Vite fait, bien fait!

Bernard Goossens

ELIAUS-DALI

Plan de l'exposé

- Descendre du source jusqu'à l'architecture
  - pour produire un code correct
  - pour produire un code rapide
- Multiplications et divisions des entiers en machine
- Flottant IEEE 754 et architecture x86_64
  - contrôler la précision
  - contrôler l'arrondi
  - contrôler les exceptions
- Mesurer précisément le temps d'exécution
  - sur les architectures x86_64

Exemple élémentaire

```
char max(char x, char y){
  if (x>y) return x;
  else return y;
}

max(130,1)=
```

Exemple élémentaire

```
char max(char x, char y){
  if (x>y) return x;
  else return y;
}
```

max(130,1)= 1

Exemple élémentaire

```
char max(char x, char y){
  if (x>y) return x;
  else return y;
}

max(130,128)=
```

Exemple élémentaire

```
char max(char x, char y){
  if (x>y) return x;
  else return y;
}
```

max(130,128)= -126

Exemple élémentaire

```
unsigned char umax(unsigned char x,
                                unsigned char y){
  if (x>y) return x;
  else return y;
}

umax(130,1)= 130
umax(130,128)= 130
```

Exemple élémentaire

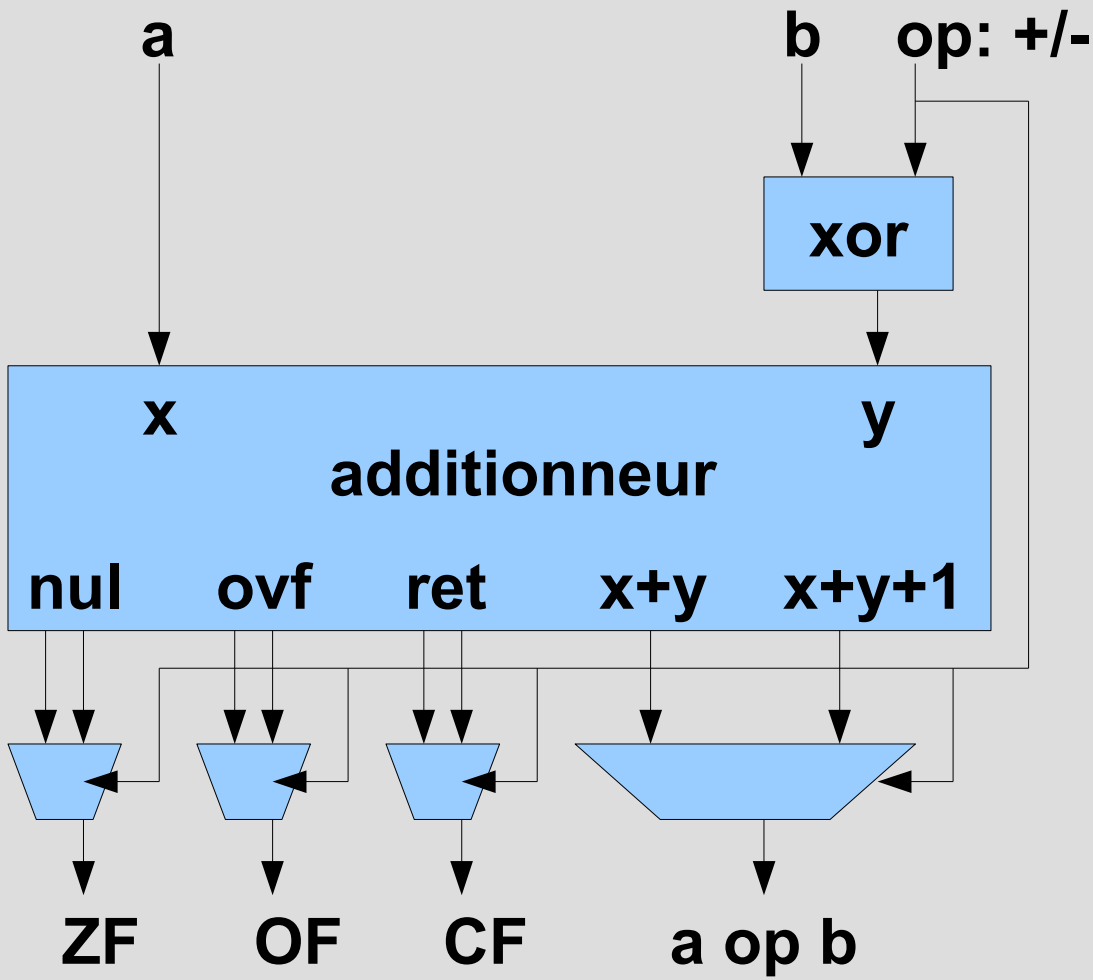
```
max.c:      max.s:

x           movsbl   %dil,%eax
y           movsbl   %sil,%edx
x-y         cmpb     %sil,%dil
            cmovle   %edx,%eax      //max=y ssi ZF v (SF!=OF)

umax.c:     umax.s:

x           movzbl   %dil,%eax
y           movzbl   %sil,%edx
x-y         cmpb     %sil,%dil
            cmovbe   %edx,%eax      //max=y ssi ZF v CF
```

