Roadmap

- Memory & data
- Integers & floats
- Machine code & C
- **x86 assembly**
- Procedures & stacks
- Arrays & structs
- Optimizations
- Memory & caches

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

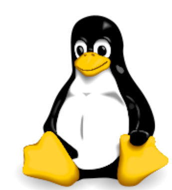
Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

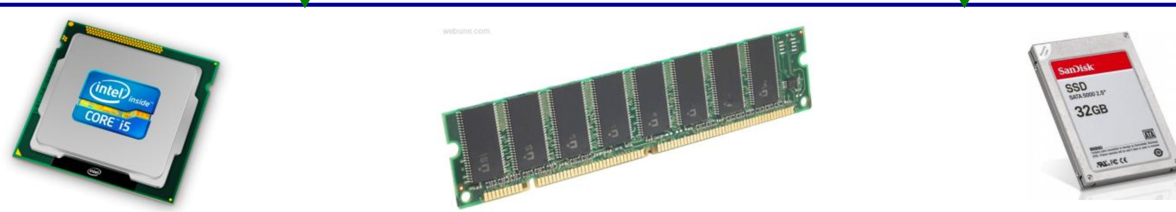
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:



Cours 4: Programmation Assembler x86

- Modes d’adressage mémoire
- Opérations arithmétiques
- Codes conditionnels
- Branches conditionnelles et inconditionnelles
- Boucles
- Switch

Différents modes d'adressage

- **Forme plus général :**

  D(Rb,Ri,S) **Mem[Reg[Rb] + S*Reg[Ri] + D]**

  - D: Constante pour l'offset (1, 2, ou 4 octets)
  - Rb: Registre de base (tous les 8/16 registres)
  - Ri: Registre d'index (tous sauf `%esp` et `%rsp`)
  - S: Multiplicateur : 1, 2, 4, ou 8

- **Cas spéciaux**

  (Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]]**

  D(Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]+D]**

  (Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]]**

Exemples

%edx	0xf000
%ecx	0x100

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]
D(,Ri,S)	Mem[S*Reg[Ri]+D]
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]
D(Rb)	Mem[Reg[Rb] +D]

Expression	Formule	Adresse
`0x8(%edx)`	`0xf000 + 0x8`	`0xf008`
`(%edx,%ecx)`	`0xf000 + 0x100`	`0xf100`
`(%edx,%ecx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%edx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Instruction de calcul d’adresse

- **`leal` *Src*, *Dest***
  - *Src* est l’expression pour le mode d’adressage
  - Met *Dest* à l’adresse calculée par l’expression
    - (lea = *load effective address)*
  - Exemple: **`leal (%edx,%ecx,4), %eax`**

- **Usage :**
  - Calculer l’adresse sans référence mémoire
    - Exemple : traduction de **`p = &x[i];`**
  - Calculer des expressions arithmétiques de forme : x + k*i
    - k = 1, 2, 4, or 8

Cours 4: Programmation Assembler x86

- Modes d’adressage mémoire
- <u>Opérations arithmétiques</u>
- Codes conditionnels
- Branches conditionnelles et inconditionnelles
- Boucles
- Switch

Quelques opérations arithmétiques

- **Instructions binaires:**

Format		*Computation*	
`addl`	*Src,Dest*	*Dest = Dest + Src*	
`subl`	*Src,Dest*	*Dest = Dest − Src*	
`imull`	*Src,Dest*	*Dest = Dest * Src*	
`sall`	*Src,Dest*	*Dest = Dest << Src*	*Also called shll*
`sarl`	*Src,Dest*	*Dest = Dest >> Src*	*Arithmetic*
`shrl`	*Src,Dest*	*Dest = Dest >> Src*	*Logical*
`xorl`	*Src,Dest*	*Dest = Dest ^ Src*	
`andl`	*Src,Dest*	*Dest = Dest & Src*	
`orl`	*Src,Dest*	*Dest = Dest | Src*	

Quelques opérations arithmétiques

- **Instructions unaires**

```
incl Dest        Dest = Dest + 1
decl Dest        Dest = Dest - 1
negl Dest        Dest = -Dest
notl Dest        Dest = ~Dest
```

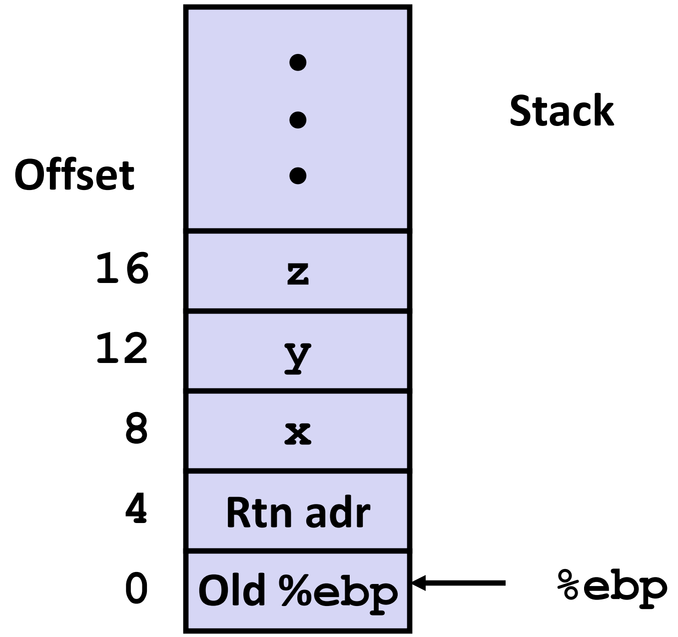
Utiliser `leal` pour des expressions arithmétiques

```c
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

