Computer Architecture (Practical Class)
Assembly: Arithmetic Operations

Luís Nogueira

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2023/2024

Introduction

- Most of the operations are given as instruction classes, as they can have different variants with different operand sizes
  - For example, the instruction class `add` consists of four addition instructions: `addb`, `addw`, `addl`, and `addq`, adding bytes, words, double words, and quad words, respectively
- Recall that type conversion has higher precedence than arithmetic operations
  - We can only operate on arguments of the same size
  - Moving from a smaller to a larger data size can involve either sign extension (for signed values) or zero extension (for unsigned values)
- Do not forget that instructions that move or generate 32-bit register values also set the upper 32 bits of the register to zero

Arithmetic operations: ADD

- The `ADD` instruction adds two integers
- Usage: `add` *origin*`,` *destination*
- Performs the operation *destination* = *destination* + *origin* (the result is placed in *destination*
- *origin* can be a memory address, a constant value or a register
- *destination* can be a memory address or a register
- As for any other instruction, a memory address for *origin* and *destination* cannot be used simultaneously
- The `ADD` instruction can add numbers of 8(b), 16(w), 32(l), or 64(q) bits

ADD instruction examples

Adding bytes, words, double words (longs), and quads

```
addb $10, %al              # adds 10 to the 8-bit AL register; AL=AL+10
addw %bx, %cx              # adds the value of BX to CX (16 bits); CX=CX+BX
addl var1(%rip), %eax      # adds the 32-bit value in var1 to EAX; EAX=EAX+var1
addl %eax, %eax            # adds EAX to itself; EAX=EAX+EAX
addq %rcx, %rax            # adds RCX to RAX; RAX=RAX+RCX
```

Arithmetic operations: SUB

- The `SUB` instruction subtracts two integers
- Usage: `sub` *origin*`,` *destination*
- Performs the operation *destination* = *destination* - *origin* (the result is placed in *destination*
- *origin* can be a memory address, a constant value or a register
- *destination* can be a memory address or a register
- A memory address for *origin* and *destination* cannot be used simultaneously
- The `SUB` instruction can subtract numbers of 8(b), 16(w), 32(l), or 64(q) bits

SUB instruction examples

Subtracting bytes, words, double words (longs), and quads

```
subl $10, %eax          # subtract 10 to the current value of EAX; EAX=EAX-10
subw %bx, %cx           # subtract the value of BX to CX (16 bits); CX=CX-BX
subb var1(%rip), %al    # subtract the 8-bit value in var1 to AL; AL=AL-var1
subq %rcx, %rax         # subtract RCX to RAX; RAX=RAX-RCX
```

Arithmetic Operations: INC and DEC

- The `INC` and `DEC` instructions increment (INC) and decrement (DEC) an integer by one, respectively
- Usage: `inc` *destination*
  `dec` *destination*
- performs the operation *destination* = *destination* +/- 1;
- *destination* can be a memory address or a register
- the `INC` and `DEC` instructions can be used in numbers of 8(b), 16(w), 32(l) or 64(q) bits

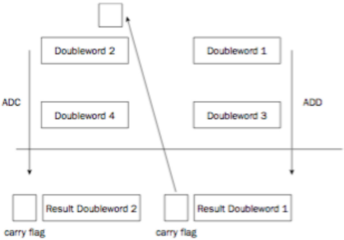
INC/DEC instructions examples

Increment/Decrement bytes, words, double words (longs), and quads

```
incq %rax    # RAX=RAX+1 (64 bits)

incl %eax    # EAX=EAX+1 (32 bits)

incw %bx     # BX=BX+1 (16 bits)

decb %cl     # CL=CL-1 (8 bits)
```

`adc` *origin*, *destination*

- The `ADC` instruction can be used to add two integer values, along with the value contained in the *carry* flag set by a previous addition
- If your program treats the bits in a register as `unsigned` numbers, you must watch to see if your arithmetic sets the carry flag on, indicating the result is wrong (more on this in the next class)
- Performs the operation: $destination = destination + origin + CF$
- The `ADC` instruction can add numbers of 8(b), 16(w), 32(l), or 64(q) bits



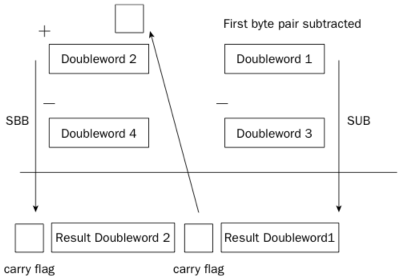
Add With Carry (ADC) Example

```
.global adctest
adctest:
    ...
    movb $0xFF, %al
    movb $0x1, %ah
    movb $0x1, %cl
    movb $0x0, %ch

    # cl = cl + al (8 bits)
    addb %al, %cl

    # ch = ch + ah + CF (8 bits)
    adcb %ah, %ch

    ...
    ret
```

