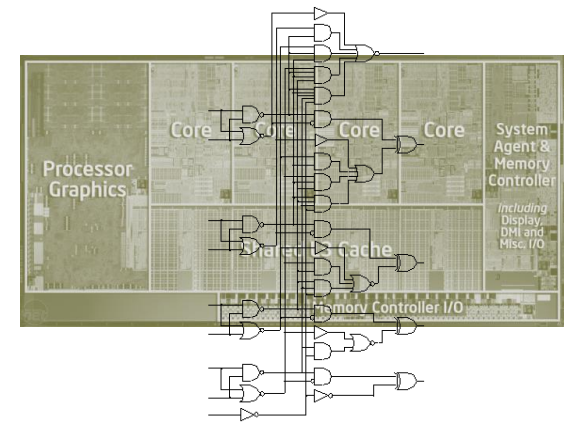




고려대학교
KOREA UNIVERSITY



COSE221, COMP211 Logic Design

Lecture 7. Subtractor

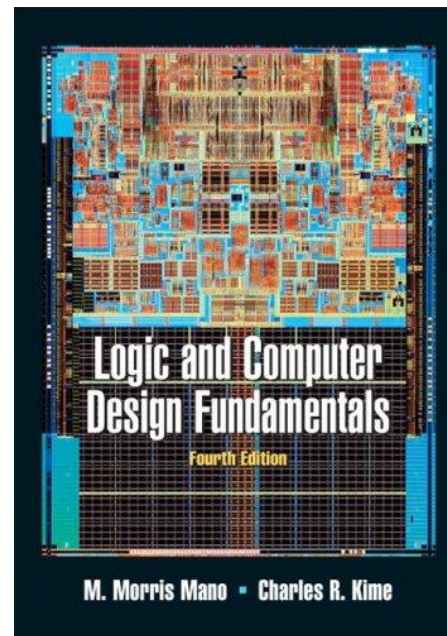
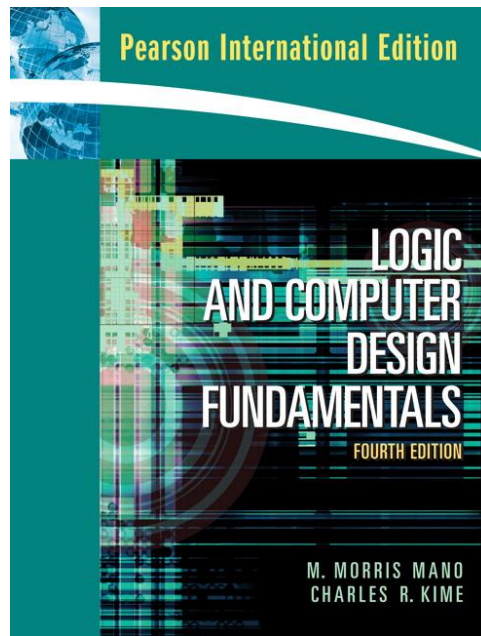
Prof. Taeweon Suh

Computer Science & Engineering

Korea University

References

- Our book covers very little about addition and subtraction
- Refer to the following books for details
 - Logic and Computer Design Fundamentals by Morris Mano, 4th edition (Ch 4. Arithmetic Functions and HDLs, pp168 ~ pp185)



Revisiting 2's Complement Number

- Given an n-bit number N, the 2s complement of N is defined as

$$\begin{aligned} 2^n - N & \text{ for } N \neq 0 \\ 0 & \text{ for } N = 0 \end{aligned}$$

- Example: 3 is 4'b0011 (in a 4-bit binary)
 - 2s complement of 3: $2^4 - 3 = 4'b1101$
- A fast way to get a 2s complement number is to **flip all the bits and add 1**
- In hardware design of computer arithmetic, the 2s complement number provides a convenient and simple way to do addition and subtraction of **unsigned and signed numbers**

Subtractor

- Suppose that we use a 4-bit computer

7 - 5

$$\begin{array}{r} 0111 \\ 0101 \\ \hline 0111 \\ + 1011 \\ \hline 10010 \end{array}$$