```
arith:
    pushl %ebp                       } Set
    movl %esp,%ebp                   } Up

    movl 8(%ebp),%eax                }
    movl 12(%ebp),%edx               }
    leal (%edx,%eax),%ecx            }
    leal (%edx,%edx,2),%edx          }
    sall $4,%edx                     } Body
    addl 16(%ebp),%ecx               }
    leal 4(%edx,%eax),%eax           }
    imull %ecx,%eax                  }

    movl %ebp,%esp                   }
    popl %ebp                        } Finish
    ret                              }
```

Explication

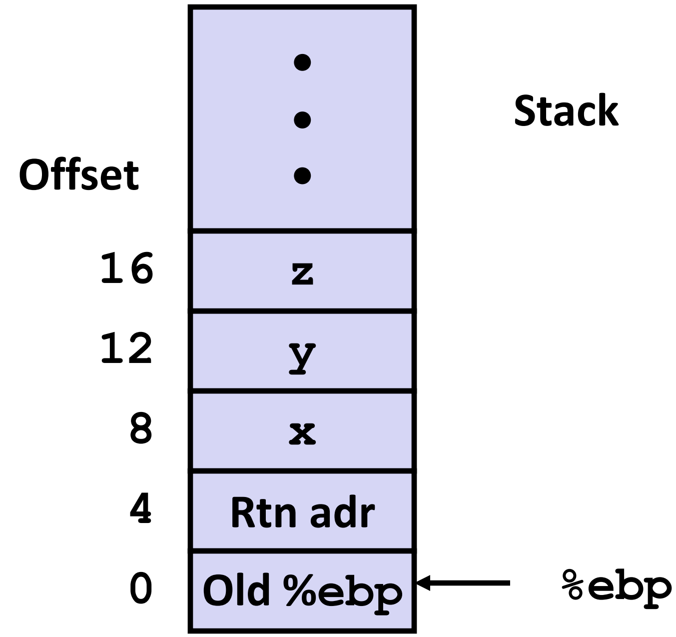
```
int arith
   (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
movl  8(%ebp),%eax          # eax = x
movl  12(%ebp),%edx         # edx = y
leal  (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal  (%edx,%edx,2),%edx    # edx = y + 2*y = 3*y
sall  $4,%edx               # edx = 48*y (t4)
addl  16(%ebp),%ecx         # ecx = z+t1 (t2)
leal  4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax             # eax = t5*t2 (rval)
```

Explication

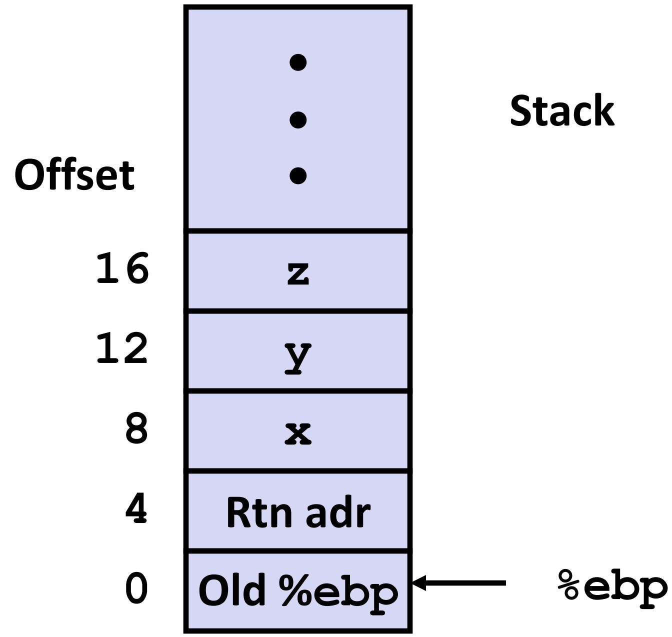
```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```




```
movl  8(%ebp),%eax          # eax = x
movl  12(%ebp),%edx         # edx = y
leal  (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal  (%edx,%edx,2),%edx    # edx = y + 2*y = 3*y
sall  $4,%edx               # edx = 48*y (t4)
addl  16(%ebp),%ecx         # ecx = z+t1 (t2)
leal  4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax             # eax = t5*t2 (rval)
```

