Exercice 1 : pgcd : algorithme d'Euclide

On peut calculer le pgcd (Plus Grand Commun Diviseur) de deux nombres entiers par l'algorithme d'Euclide. Par exemple

```
pgcd(49,91)=pgcd(91,49%91)=pgcd(91,49)=pgcd(49,91%49)=
pgcd(49,42)=pgcd(42,49%42)=pgcd(42,7)=pgcd(7,42%7)=pgcd(7,0)=7
```

En C ce calcul est plus simple avec une fonction récursive:

```
int pgcd(int a,int b) { return b? pgcd(b,a%b) : a; }
```

Mais un compilateur intelligent reconnaît la récursivité terminale (la valeur de la fonction est la valeur rendue par un appel récursif) et la remplace par un branchement au début de la procédure après avoir changé les arguments. Donc, le compilateur va d'abord transformer la fonction en:

```
int pgcd(int a,int b)
{ while(b) { int c=a%b; a=b, b=c; }  // (a,b):=(b,a%b)
  return a;
}
```

En assembleur cela donne:

```
pgcd:              int pgcd(int a, int b) { int c;
and r1,r1,r1                 // r0     r1       r2
je fin:              while(b!=0)
sarimm r0,31,r2      { c=a>>31;  // =a<0? -1 :0;
idiv r0,r2,r1          a%=b;     // on divise (c<<32)+a par b  a=reste c=quotient
mov r0,r2              c=a;      // on échange a et b
mov r1,r0              a=b;
mov r2,r1              b=c;
jmp pgcd             }
fin:
ret                }
```

Ecrivez cette fonction en binaire. Ecrivez en assembleur la fonction

```
unsigned pgcdu(unsigned a,unsigned d);
```

Il n'y a qu'à remplacer `c=a>>31;` par `c=0` et `idiv` par `div`.

En supposant que la division entre nombres non signés est un peu plus rapide que la division entre nombres signés, on peut espérer que `pgcdu` sera plus rapide que `pgcd` et donc on peut réécrire cette dernière fonction, en C comme:

```
int pgcd(int a,int b) { return pgcdu(|a|,|b|); }
```

et en assembleur comme 5 lignes à rajouter avant `pgcdu`. Ecrivez ces 5 lignes.

Exercice 2 : pgcd sans division : algorithme binaire

Dans l'algorithme d'Euclide, la plupart des divisions ont un très petit quotient. Il y a donc beaucoup de divisions qui sont très lentes. On peut donc trouver un algorithme sans division qui est plus rapide en général. Pour trouver le pgcd de deux nombres impairs, on peut, tant qu'ils sont différents, remplacer le plus grand des deux par leur différence divisée par la plus grande puissance de 2 possible. Par exemple:

```
pgcd(49,91)=pgcd(49,42)=pgcd(49,21)=pgcd(28,21)=pgcd(7,21)=pgcd(7,14)=pgcd(7,7)=7
```

Mais en général les deux nombres donnés ne sont pas nécessairement impairs. On va donc d'abord chercher la plus grande puissance de 2 qui divise `a` et celle qui divise `b`. La plus petite des deux sera la plus grande puissance de 2 qui divise le pgcd. On multipliera le résultat final par ce nombre. En C cela donne:

```
unsigned pgcdb(unsigned a,unsigned b)
{ unsigned j=__builtin_ctz(a|b), d;
  a>>=__builtin_ctz(a);
  b>>=__builtin_ctz(b);
  if(a && b)
  while(a!=b)
  { if(a<b) d=a,a=b,b=d;
    a-=b;
    a>>=__builtin_ctz(a);
  }
  return (a|b)<<j;
}
```

Pour cette fonction, avec `gcc -O3`, le compilateur GNU génére un code optimisé qui est en moyenne presque 2 fois plus rapide que `pgcdu` pour des pgcd de nombres aléatoires.

La fonction `__builtin_ctz(a)` (Count Trailing Zeros) donne le nombre de bits à zéro de poids faible de `a`. Pour un pentium elle utilise l'instruction appelée `bsf` (Bit Scan Forward) en assembleur. On supposera donc que notre machine possède aussi une instruction `bsf` de code opératoire 30. Donc `a>>=__builtin_ctz(a)` divise `a` par la plus grande puissance de 2 possible. Par exemple la valeur de `a` passe de 12 à 3 ou de 1000 à 125 ou de 13 à 13. Dans la boucle on échange éventuellement `a` et `b` pour que `a` soit toujours le plus grand. Cet échange utilise trois moves conditionnels. Il n'y a pas de branchement conditionnel. On ne perdra donc pas de temps pour une mauvaise prédiction de branchement.

```
                  //    r0          r1    r2  r3
unsigned pgcdb(unsigned a,unsigned b)//  d   j
{ unsigned d=a|b,               or  r0,r1,r2
  j=__builtin_ctz(d);           bsf r2,r3
  d=__builtin_ctz(a);           bsf r0,r2
  a>>=d;                        shr r0,r2,r0
  d=__builtin_ctz(b);           bsf r1,r2
  b>>=d;                        shr r1,r2,r1
  if(a==0)                      and r0,r0,r0
   goto fin;                    je fin
  if(b==0)                      and r1,r1,r1
   goto fin;                    je fin
  debut:                        debut:
  d=a-b;                        sub r0,r1,r2
  if(a==b) goto fin;            je fin
  if(a<b) d=a,                  movcb r0,r2
          a=b,                  movcb r1,r0
          b=d;                  movcb r2,r1
  a-=b;                         sub r0,r1,r0
  d=__builtin_ctz(a);           bsf r0,r2
  a>>=d;                        shr r0,r2,r0
  goto debut;                   jmp debut
  fin:                          fin:
  a|=b;                         or  r0,r1,r0
  a<<=j;                        shl r0,r3,r0
  return a;                     ret
}
```

Ecrivez cette fonction en binaire.