Indicateurs EFLAGS



Le même exemple en Ada

```
subtype Signed_Byte is Integer range -128..+127;
function max(x, y: Signed_Byte) return Signed_Byte is
 begin
  if (x>y) then return x;
           else return y;
  end if;
end max;
```

max(130,1)= Constraint Error

Le même exemple en Ada

```
subtype Unsigned_Byte is Integer range 0..255;
function umax(x, y: Unsigned_Byte) return Unsigned_Byte is
 begin
  if (x>y) then return x;
           else return y;
  end if;
end umax;
```

max(130,1)= 130

Première leçon de l'exemple

- En C, le contrôle de type n'est pas strict.
  La programmation n'est pas sure.
- En Ada, c'est mieux.
- Le typage est une abstraction qui semble permettre de s'affranchir de l'architecture.
- Mieux fait, mais aussi vite fait?

Un autre exemple en Ada

```
subtype Signed_Byte is Integer range -128..+127;
function add(x, y: Signed_Byte) return Signed_Byte is
 begin
  return x+y;
end add;
```

```
leal    (%rbx,%r12),%esi      //x+y (add 64 bits signée)
leal    128(%rsi),%eax        //x+y+128 (sadd64)
cmpl    $255,%eax             //x+y+128  > 255
ja      .L18                  //x+y>+127
...                           //x+y<=+127
```

Seconde leçon de l'exemple

- S'appuyer aveuglément sur les types conduit à du code peu efficace.
- En Ada, l'implémentation Gnat propose un type Short_Short_Integer, mappé sur le type char de C.
- Le bon choix est l'implémentation la plus rapide, accompagnée des contrôles nécessaires à la sureté.
- Il est préférable de ne pas faire l'économie de la connaissance de l'architecture.

Multiplication des entiers

```
int mul10(int a){
  return a*10;
}

mul10:    leal    (%rdi,%rdi,4),%edi      //edi=4a+a
          leal    (%rdi,%rdi),%eax        //eax=5a+5a

int mul11(int a){
  return a*11;
}

mul11:    leal    (%rdi,%rdi,4),%eax      //eax=4a+a
          leal    (%rdi,%rax,2),%eax      //eax=10a+a
```

Multiplication des entiers

```
int mul28(int a){return a*28;}
mul28:    movl  %edi,%eax      //eax=a
          movl  $28,%edx       //edx=28
          imull %edx,%eax      //eax=a*28

int mul33(int a){return a*33;}
mul33:    movl  %edi,%eax      //eax=a
          sall  $5,%eax        //eax=a>>5=a*32
          addl  %edi,%eax      //eax=a+32a

int mul(int a, int b){return a*b;}
mul:      imull %edi,%esi      //esi=a*b
```

Division des entiers

```
int div10(int a){return a/10;}

div10:  movl  %edi,%eax          //eax=a
        movl  $1717986919,%edx   //edx=0,4
        sarl  $31,%edi           //edi=(a<0)?-1:0
        imull %edx               //edx:eax=0,4*a
        sarl  $2,%edx            //edx=0,4*a>>2=a/10
        subl  %edi,%edx          //edx-=(a<0)?1:0 (arrondi)
        movl  %edx,%eax          //eax=a/10
```

$1717986919 = (0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0111)_2$

$0.0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0111 = 0{,}40000000014_{10}$

Division des entiers

```
int div123456789(int a){return a/123456789;}

div1_9:   movl  %edi,%eax              //eax=a
          movl  $-1960301227,%edx      //edx=-0,456418196
          imull %edx                   //edx:eax=a*(-0,456418196)
          leal  (%rdx,%rdi),%eax       //eax=a*0,543581804
          sarl  $31,%edi               //edi=(a<0)?-1:0
          sarl  $26,%eax               //eax=0,54a>>26
                                       //eax=a/123456789
          subl  %edi,%eax              //eax-=(a<0)?1:0 (arrondi)
```

$-1960301227 = (1000\ 1011\ 0010\ 1000\ 0010\ 1101\ 0101\ 0101)_2$

$1000\ 1011\ 0010\ 1000\ 0010\ 1101\ 0101\ 0101 = -0,456418196_{10}$

Division des entiers

```
#include "signal.h"
int d=0;
void handler(int signum){
 printf("division par 0\n");
 d=1;
}
main(){
 signal(SIGFPE,handler);
 printf("8/%d=%d\n",d,8/d);
}
```

Division des entiers

```
#include "signal.h"
int d=0;
void handler(int signum){
 printf("division par 0\n");
 d=1;
}
main(){
 signal(SIGFPE,handler);
 printf("8/%d=%d\n",d,8/d);
}
```

Ca boucle! (division par 0, ...)