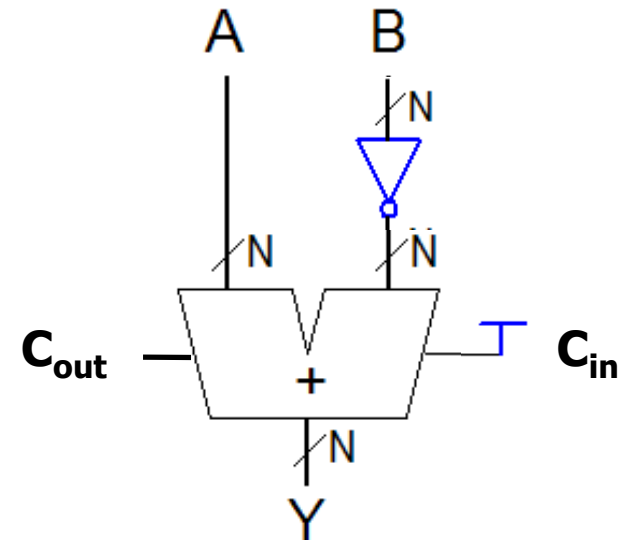
Result = 2

3 - 7

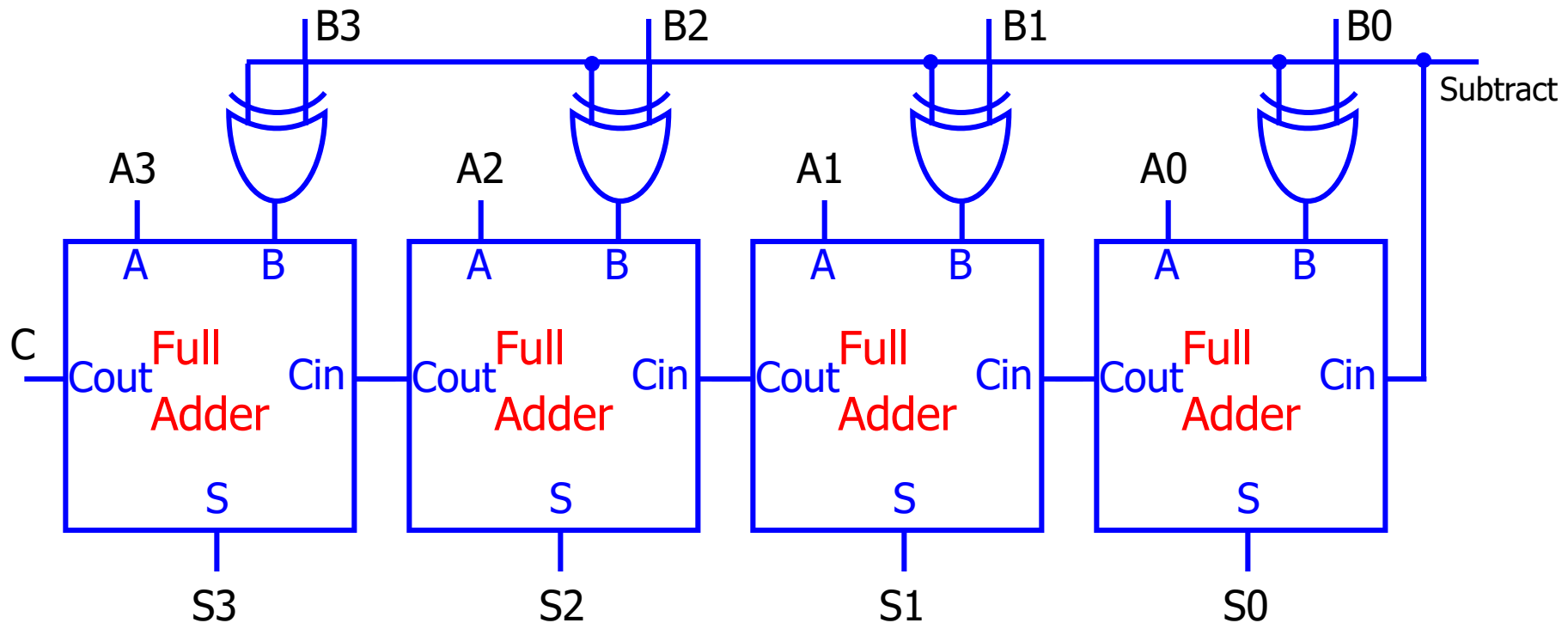
$$\begin{array}{r} 0011 \\ 0111 \\ \hline 0011 \\ + 1001 \\ \hline 01100 \end{array}$$