Exercice 3 : fractions

Voici quelques fonctions qui font les opérations arithmétiques élémentaires sur les fractions.
On suppose que `pgcd` utilise (et modifie) seulement les registres `r0` à `r3`.
`reduit` utilise `r0` à `r5`.
`addfrac`, `subfrac`, `divfrac` et `mulfrac` utilisent `r0` à `r9`.

```
typedef struct{ int num,den; } frac;
// r0    r1     r2    r3     r4    r5     r6    r7     r8    r9
// a.num a.den  b.num b.den  e.num e.den  c.num c.den  d.num d.den
// p            d     j      x                                dd
frac neofrac(int n,int d)        neofrac:
{ frac a={n,d};
  return reduit(a);              // jmp reduit
}
frac reduit(frac a)              reduit:
{ frac e=a;                      mov r0,r4; mov r1,r5;
  int p=pgcd(a.num,a.den);       call pgcd
  a.den=e.den/p;                 idiv r5,r0,r1
  a.num=e.num/p;                 idiv r4,r0,r0
  return a;                      ret
}
frac divfrac(frac a,frac b)      divfrac:
{ int x=b.num;                   mov r2,r4
  b.num=b.den;                   mov r3,r2
  b.den=x;                       mov r4,r3
  return mulfrac(a,b);           // jmp mulfrac
}
frac mulfrac(frac a,frac b)      mulfrac:
{ frac c={a.num,b.den};          mov r0,r6; mov r3,r7
  a.num=b.num;                   mov r2,r1
  a=reduit(a);                   call reduit
  frac d=a;                      mov r0,r8; mov r1,r9
  a=c;                           mov r6,r0; mov r7,r1
  a=reduit(a);                   call reduit
  a.num*=d.num;                  mul r0,r8,r0
  a.den*=d.den;                  mul r1,r9,r1
  return a;                      ret
}
frac subfrac(frac a,frac b)      subfrac:
{ int x=0;                       xor r4,r4,r4
  b.num=x-b.num;                 sub r4,r2,r2
  return addfrac(a,b);           // jmp addfrac
}
frac addfrac(frac a,frac b)      addfrac:
{ frac c=a,                      mov r0,r6; mov r1,r7
       d=b;                      mov r2,r8; mov r3,r9
  int p=pgcd(b.den,a.den);       mov r2,r0; call pgcd
```

```
  c.den/=p;                       idiv r7,r0,r7
  d.den/=p;                       idiv r8,r0,r8
  a.den=p;                        mov r0,r1
  a.num=c.num*d.den               mul r6,r9,r2
       +d.num*c.den;              mul r7,r8,r0; add r0,r2,r0
  int dd=c.den*d.den;             mul r7,r9,r9
  a=reduit(a);                    call reduit
  a.den*=dd;                      mul r1,r9,r1
  return a;                       ret
}
```

La plupart des instructions en assembleur sont des moves, qui servent à mettre les arguments des procédures dans le bon registre avant un appel de fonction. Les fonctions en C traduisent ces moves en des affectations entre variables. Mais normalement, on écrit du C sans ces affectations et le compilateur se débrouille pour rajouter tous ces moves. Les fonctions en C devraient donc plutôt ressembler à:

```
typedef struct{ int num,den; } frac;
frac reduit(frac a)
{ int p=pgcd(a.num,a.den);
  a.den/=p;
  a.num/=p;
  return a;
}
frac divfrac(frac a,frac b)
{ frac c={b.den,b.num};
  return mulfrac(a,c);
}
frac mulfrac(frac a,frac b)
{ frac c={a.num,b.den};
  a.num=b.num;
  a=reduit(a);
  c=reduit(c);
  a.num*=c.num;
  a.den*=c.den;
  return a;
}
frac subfrac(frac a,frac b)
{ b.num=-b.num;
  return addfrac(a,b);
}
frac addfrac(frac a,frac b)
{ int p=pgcd(b.den,a.den);
  a.den/=p;
  b.den/=p;
  frac c={a.num*b.den+b.num*a.den,p};
  c=reduit(c);
  c.den*=a.den*b.den;
  return c;
}
```

Ecrivez en C et en assembleur, une fonction qui calcule $\sum_{i=10}^{100} \frac{i+2}{i^3-i}$

Exercice 4 **:**

```
p:
loadimm16 r3,1
l1:
sub r2,r3,r6
jc l2
load r0,r4
load r1,r5
xor r4,r5,r4
sub r4,r5,r6
movcl r0,r6
movcl r1,r0
movcl r6,r1
store r0,r4
store r1,r5
add r0,r3,r0
sub r1,r3,r1
jmp l1
l2:
ret
```

Ecrivez la procédure précédente en binaire.

Que contient la tableau `t` après:

```
int t[]={1,2,8,4,5,6,7,8,9,10}; p(t+3,t+6,4); p(t+4,t+5,5);
```