Explication

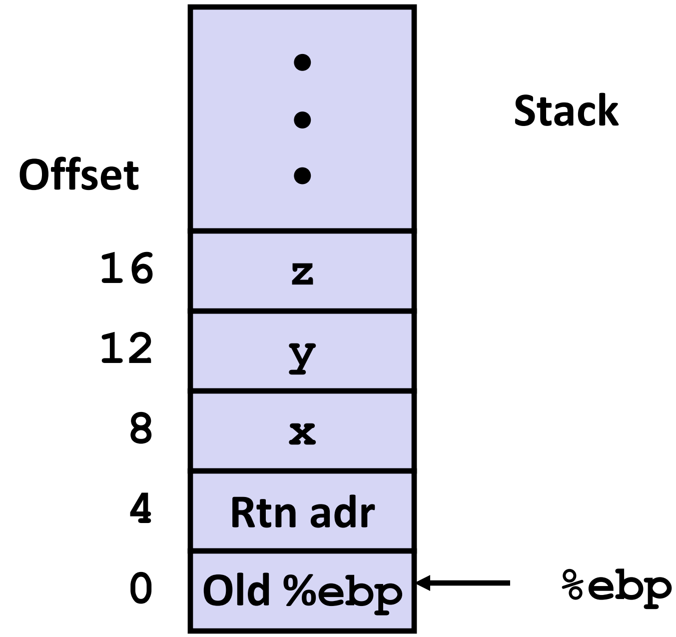
```
int arith
   (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
movl  8(%ebp),%eax          # eax = x
movl  12(%ebp),%edx         # edx = y
leal  (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal  (%edx,%edx,2),%edx    # edx = y + 2*y = 3*y
sall  $4,%edx               # edx = 48*y (t4)
addl  16(%ebp),%ecx         # ecx = z+t1 (t2)
leal  4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax             # eax = t5*t2 (rval)
```

Explication

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```




```
movl  8(%ebp),%eax          # eax = x
movl  12(%ebp),%edx         # edx = y
leal  (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal  (%edx,%edx,2),%edx    # edx = y + 2*y = 3*y
sall  $4,%edx               # edx = 48*y (t4)
addl  16(%ebp),%ecx         # ecx = z+t1 (t2)
leal  4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax             # eax = t5*t2 (rval)
```

Observations

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```

- Instructions en ordre différent de C
- Quelques expressions demandent plusieurs instructions
- Quelques instructions correspondent aux plusieurs expressions
- Même code :
- `(x+y+z)*(x+4+48*y)`

```
movl  8(%ebp),%eax           # eax = x
movl  12(%ebp),%edx          # edx = y
leal  (%edx,%eax),%ecx       # ecx = x+y   (t1)
leal  (%edx,%edx,2),%edx     # edx = y + 2*y = 3*y
sall  $4,%edx                # edx = 48*y (t4)
addl  16(%ebp),%ecx          # ecx = z+t1 (t2)
leal  4(%edx,%eax),%eax      # eax = 4+t4+x (t5)
imull %ecx,%eax              # eax = t5*t2 (rval)
```

Un autre exemple

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

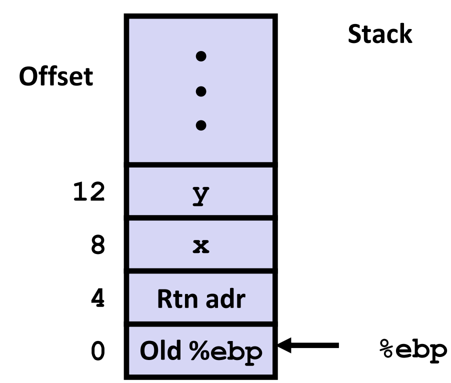
```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up: pushl %ebp, movl %esp,%ebp

Body: movl 8(%ebp),%eax … andl $8185,%eax

Finish: movl %ebp,%esp, popl %ebp, ret

```
movl 8(%ebp),%eax       # eax = x
xorl 12(%ebp),%eax      # eax = x^y
sarl $17,%eax           # eax = t1>>17
andl $8185,%eax         # eax = t2 & 8185
```



Un autre exemple

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

- pushl / movl %esp,%ebp: **Set Up**
- movl, xorl, sarl, andl: **Body**
- movl %ebp,%esp / popl / ret: **Finish**

```
movl 8(%ebp),%eax       eax = x
xorl 12(%ebp),%eax      eax = x^y      (t1)
sarl $17,%eax           eax = t1>>17   (t2)
andl $8185,%eax         eax = t2 & 8185
```

Un autre exemple

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp              } Set
    movl %esp,%ebp          } Up

    movl 8(%ebp),%eax       }
    xorl 12(%ebp),%eax      }
    sarl $17,%eax           }
    andl $8185,%eax         } Body

    movl %ebp,%esp          }
    popl %ebp               } Finish
    ret                     }
```

```
movl 8(%ebp),%eax      eax = x
xorl 12(%ebp),%eax     eax = x^y      (t1)
sarl $17,%eax          eax = t1>>17   (t2)
andl $8185,%eax        eax = t2 & 8185
```

