Introduction to Computer Systems

Lecture #2

Signed Fixed Point Numbers

Lecture#2 Agenda - 2’s Comp. Numbers

- **Addition & subtraction**
  - Adding 2 unsigned binary numbers
  - Subtraction of binary numbers
- **2’s complement integers**
  - Intuition
  - Mathematic representation
  - Examples
  - Negation
  - Relation to Unsigned Numbers
  - Sign extension
- **Overflow**
  - Unsigned addition & subtraction
  - 2’s Complement addition
- **Additional related topics**
  - Long addition

Addition of 2 binary numbers

Addition of 2 binary numbers

Let us try to add two 6 digits binary numbers: X=[001010] and Y=[001100]. As in adding decimal numbers we perform the operation a digit by digit starting from the LSB. Let us write the two numbers and try to add them:

```
                                        01000
 [001010]                              [001010]
+[001100]            =>               +[001100]
 --------------                        --------------
 [  _?110]                             [010110]
```

We have no problem adding X_0 and Y_0: 0+0=0. We have no problem adding X_1 and Y_1 or X_2 and Y_2 since 0+1=1+0=1. We do have a problem adding X_3 and Y_3 since 1+1=2 and we do not have 2 in our alphabet. What do we do in a similar case in adding decimal numbers? When we add 8 and 7 we get 15. We write the digit 5, which is the excess value above 10, in the proper location and "remember" to add 1 when we get to the addition of the next digit in the numbers. That 1 is called the Carry. In binary numbers we do exactly the same. Here everything up to 2 has no carry. From 2 and above we'll write the excess above 2 in the current digit of the result and add an extra 1 during the addition of the next digit.

Addition of 2 binary numbers

WHY IS THIS CORRECT?
Let us try to add two n bits binary numbers: [X] and [Y]:

$$
\begin{array}{cccl}
 & [X] & = & [X_{n-1}\ X_{n-2} \dots X_{K+2}\ X_{K+1}\ \ X_K\ \ X_{K-1}\ X_{K-2} \dots X_1\ X_0\,] \\
+ & & & + \\
 & [Y] & = & [Y_{n-1}\ Y_{n-2} \dots Y_{K+2}\ \ Y_{K+1}\ \ \ Y_K\ \ Y_{K-1}\ Y_{K-2} \dots Y_1\ Y_0\ ] \\
 & \text{--------} & & \text{---} \\
 & [Z] & = & [Z_{n-1}\ Z_{n-2} \dots Z_{K+2}\ \ Z_{K+1}\ \ \ Z_K\ \ Z_{K-1}\ Z_{K-2} \dots Z_1\ Z_0\,]
\end{array}
$$

Say that for i=0 to K-1 we had $X_i+Y_i<2$, then:

$$Z = \sum_{i=0}^{n-1} Z_i \cdot 2^i = \sum_{i=0}^{n-1} X_i \cdot 2^i + \sum_{i=0}^{n-1} Y_i \cdot 2^i =$$

$$= \sum_{i=0}^{K-1} (X_i+Y_i) \cdot 2^i + (X_K+Y_K) \cdot 2^K + (X_{K+1}+Y_{K+1}) \cdot 2^{K+1} + \sum_{i=K+2}^{n-1} (X_i+Y_i) \cdot 2^i$$

It is clear that for i=0-(K-1) we have $Z_i=X_i+Y_i$.

But what happens to the powers of 2^K and 2^{K+1} if $X_K=1$ and also $Y_k=1$?

Addition of 2 binary numbers

WHY IS THIS CORRECT?

Let us try to add two n bits binary numbers: [X] and [Y]:

$$
\begin{array}{rcl}
[X] & = & [X_{n-1}\ X_{n-2}\ \dots\ X_{K+2}\ X_{K+1}\ X_K\ X_{K-1}\ X_{K-2}\ \dots\ X_1\ X_0] \\
+ & & + \\
[Y] & = & [Y_{n-1}\ Y_{n-2}\ \dots\ Y_{K+2}\ Y_{K+1}\ Y_K\ Y_{K-1}\ Y_{K-2}\ \dots\ Y_1\ Y_0] \\
\hline
[Z] & = & [Z_{n-1}\ Z_{n-2}\ \dots\ Z_{K+2}\ Z_{K+1}\ Z_K\ Z_{K-1}\ Z_{K-2}\ \dots\ Z_1\ Z_0]
\end{array}
$$

Say that for i=0 to K-1 we had $X_i+Y_i<2$, then:

$$Z = \sum_{i=0}^{n-1} Z_i \cdot 2^i = \sum_{i=0}^{n-1} X_i \cdot 2^i + \sum_{i=0}^{n-1} Y_i \cdot 2^i =$$

$$= \boxed{\sum_{i=0}^{K-1}(X_i+Y_i)\cdot 2^i} + \boxed{(X_K+Y_K)\cdot 2^K} + \boxed{(X_{K+1}+Y_{K+1})\cdot 2^{K+1}} + \boxed{\sum_{i=K+2}^{n-1}(X_i+Y_i)\cdot 2^i}$$

It is clear that for i=0-(K-1) we have $\boxed{Z_i=X_i+Y_i}$.

But what happens to the powers of 2^K and 2^{K+1} if $X_K=1$ and also $Y_k=1$?

Addition of 2 binary numbers

WHY IS THIS CORRECT?

Let us try to add two n bits binary numbers: [X] and [Y]:

$$\begin{array}{rcl} [X] & = & [X_{n-1}\ X_{n-2}\ \dots\ X_{K+2}\ \boxed{X_{K+1}}\ \boxed{X_K}\ X_{K-1}\ X_{K-2}\ \dots\ X_1\ X_0\,] \\ + & & + \\ [Y] & = & [Y_{n-1}\ Y_{n-2}\ \dots\ Y_{K+2}\ \boxed{Y_{K+1}}\ \boxed{Y_K}\ Y_{K-1}\ Y_{K-2}\ \dots\ Y_1\ Y_0\,] \\ \text{--------} & & \text{--} \\ [Z] & = & [Z_{n-1}\ Z_{n-2}\ \dots\ Z_{K+2}\ Z_{K+1}\ Z_K\ Z_{K-1}\ Z_{K-2}\ \dots\ Z_1\ Z_0\,] \end{array}$$

Say that for i=0 to K-1 we had $X_i+Y_i<2$, then:

$$(1+1)\cdot 2^K = 2\cdot 2^K = 2^{K+1}$$
$$= 0\cdot 2^K + 1\cdot 2^{K+1}$$

$$Z = \sum_{i=0}^{n-1} Z_i \cdot 2^i = \sum_{i=0}^{n-1} X_i \cdot 2^i + \sum_{i=0}^{n-1} Y_i \cdot 2^i =$$

$$= \sum_{i=0}^{K-1} (X_i+Y_i)\cdot 2^i + \boxed{(1+1)\cdot 2^K} + \boxed{(X_{K+1}+Y_{K+1})\cdot 2^{K+1}} + \sum_{i=K+2}^{n-1} (X_i+Y_i)\cdot 2^i$$

It is clear that for i=0-(K-1) we have $Z_i=X_i+Y_i$.

