 1 (arithmetic addition)
 */

CHIP Inc16 {
    IN in[16];
    OUT out[16];

    PARTS:
   // Put you code here:
   Add16(a[0..15]=in[0..15], b[0]=true, b[1..15]=false, out[0..15]=out[0..15]);
}
```

```
load Inc16.hdl,
output-file Inc16.out,
compare-to Inc16.cmp,
output-list in%B1.16.1 out%B1.16.1;

set in %B0000000000000000,  // in = 0
eval,
output;

set in %B1111111111111111,  // in = -1
eval,
output;

set in %B0000000000000101,  // in = 5
eval,
output;

set in %B1111111111111011,  // in = -5
eval,
output;
```

Problem 3

ALU.cmp

x	y	zx	nx	zy	ny	f	no	out	zr	ng
0000000000000000	1111111111111111	1	0	1	0	1	0	0000000000000000	1	0
0000000000000000	1111111111111111	1	1	1	1	1	1	0000000000000001	0	0
0000000000000000	1111111111111111	1	1	1	0	1	0	1111111111111111	0	1
0000000000000000	1111111111111111	0	0	1	1	0	0	0000000000000000	1	0
0000000000000000	1111111111111111	1	1	0	0	0	0	1111111111111111	0	1
0000000000000000	1111111111111111	0	0	1	1	0	1	1111111111111111	0	1
0000000000000000	1111111111111111	1	1	0	0	0	1	0000000000000000	1	0
0000000000000000	1111111111111111	0	0	1	1	1	1	0000000000000000	1	0
0000000000000000	1111111111111111	1	1	0	0	1	1	0000000000000001	0	0
0000000000000000	1111111111111111	0	1	1	1	1	1	0000000000000001	0	0
0000000000000000	1111111111111111	1	1	0	1	1	1	0000000000000000	1	0
0000000000000000	1111111111111111	0	0	1	1	1	0	1111111111111111	0	1
0000000000000000	1111111111111111	1	1	0	0	1	0	1111111111111110	0	1
0000000000000000	1111111111111111	0	0	0	0	1	0	1111111111111111	0	1
0000000000000000	1111111111111111	0	1	0	0	1	1	0000000000000001	0	0
0000000000000000	1111111111111111	0	0	0	1	1	1	1111111111111111	0	1
0000000000000000	1111111111111111	0	0	0	0	0	0	0000000000000000	1	0
0000000000000000	1111111111111111	0	1	0	1	0	1	1111111111111111	0	1
0000000000010001	0000000000000011	1	0	1	0	1	0	0000000000000000	1	0
0000000000010001	0000000000000011	1	1	1	1	1	1	0000000000000001	0	0
0000000000010001	0000000000000011	1	1	1	0	1	0	1111111111111111	0	1
0000000000010001	0000000000000011	0	0	1	1	0	0	0000000000010001	0	0
0000000000010001	0000000000000011	1	1	0	0	0	0	0000000000000011	0	0
0000000000010001	0000000000000011	0	0	1	1	0	1	1111111111101110	0	1
0000000000010001	0000000000000011	1	1	0	0	0	1	1111111111111100	0	1
0000000000010001	0000000000000011	0	0	1	1	1	1	1111111111101111	0	1
0000000000010001	0000000000000011	1	1	0	0	1	1	1111111111111101	0	1
0000000000010001	0000000000000011	0	1	1	1	1	1	0000000000010010	0	0
0000000000010001	0000000000000011	1	1	0	1	1	1	0000000000000100	0	0
0000000000010001	0000000000000011	0	0	1	1	1	0	0000000000010000	0	0
0000000000010001	0000000000000011	1	1	0	0	1	0	0000000000000010	0	0
0000000000010001	0000000000000011	0	0	0	0	1	0	0000000000010100	0	0
0000000000010001	0000000000000011	0	1	0	0	1	1	0000000000001110	0	0
0000000000010001	0000000000000011	0	0	0	1	1	1	1111111111110010	0	1
0000000000010001	0000000000000011	0	0	0	0	0	0	0000000000000001	0	0
0000000000010001	0000000000000011	0	1	0	1	0	1	0000000000010011	0	0

```
CHIP ALU {
    IN
        x[16], y[16],  // 16-bit inputs
        zx, // zero the x input?
        nx, // negate the x input?
        zy, // zero the y input?
        ny, // negate the y input?
        f,  // compute out = x + y (if 1) or x & y (if 0)
        no; // negate the out output?

    OUT
        out[16], // 16-bit output
        zr, // 1 if (out == 0), 0 otherwise
        ng; // 1 if (out < 0),  0 otherwise

    PARTS:
   // Put you code here:
   Mux16(a=x, b=false, sel=zx, out=zeroedX);
   Not16(in=zeroedX, out=notX);
   Mux16(a=zeroedX, b=notX, sel=nx, out=negX);
   Mux16(a=y, b=false, sel=zy, out=zeroedY);
   Not16(in=zeroedY, out=notY);
   Mux16(a=zeroedY, b=notY, sel=ny, out=negY);

   Add16(a=negX, b=negY, out=xPlusY);
   And16(a=negX, b=negY, out=xAndY);
   Mux16(a=xAndY, b=xPlusY, sel=f, out=funcVar);
   Not16(in=funcVar, out=negFuncVar);
   Mux16(a=funcVar, b=negFuncVar, sel=no, out=out, out[0..7]=lowOut, out[8..15]=highOut, out[15]=ng);

   Or8Way(in=lowOut, out=zero1);
   Or8Way(in=highOut, out=zero2);
   Or(a=zero1, b=zero2, out=notZr);
   Not(in=notZr, out=zr);
}
```

```
load ALU.hdl,
output-file ALU2.out,
compare-to ALU.cmp,
output-list x%B1.16.1 y%B1.16.1 zx%B1.1.1 nx%B1.1.1 zy%B1.1.1
            ny%B1.1.1 f%B1.1.1 no%B1.1.1 out%B1.16.1 zr%B1.1.1
            ng%B1.1.1;
set x %B0000000000000000,  // x = 0
set y %B1111111111111111;  // y = -1
// Test0,
set zx 1,
set nx 0,
set zy 1,
set ny 0,
set f 1,
set no 0,
eval,
output;

// Test0,
set zx 1,
set nx 1,
set zy 1,
set ny 1,
set f 1,
set no 1,
eval,
output;
```

Here are just a couple of tests for the ALU, there are more tests in the ALU.tst file.

a. less than 1296.

b. 2 tests with 36 cases total

c. Unlike the full ALU.cmp, the zr and ng status outputs are ignored. This is only a partial test, so the testing and implementation of the gates used the construct the ALU will be the only things needed. The nostat.tst helps concentrate on getting the correct ALU computation.

d. 1296 and greater.

e. 2 tests with 36 tests total

f. To thoroughly test our ALU, we needed to include tests for the gates used to construct our ALU. We tested our 16 bit AND gate, 16 bit NOT gate, 16 bit Multiplexer, an 8 bit way OR and a regular NOT gate. We had 6 inputs and 3 outs, but if you take the powers of two, it is actually a total of 64 and 8.

Also, ALU NoStat test needed to be included because any comparison failures during the ALU.tst will be caused by errors in the handling of the ignored status outputs.