Result = -4

Implementation



An Implementation of a 4-bit Adder and Subtractor



Hmmm.. So, it looks simple! Are we done?

Not Really!!

Overflow/Underflow

- Overflow/Underflow
 - The answer to an addition or subtraction exceeds the magnitude that can be represented with the allocated number of bits
- Overflow/Underflow is a problem in computers because the number of bits to hold a number is fixed
 - For this reason, computers detect and flag the occurrence of an overflow/underflow
- Detection of an overflow/underflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned

Overflow/Underflow in Unsigned Numbers

- When two unsigned numbers are **added**, **overflow** is detected from the **end carry-out** of the most significant position
 - If the end carry is 1, there is an overflow
- When two unsigned numbers are **subtracted**, **underflow** is detected when **the end carry is 0**

Subtraction of Unsigned Numbers

- Unsigned number is either positive or zero
 - There is **no sign bit**
 - So, a n-bit can represent numbers from 0 to $2^n - 1$
 - For example, a 4-bit can represent 0 to 15 ($=2^4 - 1$)
 - To declare an unsigned number in C language,
 - `unsigned int a;`
 - x86 allocates a 32-bit for a variable of `unsigned int`
- Subtraction of **unsigned** integers
 - $M - N$ in binary can be done as follows:
 - $M + (2^n - N) = M - N + 2^n$
 - If $M \geq N$, the sum does **produce an end carry**, which is 2^n
 - Subtraction result is zero or a positive number
 - If $M < N$, the sum does **not produce an end carry** since it is equal to $2^n - (N - M)$
- **Unsigned Underflow**
 - If there is no carry-out from adder, the subtraction result is negative (and unsigned number can't represent negative numbers)

Example

- Suppose that we use a 4-bit computer
 - 4-bit can represent 0 to 15

10 - 5

$$\begin{array}{r} 1010 \\ 0101 \\ \hline 1010 \\ + 1011 \\ \hline 10101 \end{array}$$

10 - 13

$$\begin{array}{r} 1010 \\ 1101 \\ \hline 1010 \\ + 0011 \\ \hline 01101 \end{array}$$

- Carry-out can be used in **comparison** of two unsigned numbers
 - If the sum produces an end carry, then the minuend (10) is bigger than or equal to the subtrahend (5)
- It is called **unsigned underflow (borrow)** when the carry-out is 0 in unsigned subtraction
- Carry-out can be used in **comparison** of two unsigned numbers
- If the sum does **not** produce an end carry, then the former (10) is smaller than the latter (13)

Overflow/Underflow in Signed Numbers

- With signed numbers, an overflow/underflow can't occur for an addition if one number is positive and the other is negative.
 - Adding a positive number to a negative number produces a result whose magnitude is equal to or smaller than the larger of the original numbers
- An **overflow** may occur in addition if two numbers are both positive
 - When x and y both have sign bits of 0 (positive numbers)
 - If the sum has sign bit of 1, there is an overflow
- An **underflow** may occur in addition if two numbers are both negative
 - When x and y both have sign bits of 1 (negative numbers)
 - If the sum has sign bit of 0, there is an underflow

Examples

8-bit **Signed** number addition

01001000 (+72)

00111001 (+57)

10000001 (+129)

What is largest
positive number
represented by 8-bit?

8-bit **Signed** number addition

10000001 (-127)

11111010 (-6)

01111011 (-133)

What is smallest
negative number
represented by 8-bit?