But what happens to the powers of 2^K and 2^{K+1} if $X_K=1$ and also $Y_k=1$?

Addition of 2 binary numbers

WHY IS THIS CORRECT?
Let us try to add two n bits binary numbers: [X] and [Y]:

$$
\begin{array}{rcl}
[X] & = & [X_{n-1}\ X_{n-2}\ \dots\ X_{K+2}\ \boxed{X_{K+1}}\ \boxed{X_K}\ X_{K-1}\ X_{K-2}\ \dots\ X_1\ X_0\ ] \\
+ & & + \\
[Y] & = & [Y_{n-1}\ Y_{n-2}\ \dots\ Y_{K+2}\ \boxed{Y_{K+1}}\ \boxed{Y_K}\ Y_{K-1}\ Y_{K-2}\ \dots\ Y_1\ Y_0\ ] \\
\text{--------} & & \text{---} \\
[Z] & = & [Z_{n-1}\ Z_{n-2}\ \dots\ Z_{K+2}\ Z_{K+1}\ Z_K\ Z_{K-1}\ Z_{K-2}\ \dots\ Z_1\ Z_0\ ]
\end{array}
$$

Say that for i=0 to K-1 we had $X_i+Y_i<2$, then:

$$(1+1)\cdot 2^K = 2\cdot 2^K = 2^{K+1}$$
$$= 0\cdot 2^K + 1\cdot 2^{K+1}$$

$$Z = \sum_{i=0}^{n-1} Z_i \cdot 2^i = \sum_{i=0}^{n-1} X_i \cdot 2^i + \sum_{i=0}^{n-1} Y_i \cdot 2^i =$$

$$= \sum_{i=0}^{K-1}(X_i+Y_i)\cdot 2^i + \boxed{0\cdot 2^K} + \boxed{(X_{K+1}+Y_{K+1}+1)\cdot 2^{K+1}} + \sum_{i=K+2}^{n-1}(X_i+Y_i)\cdot 2^i$$

It is clear that for i=0-(K-1) we have $Z_i=X_i+Y_i$.

But what happens to the powers of 2^K and 2^{K+1} if $X_K=1$ and also $Y_k=1$?

Addition of 2 binary numbers

What happens if $X_K=1$ and also $Y_k=1$?

Let's look at 2^K and 2^{K+1}:

$$(X_K+Y_K)\cdot 2^K + (X_{K+1}+Y_{K+1})\cdot 2^{K+1} = (1+1)\cdot 2^K + (X_{K+1}+Y_{K+1})\cdot 2^{K+1} = 2\cdot 2^K + (X_{K+1}+Y_{K+1})\cdot 2^{K+1} =$$

$$= 1\cdot 2^{K+1} + (X_{K+1}+Y_{K+1})\cdot 2^{K+1} = 0\cdot 2^K + (X_{K+1}+Y_{K+1}+1)\cdot 2^{K+1}$$

We call the 0 at the K-th position the result and the 1 that is added to the K+1 position – the carry.

We conclude that the addition of the K-th bits can be written as resulting with a 2-bit number:

$[C_{K+1}, Z_k] = X_K+Y_K$ indeed the carry bit weighs twice as much (2^{K+1}) than the result bit (2^K).

We should also take into account the case in which we did have carry from the (K-1) position. Thus a more accurate representation of addition is:

$\boxed{[C_{K+1}, Z_k] = X_K+Y_K+C_K}$ We see that we cover all possible values 0 to 3 (0=0+0+0, 3=1+1+1)

We will use this to build Unsigned adders later in the course.

Subtraction of binary numbers

Subtraction of binary numbers

Let us try to add two 6 digits binary numbers: X=[001010] and Y=[001100]. As in adding decimal numbers we perform the operation a digit by digit starting from the LSB. Let us write the two numbers and try to add them:

[010011]
- [001010]

[__?001]

We have no problem subtracting Y_0 from X_0: 1-0=1. We have no problem subtracting Y_1 from X_1 or Y_1 from X_2 since 1-1=0 and also 0-0=0. We do have a problem subtracting Y_3 from X_3 since 0-1=(-1) and we do not have (-1) in our alphabet. What do we do in a similar case in adding decimal numbers? When we add calculate 3 - 8 we "borrow" 1 from the next digit and calculate 13-8=5. We write the digit 5 as the result and subtract the borrowed 1 from the next digit. That 1 is called the Borrow. In binary numbers we do exactly the same.

[010011]
- [001010]
- [010000]

[001001]

Subtraction of binary numbers

Let us try to add two n bits binary numbers: [X] and [Y]:

$$
\begin{array}{cccl}
 & [X] & = & [X_{n-1}\ X_{n-2} \dots X_{K+2}\ X_{K+1}\ \ X_K\ \ X_{K-1}\ X_{K-2} \dots X_1\ X_0\,] \\
- & & - & \\
 & [Y] & = & [Y_{n-1}\ Y_{n-2} \dots Y_{K+2}\ \ Y_{K+1}\ \ \ Y_K\ \ Y_{K-1}\ Y_{K-2} \dots Y_1\ Y_0\ ] \\
 & \text{--------} & & \text{--} \\
 & [Z] & = & [Z_{n-1}\ Z_{n-2} \dots Z_{K+2}\ \ Z_{K+1}\ \ \ Z_K\ \ Z_{K-1}\ Z_{K-2} \dots Z_1\ Z_0\,]
\end{array}
$$

Say that for i=0 to K-1 we had X_i-Y_i≥0, then:

$$Z = \sum_{i=0}^{n-1} Z_i \cdot 2^i = \sum_{i=0}^{n-1} X_i \cdot 2^i - \sum_{i=0}^{n-1} Y_i \cdot 2^i =$$

$$= \sum_{i=0}^{K-1}(X_i - Y_i) \cdot 2^i + (X_K - Y_K) \cdot 2^K + (X_{K+1} - Y_{K+1}) \cdot 2^{K+1} + \sum_{i=K+2}^{n-1}(X_i - Y_i) \cdot 2^i$$

It is clear that for i=0-(K-1) we have Z_i=X_i-Y_i.

But what happens to the powers of 2^K and 2^{K+1} if X_K=0 and Y_k=1?

Subtraction of binary numbers

Let us try to add two n bits binary numbers: [X] and [Y]:

$$\begin{array}{rcl} [X] & = & [X_{n-1}\ X_{n-2}\ \ldots\ X_{K+2}\ X_{K+1}\ X_K\ X_{K-1}\ X_{K-2}\ \ldots\ X_1\ X_0] \\ - & & - \\ [Y] & = & [Y_{n-1}\ Y_{n-2}\ \ldots\ Y_{K+2}\ Y_{K+1}\ Y_K\ Y_{K-1}\ Y_{K-2}\ \ldots\ Y_1\ Y_0] \\ \text{--------} & & \text{---} \\ [Z] & = & [Z_{n-1}\ Z_{n-2}\ \ldots\ Z_{K+2}\ Z_{K+1}\ Z_K\ Z_{K-1}\ Z_{K-2}\ \ldots\ Z_1\ Z_0] \end{array}$$

Say that for i=0 to K-1 we had X_i-Y_i≥0, then:

$$Z = \sum_{i=0}^{n-1} Z_i \cdot 2^i = \sum_{i=0}^{n-1} X_i \cdot 2^i - \sum_{i=0}^{n-1} Y_i \cdot 2^i =$$

$$= \sum_{i=0}^{K-1}(X_i - Y_i) \cdot 2^i + (X_K - Y_K) \cdot 2^K + (X_{K+1} - Y_{K+1}) \cdot 2^{K+1} + \sum_{i=K+2}^{n-1}(X_i - Y_i) \cdot 2^i$$

It is clear that for i=0-(K-1) we have $Z_i=X_i-Y_i$.