Division des entiers

```
main:   ...
        movl  d(%rip),%ecx    //ecx=d
        movl  $8,%eax         //eax=8
        idivl %ecx            //eax=8/d
        ...                   //exception

handler: ...
        movl  $1,d(%rip)      //d=1
        ...
```

On modifie « d » dans « handler » mais l'instruction à reprendre utilise une copie de « d » en registre.

Division des entiers

```
#include "signal.h"
int d=0,q;
void handler(int signum){
 printf("division par 0\n");
 d=1;
}
main(){
 signal(SIGFPE,handler);
 asm("movl $8,%%eax" : : );
 asm("idivl %[d]" : : [d] "m" (d));
 asm("movl %%eax,%[q]" : [q] "=m" (q) : );
 printf("8/%d=%d\n",d,q);
}
```

Multiplication et division des entiers

- La division « a/k » est calculée par « a*k' » avec « k'=1/k » calculée par le compilateur.
- La multiplication « a*b » est calculée par décalages et additions quand l'un ou l'autre de « a » et « b » est proche d'une puissance de 2.
- Attention à l'interaction entre une exception et son gestionnaire: celui-ci ne peut agir sur les variables du programme principal copiées en registre au moment de l'exception.

L'implémentation x86_64 de l'IEEE 754

- X87: format 80 bits: (1,15,64) avec partie entière explicite.
  8 registres 80 bits en pile ST(0)-ST(7) (SP=ST(0))
  FADD, FSUB, FMUL, FDIV, FSQRT
  exemple: FADDP //ST(0)+ST(1)->ST(1); POP
- SSE: format simple précision IEEE 754.
  16 registres 128 bits XMM0-XMM15 (x86_64)
  1 donnée scalaire (S) ou 4 données vectorielles (P)
  ADDxS, SUBxS, MULxS, DIVxS, SQRTxS
  exemple: ADDSS %XMM0, %XMM1 //XMM0+XMM1->XMM0
- SSE2: format double précision IEEE 754.
  16 registres 128 bits XMM0-XMM15 (x86_64)
  1 donnée scalaire (S) ou 2 données vectorielles (P)
  ADDxD, SUBxD, MULxD, DIVxD, SQRTxD
  exemple: ADDSD %XMM0, %XMM1 //XMM0+XMM1->XMM0

Précision des calculs en x87

- L'architecture x87 fonctionne en précision étendue (80 bits).

- Les données chargées de la mémoire en registre ou pour un calcul sont automatiquement étendues (sp->ep ou dp->ep).
- La précision des calculs n'est pas modifiable (ep, 80 bits).
- La précision du résultat lors de son écriture en registre ou en mémoire est paramétrable (sp, dp ou ep).

- Le résultat écrit en registre conserve une mantisse de 64 bits, arrondie selon la précision. L'exposant est exprimé en ep.
- Le résultat écrit en mémoire est converti pour avoir la taille de la destination (ep: 10 octets, dp: 8 octets, sp: 4 octets).

Calculer en ep n'est pas IEEE standard

X87 ep:

double x,y,z;

$x = 1.0 + 2^{-52}$;	**1.000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1000 0000 0000**	**000 (grt)**
$y = 2^{-53} - 2^{-69}$;	**0.000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0011 1111 1111**	**111 (grt)**
x + y;	**1.000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1011 1111 1111**	**111 (grt)**
	1.000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1100 0000 0000	**arrondi**
z = x + y;	**1.000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 0**	

IEEE dp:

$X = 1.0 + 2^{-52}$;	**1.000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1**	**000 (grt)**
$y = 2^{-53} - 2^{-69}$;	**0.000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0**	**011 (grt)**
x + y;	**1.000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1**	**011 (grt)**
	1.000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1	**arrondi**
z = x + y;	**1.000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1**	

Contrôler la précision des calculs x87

- La précision des calculs des instructions FADD, FSUB, FMUL, FDIV et FSQRT peut être imposée.
- Le registre CW contient deux bits indicatifs de la précision applicable (bits 8 et 9).
- 00: sp, 01: indéfini, 10: dp, 11: ep.
- Par défaut, CW[9,8]=11.
- Pour fixer sp, CW = CW & 0xfffffcff.
- Pour fixer dp, CW = (CW & 0xfffffeff) | 0x200.
- Pour fixer ep, CW = CW | 0x300.