Un autre exemple

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192$, $2^{13} - 7 = 8185$

…0010000000000000, …0001111111111001

```
logical:
    pushl %ebp              } Set Up
    movl %esp,%ebp

    movl 8(%ebp),%eax       }
    xorl 12(%ebp),%eax      }
    sarl $17,%eax           } Body
    andl $8185,%eax         }

    movl %ebp,%esp          }
    popl %ebp               } Finish
    ret                     }
```

```
movl 8(%ebp),%eax       eax = x
xorl 12(%ebp),%eax      eax = x^y      (t1)
sarl $17,%eax           eax = t1>>17   (t2)
andl $8185,%eax         eax = t2 & 8185
```

Cours 4: Programmation Assembler x86

- Modes d'adressage mémoire
- Opérations arithmétiques
- <u>Codes conditionnels</u>
- Branches conditionnelles et inconditionnelles
- Boucles
- Switch

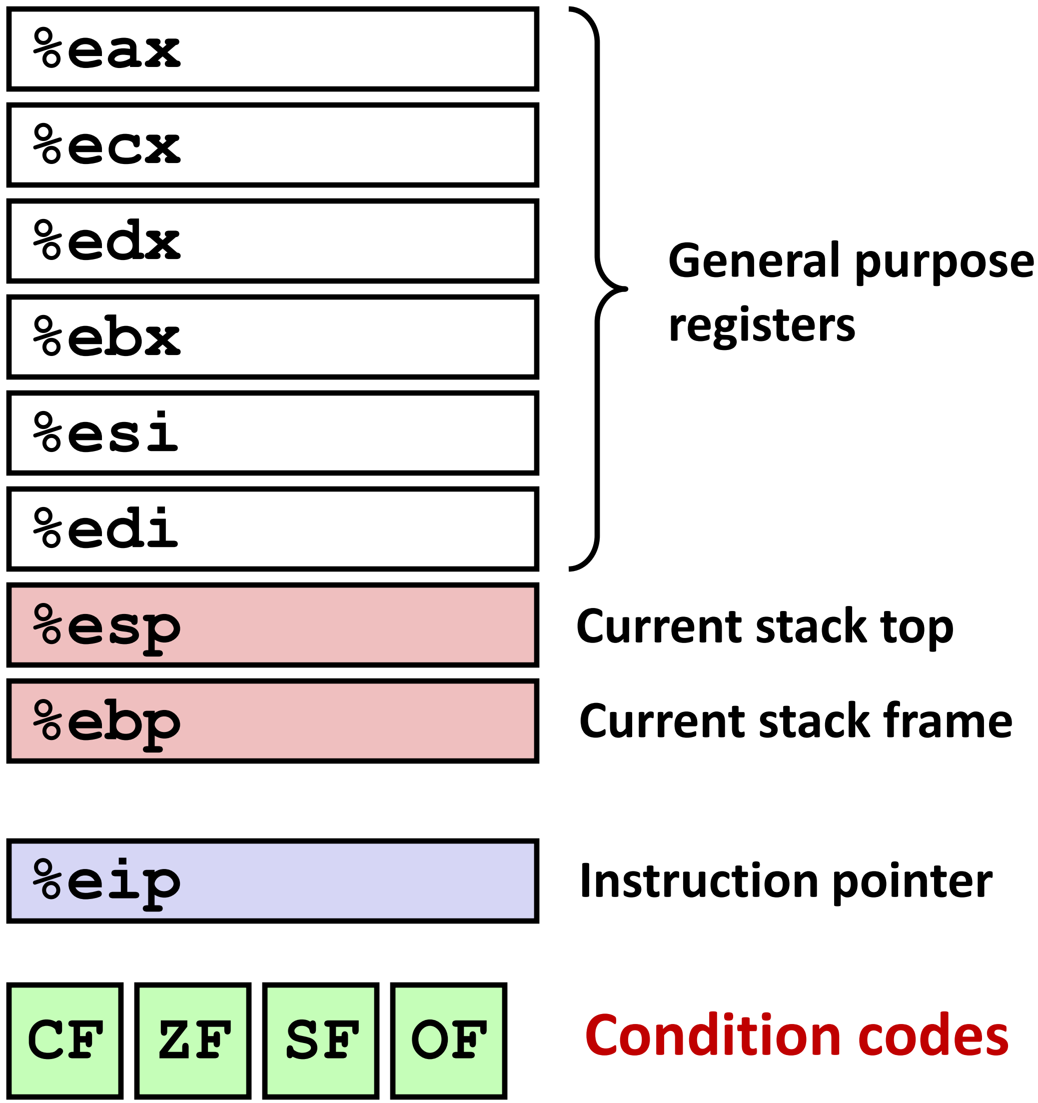
Jumping

- jX Instructions

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

État de processeur (IA32)