But what happens to the powers of 2^K and 2^{K+1} if $X_K=0$ and $Y_k=1$?

Subtraction of binary numbers

Let us try to add two n bits binary numbers: [X] and [Y]:

$$
\begin{array}{rcl}
[X] & = & [X_{n-1}\ X_{n-2}\ \dots\ X_{K+2}\ \boxed{X_{K+1}}\ \boxed{X_K}\ X_{K-1}\ X_{K-2}\ \dots\ X_1\ X_0\ ] \\
- & & - \\
[Y] & = & [Y_{n-1}\ Y_{n-2}\ \dots\ Y_{K+2}\ \boxed{Y_{K+1}}\ \boxed{Y_K}\ Y_{K-1}\ Y_{K-2}\ \dots\ Y_1\ Y_0\ ] \\
\text{--------} & & \text{--} \\
[Z] & = & [Z_{n-1}\ Z_{n-2}\ \dots\ Z_{K+2}\ Z_{K+1}\ Z_K\ Z_{K-1}\ Z_{K-2}\ \dots\ Z_1\ Z_0\ ]
\end{array}
$$

Say that for i=0 to K-1 we had X_i-Y_i≥0, then:

$$(0-1)\cdot 2^K = (1-2)\cdot 2^K =$$
$$= 1\cdot 2^K + (-1)\cdot 2^{K+1}$$

$$Z = \sum_{i=0}^{n-1} Z_i \cdot 2^i = \sum_{i=0}^{n-1} X_i \cdot 2^i - \sum_{i=0}^{n-1} Y_i \cdot 2^i =$$

$$= \sum_{i=0}^{K-1}(X_i - Y_i)\cdot 2^i + \boxed{(0-1)\cdot 2^K} + \boxed{(X_{K+1} - Y_{K+1})\cdot 2^{K+1}} + \sum_{i=K+2}^{n-1}(X_i - Y_i)\cdot 2^i$$

It is clear that for i=0-(K-1) we have $Z_i=X_i-Y_i$.

But what happens to the powers of 2^K and 2^{K+1} if $X_K=0$ and $Y_k=1$?

Subtraction of binary numbers

Let us try to add two n bits binary numbers: [X] and [Y]:

$$
\begin{array}{rcl}
 & [X] = & [X_{n-1}\ X_{n-2} \ldots X_{K+2}\ \boxed{X_{K+1}}\ \boxed{X_K}\ X_{K-1}\ X_{K-2} \ldots X_1\ X_0] \\
- & & - \\
 & [Y] = & [Y_{n-1}\ Y_{n-2} \ldots Y_{K+2}\ \boxed{Y_{K+1}}\ \boxed{Y_K}\ Y_{K-1}\ Y_{K-2} \ldots Y_1\ Y_0] \\
 & \text{--------} & \text{---} \\
 & [Z] = & [Z_{n-1}\ Z_{n-2} \ldots Z_{K+2}\ Z_{K+1}\ Z_K\ Z_{K-1}\ Z_{K-2} \ldots Z_1\ Z_0]
\end{array}
$$

Say that for i=0 to K-1 we had X_i-Y_i≥0, then:

$$(0-1)\cdot 2^K = (1-2)\cdot 2^K =$$
$$= 1\cdot 2^K + (-1)\cdot 2^{K+1}$$

$$Z = \sum_{i=0}^{n-1} Z_i \cdot 2^i = \sum_{i=0}^{n-1} X_i \cdot 2^i - \sum_{i=0}^{n-1} Y_i \cdot 2^i =$$

$$= \sum_{i=0}^{K-1}(X_i - Y_i)\cdot 2^i + \boxed{1 \cdot 2^K} + \boxed{(X_{K+1} - Y_{K+1} - 1)\cdot 2^{K+1}} + \sum_{i=K+2}^{n-1}(X_i - Y_i)\cdot 2^i$$

It is clear that for i=0-(K-1) we have $Z_i=X_i-Y_i$.

But what happens to the powers of 2^K and 2^{K+1} if $X_K=0$ and $Y_k=1$?

Subtraction of binary numbers

$$\begin{array}{l} \quad -2 \quad 1 \\ \langle B_{K+1}, Z_k \rangle = X_K - Y_K - B \\ \langle 0, 1 \rangle = 1 \\ \langle 0, 0 \rangle = 0 \\ \langle 1, 1 \rangle = -1 \\ \langle 1, 0 \rangle = -2 \end{array}$$

What happens if $X_K=0$ and also $Y_k=1$?

Let's look at 2^K and 2^{K+1}:

$$(X_K - Y_K) \cdot 2^K + (X_{K+1} - Y_{K+1}) \cdot 2^{K+1} = (0-1) \cdot 2^K + (X_{K+1} - Y_{K+1}) \cdot 2^{K+1} = (-1) \cdot 2^K + (X_{K+1} - Y_{K+1}) \cdot 2^{K+1} =$$

$$= 1 \cdot 2^K + (-1) \cdot 2^{K+1} + (X_{K+1} - Y_{K+1}) \cdot 2^{K+1} = 1 \cdot 2^K + (X_{K+1} - Y_{K+1} - 1) \cdot 2^{K+1}$$

We call the 1 at the K-th position the result and the 1 that is subtracted from the to the (K+1)-th position – the borrow.

We conclude that subtraction of the K-th bits can be written as resulting with a 2-bit number:

$\langle B_{K+1}, Z_k \rangle = X_K - Y_K$ where the borrow bit weighs (-2) and the bit weighs 1.

We should also take into account the case in which we did have borrow from the (K-1) position. Thus a more accurate representation of subtraction is:

$\boxed{\langle B_{K+1}, Z_k \rangle = X_K - Y_K - B_K}$ We see that we cover all possible values of -2 to 1 (-2=0-1-1, 1=1-0-0)

Signed numbers

This is a good point to discuss negative numbers

We will try to get some intuition on negative numbers , then turn to mathematic explanation

Negative numbers – Intuition1

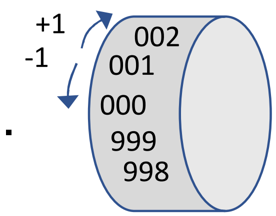
- Let's try to find (-1) by subtracting 1 from 0. We will use 6-bit numbers and we disregard the Carry from the MSB – i.e., we look only at 6 bits:

```
  [000000]
- [000001]
 ~~1~~11111
  =======
  [111111]
```

- Does [111111] equal (-1)? Surprisingly, it is. Let's check on 20+(-1) :

```
  [010100]   = 20
+ [111111]   =(-1)
  =======
  [010011]   = 19
```

Negative numbers – Intuition2



- In a car's odometer we add 1 to the display when we advance 1 Km.

