```
 0, 7, 5, 3        add r7,r5,r3        r3=r7+r5
 1, 3, 4, 5        adc r3,r4,r5        r5=r3+r4+cf
 2, 9, 3, 4        sub r9,r3,r4        r4=r9-r3
 3, 8, 1, 0        sbb r8,r1,r0        r0=r8-r1-cf
 4, 4, 6, *        com r4,r6           r6=~r4
 5, 5, 7, *        neg r5,r7           r7=-r5
 6, 4, 5, 7        or r4,r5,r7         r7=r5|r4
 7, 5, 3, 8        and r5,r3,r8        r8=r5&r3
 8, 3, 6, 4        xor r3,r6,r4        r4=r3^r6
10, 5, 3, 4        shr r5,r3,r4        r4=r5>>r3  non signé
11, 7, 4, 3        shrimm r7,4,r3      r3=r7>>3
12, 5, 6, 9        shl r5,r6,r9        r9=r5<<r6
13, 5, 6, 9        shlimm r5,6,r9      r9=r5<<6
14, 6, 5, 8        sar r6,r5,r8        r8=r6>>r5  signé
15                 sarimm
16, 1, 5, 4        idiv r1,r5,r4       r4=r1/r5  signé
17,                div                           non signé
18, 8, 4, 2        idiv2 r8,r4,r2      (r4<<32+r8) divisé par r2  signé
19, 8, 4, 2        div2 r8,r4,r2       r8=reste r4=quotient      non signé
20, 1, 5, 8        mul r1,r5,r8        r8=r1*r5
22, 2, 6, 9        imul2 r2,r6,r9      (r6<<32+r9)=r2*r6  signé
23,                mul2                                  non signé
24, 1, 3, *        load r1,r3          r3=*r1
25, 1, 3, *        store r1,r3         *r1=r3
26, 2, 5, 4        loadimm16  r2,1284  r2=5*256+4=0x504
27, 3, *, *, 34567 loadimm32  r3,34567 r3=34567
28, 4, 5, 6        call $+0x40506
29, *, *, *        ret
32, 1, 2, 3        jc $+0x10203        if(CF)  goto $+0x10203
                   jb $+0x10203        if(*<*)       Below      non signé
                   jnae $+0x10203      if(!(*>=*))   Not ( Above or Equal)
33,                jnc = jnb = jae     if(!CF) goto            if(*>=*)
34,                je  = jz            if(ZF)  goto            if(*==*)
35,                jne = jnz           if(!ZF) goto            if(*!=*)
36,                ja  = jnbe          if(!CF && !ZF) goto     if(*>*)
38,                jg  = jnle             SF==OF && !ZF            >
40,                jge                    SF==OF                   >=
41,                jl                     SF!=OF                   <
42,                jo                     OF
44, 4, 5, 6        jmp $+0x40506
45, *, *, *        nop                 ne fait rien
48, 1, 3, *        movcc r1,r3         if(CF) r3=r1
49, 5, 6, *        movcnc r5,r6        if(!CF) r6=r5
59, 6, 1, *        movcno r6,r1        if(!OF) r1=r6
60, 7, 2, *        mov r7,r2           r2=r7
61  *, *, *        nop
```

```
void echtab(int n, int *t, int *u)
{ while(n--) { int x=*t, y=*u; *t++=y, *u++=x; } }
echtab:  //  r0  r1  r2  r3  r4  r5
         //   n   t   u   1   x   y
  loadimm16  r3,1       // r3=1
  sub        r0,r3,r0   // if(!n--) goto l2
  jc l2
l1:
  load       r1,r4      // x=*t
  load       r2,r5      // y=*u
  store      r2,r4      // *u=x
  store      r1,r5      // *t=y
  add        r1,r3,r1   // ++t
  add        r2,r3,r2   // ++u
  sub        r0,r3,r0   // if(n--) goto l1
  jnc l1
l2:
  ret
void rettab(int n, int *t)
{ int *u=t+n-1; while(u>t) { int x=*t, y=*u; *t--=y, *u++=x; } }
rettab:  //  r0  r1  r2  r3  r4  r5
         //   n   t   u   1   x   y
  loadimm16  r3,1       // r3=1
  add        r1,r0,r2   // u=t+n
  sub        r2,r3,r2   // u=t+n-1
  sub        r1,r2,r4   // if(t>=u) goto l4
  jnc l4
l3:
  load       r1,r4      // x=*t
  load       r2,r5      // y=*u
  store      r2,r4      // *u=x
  store      r1,r5      // *t=y
  add        r1,r3,r1   // ++t
  sub        r2,r3,r2   // --u
  sub        r1,r2,r4   // if(t<u) goto l3
  jc l3
l4:
  ret
```

Dans la procédure précédente, on peut économiser un registre en rangeant `u` et `n` dans le même registre. On peut donc remplacer toutes les occurences de `r2` par `r0`.