Contrôler le mode d'arrondi du x87

- Le mode d'arrondi peut être imposé.
- Le registre CW contient deux bits indicatifs du mode d'arrondi applicable (bits 10 et 11).
- 00: plus près, 01: -∞, 10: +∞, 11: zéro.
- Par défaut, CW[11,10]=00.
- Pour fixer l'arrondi au plus près: CW = CW & 0xfffff3ff.
- Pour fixer l'arrondi vers -∞: CW = (CW & 0xfffff7ff) | 0x400.
- Pour fixer l'arrondi vers +∞: CW = (CW & 0xffffbff) | 0x800.
- Pour fixer l'arrondi vers zéro: CW = CW | 0xc00.

Pièges du x87

- Quand CW fixe la précision à sp ou dp, les opérandes du calcul sont en ep et la mantisse du résultat est arrondie à la précision fixée. L'exposant reste en précision ep (le résultat peut être infini, dénormalisé ou NaN en dp, mais pas en ep: le signalement par une exception est retardé jusqu'à l'écriture du résultat en mémoire).

- Quand on veut que la précision soit fixée à sp ou dp, il est prudent de réajuster CW à chaque retour d'une fonction de librairie si elle est susceptible de revenir à la précision par défaut ep.

- Initialiser CW est couteux (deux accès mémoire).

Contrôler les exceptions flottantes

- Les exceptions flottantes sont signalées dans le registre d'état SW.

- SW[0]: Invalid operation (par exemple, 0/0),
  SW[1]: Denormal operand,
  SW(2]: Divide by zero,
  SW[3]: Overflow,
  SW[4]: Underflow,
  SW[5]: Precision (résultat arrondi),
  SW[6]: Stack fault,
  SW[7]: Summary.

- Une exception met à 1 le bit de SW concerné et SW[7].

Contrôler les exceptions flottantes

- Les exceptions flottantes peuvent être masquées. Le registre de contrôle CW regroupe les bits de masquage.

- CW[0]: Invalid operation (par exemple, 0/0),
  CW[1]: Denormal operand,
  CW(2]: Divide by zero,
  CW[3]: Overflow,
  CW[4]: Underflow,
  CW[5]: Precision (résultat arrondi).

- Par défaut, toutes les exceptions sont masquées (CW[5,0]=0).

Contrôler les exceptions flottantes

Voici une macro pour démasquer les exceptions d'overflow:

```
#define UNMASK_OM(CW){                                          \\
    asm("fstcw %[cw]" : [cw] "=m" (CW));                        \\
    asm("andl $0xfffffff7,%[cw]" : [cw] "+m" (CW));             \\
    asm("fldcw %[cw]" : : [cw] "m" (CW));                       \\
}

// effectue CW[3]=0
```

Contrôler les exceptions flottantes

```
#include "emask.h"
main(){
    int cw;
    union {float f; int i;} u1, u2, u3;
    UNMASK_OM(cw);
    u1.i = 0x7f7fffff;
    u2.i = 0x7f000001;
    u3.f = u1.f + u2.f;//u3.f=+∞
}
```

Exception flottante signalée.

Contrôler les exceptions flottantes

```
#include "emask.h"
#include <signal.h>
void handler_fpe(int signum){
      CLEAR_OE();
      printf("notre handler\n");
      exit(0);
}
main(){
      int cw;
      union {float f; int i;} u1, u2, u3;
      signal(SIGFPE, handler_fpe);
      UNMASK_OM(cw);
      u1.i = 0x7f7fffff;
      u2.i = 0x7f000001;
      u3.f = u1.f + u2.f;
}
```

Contrôler les exceptions flottantes

```
handler_fpe:    ...
                andl    $0xffffff77,0x1d2(%rbp)    //SW[3]=0: copie en pile
                ...                                //SW[7]=0

main:           ...
                fstcw   -4(%rbp)
                andl    $0xfffffff7,-4(%rbp)       //CW[3]=0 (OMask)
                fldcw   -4(%rbp)
                ...
                flds    -16(%rbp)                  //u1.f
                flds    -32(%rbp)                  //u2.f
                faddp   %st,%st(1)                 //pas d'exception
                fstps   -48(%rbp)                  //exception levée
                flds    -48(%rbp)                  //exception déclenchée
                ...
```