Overflow/Underflow in Signed Numbers

- We can detect overflow/underflow with the following logic

- Suppose that we add two k-bit numbers

$$x_{k-1}x_{k-2}\dots x_0 + y_{k-1}y_{k-2}\dots y_0 = s_{k-1}s_{k-2}\dots s_0$$

$$\text{Overflow} = \overline{x_{k-1}}\overline{y_{k-1}}s_{k-1} + \overline{x_{k-1}}y_{k-1}\overline{s_{k-1}}$$

- There is an easier formula

- Let the carry-out of a k-bit full adder be $c_{k-1}c_{k-2}\dots c_0$

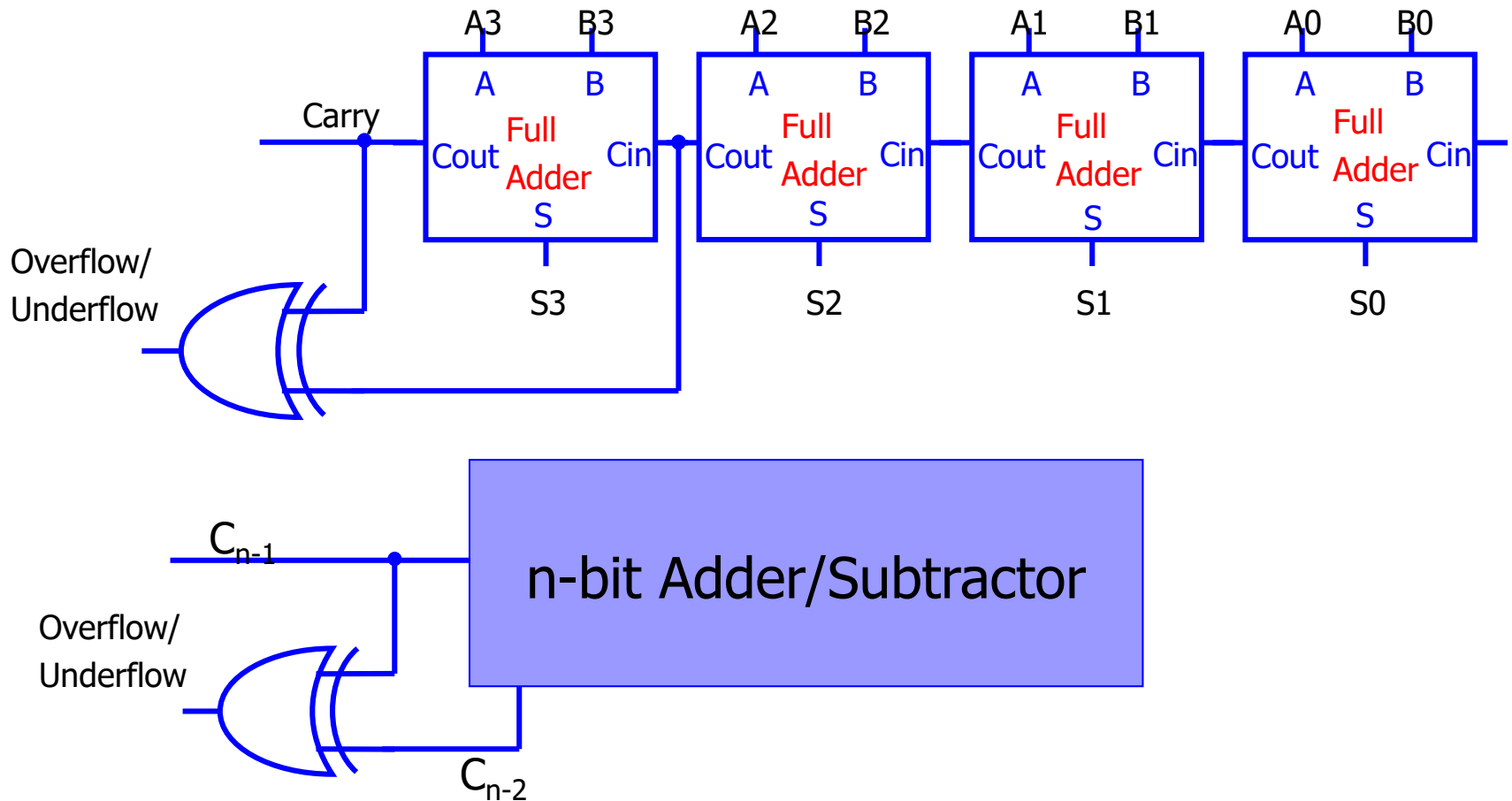
$$\text{Overflow} = c_{k-1} \oplus c_{k-2}$$