- **Information sur le programme en cours d'exécution**
  - Données temporaires (**%eax**, …)
  - Pile d'exécution (**%ebp**, **%esp**)
  - Point de contrôle du code actuel (**%eip**)
  - États de tests récents (**CF**, **ZF**, **SF**, **OF**)



Codes de condition (mis implicitement)

- **Registres d'un bit**

  `CF` Carry Flag (unsigned)   **`SF`** Sign Flag (signed)

  `ZF` Zero Flag   **`OF`** Overflow Flag (signed)

- **Mis implicitement par les opérations arithmétiques**

  Exemple: **`addl/addq`** *Src,Dest* ↔ **`t = a+b`**

  - **CF mis** si retenue (unsigned overflow)
  - **ZF mis** si `t == 0`
  - **SF mis** si `t < 0` (signed)
  - **OF mis** si overflow avec complément à 2
    `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

- ***Ne sont pas mis*** **par l'instruction `lea` !**

Codes de condition (mis explicitement)

- **Registres d'un bit**

  `CF` Carry Flag (unsigned) **`SF`** Sign Flag (signed)

  `ZF` Zero Flag **`OF`** Overflow Flag (signed)

- **Mis explicitement par l'instruction de comparaison**

  `cmpl/cmpq` *Src2,Src1*

  `cmpl b,a` (calculer **`a-b`**)

  - **CF mis** si retenue (unsigned)
  - **ZF mis** si **`a == b`**
  - **SF mis** si **`(a-b) < 0`** (signed)
  - **OF mis** si overflow avec complément à 2
    `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

Codes de condition (mis explicitement)

- **Registres d'un bit**

  `SF` Sign Flag (signed)

  `ZF` Zero Flag

- **Mis explicitement par l'instruction de test**

  `testl`/**`testq`** *Src2,Src1*

  `testl b,a` (calculer **`a & b`**)

  - **ZF mis** si **`a&b == 0`**
  - **SF mis** si **`a&b < 0`**

  - **`testl %eax, %eax`**
    - Met SF et ZF, test si eax est +,0,-

Cours 4: Programmation Assembler x86

- Modes d'adressage mémoire
- Opérations arithmétiques
- Codes conditionnels
- <u>Branches conditionnelles et inconditionnelles</u>
- Boucles
- Switch

Branche conditionnelle : exemple

```c
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff:
    pushl   %ebp                 } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx        ┐
    movl    12(%ebp), %eax       │
    cmpl    %eax, %edx           │
    jle     .L7                  ├ Body1
    subl    %eax, %edx           │
    movl    %edx, %eax           ┘
.L8:
    leave                        } Finish
    ret
.L7:
    subl    %edx, %eax           } Body2
    jmp     .L8
```

Branche conditionnelle : exemple

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
int x          %edx
int y          %eax
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

Branche conditionnelle : exemple

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

int x	%edx
int y	%eax

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

Branche conditionnelle : exemple

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

int x	%edx
int y	%eax

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

Branche conditionnelle : exemple

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

int x	%edx
int y	%eax

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

Branche conditionnelle : exemple

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

int x	%edx
int y	%eax

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

Conditionnelles : x86-64

```
int absdiff(
    int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff: # x in %edi, y in %esi
  movl   %edi, %eax   # eax = x
  movl   %esi, %edx   # edx = y
  subl   %esi, %eax   # eax = x-y
  subl   %edi, %edx   # edx = y-x
  cmpl   %esi, %edi   # x:y
  cmovle %edx, %eax   # eax=edx if <=
  ret
```

- **Instruction move conditionnelle**
  - **`cmov`*C*** src, dest
  - Plus efficace que la branche conditionnelle : pipelining !
  - Mais surcharge : 2 branches sont évaluées

Transfert de control vs transfert de données

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

Goto Version : jX

```
   nt = !Test;
   if (nt) goto Else;
   val = Then_Expr;
   goto Done;
Else:
   val = Else_Expr;
Done:
   . . .
```

Goto Version : cmov

```
   tval = Then_Expr;
   result = Else_Expr;
   t = Test;
   if (t) result = tval;
   return result;
```

```
absdiff Intel Haswell
• jX : 8 - 17.50 clock cycles
• cmov : 8 clock cycles
```

Mauvais cas pour cmove !

