Adders, Subtractors, Comparators

Instructor: Dr. Vinicius Prado da Fonseca (vpradodafons@online.mun.ca)

Adding One Bit To Another, A Half-Adder

- All the possible additions of two 1-bit numbers expressed in base 2:

  0 + 0 = 00 (0)

  0 + 1 = 01 (1)

  1 + 0 = 01 (1)

  1 + 1 = 10 (2)

- The left most bit in the above binary example is the carry bit.
- The right most bit is the sum bit

Adding One Bit To Another, A Half-Adder

- A table showing the addition of a and b.
- c is the carry out, and s is the sum.
- The carry output is given by

$$a \,\&\, b$$

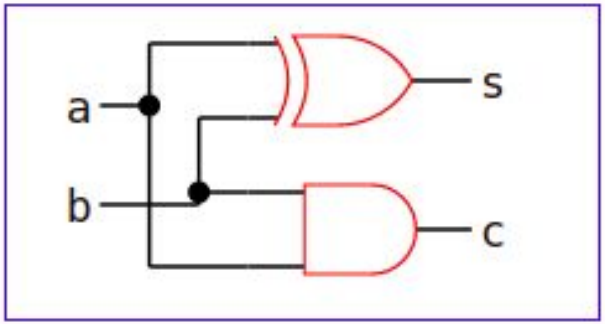
- The sum output is given by

$$a \,\&\, b' + a' \,\&\, b$$

  - equal to

$$a \oplus b$$

SOP	a	b	c	s
$\overline{a} \cdot \overline{b}$	0	0	0	0
$\overline{a} \cdot b$	0	1	0	1
$a \cdot \overline{b}$	1	0	0	1
$a \cdot b$	1	1	1	0

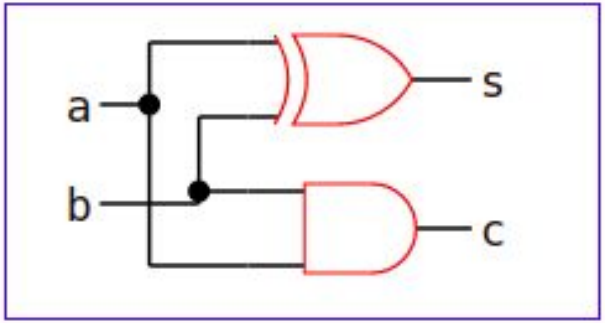


Adding One Bit To Another, A Half-Adder

- model half adder using gates

```
module ha_gates( output logic c, s, input logic a, b);
     and  carry(c, a, b);
     xor  sum(s, a, b);
endmodule
```

SOP	a	b	c	s
$\overline{a} \cdot \overline{b}$	0	0	0	0
$\overline{a} \cdot b$	0	1	0	1
$a \cdot \overline{b}$	1	0	0	1
$a \cdot b$	1	1	1	0

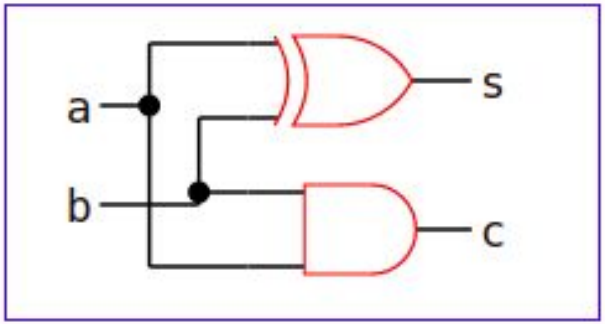


Adding One Bit To Another, A Half-Adder

- model half adder using Boolean algebra

```
module ha_boolean( output logic c, s,  input logic a, b);
      assign      c = a & b;
      assign      s = a ^ b;
endmodule
```

SOP	a	b	c	s
$\overline{a} \cdot \overline{b}$	0	0	0	0
$\overline{a} \cdot b$	0	1	0	1
$a \cdot \overline{b}$	1	0	0	1
$a \cdot b$	1	1	1	0