- When $x_{k-1} = 0$ and $y_{k-1} = 0$, the only way that $s_{k-1} = 1$
 - 1 (c_{k-2}) is carried in, then 0 (c_{k-1}) is carried out
 - Adding two positive numbers results in a negative number
- When $x_{k-1} = 1$ and $y_{k-1} = 1$, the only way that $s_{k-1} = 0$
 - 0 (c_{k-2}) is carried in, then 1 (c_{k-1}) is carried out
 - Adding two negative numbers results in a non-negative number

Subtraction of Signed Numbers

- Signed number represents positive or negative number
 - There is a sign bit (MSB)
 - A n-bit can represent numbers from -2^{n-1} to $2^{n-1}-1$
 - For example, a 4-bit can represent -8 (-2^3) to 7 ($=2^3 - 1$)
 - To declare a signed number in C language,
 - `int a; // signed is implicit`
 - x86 allocates a 32-bit for a variable of signed int
- Subtraction of signed integers
 - Same as the unsigned number subtraction: addition of two binary numbers in 2s complement form

Overflow/Underflow Detection of Signed Numbers



Recap

- **Unsigned numbers**

- **Overflow** could occur when 2 unsigned numbers are added
 - An end carry of 1 indicates an overflow
- **Underflow** could occur when 2 unsigned numbers are subtracted
 - An end carry of 0 indicates an underflow (minuend < subtrahend)

- **Signed numbers**

- **Overflow** could occur when 2 signed positive numbers are added
- **Underflow** could occur when 2 signed negative numbers are added
- Overflow flag ($C_{n-1} \wedge C_{n-2}$) indicates either overflow or underflow

Recap

- Binary numbers in 2s complement system are added and subtracted by the **same** basic addition and subtraction rules as used in unsigned numbers
 - Therefore, computers need only one common hardware circuit to handle both types (signed, unsigned numbers) of arithmetic
- **The programmer must interpret the results** of addition or subtraction differently, depending on whether it is assumed that the numbers are signed or unsigned

Backup Slides

Flags in CPU

- In general, computer has several flags (registers) to indicate state of operations such as addition and subtraction
 - N: Negative
 - Z: Zero
 - C: Carry
 - V: Overflow
- We have only one adder inside a computer
 - CPU does comparison of signed or unsigned numbers by subtraction using adder
 - Computer sets the flags depending on the operation result
 - Then, do these flags provide enough information to judge that one is bigger than or less than the other?

Example

```
void example(void)
{
    unsigned int a, b, c;
    signed int aa, bb, cc;

    a = 0x10;
    b = 0x20;
    aa = 0x30;
    bb = 0x40;

    if (a > b)    c = a + b;
    else         c = a - b;

    if (aa > bb) cc = aa + bb;
    else        cc = aa - bb;

    return;
}
```

- Equality
 - $a == b$?
 - Do subtraction
 - True if the Z flag is set
- Unsigned number comparison
 - $a > b$?
 - Do subtraction
 - True if C is set and Z is clear
- Signed number comparison
 - $a > b$?
 - Do subtraction
 - True if $N == V$, meaning either
 - Both N and V are set (1) or
 - Both N and V are clear (0)

Example

- Signed number comparison
 - $a > b$?
 - Do subtraction
 - True if $N == V$, meaning either
 - Both N and V are set (1) or
 - Both N and V are clear (0)

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

- $N == V$
 - Both are 0, meaning that overflow didn't occur
 - Examples: $5 - 1$, $3 - (-4)$, $(-3) - (-4)$
 - Both are 1, meaning that overflow did occur
 - Examples: $5 - (-3)$, $7 - (-4)$

sa > sb

	Which flags would you check? (N, Z, C, V)	
Unsigned higher	ua > ub ?	C = 1
Unsigned lower	ua < ub ?	C = 0
Signed greater than	sa > sb ?	
Signed less than	sa < sb ?	