  It goes from 000 to 001, then 002, etc. till 999.

  (then it become 000, 001 ... again)

- Say we have a special kind of odometer that subtracts 1 when we reverse 1 Km. In this case when we go 1 Km back from 003 we get to 002. Then to 001.

  Then to 000 and if we go 1 Km back once more we get to 999.

- Is 999 the same as (-1)? Let's check that on 430+999. Again, we only look at the 3 digits numbers

430	It works. Also 430+998=430+(-2). And so on.
+ 999	This is so since 999=1000-1 and we ignore the 1000
===	since we only look at the 3 right-hand side digits.
429	(This is 10's complement)

Negative numbers – Intuition2 (cont.)

- The equivalence of $[99\ldots99]_{10}$ in base 2 is $[111\ldots11]_2$
  Numbers start with [00…00] climb to [00…01], then [00..10], etc. till [11…11].
- If we add a binary point then [1.111…11] is almost 2 and if we consider its value as [10.000…00] - [0.000..01] we see that it is 2's complement

  i.e., in 2's complement [1.111…11] actually means (-[0.000..01])
- We would like now to formalize this

2's complement numbers – Mathematic representation

ההסבר המתמטי של כיצד מייצגים מספרים בשיטת המשלים ל-2

- We represent 2's complement numbers with triangular brackets <X>

- $<X> = <X_{n-1}, X_{n-2}, \ldots X_2, X_1, X_0>$ $(X_i \in \{0,1\})$

- The value of <X> is given by the formula:

$$X = X_{n-1}\cdot(-2^{n-1}) + X_{n-2}\cdot 2^{n-2} + \ldots + X_2\cdot 2^2 + X_1\cdot 2^1 + X_0\cdot 2^0$$

or

$$X = X_{n-1}\cdot(-2^{n-1}) + \sum_{i=0}^{n-2} X_i \cdot 2^i$$

Thus the weight of X_{n-1} is (-2^{n-1}) instead of 2^{n-1} in Unsigned numbers

Range of 2's comp. integers

- The range that can be represented by n bits is (-2^{n-1}) - $(2^{n-1}-1)$

- We have 2^n possible digit combinations

  From <0,0,…,0> till <1,1,…,1>.

- Half of them are negative!

  This is so since <0,1,1,…,1,1> equals $(2^{n-1}-1)$ and the MSB weighs (-2^{n-1}), which has a higher absolute value. Thus if the MSB is 1, the number is negative.

- The MSB is therefore called the Sign Bit.

<0,0,0,…,0,0> to <0,1,1,…,1,1> = 0 to $(2^{n-1}-1)$ = non-negative = positives + zero

<1,0,0,…,0,0> to <1,1,1,…,1,1> = (-2^{n-1}) to (-1) = negatives

8-bit 2's comp. numbers

$(-2^{n-1}) - (2^{n-1}-1)$

$(-2^{7}) - (2^{7}-1)$

-128 - 127

Negative numbers

<10000000>= (-128)

<10000001>= (-128)+1 = (-127)

<10000010>= (-128)+2 = (-126)

…

<11111110>= (-128)+126 = (-1)-1 = (-2)

<11111111>= (-128)+127 = (-1)

<00000000>=0

<00000001>=1

<00000010>=2

…

<01111111>=127

The MSB, bit 7, weighs (-128).

It is the sign bit. If it is 1 the number is negative.

The LSB, bit 0, still determines whether the number is even (if 0) or odd (if 1).

From the 256 possible bit combinations, 128 are negative, 127 positive and 1 combination for zero.

Range of numbers is -128 to 127

BTW, we could also define 2's comp. as: If the number is above 127 then its' value is given by its' Unsigned value-256. This is equivalent to our definition of the MSB as having weight of (-2^{n-1}).

Negating a 2’s comp. number

הפיכת סימן של מספר המיוצג בשיטת המשלים ל-2

Negating a 2's complement number

  <0011000>
+ <1100111>
=========
  <1111111>

- $< X > = < X_{n-1}, X_{n-2}, \dots, X_2, X_1, X_0 >$
- Let us denote $< \overline{X} > = < \overline{X_{n-1}}, \overline{X_{n-1}}, \dots, \overline{X_2}, \overline{X_1}, \overline{X_0} >$
- It is clear that $< X > + < \overline{X} > = < 1,1, \dots, 1,1,1 > = (-1)$

$(-2^{n-1}) + (2^{n-1}-1) = (-1)$

- Thus negating $< X >$ is by:

$$\boxed{- < X > = < \overline{X} > + 1}$$

- Example: <0011000> = 24 <1100111> +1 = <1101000>=(-24)

Indeed (-64)+32+8= (-24)

Negating a 2's complement number - examples

$$\langle X \rangle = \langle 001\boxed{1000} \rangle = 24$$

Carry bits during the calculation

$$\begin{array}{lll} & & \quad\ \ 1\ 1\ 1 \\ \langle \overline{X} \rangle & = & \langle 1100111 \rangle \\ +1 & = & \langle 0000001 \rangle \\ \text{-----} & & \text{------------} \\ -\langle X \rangle & = & \langle 110\boxed{1000} \rangle = (-24) \end{array}$$

$$\langle X \rangle = \langle 110\boxed{1000} \rangle = (-24)$$

Carry bits during the calculation

$$\begin{array}{lll} & & \quad\ \ 1\ 1\ 1 \\ \langle \overline{X} \rangle & = & \langle 0010111 \rangle \\ +1 & = & \langle 0000001 \rangle \\ \text{-----} & & \text{------------} \\ -\langle X \rangle & = & \langle 001\boxed{1000} \rangle = 24 \end{array}$$

We see that for humans it is easier to negate by starting from LSB, leaving all bits until the 1st 1 (included) unchanged and inverting all next bits.

Relation of 2’s comp. and Unsigned numbers

The relation to Unsigned Numbers

When the MSB, bit 7, is 0 – there is no difference between Unsigned 8 bits number and the same number in 2's Complement.

The MSB, bit 7, weighs 128 in Unsigned and (-128) in 2's complement. Thus there is a difference of 256 = 128-(-128) between Unsigned 8 bits number and the same number in 2's Complement when the MSB is 1.