Contrôler les exceptions flottantes

```
#include "emask.h"
#include <signal.h>
int i=0;
void handler_fpe(int signum){
        CLEAR_OE();
        printf("i=%d\n",i);
}
main(){
        int cw;
        union {float f; int i;} u1, u2, u3;
        signal(SIGFPE, handler_fpe);
        UNMASK_OM(cw);
        u1.i = 0x7f7fffff;
        u2.i = 0x7f000001;
        u3.f = u1.f + u2.f;
        i++;
        WAIT();
        printf("i=%d\n",i);
}

i=1
i=1
```

Contrôler les exceptions flottantes

```
handler_fpe:  ...
              andl    $0xffffff77,0x1d2(%rbp)   //SW[3]=0: copie en pile
              ...                               //SW[7]=0

main:         ...
              fstcw   -4(%rbp)
              andl    $0xfffffff7,-4(%rbp)      //CW[3]=0 (OMask)
              fldcw   -4(%rbp)
              ...
              flds    -16(%rbp)                 //u1.f
              flds    -32(%rbp)                 //u2.f
              faddp   %st,%st(1)                //pas d'exception
              fstps   -48(%rbp)                 //exception levée
              addl    $1, i(%rip)               //i++
              wait                              //exception déclenchée
              ...
```

Contrôler les exceptions flottantes

```
#include "emask.h"
#include <signal.h>
void handler_fpe(int signum){
	CLEAR_OE();
	ZERO_TOP();//0 dans rbp+[0x1f0-0x1f9], copie de st(0)
	RESTART();//copier rbp+0x1d8 dans rbp+0x1ae, reprise de faddp
}
main(){
	int cw;
	union {float f; int i;} u1, u2, u3;
	signal(SIGFPE, handler_fpe);
	UNMASK_OM(cw);
	u1.i = 0x7f7fffff;
	u2.i = 0x7f000001;
	u3.f = u1.f + u2.f;
	WAIT();
}
```

Particularités des exceptions x87

- Les calculs étant effectués en précision étendue, les exceptions ne sont signalées qu'à l'écriture du résultat en mémoire (instruction « fstp »).
- Pour des raisons historiques, le gestionnaire d'exception n'est pas activé par l'instruction levant l'exception, mais par l'instruction x87 suivante.
- Entre la levée de l'exception et le déroutement vers le gestionnaire, des instructions entières postérieures peuvent être exécutées. On peut l'empêcher en forçant le déclenchement du gestionnaire par « wait ».
- Pour retourner du gestionnaire à l'instruction fautive, il faut manipuler la pile du gestionnaire (copie des registres x87 et adresse de retour).

Mesurer le temps d'exécution

```
unsigned char x[8][8],y[8][8],z[8][8];
int i,j;
saddb(){
 for(i=0;i<8;i++)
  for(j=0;j<8;j++)
   z[i][j]=x[i][j]+y[i][j];
}
main(){
 ... //initialisation de x et y
 saddb();
 ... //affichage de z
}
```

La fonction « saddb » additionne les matrices d'octets « x » et « y », octet par octet.