Calculs chers

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- **Deux expressions sont calculées**
- **Seulement quand les calculs sont simples !**

Calculs risqués

```
val = p ? *p : 0;
```

- Deux expressions sont calculées
- **Peut avoir des effets non désirables**

Calcul avec des effets de borne

```
val = x > 0 ? x*=7 : x+=3;
```

- Deux expressions sont calculées
- **Ne doit pas avoir des effets de borne**

Cours 4: Programmation Assembler x86

- Modes d’adressage mémoire
- Opérations arithmétiques
- Codes conditionnels
- Branches conditionnelles et inconditionnelles
- <u>Boucles</u>
- Switch

Boucle “Do-While”

C Code

```
int fact_do(int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1) goto loop;
  return result;
}
```

Boucle "Do-While"

Registers:

`%edx`	`x`
`%eax`	`result`

Goto Version

```
int
fact_goto(int x)
{
  int result = 1;


loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;

  return result;
}
```

Assembly

```
fact_goto:
  pushl %ebp            # Setup
  movl %esp,%ebp        # Setup
  movl $1,%eax          # eax = 1
  movl 8(%ebp),%edx     # edx = x

.L11:
  imull %edx,%eax       # result *= x
  decl %edx             # x--
  cmpl $1,%edx          # Compare x : 1
  jg .L11               # if > goto loop

  movl %ebp,%esp        # Finish
  popl %ebp             # Finish
  ret                   # Finish
```

Boucle “While”

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

Goto Version

```
int fact_while_goto(int x)
{
  int result = 1;
  goto middle;
loop:
  result *= x;
  x = x-1;
middle:
  if (x > 1)
    goto loop;
  return result;
}
```

Boucle “While”

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x--;
  };
  return result;
}
```

```
# x in %edx, result in %eax
  jmp   .L34         #
.L35:                # Loop:
  imull %edx, %eax   #   result *= x
  decl  %edx         #   x--
.L34:                #
  cmpl  $1, %edx     #   x:1
  jg    .L35         #   if >, goto
                     #          Loop
```

Boucle “For”: Square-and-Multiply

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int p)
{
  int result;
  for (result = 1; p != 0; p = p>>1) {
    if (p & 0x1)
      result *= x;
    x = x*x;
  }
  return result;
}
```

- **Algorithme**
  - Complexité $O(\log p)$

Boucle “For”: Square-and-Multiply

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int p)
{
   int result;
   for (result = 1; p != 0; p = p>>1) {
      if (p & 0x1)
         result *= x;
      x = x*x;
   }
   return result;
}
```

before iteration	result	x=3	p=10
1	1	3	$10=1010_2$
2	1	9	$5= 101_2$
3	9	81	$2= 10_2$
4	9	6561	$1= 1_2$
5	59049	43046721	0_2

"For"→ "While"

For Version

```
for (Init; Test; Update )

      Body
```

While Version

```
Init;
while (Test ) {
      Body
      Update ;
}
```

Goto Version

```
   Init;
   goto middle;
loop:
   Body
   Update ;
middle:
   if (Test)
      goto loop;
done:
```

Cours 4: Programmation Assembler x86

- Modes d'adressage mémoire
- Opérations arithmétiques
- Codes conditionnels
- Branches conditionnelles et inconditionnelles
- Boucles
- Switch

```
long switch_eg (unsigned
   long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Switch : exemple