It is easy to see that:

$$[X] = <X> + X_{n-1} \cdot 2^n$$

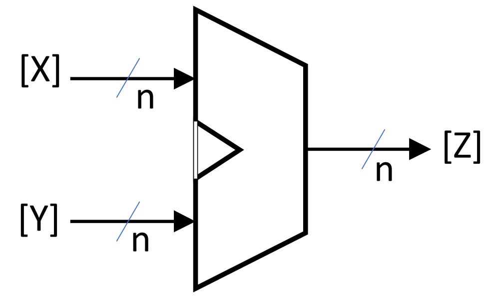
	<10000000>= (-128)
	<10000001>= (-127)
	<10000010>= (-126)
	…
	<11111110>= (-2)
	<11111111>= (-1)
[00000000]=0	<00000000>=0
[00000001]=1	<00000001>=1
[00000010]=2	<00000010>=2
…	…
[01111111]=127	<01111111>=127
[10000000]=128	
[10000001]=129	
[10000010]=130	
…	
[11111111]=254	
[11111111]=255	

The relation to Unsigned Numbers (cont.)

Say we have an n-bit Unsigned Adder built using the equation

$[C_{K+1}, Z_k] = X_K+Y_K+C_K$

Say we feed the adder with the bit strings of <X> and <Y>. The adder does not know that these are 2's component bits and it calculates [Z]=[X]+[Y] as if the bits represent two Unsigned numbers.



However, it is easy to see that:

$$
\begin{aligned}
[Z]=[X]+[Y] &= \langle X\rangle + X_{n-1}\cdot 2^n + \langle Y\rangle + Y_{n-1}\cdot 2^n = \\
&= \langle X\rangle + \langle Y\rangle + (X_{n-1}+Y_{n-1})\cdot 2^n = \\
&= \langle Z\rangle + (X_{n-1}+Y_{n-1})\cdot 2^n
\end{aligned}
$$

Where is $\mathbf{1\cdot 2^8}$?

I.e., we get the correct result <Z> with additional value that is shifted left n positions! So if we take the n LSBs we just get <Z>.

Thus: An Unsigned Adder also adds 2's Complement numbers as is. No HW changes are required. This is the reason for using 2's comp and not Sign & Magnitude

Sign Extension

הארכת סימן

Sign extension

<0000 0000 0111 0011> ← <0111 0011>

<1111 1111 1111 0011> ← <1111 0011>

Say we want to copy an 8-bit unsigned number into 16-bit unsigned number. This is easily done by copying the 8-bit number into the 8 LSBs of the 16-bit number and filling the rest 8 MSBs with 0-s. This is correct since leading zeros do not change the value of the number.

This is also the case when we want to copy a positive 8-bit 2's complement number into a 16-bit 2's complement number. Again it is done by copying the 8-bit number into the 8 LSBs of the 16-bit number and filling the rest 8 MSBs with 0-s. This is still correct since leading zeros do not change the value of the number also for 2's comp. numbers.

But what happens when this is a negative 8-bit 2's complement number?

The MSB is 1. The MSB of the copied number should be 1 as well – it is still negative. So how do we keep the value unchanged? Let's add a single bit first.

<1xxxxxxx> After adding 1 bit: <11xxxxxxx>

MSB= bit7 = -128 MSB = bit8 = -256, bit7 = +128 together (-256)+128= (-128)

Similarly, adding two bits of 1 will not change the value =>

When we copy a short 2's comp. number into a longer number we should extend the sign bit to the additional bits (= if it is 0, add 0-s. If it is 1, add 1-s)

Sign extension in right shift

Let’s discuss shifting right of numbers. We mean shift right one position and truncate the fraction.

<01010> >> 1 = <00101> $10 >> 1 = \lfloor 10/2 \rfloor = 5$

<01010>
<00101>

We shift the bits to the right and add 0 to the MSB.

This is OK since leading zeros do not change the value in positive numbers (Also so in unsigned numbers)

<01011> >> 1 = <00101> $11 >> 1 = \lfloor 11/2 \rfloor = 5$

<01011>
<00101>

We shift the bits to the right and add 0 to the MSB.

This is OK since leading zeros do not change the value in positive numbers (Also so in unsigned numbers)

<10110> >> 1 = <11011> $-10 >> 1 = -\lfloor 10/2 \rfloor = -5$

<10110>
<11011>

We shift the bits to the right and add 1 to the MSB.

This is OK since we can first extend the number to 6 bits and then cut the LSB

<10101>
<11010>

<10101> >> 1 = <11010> $-11 >> 1 = -6 \neq -\lfloor 11/2 \rfloor = -5$

We shift the bits to the right and add 1 to the MSB and get an unexpected result.

If we shift right a negative odd number we should add 1 to the result to get consistency with positive number shifts.

Thus, a correct arithmetic shift right means: sign extension then truncation, then add X_{n-1}*X_0 (of the orig. num.)

Overflow - Unsigned numbers

Overflow in Unsigned Numbers

Let’s discuss 8-bit unsigned numbers. The range is $0\text{-}(2^8\text{-}1)=0\text{-}255$. When we add 240 and 35, we suppose to get 275. However, this number CANNOT be represented in 8 bits. It requires 9 bits to represent. This is a case of Overflow.

Similarly, 10 - 13 = (-3). However, we cannot represent numbers below 0. Thus, we will have an Overflow also in this case.

```
            1 11     <=carry bits
 Cn          [11110000] = 240
           + [00100011] = 35
          --------------
             [00010011] = 19
```

```
     [00001010]  =10
   - [00001101]  =13
    1 1111101    <=borrow bits
 Bn -------------
     [11111101] = 253
```

Overflow -

When the result of a mathematical operation cannot be represented by the n-bit number format we use, we have an overflow.

Overflow (OVF) in addition of unsigned numbers

Let us add two n bits binary numbers [X] and [Y] and check when we have an overflow.

$$
\begin{array}{rcll}
 & [X] = & [X_{n-1}\, X_{n-2} \ldots X_2\, X_1\, X_0] & \text{range of 0 to } (2^n-1) \\
+ & & + & \\
 & [Y] = & [Y_{n-1}\, Y_{n-2} \ldots Y_2\, Y_1\, Y_0] & \text{range of 0 to } (2^n-1) \\
 & \text{--------} & \text{-------------------------} & \\
 & [Z] = & [Z_{n-1}\, Z_{n-2} \ldots Z_2\, Z_1\, Z_0] & \text{range of 0 to } 2\cdot(2^n-1)=(2^{n+1}-2)
\end{array}
$$

If the result is above (2^n-1) we have Overflow! If we use n+1 bits for the calculation we won't have an overflow since in n+1 bits we can represent values in the range of 0 to $(2^{n+1}-1)$ – more than the maximal result:

$$
\begin{array}{rcll}
 & [X] = & [0\ \ X_{n-1}\, X_{n-2} \ldots X_2\, X_1\, X_0] & \\
+ & & + & \\
 & [Y] = & [0\ \ Y_{n-1}\, Y_{n-2} \ldots Y_2\, Y_1\, Y_0] & \\
 & \text{--------} & \text{----------------------------} & \\
 & [Z] = & [Z_n\, Z_{n-1}\, Z_{n-2} \ldots Z_2\, Z_1\, Z_0] & \text{result is above } (2^n-1) \text{ only if } Z_n=1.
\end{array}
$$

But $Z_n = 0 + 0 + C_n$ where C_n is the carry coming out of the (n-1) bit addition.