Signed greater than

sa > sb? Yes if (N == V)

- (+) - (+) : N=0 & V=0
- (+) - (-) : N=0 & V=0 or
: N=1 & V=1
- (-) - (+) : N=1 & V=0 or
: N=0 & V=1
- (-) - (-) : N=0 & V=0

Signed less than

sa < sb? Yes if (N != V)

- (+) - (+) : N=1 & V=0
- (+) - (-) : N=0 & V=0 or
: N=1 & V=1
- (-) - (+) : N=1 & V=0 or
: N=0 & V=1
- (-) - (-) : N=1 & V=0

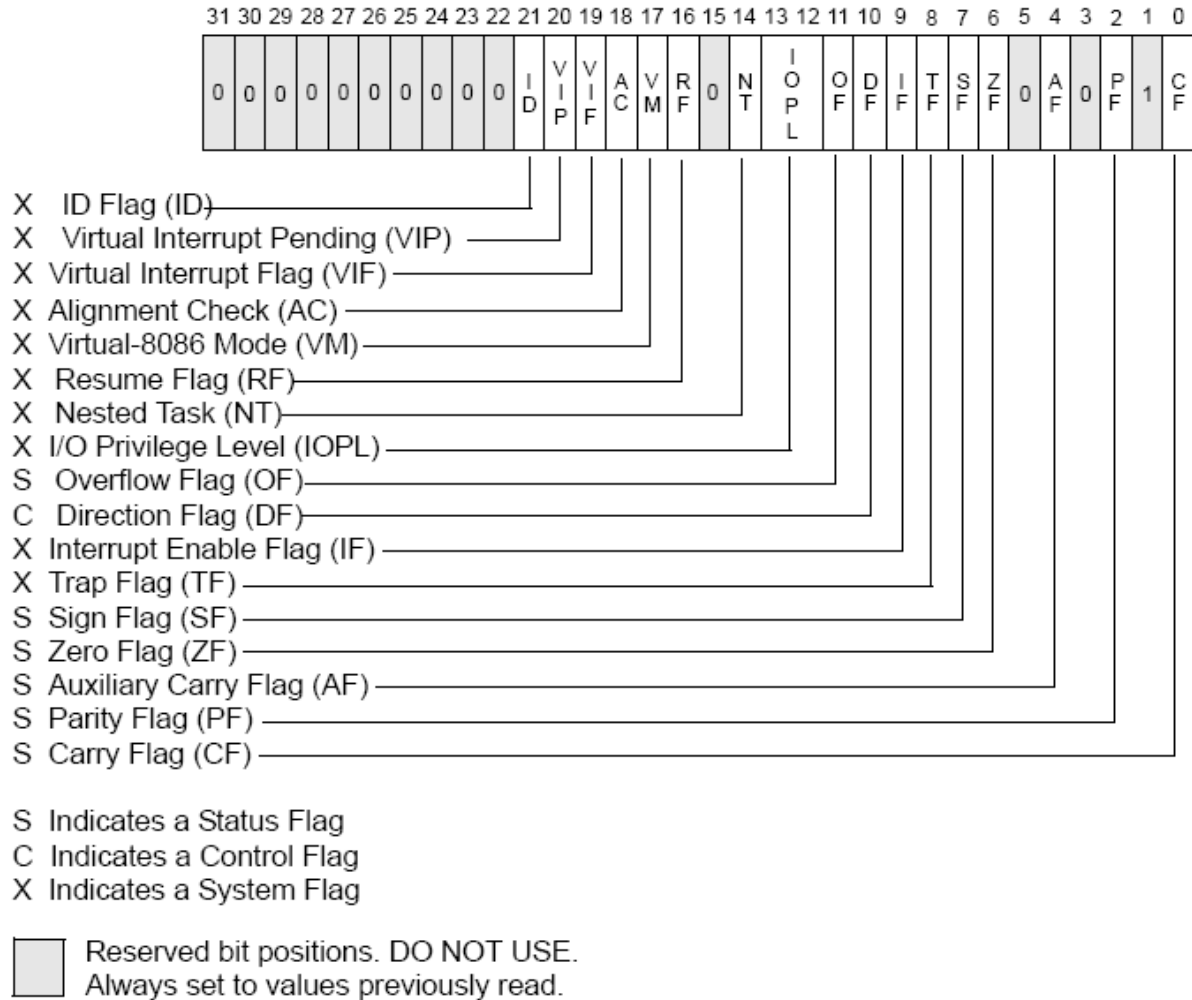
CPSR in ARM

N	Is set to bit 31 of the result of the instruction. If this result is regarded as a two's complement signed integer, then $N = 1$ if the result is negative and $N = 0$ if it is positive or zero.
Z	Is set to 1 if the result of the instruction is zero (this often indicates an <i>equal</i> result from a comparison), and to 0 otherwise.
C	Is set in one of four ways: • For an addition, including the comparison instruction CMN, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise. • For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise. • For non-addition/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter. • For other non-addition/subtractions, C is normally left unchanged (but see the individual instruction descriptions for any special cases).
V	Is set in one of two ways: • For an addition or subtraction, V is set to 1 if signed overflow occurred, regarding the operands and result as two's complement signed integers. • For non-addition/subtractions, V is normally left unchanged (but see the individual instruction descriptions for any special cases).

CPSR in ARM

Unsigned higher	C set and Z clear
Unsigned lower or same	C clear or Z set
Signed greater than or equal	N set and V set, or N clear and V clear ($N == V$)
Signed less than	N set and V clear, or N clear and V set ($N != V$)
Signed greater than	Z clear, and either N set and V set, or N clear and V clear ($Z == 0, N == V$)
Signed less than or equal	Z set, or N set and V clear, or N clear and V set ($Z == 1$ or $N != V$)

EFLGAS in x86



EFLAGS in x86

3.4.3.1 Status Flags

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The status flag functions are:

CF (bit 0) **Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared

otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.

PF (bit 2) **Parity flag** — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

AF (bit 4) **Adjust flag** — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.

ZF (bit 6) **Zero flag** — Set if the result is zero; cleared otherwise.

SF (bit 7) **Sign flag** — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

OF (bit 11) **Overflow flag** — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

Subtraction of Unsigned Numbers

- Unsigned number is either positive or zero
 - There is **no sign bit**
 - So, a n-bit can represent numbers from 0 to $2^n - 1$
 - For example, a 4-bit can represent 0 to 15 ($=2^4 - 1$)
 - To declare an unsigned number in C language,
 - `unsigned int a;`
 - x86 allocates a 32-bit for a variable of `unsigned int`
- Subtraction of **unsigned** integers
 - $M - N$ in binary can be done as follows:
 - $M + (2^n - N) = M - N + 2^n$
 - If $M \geq N$, the sum does **produce an end carry**, which is 2^n
 - Subtraction result is zero or a positive number
 - If $M < N$, the sum does **not produce an end carry** since it is equal to $2^n - (N - M)$
- **Unsigned Underflow**
 - If there is no carry-out from adder, the subtraction result is negative (and unsigned number can't represent negative numbers)

Example

- Suppose that we use a 4-bit computer
 - 4-bit can represent 0 to 15

10 - 5
1010
0101

1010
+ 1011

10101

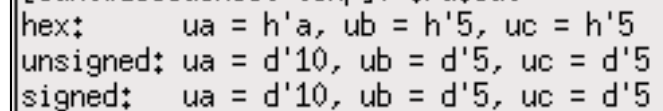
```
#include <stdio.h>

void main()
{
    unsigned int ua, ub, uc;

    ua = 10;
    ub = 5;

    uc = ua - ub ;

    printf("hex:      ua = h'x, ub = h'x, uc = h'x\n", ua, ub, uc);
    printf("unsigned: ua = d'u, ub = d'u, uc = d'u\n", ua, ub, uc);
    printf("signed:   ua = d'd, ub = d'd, uc = d'd\n", ua, ub, uc);
}
```



```
hex:      ua = h'a, ub = h'5, uc = h'5
unsigned: ua = d'10, ub = d'5, uc = d'5
signed:   ua = d'10, ub = d'5, uc = d'5
```