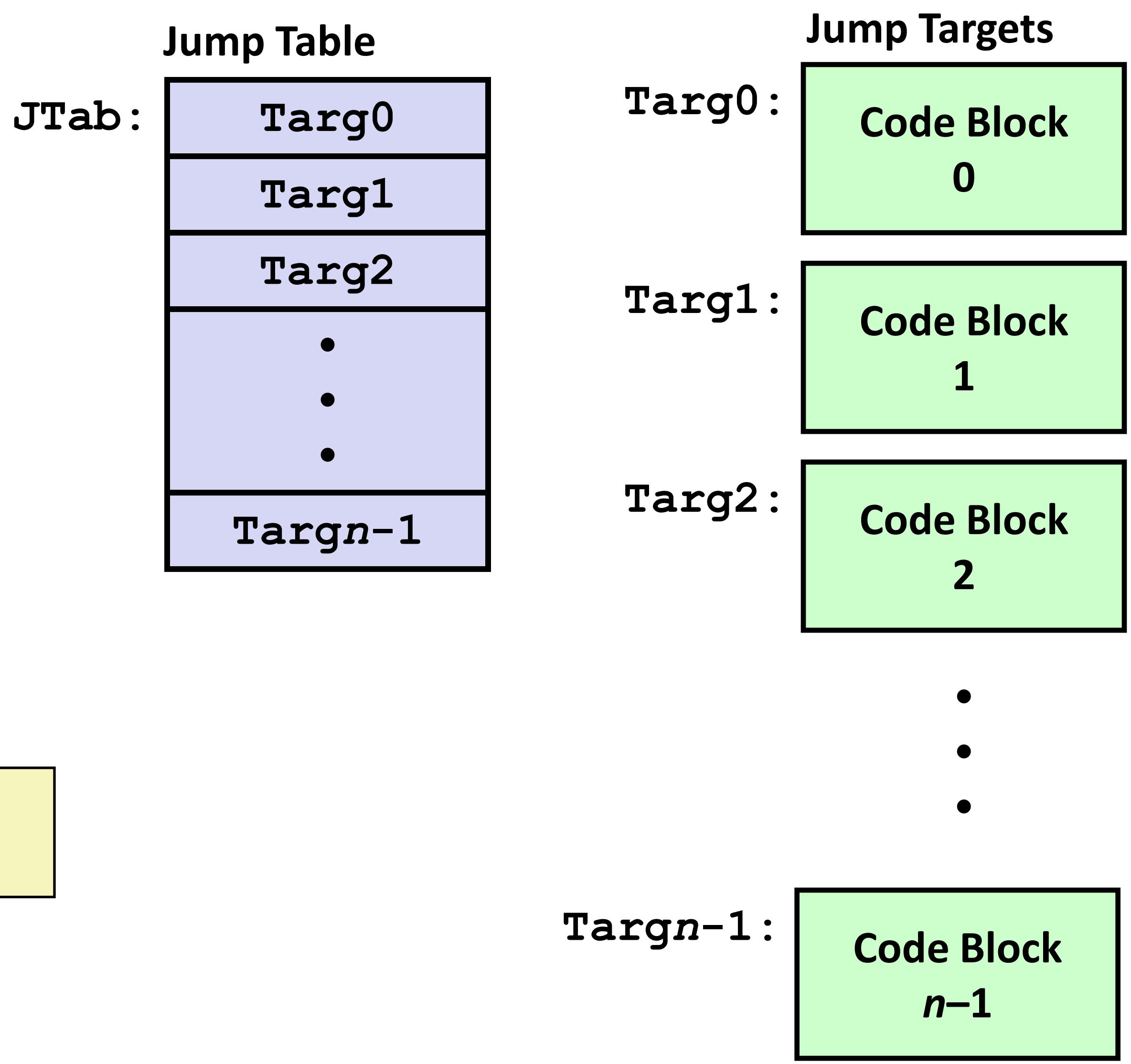
- **Labels multiples**
  - 5, 6
- **Cas “fall through”**
  - 2
- **Cas manquant**
  - 4

- **=> *jump table***

Jump Table

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```



Traduction approximative

```
target = JTab[x];
goto *target;
```

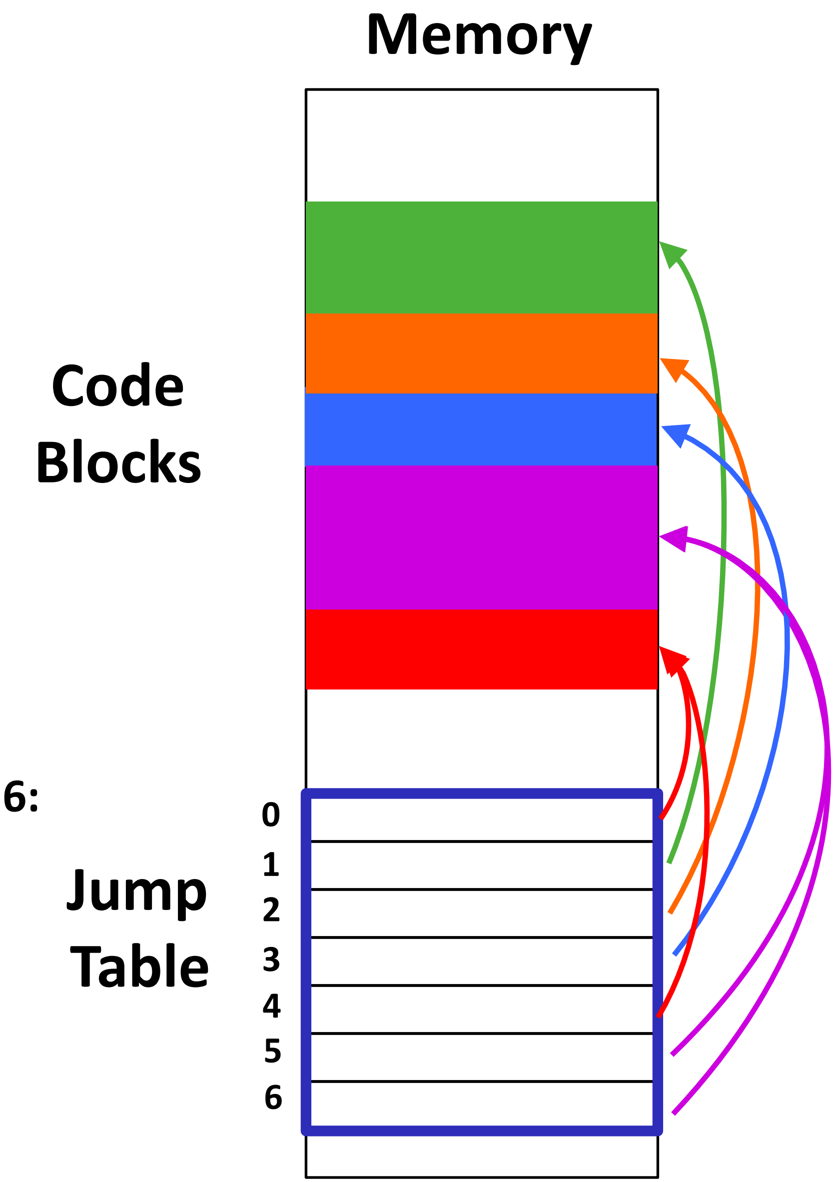
Jump Table

C code:

```
switch(x) {
  case 1: <some code>
          break;
  case 2: <some code>
  case 3: <some code>
          break;
  case 5:
  case 6: <some code>
          break;
  default: <some code>
}
```