Mesurer le temps d'exécution

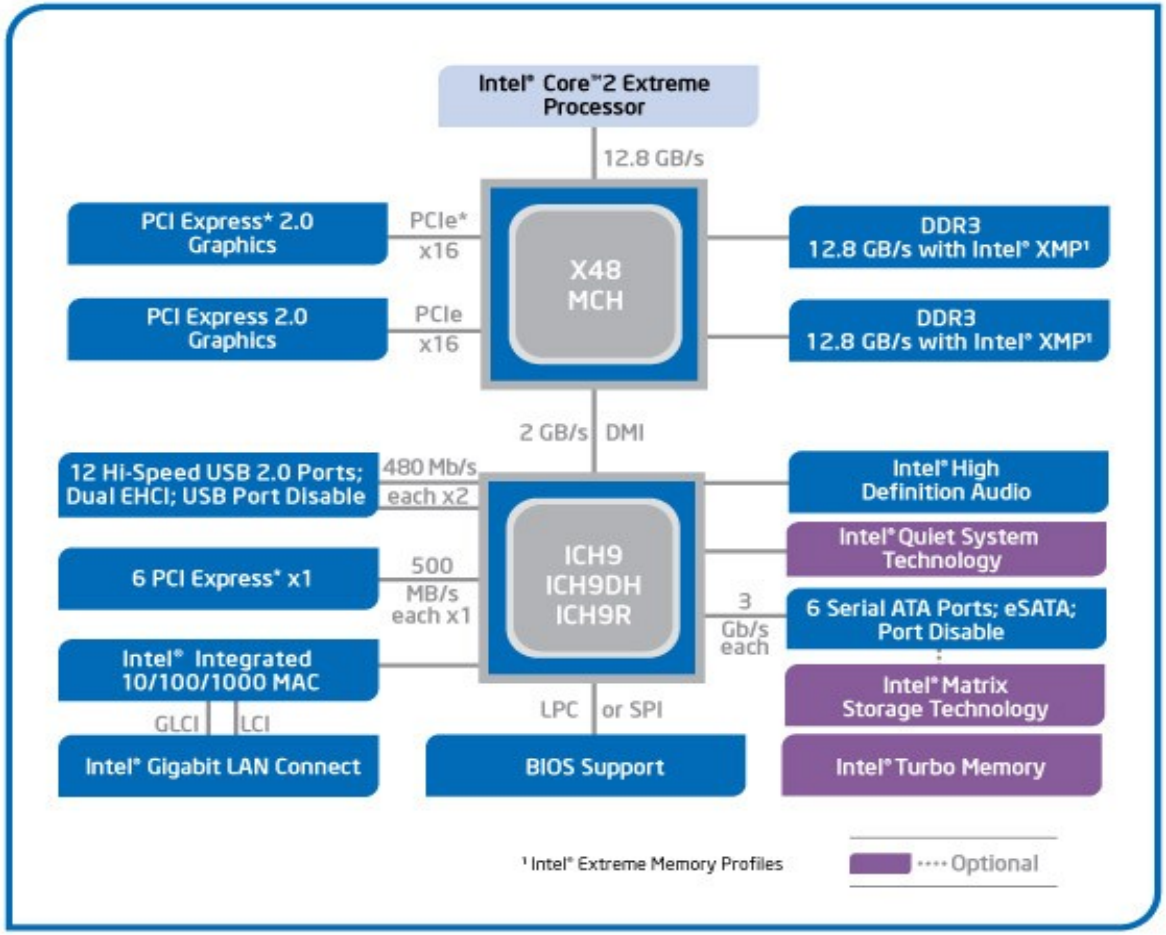
```
typedef unsigned char v8qi __attribute__ (vector_size(8));
typedef union {v8qi p[8]; unsigned char s[8][8];} V8;
V8 x,y,z;
int i;
vaddb(){
 for(i=0;i<8;i++)
  z.p[i]=__builtin_ia32_paddb (x.p[i],y.p[i]);
}
main(){
 ... //initialisation de x.s et y.s
 vaddb();
 ... //affichage de z.s
}
```

La fonction « vaddb » additionne les matrices d'octets « x » et « y », ligne par ligne.