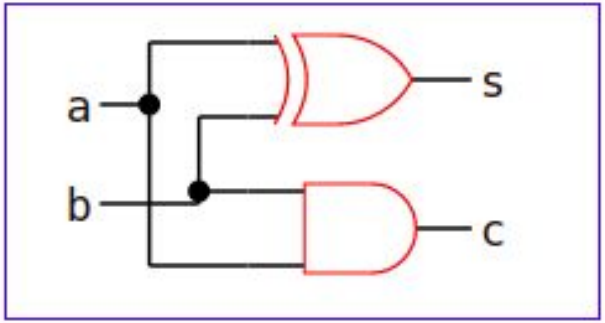
Adding One Bit To Another, A Half-Adder

- Model half adder using Verilog's “+” operator

```
module ha_functional( output logic c,s, input logic a,b);
      assign {c, s} = a + b;
endmodule
```

SOP	a	b	c	s
$\overline{a} \cdot \overline{b}$	0	0	0	0
$\overline{a} \cdot b$	0	1	0	1
$a \cdot \overline{b}$	1	0	0	1
$a \cdot b$	1	1	1	0

- The concatenation {c,s} is a 2-bit number



Adding 4-bit Binary Numbers

```
  1 1
  0 1 1 0 (6)
+ 0 0 1 1 (3)
  ---------
  1 0 0 1 (9)
```

Adding 4-bit Binary Numbers

```
  1 1
  0 1 1 0 (6)
+ 0 0 1 1 (3)
  ---------
  1 0 0 1 (9)
```

Adding 4-bit Binary Numbers

```
  1 1
  0 1 1 0 (6)
+ 0 0 1 1 (3)
  ---------
  1 0 0 1 (9)
```

Adding 4-bit Binary Numbers

```
  1 1
  0 1 1 0 (6)
+ 0 0 1 1 (3)
  ----------
  1 0 0 1 (9)
```

Adding 4-bit Binary Numbers

```
  1 1 1
  0 1 1 1 (7)
+ 0 0 1 1 (3)
  ----------
  1 0 1 0 (10)
```

Adding 4-bit Binary Numbers

```
  1 1 1
  0 1 1 1 (7)
+ 0 0 1 1 (3)
  ----------
  1 0 1 0 (10)
```

Adding 4-bit Binary Numbers

```
  1 1 1
  0 1 1 1 (7)
+ 0 0 1 1 (3)
  ----------
  1 0 1 0 (10)
```

Adding 4-bit Binary Numbers

```
  1 1 1
  0 1 1 1 (7)
+ 0 0 1 1 (3)
  ----------
  1 0 1 0 (10)
```

Full Adder

- Three bits are being added for each of these digits
  - a, b, cin
- For a n-bit number the adder must be able to handle a carry in

SOP	a	b	cin	c	s
$\overline{a} \cdot \overline{b} \cdot \overline{cin}$	0	0	0	0	0
$\overline{a} \cdot \overline{b} \cdot cin$	0	0	1	0	1
$\overline{a} \cdot b \cdot \overline{cin}$	0	1	0	0	1
$\overline{a} \cdot b \cdot cin$	0	1	1	1	0
$a \cdot \overline{b} \cdot \overline{cin}$	1	0	0	0	1
$a \cdot \overline{b} \cdot cin$	1	0	1	1	0
$a \cdot b \cdot \overline{cin}$	1	1	0	1	0
$a \cdot b \cdot cin$	1	1	1	1	1

Full Adder

SOP	a	b	cin	c	s
$\overline{a} \cdot \overline{b} \cdot \overline{cin}$	0	0	0	0	0
$\overline{a} \cdot \overline{b} \cdot cin$	0	0	1	0	1
$\overline{a} \cdot b \cdot \overline{cin}$	0	1	0	0	1
$\overline{a} \cdot b \cdot cin$	0	1	1	1	0
$a \cdot \overline{b} \cdot \overline{cin}$	1	0	0	0	1
$a \cdot \overline{b} \cdot cin$	1	0	1	1	0
$a \cdot b \cdot \overline{cin}$	1	1	0	1	0
$a \cdot b \cdot cin$	1	1	1	1	1