Nous pouvons utiliser jump table quand x <= 6:

```
if (x <= 6)
  target = JTab[x];
  goto *target;
else
  goto default;
```



Jump Table

Jump table

```
.section .rodata
   .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

```
switch(x) {
case 1:       // .L56
    w = y*z;
    break;
case 2:       // .L57
    w = y/z;
    /* Fall Through */
case 3:       // .L58
    w += z;
    break;
case 5:
case 6:       // .L60
    w -= z;
    break;
default:      // .L61
    w = 2;
}
```

Switch (IA32)

```
long switch_eg(unsigned long x, long y,
   long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
      pushl  %ebp              # Setup
      movl   %esp, %ebp        # Setup
      pushl  %ebx              # Setup
      movl   $1, %ebx          # w = 1
      movl   8(%ebp), %edx     # edx = x
      movl   16(%ebp), %ecx    # ecx = z
      cmpl   $6, %edx
      ja     .L61
      jmp    *.L62(,%edx,4)
```

Jump table

```
.section .rodata
   .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

Translation?

Switch (IA32)

```
long switch_eg(unsigned long x, long y,
   long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    pushl  %ebp              # Setup
    movl   %esp, %ebp        # Setup
    pushl  %ebx              # Setup
    movl   $1, %ebx          # w = 1
    movl   8(%ebp), %edx     # edx = x
    movl   16(%ebp), %ecx    # ecx = z
    cmpl   $6, %edx          # x:6
    ja     .L61              # if > goto default
    jmp    *.L62(,%edx,4)    # goto JTab[x]
```

Indirect jump → `jmp *.L62(,%edx,4)`

Jump table

```
.section .rodata
   .align 4
.L62:
  .long    .L61  # x = 0
  .long    .L56  # x = 1
  .long    .L57  # x = 2
  .long    .L58  # x = 3
  .long    .L61  # x = 4
  .long    .L60  # x = 5
  .long    .L60  # x = 6
```

Explication

- **Structure de la table**
  - Chaque cible demande 4 octets
  - Adresse de base à **`.L62`**
- **Jumping**
  - **Direct:** **`jmp .L61`**
  - **Indirect:** **`jmp *.L62(,%edx,4)`**
    - Adresse de départ : **`.L62`**
    - Multiplicateur 4 (labels sont de 32-bits = 4 octets en IA32)
    - Adresse effective : **`.L62 + edx*4`**

`target = JTab[x]; goto *target;` (pour $0 \leq \mathbf{x} \leq 6$)

Jump table

```
.section .rodata
   .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

Code Blocks

```
switch(x) {
  . . .
case 2:        // .L57
    w = y/z;
    /* Fall Through */
case 3:        // .L58
    w += z;
    break;
  . . .
default:       // .L61
    w = 2;
}
```

```
.L61:  // Default case
   movl  $2, %ebx     # w = 2
   movl  %ebx, %eax   # Return w
   popl  %ebx
   leave
   ret
.L57:  // Case 2:
   movl  12(%ebp), %eax  # y
   cltd                 # Div prep
   idivl %ecx           # y/z
   movl  %eax, %ebx # w = y/z
# Fall through
.L58:  // Case 3:
   addl  %ecx, %ebx # w+= z
   movl  %ebx, %eax # Return w
   popl  %ebx
   leave
   ret
```

Code Blocks

```
switch(x) {
case 1:         // .L56
    w = y*z;
    break;
  . . .
case 5:
case 6:         // .L60
    w -= z;
    break;
  . . .
}
```

```
.L60: // Cases 5&6:
  subl  %ecx, %ebx   # w -= z
  movl  %ebx, %eax   # Return w
  popl  %ebx
  leave
  ret
.L56: // Case 1:
  movl  12(%ebp), %ebx # w = y
  imull %ecx, %ebx     # w*= z
  movl  %ebx, %eax   # Return w
  popl  %ebx
  leave
  ret
```

Question

- **Allez vous implémenter ce switch avec un jump table?**

```
switch(x) {
  case 0:     <some code>
              break;
  case 10:    <some code>
              break;
  case 52000: <some code>
              break;
  default:    <some code>
              break;
}
```

  - Jump table avec 52001 entrées ?