Thus, **in unsigned addition we have OVF iff $C_n = 1$**

Overflow (OVF) in subtraction of unsigned numbers

Let us subtract two n bits binary numbers [X] and [Y] and check when we have an overflow.

$[X] \quad = \quad [X_{n-1}\, X_{n-2} \dots X_2\, X_1\, X_0]$ range of 0 to (2^n-1) -

\- \-

$[Y] \quad = \quad [Y_{n-1}\, Y_{n-2} \dots Y_2\, Y_1\, Y_0]$ range of 0 to (2^n-1)

-------- -------------------------

$[Z] \quad = \quad [Z_{n-1}\, Z_{n-2} \dots Z_2\, Z_1\, Z_0]$ range of $-(2^n-1)$ to (2^n-1). Might Overflow!

If the result is below 0 we have Overflow! If we use n+1 bits 2’s complement for the calculation we won’t have an overflow since in n+1 bits we can represent values in the range of $-(2^n)$ to (2^n-1) - more than expected here:

$<X> \quad = \quad <0\ X_{n-1}\, X_{n-2} \dots X_2\, X_1\, X_0>$

\- \-

$<Y> \quad = \quad <0\ Y_{n-1}\, Y_{n-2} \dots Y_2\, Y_1\, Y_0>$

-------- -------------------------------

$<Z> \quad = \quad <1\ Z_{n-1}\, Z_{n-2} \dots Z_2\, Z_1\, Z_0>$ result is negative only if $Z_n=1$.

But $Z_n = 0 - 0 - B_n$ where B_n is the borrow coming out of the (n-1) bit addition.

Thus, **in unsigned subtraction we have OVF iff $B_n = 1$**

Overflow – 2’s comp. numbers

Examples of overflow in addition of two 2's comp. numbers

We can detect OVF by: $\boxed{X_{n-1}=Y_{n-1} \neq Z_{n-1} \Leftrightarrow \text{OVF}}$

An equivalent condition is: $\boxed{C_n \neq C_{n-1} \Leftrightarrow \text{OVF}}$

```
   011000      <= carry bits
   <011100> = 28
+  <001001> =  9
   -------------
   <100101> = -27

=> OVF
```

```
   100100      <= carry bits
   <100100> = -28
+  <110111> =  -9
   -------------
   <011011> = 27

=>  OVF
```

The number range for 6 bits is **-32** to **31**

We **cannot** represent 37 or -37

Examples of addition of two 2's comp. numbers

We can detect OVF by: $\boxed{X_{n-1}=Y_{n-1} \neq Z_{n-1} \Leftrightarrow \text{OVF}}$

An equivalent condition is: $\boxed{C_n \neq C_{n-1} \Leftrightarrow \text{OVF}}$

000001 <= carry bits
<010101> = 21
+ <001001> = 9

<011110> = 30

111111 <= carry bits
<101011> = -21
+ <110111> = -9

<100010> = -30

011000 <= carry bits
<011100> = 28
+ <001001> = 9

<100101> = -27

=> OVF

100100 <= carry bits
<100100> = -28
+ <110111> = -9

<011011> = 27

=> OVF

111100 <= carry bits
<011100> = 28
+ <110111 > = -9

<010011> = 19

Overflow in 2's Complement Numbers

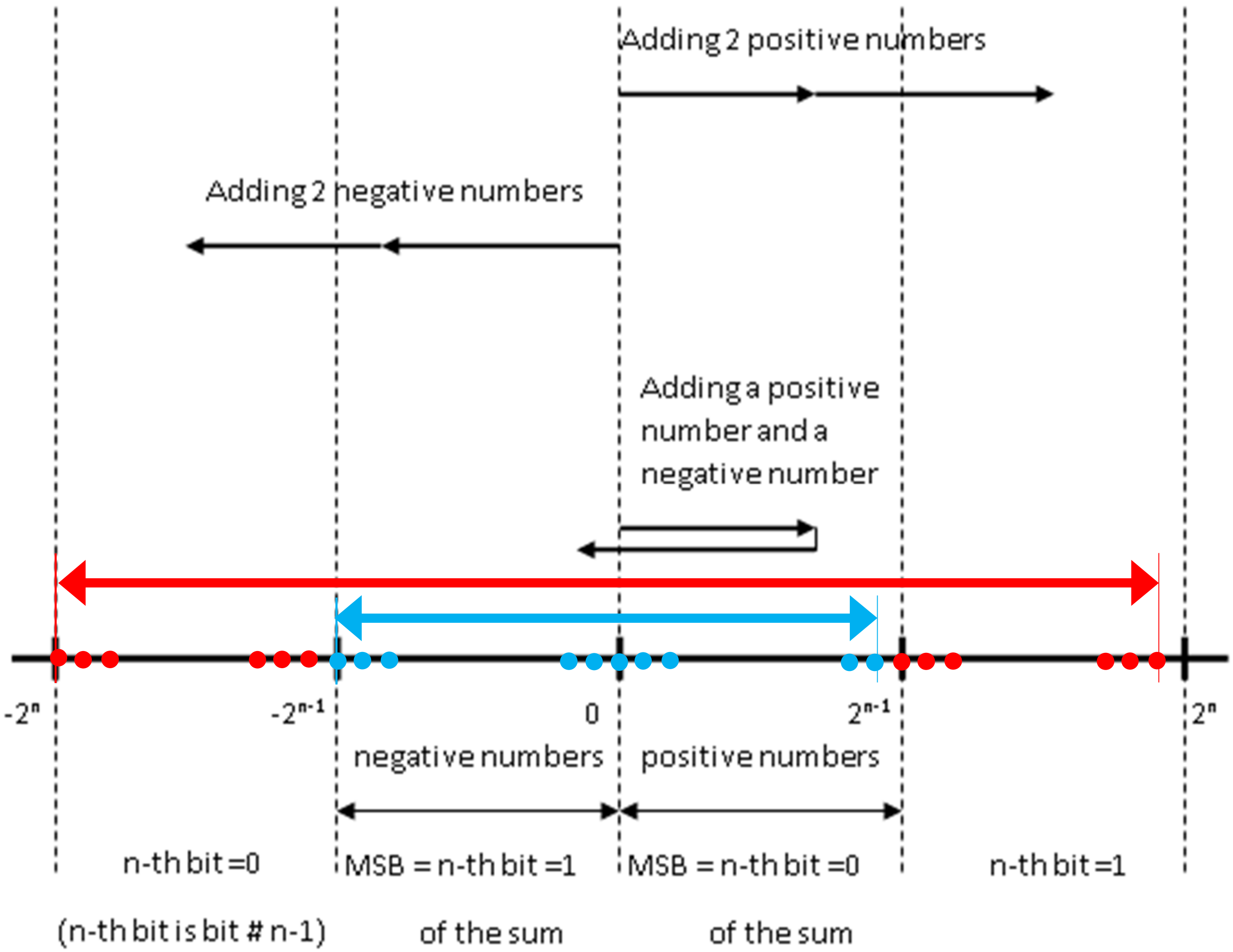
Let's look at the drawing on the right:

The blue arrow shows the range represented by n bits, (-2^{n-1}) to $(2^{n-1}-1)$

The red arrow shows the range represented by n+1 bits, (-2^{n}) to $(2^{n}-1)$

We might have an overflow if we add 2 positive numbers or 2 negative numbers.