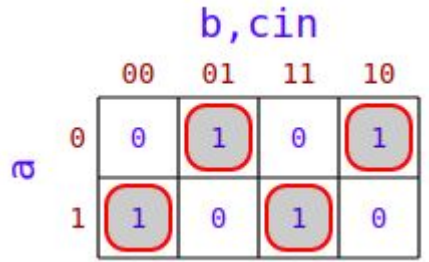
The k-map for the s, sum output is:

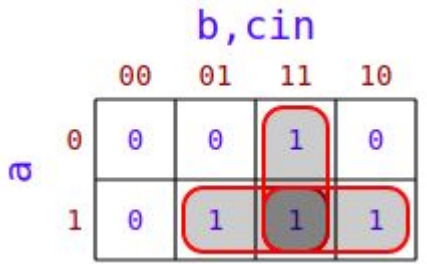
three input XOR

$s = a \oplus b \oplus cin$

The k-map for the c, carry output is:

$c = (a \cdot cin) + (b \cdot cin) + (a \cdot b)$





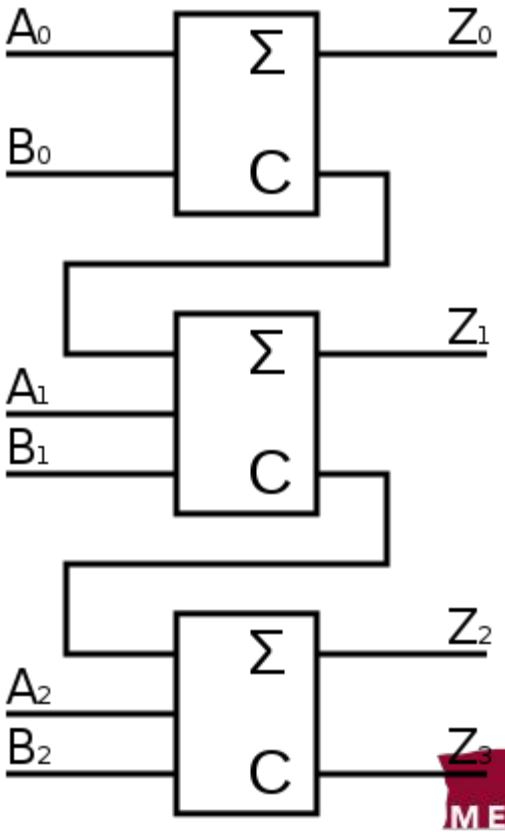
Full Adder

```
module fa_gates( output logic c, s, input logic a, b, cin);
        ha_gates a1(c1, s0, a, b);
        ha_gates a2(c2, s, s0, cin);
        or (c, c1, c2);

module fa_boolean( output logic c,s, input logic a,b, cin);
        assign c = a & b | a & cin | b & cin;
        assign s = a ^ b ^ cin;

module fa_functional( output logic c,s, input logic a,b,cin);
        assign {c,s} = a + b + cin;
```

Adding 3-bit Numbers

- A 3-bit adder
  - one half adder
  - two full adders
  - first bit doesn't need cin
- Connected in a chain
- The carry out of the previous adder is connected to the carry in of the next adder.
- More bits, more full-adders
- Parameterized modules

```
module add #(parameter N=4)
```



Signed Addition

- All 3-bit two's complement numbers in decimal are:

100 = -4, 101 = -3, 110 = -2, 111 = -1, 000 = 0, 001 = 1, 010 = 2, 011 = 3

- Two's complement binary numbers can be added as if they were unsigned

1 + 2 = 001 + 010 = ?

-2 + 1 = 110 + 001 = ?

-1 + 1 = 111 + 001 = ?

6 - 2 = 0110 - 1110 = ?