Arithmetic Operations: Subtract with Borrow (SBB)

`sbb` *origin*, *destination*

- The `SBB` instruction can be used to subtract two integer values, along with the value contained in the *carry* flag set by a previous subtraction
- If your program treats the bits in a word as `unsigned` numbers, you must watch to see if your arithmetic sets the carry flag on, indicating the result is wrong (more on this in the next class)
- Performs the operation: $destination = destination - (origin + CF)$
- The `SBB` instruction can subtract numbers of 8(b), 16(w), 32(l), or 64(q) bits.



Practical problem

- When mixing data of different sizes, you must make the right choice of arithmetic instructions, sign extensions, and zero extensions. These depend on subtle aspects of type conversion and the interactions between the arithmetic instructions
- This is illustrated by the following C function:

Mixing different sizes in arithmetic instructions

```c
int x,y;  // 32 bits (signed)

long fun(){
  long t1 = (long)x + y;
  long t2 = (long)(x+y);
  return t1 - t2;
}
```

- The two additions in this function proceed as follows:
  - Recall that type conversion has higher precedence than addition, and so the first addition calls for x to be converted to 64 bits, and by operand promotion y is also converted. Value $t1$ is then computed using 64-bit addition
  - On the other hand, $t2$ is computed by performing a 32-bit addition and then extending this value to 64 bits

Practical problem

- The equivalent Assembly code for this function is as follows:

Mixing different sizes in arithmetic instructions

```
fun:
    movslq x(%rip),%rax      # Convert x to long
    movslq y(%rip),%rdx      # Convert y to long
    addq   %rdx,%rax         # t1 (64-bit addition)
    movl   y(%rip), %edx     # Move y to %edx (lower 32 bits of RDX)
    addl   x(%rip), %edx     # t2 (32-bit addition)
    movslq %edx, %rdx        # Convert t2 to long
    subq   %rdx, %rax        # Return t1 - t2
    ret
```

- The two additions in this function proceed as follows:
  - $t1$ is computed by first sign extending the arguments. The *movslq* instructions take the 32 bits of x and y and sign extend them to 64 bits in registers *%rax* and *%rdx*. The *addq* instruction then performs 64-bit addition to get $t1$
  - $t2$ is computed by performing 32-bit addition with x and y. This value is sign extended to 64 bits (within a single register) to get $t2$

Arithmetic Operations: Multiplication

- Multiplying two 64-bit *signed* or *unsigned* integers can yield a product that requires 128 bits to represent. Intel refers to a 16-byte quantity as an *oct word*
- There are "one-operand" instructions that support generating the full 128-bit product of two 64-bit numbers
  - One argument must be in register %*rax*, and the other is given as the instruction source operand
  - The product is then stored in registers %*rdx* (high-order 64 bits) and %*rax* (low-order 64 bits)
- There is also a "two-operand" multiply instruction, generating a 64-bit product from two 64-bit operands
  - Recall that when truncating the product to 64 bits, both unsigned multiply and two's-complement multiply have the same bit-level behaviour

Arithmetic Operations: Unsigned Multiplication (MUL)

`mul` *origin*

- The `MUL` instruction multiplies two **unsigned** integers
- Performs the operation: $destination = origin \times operand2$
- *origin* can be a memory address or a register
- *operand2* is the RAX, EAX, AX or AL register, depending on the size of *origin*
- *destination* is the RDX:RAX, EDX:EAX, DX:AX or AX registers, depending on the size of *origin*;

Size of *origin*	*operand2*	*destination*
8 bits	AL	AX
16 bits	AX	DX:AX
32 bits	EAX	EDX:EAX
64 bits	RAX	RDX:RAX

- The `MUL` instruction can multiply numbers of 8(b), 16(w), 32(l), or 64(q) bits.

Unsigned Multiplication Example

```
.global multest
multest:

    ...

    movw $200, %ax
    movw $2, %cx

    # multiply %cx by %ax
    # result in %dx:%ax
    mulw %cx

    ...

    ret
```

Arithmetic Operations: Signed Multiplication (IMUL)

`imul` *origin*
`imul` *origin*, *destination*

- The `IMUL` instruction multiplies two **signed** integers
- `imul` *origin*
  - The use of registers is the same as presented in `MUL`
- `imul` *origin*, *destination*
  - Performs $destination = destination \times origin$
  - *origin* can be a memory address, a constant value or a register (16, 32, or 64-bit)
  - *destination* can be a 16,32, or 64-bit register