Adding a positive number and a negative one will never overflow



Overflow – 2's comp. numbers
Detailed discussion   (If time permits)

How to detect overflow in addition of 2’s comp numbers

Let us add two n bits 2’s comp. numbers <X> and <Y> and check when we have an overflow.

$$
\begin{array}{lclcl}
 & <X> & = & <X_{n-1}\ X_{n-2}\ \ldots\ X_2\ X_1\ X_0> & \text{range of } (-2^{n-1}) \text{ to } (2^{n-1}-1) \\
+ & & + & & \\
 & <Y> & = & <Y_{n-1}\ Y_{n-2}\ \ldots\ Y_2\ Y_1\ Y_0> & \text{range of } (-2^{n-1}) \text{ to } (2^{n-1}-1) \\
 & \text{--------} & & \text{-------------------------} & \\
 & <Z> & = & <Z_{n-1}\ Z_{n-2}\ \ldots\ Z_2\ Z_1\ Z_0> & \text{range of } (-2^{n}) \text{ to } (2^{n}-2) \quad \text{Might Overflow!}
\end{array}
$$

If the result is above $(2^{n-1}-1)$ we have an overflow! This can only happen if we add 2 positive numbers!
If the result is below (-2^{n-1}) we have an overflow! This can only happen if we add 2 negative numbers!

If we use n+1 bits 2’s comp. for the calculation we won’t have an overflow since in n+1 bits we can represent values in the range of $-(2^n)$ to (2^n-1) - more than expected rage.

In the next two slide we will check the two cases of adding 2 positive numbers and adding 2 negative ones.

How to detect overflow in addition of 2 positive numbers

Let us add two n bits 2's comp. positive numbers <X> and <Y> and check when we have an overflow.

$$
\begin{array}{lcll}
 & \langle X\rangle = & \langle 0 \quad X_{n-2} \ldots X_2\, X_1\, X_0\rangle & \text{range of } (-2^{n-1}) \text{ to } (2^{n-1}-1) \\
+ & & + & \\
 & \langle Y\rangle = & \langle 0 \quad Y_{n-2} \ldots Y_2\, Y_1\, Y_0\rangle & \text{range of } (-2^{n-1}) \text{ to } (2^{n-1}-1) \\
 & \text{--------} & \text{-------------------------} & \\
 & \langle Z\rangle = & \langle Z_{n-1}\, Z_{n-2} \ldots Z_2\, Z_1\, Z_0\rangle & \text{range of } (-2^{n}) \text{ to } (2^{n}-2)
\end{array}
$$

If the result is above (2^{n-1}-1) we have an overflow! If we use n+1 bits 2's comp. for the calculation we won't have an overflow since in n+1 bits we can represent values in the range of -(2^n) to (2^n-1) – more than expected:

$$
\begin{array}{lcll}
 & \langle X\rangle = & \langle 0 \;\; 0 \;\; X_{n-2} \ldots X_2\, X_1\, X_0\rangle & \\
+ & & + & \\
 & \langle Y\rangle = & \langle 0 \;\; 0 \;\; Y_{n-2} \ldots Y_2\, Y_1\, Y_0\rangle & \\
 & \text{--------} & \text{-------------------------------} & \\
 & \langle Z\rangle = & \langle 0\, Z_{n-1}\, Z_{n-2} \ldots Z_2\, Z_1\, Z_0\rangle & \text{The result is correct and } Z_n=0.
\end{array}
$$

The result is above (2^{n-1}-1) only if Z_{n-1}=1. If we look at the original n-bit numbers we see that we have an OVF if:

$$\boxed{\mathbf{0 = X_{n-1}=Y_{n-1} \neq Z_{n-1}}}$$

How to detect overflow in addition of 2 negative numbers

Let us add two n bits 2's comp. negative numbers <X> and <Y> and check when we have an overflow.

$$
\begin{array}{lcll}
 & \langle X \rangle = & \langle 1 \quad X_{n-2} \ldots X_2\, X_1\, X_0 \rangle & \text{range of } (-2^{n-1}) \text{ to } (2^{n-1}-1) \\
+ & & + & \\
 & \langle Y \rangle = & \langle 1 \quad Y_{n-2} \ldots Y_2\, Y_1\, Y_0 \rangle & \text{range of } (-2^{n-1}) \text{ to } (2^{n-1}-1) \\
 & \text{--------} & \text{-------------------------} & \\
 & \langle Z \rangle = & \langle Z_{n-1}\, Z_{n-2} \ldots Z_2\, Z_1\, Z_0 \rangle & \text{range of } (-2^{n}) \text{ to } (2^{n}-2)
\end{array}
$$

If the result is below (-2^{n-1}) we have an overflow! If we use n+1 bits 2's comp. for the calculation we won't have an overflow since in n+1 bits we can represent values in the range of $-(2^n)$ to (2^n-1) – more than expected:

$$
\begin{array}{lcll}
 & \langle X \rangle = & \langle 1 \;\; 1 \;\; X_{n-2} \ldots X_2\, X_1\, X_0 \rangle & \\
+ & & + & \\
 & \langle Y \rangle = & \langle 1 \;\; 1 \;\; Y_{n-2} \ldots Y_2\, Y_1\, Y_0 \rangle & \\
 & \text{--------} & \text{-------------------------------} & \\
 & \langle Z \rangle = & \langle 1 \;\; Z_{n-1}\, Z_{n-2} \ldots Z_2\, Z_1\, Z_0 \rangle & \text{The result is correct and } Z_n=1.
\end{array}
$$

The result is below (-2^{n-1}) only if $Z_{n-1}=0$. Thus we have an OVF if: $1 = X_{n-1}=Y_{n-1} \neq Z_{n-1}$

Combining this from the previous slide we conclude that $\boxed{\mathbf{X_{n-1}=Y_{n-1} \neq Z_{n-1} \;\Leftrightarrow\; OVF}}$

Examples of overflow in addition of two 2's comp. numbers

We found that we can detect OVF by: $\boxed{X_{n-1}=Y_{n-1} \neq Z_{n-1} \Leftrightarrow \text{OVF}}$

An equivalent condition is: $\boxed{C_n \neq C_{n-1} \Leftrightarrow \text{OVF}}$

```
   011000      <= carry bits
   <011100> = 28
+  <001001> =  9
   -------------
   <100101> = -27

=> OVF
```

```
   100100      <= carry bits
   <100100> = -28
+  <110111> =  -9
   -------------
   <011011> = 27

=>  OVF
```

The number range
for 6 bits is **-32** to **31**

We **cannot** represent
37 or -37

Examples of addition of two 2's comp. numbers

We found that we can detect OVF by: $\boxed{X_{n-1}=Y_{n-1} \neq Z_{n-1} \quad \Leftrightarrow \quad \text{OVF}}$

An equivalent condition is: $\boxed{C_n \neq C_{n-1} \quad \Leftrightarrow \quad \text{OVF}}$

000001 <= carry bits
<010101> = 21
+ <001001> = 9

<011110> = 30

111111 <= carry bits
<101011> = -21
+ <110111> = -9

<100010> = -30

011000 <= carry bits
<011100> = 28
+ <001001> = 9

<100101> = -27

=> OVF

100100 <= carry bits
<100100> = -28
+ <110111> = -9

<011011> = 27

=> OVF

111100 <= carry bits
<011100> = 28
+ <110111 > = -9

<010011> = 19

Interesting stuff

A note on comparing numbers

We have the following C code:

short A,B;
unsigned short X,Y:

if (A>B) { … do this ..}
if (X>Y) { … do that ..}

In the 1st case the compiler will do if (A-B>0) { … do this ..}   and since these are two 2's comp. numbers, it should choose an "if" instruction that checks whether the sign bit of the result is 0 and that the result is not zero. In 8086 the appropriate instruction that check these conditions is **jg** – jump if greater.

