2.5 Addition and subtraction

> "Subtraction: Addition's Tricky Pal
> *No. 10, Top Ten Courses for Athletes at a Football Factory, David Letterman et al., Book of Top Ten Lists, 1990*

Addition is just what you would expect in computers. Digits are added bit by bit from right to left, with carries passed to the next digit to the left, just as you would do by hand. Subtraction uses addition: the appropriate operand is simply negated before being added.

PARTICIPATION ACTIVITY 2.5.1: Example of binary addition (COD Figure 3.1).

Start 2x speed

```
                                                                  1 1 0
  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000111two  = 7ten
+ 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000110two  = 6ten
  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001101two  = 13ten
```

PARTICIPATION ACTIVITY 2.5.2: Example of binary subtraction.

Start 2x speed

```
  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000111two  = 7ten
- 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000110two  = 6ten
  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001two  = 1ten

  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000111two  = 7ten
+ 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111010two  = -6ten
  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001two  = 1ten
```

PARTICIPATION ACTIVITY 2.5.3: Binary addition.

Consider the addition of base ten numbers 3 and 2 using 64-bit binary numbers. Fill in the missing values.

```
          b a
  0 ... 0 0 1 1
+ 0 ... 0 0 1 0
----------------
  0 ... 0 y x w
```

1) w
   - ○ 0
   - ○ 1

2) a
   - ○ 0
   - ○ 1

3) x
   - ○ 0
   - ○ 1

4) b
   - ○ 0
   - ○ 1

5) y
   - ○ 0
   - ○ 1

6) 0…0011 + 0…0010 = ?
   - ○ 0…0100
   - ○ 0…0101

PARTICIPATION ACTIVITY 2.5.4: Binary subtraction.

Consider the subtraction of base ten numbers 5 - 4 using 64-bit binary numbers, and achieved by adding 5 with the two's complement of 4:

```
  0 0 ... 0 1 0 1      0 0 ... 0 1 0 1
- 0 0 ... 0 1 0 0    + 1 1 ... d c b a
-----------------    -----------------
                     s 0 0 ... z y x w
```

1) dcba
   - 1011
   - 1100

2) zyxw
   - 0000
   - 0001

3) If a 65th sum bit, s, existed on the left, what value would that bit get?
   - 0
   - 1

CHALLENGE ACTIVITY 2.5.1: Add two numbers.

Start

Add 4 and 2. Work from right to left.

	...						
							Carry bits
	0 0 ...	0	0	1	0	0	4 64-bit number
+	0 0 ...	0	0	0	1	0	2 64-bit number
	0 0 ...	-	-	-	-	-	64-bit number

1 2 3 4 5

Check Next

Overflow

Recall that overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 64-bit word. When can overflow occur in addition? When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, -10 + 4 = -6. Since the operands fit in 64 bits and the sum is no larger than an operand, the sum must fit in 64 bits as well. Therefore, no overflow can occur when adding positive and negative operands.

There are similar restrictions to the occurrence of overflow during subtract, but it's just the opposite principle: when the signs of the operands are the *same*, overflow cannot occur. To see this, remember that *c - a = c + (−a)* because we subtract by negating the second operand and then add. Therefore, when we subtract operands of the same sign we end up by *adding* operands of *different* signs. From the prior paragraph, we know that overflow cannot occur in this case either.

Knowing when overflow cannot occur in addition and subtraction is all well and good, but how do we detect it when it *does* occur? Clearly, adding or subtracting two 64-bit numbers can yield a result that needs 65 bits to be fully expressed.

The lack of a 65th bit means that when overflow occurs, the sign bit is set with the *value* of the result instead of the proper sign of the result. Since we need just one extra bit, only the sign bit can be wrong. Hence, overflow occurs when adding two positive numbers and the sum is negative, or vice versa. This spurious sum means a carry out occurred into the sign bit.

Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result. Such a ridiculous result means a borrow occurred from the sign bit. The figure below shows the combination of operations, operands, and results that indicate an overflow.

PARTICIPATION ACTIVITY 2.5.5: Overflow conditions for addition and subtraction (COD Figure 3.2).

Start 2x speed

```
 01111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111  9223372036854775807ten
+00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000011  +                   3ten
 10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000010  -9223372036854775806ten
```

Operation	Operand A	Operand B	Result indicating overlow	Wrong	Right	
A + B	≥ 0	≥ 0	< 0	Pos + Pos = Neg	Pos + Pos = Pos	Ex: 10 + 20 = 30
A + B	< 0	< 0	≥ 0	Neg + Neg = Pos	Neg + Neg = Neg	Ex: -10 + -20 = -30
A - B	≥ 0	< 0	< 0	Pos - Neg = Neg	Pos - Neg = Pos	Ex: 10 - -20 = 30
A - B	< 0	≥ 0	≥ 0	Neg - Pos = Pos	Neg - Pos = Neg	Ex: -10 - 20 = -30

PARTICIPATION ACTIVITY 2.5.6: Overflow.