Signed Addition

- All 3-bit two's complement numbers in decimal are:

100 = -4, 101 = -3, 110 = -2, 111 = -1, 000 = 0, 001 = 1, 010 = 2, 011 = 3

- Two's complement binary numbers can be added as if they were unsigned

1 + 2 = 001 + 010 = 011 = 3_{10}

-2 + 1 = 110 + 001 = 111 = -1_{10}

-1 + 1 = 111 + 001 = 1 000 = 0 // correct, the 4th bit is ignored (carry)

6 - 2 = 0110 + 1110 = 1 0100 = 4 // correct, the 5th bit is ignored (carry)

Signed Addition

- It is possible to add two number where the result will overflow
  - (MSBa == MSBb) && (MSBout != MSBa)
- Overflows when addition two positives yield a negative:
  2 + 2 = 010 + 010 = 100 = 4_{10} (?)
  (0 == 0) && (1 != 0) // Overflow
- Overflows when addition two negatives yield a positive:
  -3 + -2 = 101 + 110 = 1 011 = 3_{10} // the 4th bit is ignored

  (1 == 1) && (0 != 1) // Overflow
- Will require a circuit to detect these situations

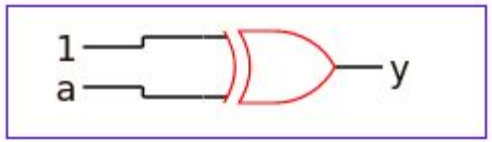
Building A Subtractor

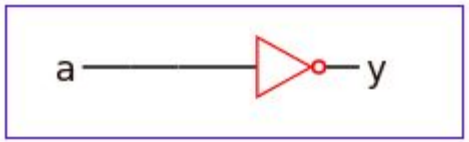
- Complementing the subtrahend and then doing the addition.
- Use an adder with a two's complement circuit
- A two's complement can be performed by
  - one's complement
  - followed by addition by 1
- A one's complement is done by inverting all the bits with NOT gates.

```
// least significant bit with 1 on carry in
not (invert_b[0], b[0]);
fa bits0 ( carries[0], diff[0], a[0], invert_b[0],1'b1 );
```

Building An Adder/Subtractor

- Using a flag (subtract) to indicate when we want to do each operation
- XOR will invert the b operand if flag "subtract" is 1






```
xor neg ( invert_b[i], b[i], subtract);

fa bits ( carries[i+1], result[i], a[i], invert_b[i], carries[i] );
```

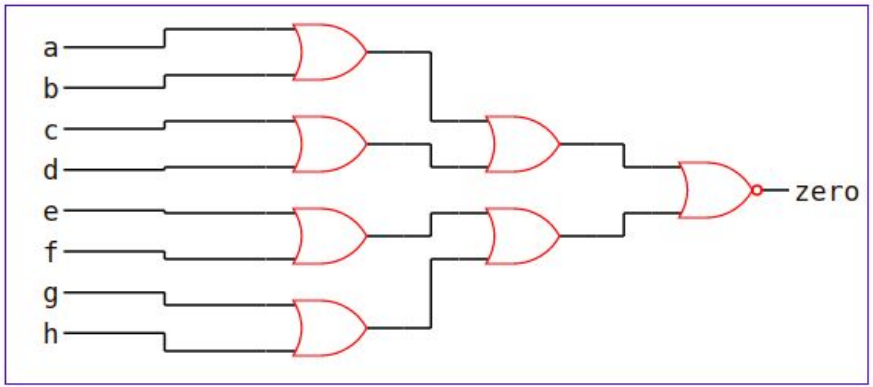
- If subtract is 1, b will be two-complement, subtraction

```
fa bits0 ( carries[0], diff[0], a[0], invert_b[0],subtract);
```

- Otherwise it proceed with a regular addition

Zero Detector

- Another common logic/arithmetic operation
- Usually outputs is high if the input is all zeros
- Given an 8-bit binary number, a zero can be detected with:



- Verilog example in the notes

Equality Checking

- Checking to see of two numbers are the same is another common operation
- This check can be done with a set of XOR gates, one for each bit, and a zero detector

0 xor 0 = 0

1 xor 1 = 0

- Detect zeros for all bits = both operands are the same
- Verilog example in the notes

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

Less/Greater Than Comparison

- Compare number for less than, greater than, or equal
- Hardware that subtracts two binary numbers can be used to test which number is smaller/larger
  - If the result of A−B is negative then A is smaller B.
  - If A−B is positive, then A is larger than B.
  - If A−B is zero, then A is equal to B
- Verilog example in the notes

Next Steps

- Run the examples in the online notes
- Memory in verilog
- Design of a mini datapath