Mesurer le temps d'exécution

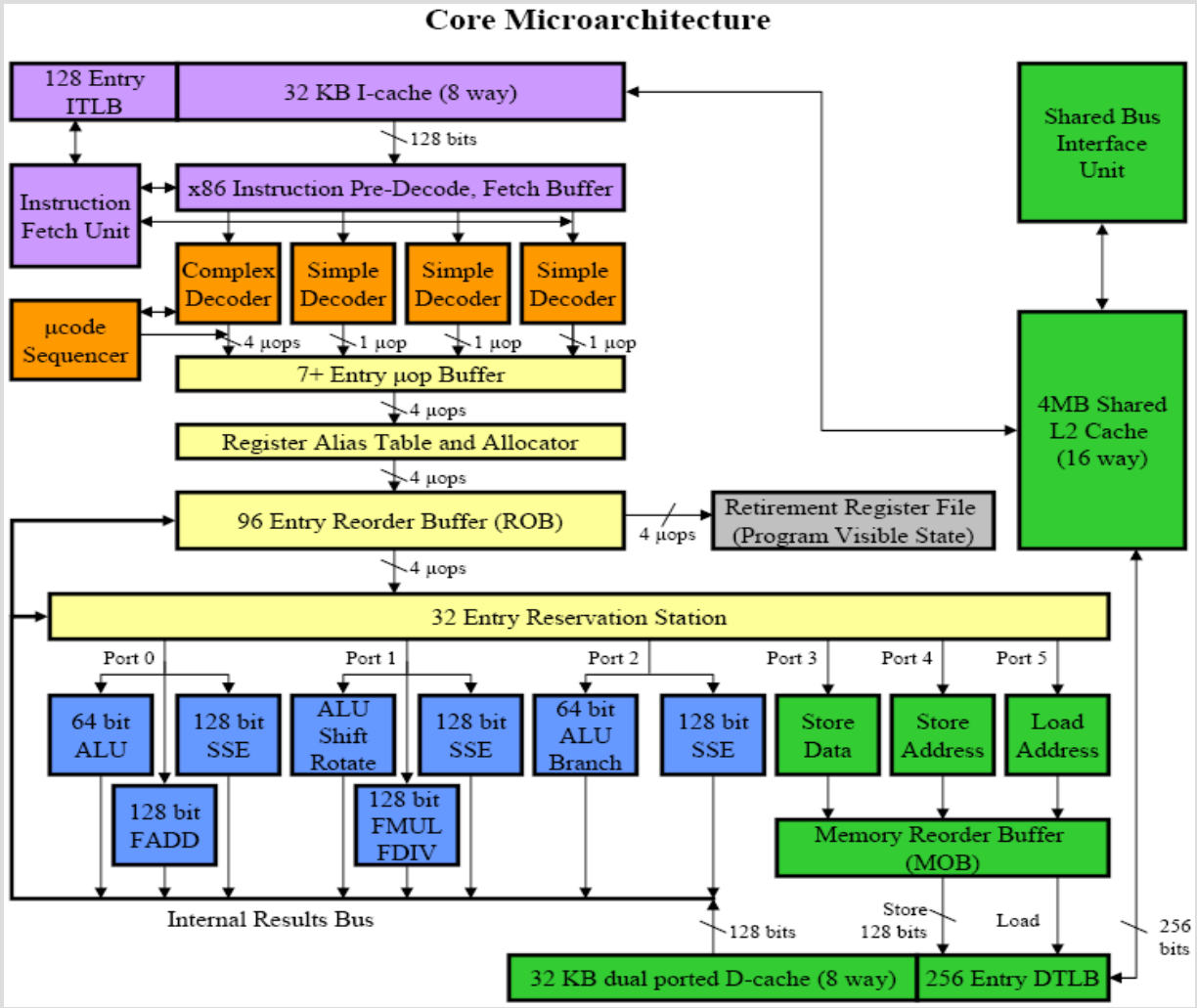
- Nous voulons comparer les vitesses d'exécution en cycles des fonctions « saddb » et « vaddb ».
- Comment isoler le temps passé dans le cœur du processeur, pour ne compter que l'exécution des instructions et éliminer tous les événements parasites, systèmes et matériels?
- Comment observer que « vaddb » est bien 8 fois plus rapide que « saddb » en procédant par ligne?
- Les processeurs « Core » sont équipés de compteurs observables et programmables.

Le processeur et ses périphériques



Intel® X48 Express Chipset Block Diagram

L'organisation du processeur



La sérialisation des exécutions

- Les instructions s'exécutent en ordre partiel de dépendances.
- Les instructions de relevé des compteurs doivent le moins possible se mélanger à la portion de code à mesurer.
- L'instruction CPUID est sérialisante: une fois décodée, elle entre dans le buffer de micro-opérations et y attend que le ROB soit vide (terminaison et validation de toutes les insructions précédentes).
- Dès qu'elle reprend sa progression (entre en ROB), le décodage des instructions qui suivent redémarre.
- L'instruction CPUID ajoute un délai de quelques centaines de cycles.

Organisation générale d'une mesure

```
xorl    %%eax,%%eax      //eax=0
cpuid
/*lecture compteur dans STAMP0*/
/*lecture compteur dans STAMP1*/
/*délai=STAMP1-STAMP0*/
/*lecture compteur dans STAMP0*/
/*portion de code à mesurer*/
xorl    %%eax,%%eax      //eax=0
cpuid
/*lecture compteur dans STAMP1*/
/*durée=STAMP1-STAMP0-délai*/
```

Le compteur TSC

- Time Stamp Counter.
- Il se situe à la frontière du FSB.
- Il s'incrémente à chaque cycle bus (cycle/11 sur mon Core2 Quad 6600 à 2.4Ghz).
- On le consulte en mode « user » avec l'instruction RDTSC qui reporte le contenu du compteur dans les registres EDX (32 bits forts) et EAX (32 bits faibles).
- Le contenu est exprimé en « cycles processeurs », donc en multiple de cycles bus.
- La lecture est effectuée par une micro-opération d'accès mémoire (compteur mappé dans l'espace mémoire).
- Elle prend une centaine de cycles et n'est pas sérialisante.

Les 5 compteurs

- 3 compteurs fixes et 2 compteurs programmables.
- CTR0 (0x309) (instruction retired) compte le nombre d'instructions x86 retirées (à chaque cycle, on lui ajoute le nombre d'instructions x86 sortant du ROB).
- CTR1 (0x30a) (unhalted core cycles) s'incrémente à chaque cycle processeur.
- CTR2 (0x30b) (unhalted reference cycles) s'incrémente à chaque cycle bus. Ce compteur est en gros une copie de TSC.
- On lit ces compteurs avec l'instruction RDMSR, en fournissant l'adresse du compteur à lire dans le registre ECX (par exemple: movl $0x30a,%ecx).
- Le contenu du compteur est reporté dans les registres EDX (32 bits forts) et EAX (32 bits faibles).