In the 2nd case the compiler will do if (X-Y>0) { … do that ..}   and since these are two unsigned numbers, it should choose an "if" instruction that checks whether the 16-th borrow bit 0 and that the result is not zero. In 8086 the appropriate instruction that check these conditions is **ja** – jump above.

The compiler will do that by its own since it know the types of these variables. It is transparent to the programmer. This is a service given by the compiler.

Long addition of unsigned numbers

Let us add two 16-bit binary numbers [A] and [B] when we have only an 8-bit adder:

$$
\begin{array}{ccccccc}
[A] & = & [A_{15}\, A_{14} \ldots A_2\, A_1\, A_0] & = & [A_{15}\, A_{14} \ldots A_9\, A_8] \cdot 256 + [A_7 \ldots A_2\, A_1\, A_0] & = & A_H \cdot 256 + A_L \\
[B] & = & [B_{15}\, B_{14} \ldots B_2\, B_1\, B_0] & = & [B_{15}\, B_{14} \ldots B_9\, B_8] \cdot 256 + [B_7 \ldots B_2\, B_1\, B_0] & = & B_H \cdot 256 + B_L \\
\hline
[Y] & = & [Y_{15}\, Y_{14} \ldots Y_2\, Y_1\, Y_0] & = & [Y_{15}\, Y_{14} \ldots Y_9\, Y_8] \cdot 256 + [Y_7 \ldots Y_2\, Y_1\, Y_0] & = & Y_H \cdot 256 + Y_L
\end{array}
$$

So let's denote $A_H = [A_{15} A_{14} \ldots A_9\, A_8]$, $A_L = [A_7 \ldots A_2\, A_1\, A_0]$ and similarly $B_H = [B_{15}\, B_{14} \ldots B_9\, B_8]$ and $B_L = [B_7 \ldots B_2\, B_1\, B_0]$ and also $Y_H = [Y_{15}\, Y_{14} \ldots Y_9\, Y_8]$ and $Y_L = [Y_7 \ldots Y_2\, Y_1\, Y_0]$.

Now we will use the **add** instruction of the 8086 CPU for example to calculate the result in two parts:

add Y_L , A_L , B_L (This instruction means $Y_L = A_L + B_L$)
add Y_H , A_H , B_H

But here in the 2nd line we disregard C_8 that resulted from the 1st line calculation. In order to fix that we will use the **adc** instruction that adds the two variable and the carry of the previous calculation:

add Y_L , A_L , B_L (This instruction means $Y_L = A_L + B_L$)
adc Y_H , A_H , B_H (This instruction means $Y_H = A_H + B_H +$ Carry_flag)

Long addition of unsigned numbers

And how the is done in adding two 32-bit binary numbers?

So let's denote $A_H = [A_{31} \dots A_{24}] = A[31:24]$, $A_{MH} = A[23:16]$, $A_{ML}=A[15:8]$, $A_L=A[7:0]$
and $B_H = [B_{31} \dots B_{24}] = B[31:24]$, $B_{MH} = B[23:16]$, $B_{ML}=B[15:8]$, $B_L=B[7:0]$ and
also $Y_H = Y[31:24]$, $Y_{MH} = Y[23:16]$, $A_{ML}=Y[15:8]$, $Y_L=Y[7:0]$

And now we'll calculate by:
add Y_L , A_L , B_L (This instruction means $Y_L = A_L + B_L$)
adc Y_{ML} , A_{ML} , B_{ML} (This instruction means $Y_{ML} = A_{ML} + B_{ML} + C_8$)
adc Y_{MH}, A_{MH} , B_{MH} (This instruction means $Y_{MH} = A_{MH} + B_{MH} + C_{16}$)
adc Y_H , A_H , B_H (This instruction means $Y_H = A_H + B_H + C_{24}$)

Note that **the order of the calculation matters**!

We do not split the calculation like that. The C compiler does that for use. It knows the size of A, B & Y since they are defined in our program (char=8-bit 2's comp. short =16-bit 2's comp., long=32-bit 2's comp., unsinged char=8-bit unsigned, unsigned short= 16-bit unsigned, etc.) and it also needs to know the width of the CPU. Then it will split the calculation as required!

Subtraction is handled similarly with **sub Y_L , A_L , B_L** and **sbb Y_H , A_H , B_H** where **sbb** stands for sub with borrow.

Summary

Summary

Addition:

We will add 2 Unsigned numbers bit by bit using the $[C_{K+1}, Z_k] = X_K + Y_K + C_K$

2's Complement numbers:

$<X> = <X_{n-1}, X_{n-2}, \ldots, X_0>$ value is calculated by $X = X_{n-1} \cdot (-2^{n-1}) + \sum_{i=0}^{n-2} X_i \cdot 2^i$

The weight of X_{n-1} is (-2^{n-1}) instead of 2^{n-1} in Unsigned numbers

The range is from -2^{n-1} to $2^{n-1}-1$. MSB is Sign bit. <X> is negative if MSB=1.

We negate by $-<X> = <\overline{X}> + 1$ or by:

Starting from LSB, leaving all bits until the 1st 1 (included) unchanged, and inverting all next bits

Unsigned Adder knows to add also 2's comp. numbers since $[X] = <X> + X_{n-1} \cdot 2^n$

Summary (cont.)

Sign extension:

When expanding the no. of bits in a 2's comp. number, the MSB (sign) need to be duplicated.

Overflow in Unsigned Numbers:

in unsigned addition we have OVF iff $C_n = 1$

in unsigned subtraction we have OVF iff $B_n = 1$

Overflow in 2's comp. Numbers:

in 2's comp. addition (Z=X+Y) we have OVF iff $X_{n-1}=Y_{n-1} \neq Z_{n-1}$

Another equivalent condition is: we have OVF iff $C_n \neq C_{n-1}$

Long addition:

Split the number to Upper half & Lower half and add with carry:

add Y_L, A_L, B_L (This means $Y_L = A_L + B_L$)

adc Y_H, A_H, B_H (This means $Y_H = A_H + B_H$ + Carry of $A_L + B_L$)

End of

Lecture #2

Signed Fixed Point Numbers