- Carry-out can be used in **comparison** of two unsigned numbers
 - If the sum produces an end carry, then the minuend (10) is bigger than or equal to the subtrahend (5)

Another Example

- Suppose that we use a 4-bit computer
 - 4-bit can represent 0 to 15

10 - 13

1010

1101

1010

+ 0011

01101

```
#include <stdio.h>
```

```
void main()
```

```
{
```

```
unsigned int ua, ub, uc;
```

```
ua = 10;
```

```
ub = 13;
```

```
uc = ua - ub ;
```

```
printf("hex:      ua = h'x, ub = h'x, uc = h'x\n", ua, ub, uc);
```

```
printf("unsigned: ua = d'u, ub = d'u, uc = d'u\n", ua, ub, uc);
```

```
printf("signed:   ua = d'd, ub = d'd, uc = d'd\n", ua, ub, uc);
```

```
}
```

hex:	ua = h'a, ub = h'd, uc = h'ffffffd
unsigned:	ua = d'10, ub = d'13, uc = d'4294967293
signed:	ua = d'10, ub = d'13, uc = d'-3

- It is called **unsigned underflow (borrow)** when the carry-out is 0 in unsigned subtraction
- Carry-out can be used in **comparison** of two unsigned numbers
- If the sum does **not** produce an end carry, then the former (10) is smaller than the latter (13)
- Be careful when you do your programming
 - Understand the consequence of the execution of your program in computer!!!

Subtraction of Signed Numbers

- Signed number represents positive or negative number
 - There is a sign bit (MSB)
 - A n-bit can represent numbers from -2^{n-1} to $2^{n-1}-1$
 - For example, a 4-bit can represent -8 (-2^3) to 7 ($=2^3 - 1$)
 - To declare a signed number in C language,
 - `int a; // signed is implicit`
 - x86 allocates a 32-bit for a variable of signed int
- Subtraction of signed integers
 - Same as the unsigned number subtraction: addition of two binary numbers in 2s complement form

Example

- Suppose that we use a 4-bit (-8 ~ 7)

7 - 5

0111

0101

0111

+ 1011

10010

```
#include <stdio.h>
```

```
void main()
```

```
{
```

```
int sa, sb, sc;
```

```
sa = 7;
```

```
sb = 5;
```

```
sc = sa - sb ;
```

```
printf("hex:      sa = h'x, sb = h'x, sc = h'x\n", sa, sb, sc);
```

```
printf("unsigned: sa = d'u, sb = d'u, sc = d'u\n", sa, sb, sc);
```

```
printf("signed:   sa = d'd, sb = d'd, sc = d'd\n", sa, sb, sc);
```

```
}
```

```
hex:      sa = h'7, sb = h'5, sc = h'2
unsigned: sa = d'7, sb = d'5, sc = d'2
signed:   sa = d'7, sb = d'5, sc = d'2
```

Example

- Suppose that we use a 4-bit (-8 ~ 7)

5 - 7

0101
0111

0101
+ 1001

01110

```
#include <stdio.h>
```

```
void main()
```

```
{
```

```
int sa, sb, sc;
```

```
sa = 5;
```

```
sb = 7;
```

```
sc = sa - sb ;
```

```
printf("hex:      sa = h'%.x, sb = h'%.x, sc = h'%.x\n", sa, sb, sc);
```

```
printf("unsigned: sa = d'%.u, sb = d'%.u, sc = d'%.u\n", sa, sb, sc);
```

```
printf("signed:   sa = d'%.d, sb = d'%.d, sc = d'%.d\n", sa, sb, sc);
```

```
}
```

hex:	sa = h'5, sb = h'7, sc = h'fffffffe
unsigned:	sa = d'5, sb = d'7, sc = d'4294967294
signed:	sa = d'5, sb = d'7, sc = d'-2