Les 2 compteurs programmables

- PerfCtr0 (0xc1) et PerfCtr1 (0xc2).
- Chaque compteur peut être programmé pour compter de façon conditionnelle.
- La condition d'incrémentation du compteur est précisée par un registre de sélection d'événement.
- PerfEvtSel0 (0x186) pour PerfCtr0 et PerfEvtSel1 (0x187) pour PerfCtr1.
- Chaque registre PerfEvtSel fixe un événement ([7-0]), des drapeaux de déclinaisons de l'événement ([15-8]), des indicateurs de mise en route ([23-16]) et un seuil d'activation du comptage ([31-24]).
- Le compteur est incrémenté à chaque événement, si le seuil est atteint.
- Par exemple, on peut compter le nombre de cycles où aucune instruction n'est lancée (« stall cycle ») en fixant un seuil < à 1.

Le compteur CTR1

```
for (i=0;i<4;i++{//répéter 4 fois pour chauffer les caches
//on ne garde que la mesure du dernier tour

  //lecture du compteur et sauvegarde en mémoire dans STAMP0
  asm ("movl $0x30a,%%ecx" : : : "ecx");
  asm ("rdmsr" : : : "ecx", "eax", "edx");
  asm ("movl %%eax, %[stamp0]" : [stamp0] "=m" (stamp0) : : "eax");

  //sérialisation
  asm ("xorl %%eax,%%eax" : : : "eax");
  asm ("cpuid" : : : "eax", "ebx", "ecx", "edx");
  //portion de code à mesurer (sadd())
  for (j=0;j<8;j++){
        z[j][0]=x[j][0]+y[j][0];

        z[j][7]=x[j][7]+y[j][7];
  }
  //sérialisation
  asm ("xorl %%eax,%%eax" : : : "eax");
  asm ("cpuid" : : : "eax", "ebx", "ecx", "edx");

  //lecture du compteur et sauvegarde en mémoire dans STAMP1
  asm ("movl $0x30a,%%ecx" : : : "ecx");
  asm ("rdmsr" : : : "ecx", "eax", "edx");
  asm ("movl %%eax, %[stamp1]" : [stamp1] "=m" (stamp1) : : "eax");
}
print_string("# core cycles elapsed");
print_int(stamp1-stamp0+1);
```

Le module noyau

```
int init_module(void)
{
        int i, j;
        int stamp0,stamp1;

        initx();
        inity();

        for (i=0;i<8;i++){
          //mesure
        }
        //affichage par « current->signal->tty->driver->write »
        print_string("# core cycles elapsed");
        print_int(stamp1-stamp0+1);
}
```

Makefile: obj-m += count_core_cycles_sadd.o

Compiler (root) avec:

```
#  make -C /usr/src/kernels/2.6.24.7-92.fc8-x86_64 SUBDIRS=$PWD modules
```

Exécuter avec (insmod exécute init_module(), rmmod exécute cleanup_module()):

```
# /sbin/insmod count_core_cycles_sadd.ko
# core cycles elapsed
697
# /sbin/rmmod count_core_cycles_sadd.ko
```

Résultat

- Le module «count_core_cycles_sadd.ko » s'exécute en 697 cycles.
- En ôtant la fonction « sadd », il s'exécute en 563 cycles.
- La fonction « sadd » prend 697-563 = 134 cycles.
- 8 tours de chacun 16 cycles et 6 cycles d'épilogue.
- Le port de lecture en mémoire est la ressource critique. Les 16 cycles sont l'enchainement des 16 lectures (une ligne de « x » et une ligne de « y »).
- Le module «count_core_cycles_vadd.ko » s'exécute en 580 cycles.
- La fonction « vadd » prend 580-563 = 17 cycles.
- 16 cycles (16 lectures) et 1 cycle d'épilogue.

Conclusion sur les mesures de cycles

- Les fonctions système de calcul des temps s'appuient sur un timer extérieur au processeur, cadencé 200 fois plus lentement. Cela ajoute aussi des parasites liés à une communication très indirecte entre le processeur et le timer (via MCH et ICH).
- Le TSC mesure des cycles bus, 10 fois plus lents que ceux du processeur. Par ailleurs, la durée des cycles du processeur peut varier pour économiser la consommation.
- Le CTR1 mesure des cycles processeur (on ne peut pas déduire un temps de calcul puisque la durée du cycle varie). Il est très précis, mais ne peut être manipulé qu'en mode « kernel ».

Merci de votre patience!

Questions?