Indicate if the binary operation resulted in overflow.

1)

```
  0 1 1 0 ...
+ 0 1 1 1 ...
-------------
  1 1 0 1 ...
```

- ○ Overflow
- ○ No overflow

2)

```
  1 1 0 0 ...
+ 1 1 0 1 ...
-------------
  1 0 0 1 ...
```

- ○ Overflow
- ○ No overflow

3)

```
  0 0 0 1 ...
+ 1 0 0 1 ...
-------------
  1 0 1 0 ...
```

- ○ Overflow
- ○ No overflow

We have just seen how to detect overflow for two's complement numbers in a computer. What about overflow with unsigned integers? Unsigned integers are commonly used for memory addresses where overflows are ignored.

Given that overflow is one of the four condition codes, it is easy for the compiler to check for overflow for add and subtract instructions if it needs to. The programming language C ignores overflows, but the programming language Fortran needs them discovered.

COD Appendix A (The Basics of Logic Design) describes the hardware that performs addition and subtraction, which is called an *Arithmetic Logic Unit* or *ALU*.

Arithmetic Logic Unit (***ALU***): Hardware that performs addition, subtraction, and usually logical operations such as AND and OR.

Hardware/Software Interface

The computer designer must decide how to handle arithmetic overflows. Although some languages like C and Java ignore integer overflow, languages like Ada and Fortran require that the program be notified. The programmer or the programming environment must then decide what to do when overflow occurs.

PARTICIPATION ACTIVITY 2.5.7: Unsigned addition and subtraction.

1) The ARM ADD, ADDI, and SUB instructions may result in overflow.
   - ○ True
   - ○ False
2) Overflow does not occur in unsigned integers.
   - ○ True
   - ○ False
3) Compilers for the C and Fortran programming languages need to identify overflow values and notify the current program.
   - ○ True
   - ○ False

Summary

A major point of this section is that, independent of the representation, the finite word size of computers means that arithmetic operations can create results that are too large to fit in this fixed word size. It's easy to detect overflow in unsigned numbers, although these are almost always ignored because programs don't want to detect overflow for address arithmetic, the most common use of natural numbers. Two's complement presents a greater challenge, yet some software systems require recognizing overflow, so today all computers have a way to detect it.

PARTICIPATION ACTIVITY 2.5.8: Check yourself: Integer arithmetic for byte and halfword variables.

Some programming languages allow two's complement integer arithmetic on variables declared byte and half, whereas LEGv8 core only has integer arithmetic operations on full words. As we recall from COD Chapter 2 (Instructions: Language of the Computer), LEGv8 core does have data transfer operations for bytes and halfwords.

1) Which LEGv8 load instructions should be generated for byte and halfword arithmetic operations?
   - ○ STUR, STURW
   - ○ LDURB, LDURH

2) Which LEGv8 arithmetic instructions should be generated for byte and halfword arithmetic operations?
   - ○ ADD, SUB, MUL, DIV
   - ○ ADD, SUB, MUL, DIV, using AND to mask result to 8 or 16 bits after each operation

3) Which LEGv8 store instructions should be generated for byte and halfword arithmetic operations?
   - ○ SBU, SHU
   - ○ STURB, STURH

Elaboration

One feature not generally found in general-purpose microprocessors is saturating operations. Saturation means that when a calculation overflows, the result is set to the largest positive number or the most negative number, rather than a modulo calculation as in two's complement arithmetic. Saturation is likely what you want for media operations. For example, the volume knob on a radio set would be frustrating if, as you turned it, the volume would get continuously louder for a while and then immediately very soft. A knob with saturation would stop at the highest volume no matter how far you turned it. Multimedia extensions to standard instruction sets often offer saturating arithmetic.

Elaboration

Determining the carry into the high-order bits earlier increases the speed of addition. There are a variety of schemes to anticipate the carry so that the worst-case scenario is a function of the $\log_2$ of the number of bits in the adder. These anticipatory signals are faster because they go through fewer gates in sequence, but it takes many more gates to anticipate the proper carry. The most popular is carry lookahead, which COD Section A.6 in Appendix A (The Basics of Logic Design) describes.