Signed Multiplication Example

```
.global imultest
imultest:
   # ...
   movl $200, %eax
   movl $-2, %ecx
   imull %ecx                # %edx:%eax = %ecx * %eax

   movq $-123, %rcx
   imulq $4, %rcx            # %rcx = 4 * %rcx
   # ...
   ret
```

Arithmetic Operations: Division

- Signed and unsigned divisions are only supported through single-operand instructions
- Their behaviour is similar to the single-operand multiply instructions
  - The dividend is the 128-bit value stored in registers $\%rdx$ (high-order 64 bits) and $\%rax$ (low-order 64 bits)
  - The divisor is given as the instruction operand
  - The instruction stores the quotient in register $\%rax$ and the remainder in register $\%rdx$
- For most practical cases, the dividend is given as a value up to 64 bits
  - This value should be stored in register $\%rax$
  - The bits of $\%rdx$ should then be set to either all zeros (unsigned arithmetic) or the sign bit of $\%rax$ (signed arithmetic)

`div` *divisor*
`idiv` *divisor*

- The `DIV/IDIV` instruction is used for unsigned/signed division
- Performs the operations:
  - $quotient = dividend \div divisor$
  - $remainder = dividend \mod divisor$
- *divisor* can be a memory address or a register
- *dividend* is the RDX: RAX, EDX:EAX, DX:AX or AX register, depending on the size of *divisor*
- The *quotient* and *remainder* of the division are put in different sections of the RAX and RDX registers, depending on the size of *divisor*

Size of *divisor*	***dividend***	***quotient***	***remainder***
8 bits	AX	AL	AH
16 bits	DX:AX	AX	DX
32 bits	EDX:EAX	EAX	EDX
64 bits	RDX:RAX	RAX	RDX

Unsigned Division Example

```
.global divtest
divtest:
...

  # dividend: ax
  movw $100, %ax
  # divisor: cl
  movb $3, %cl

  # divides %ax by %cl
  # remainder in %ah
  # quotient in %al
  divb %cl

  ...
  ret
```

Important note

- Always initialize **all** bits of the dividend!

Very bad idea!

```
.global bad_div

bad_div:
  ...
  # dividend: eax
  movl $100, %eax
  # divisor: ecx
  movl $3, %ecx

  # divides %edx:%eax by %ecx
  # Problem: the unkown content in %edx becomes part of the dividend

  # remainder in %edx
  # quotient in %eax
  divl %ecx

  ...
  ret
```

Arithmetic Operations: Unsigned and Signed Division (DIV and IDIV)

- Preparing the dividend depends on whether unsigned (*div*) or signed (*idiv*) division is to be performed
- In the former case, the higher order part of the dividend must be set to 0
- In the latter case, a set of instructions is used to produce a correct dividend before a division instruction through sign extension
- The Intel-syntax conversion instructions:
  - *cbw* — sign-extend byte in %*al* to word in %*ax*
  - *cwde* — sign-extend word in %*ax* to double word (long) in %*eax*
  - *cwd* — sign-extend word in %*ax* to double word (long) in %*dx* : %*ax*
  - *cdq* — sign-extend double word (long) in %*eax* to quad word in %*edx* : %*eax*
  - *cdqe* — sign-extend double word (long) in %*eax* to quad word in %*rax*
  - *cqo* — sign-extend quad in %*rax* to oct word in %*rdx* : %*rax*
- are called *cbtw*, *cwtl*, *cwtd*, *cltd*, *cltq*, and *cqto* in *AT&T* naming
  - This is one of the few instructions whose GAS name is very different from the Intel version. *GAS* accepts either mnemonic, but Intel-syntax assemblers like NASM may only accept the Intel names

Correct sign extension

```
.global div_ok

div_ok:
  ...
  # dividend: %eax
  movl $-100, %eax
  # converts the signed long in %eax to the signed double long in %edx:%eax
  cltd

  # divisor: ecx
  movl $3, %ecx

  # divides %edx:%eax by %ecx (remainder in %edx, quotient in %eax)
  idivl %ecx
  ...
  ret
```

A C function *arithprob* that accesses the global variables *a*, *b*, *c*, and *d* has the following body:

*return a*b + c*d;*

It compiles to the following Assembly code:

arithprob.s

```
arithprob:
   movl    d(%rip), %ecx
   movslq  %ecx,%rcx
   movb    b(%rip),%sil
   movsbl  %sil,%esi
   imulq   c(%rip),%rcx
   imull   a(%rip),%esi
   movslq  %esi, %rsi
   addq    %rsi,%rcx
   movq    %rcx,%rax
   ret
```

Based on this Assembly code, determine the types of each of the global variables and write a function prototype describing the return type for *arithprob*. The return of an Assembly function up to a 64-bit value is placed in *%rax* (or parts of it)