```
int somt4(int n, int *t) // ∑_{i=0}^{n-1} t[i]^4
{ int s=0, x; while(n--) x=*++t, x*=x, s+=x*x; return s; }
sont4:  //  r0  r1  r2  r3  r4
        //   n   t   1   s   x
  loadimm16  r2,1       // r2=1
  xor        r3,r3,r3   // s=0
  sub        r0,r2,r0   // if(!n--) goto l6
  jc L6
L5:
  load       r1,r4      // x=*t
  mul        r4,r4,r4   // x*=x
  mul        r4,r4,r4   // x*=x
  add        r3,r4,r3   // s+=x
  add        r1,r2,r1   // t++
  sub        r0,r2,r0   // if(n--) goto l5
  jnc L5
L6:
  mov r3,r0             // return s
  ret
int pgcd(int a, int b) { int c; while(b) c=a%b, a=b, b=c; return a; }
pgcd: //  r0  r1  r2
      //   a   b   c
   and       r1,r1,r1
   je        l10
l9:
   mov       r0,r2      // c=a
   sarimm    r0,r0,31   // r0,r2=a (extension du signe)
   idiv2     r2,r0,r1   // c=a%b, a=a/b
   mov       r1,r0      // a=b
   and       r2,r2,r1   // b=c
   jne       l9
l10:
   ret
int ppcm(int a, int b) { return a/pgcd(a,b)*b; }
ppcm: //  r0  r1  r2  r3  r4
      //   a   b   *   a   b
   mov       r0,r3      // a
   mov       r1,r4      // b
   call      pgcd       // pgcd(a,b)
   div       r3,r0,r0   // a/pgcd(a,b)
   mul       r4,r0,r0   // a/pgcd(a,b)*b
   ret
```

```
void mul224(int a[2], int b[2], int c[4]); // c[0..3]=a[0..1]*b[0..1]
// r0  r1  r2  r3   r4   r5   r6   r7   r8  r9  r10  r11  r12 r13 r14
//  a   b   c   1 a[0] a[1] b[0] b[1] c[0] c[1] c[2] c[3]   l   h   0
mul224:
  xor        r14,r14,r14// r14=0
  loadimm16  r3,1       // r3=1
  load       r0,r4      // a[0]
  add        r0,r3,r0   // a+1
  load       r0,r5      // a[1]
  load       r1,r6      // b[0]
  add        r1,r3,r1   // b+1
  load       r1,r7      // b[1]
  mov        r4,r9      // a[0]
  imul2      r6,r9,r8   // r9,r8=a[0]*b[0]
  mov        r5,r11     // a[1]
  imul2      r7,r11,r10 // r11,r10=a[1]*b[1]
  mov        r4,r13     // a[0]
  imul2      r7,r13,r12 // r13,r12=a[0]*b[1]
  add        r9,r12,r9
  adc        r10,r13,r10  // r11,r10,r9+=r14,r13,r12
  adc        r11,r14,r11
  mov        r5,r13     // a[1]
  imul2      r6,r13,r12 // r13,r12=a[1]*b[0]
  add        r9,r12,r9
  adc        r10,r13,r10  // r11,r10,r9+=r14,r13,r12
  adc        r11,r14,r11
  and        r5,r5,r5
  jge l7                // if(a[1]<0) c[3],c[2]-=b[1],b[0]
  sub        r10,r6,r10
  sbb        r11,r7,r11
l7:
  and        r7,r7,r7
  jge l8                // if(b[1]<0) c[3],c[2]-=a[1],a[0]
  sub        r10,r4,r10
  sbb        r11,r5,r11
l8:
  store      r2,r8      // c[0]
  add        r2,r3,r2   // c+1
  store      r2,r9      // c[1]
  add        r2,r3,r2   // c+2
  store      r2,r10     // c[2]
  add        r2,r3,r2   // c+3
  store      r2,r11     // c[3]
  ret
```

En supposant maintenant que les arguments sont passés dans la pile, on va compiler le programme suivant sans optimiser en mettant les variables locales dans la pile :

```
int sqr(int a) { return a*a; }
int somt2(int n, int t[])
{ int s=0, i;
  for(i=0;i<n;i++) s+=sqr(t[i]);
  return s;
}
sqr:   // r0      *r128     r128[1]
       // a*a     &retour     a
  loadimm16  r0,1       // 1
  add        r128,r0,r0 // &a
  load       r0,r0      // a
  mul        r0,r0,r0   // a*a
  ret                   // return a*a
somt2:          //    *r128     r128[1]  r128[2]
// r128[0]  r128[1]  r128[2]    r128[3]  r128[4]
//    s        i     &retour       n        t
  loadimm16  r0,2       // 2     allocation de mémoire dans la pile
  sub        r128,r0,r128// r128-=2        pour s et i
  xor        r0,r0,r0   // 0
  store      r128,r0    // s=0
  xor        r0,r0,r0   // 0
  loadimm16  r0,1       // 1
  add        r128,r0,r1 // &i
  store      r128,r0    // i=0
l11:
  loadimm16  r0,1       // 1
  add        r0,r128,r0 // &i
  load       r0,r0      // i
  loadimm16  r1,3       // 3
  add        r1,r128,r1 // &n
  load       r1,r1      // n
  sub        r0,r1,r1   // i-n
  jge l12               // if(i>=n) goto l12
  loadimm16  r0,1       // 1
  add        r0,r128,r0 // &i
  load       r0,r0      // i
  loadimm16  r1,4       // 4
  add        r1,r128,r1 // &t
  load       r1,r1      // t
  add        r0,r1,r0   // t+i
  load       r0,r0      // t[i]
  loadimm16  r1,1       // 1
```

```
  sub         r128,r1,r118// r128--
  store       r128,r0     // on empile t[i]
  call        sqr         // r0=sqr(t[i])
  loadimm16   r1,1        // 1
  add         r128,r1,r118// r128++
  load        r128,r1     // s
  add         r1,r0,r1    // s+sqr(t[i])
  store       r128,r1     // s+=sqr(t[i])
  loadimm16   r0,1        // 1
  add         r0,r128,r0  // &i
  load        r0,r1       // i
  loadimm16   r2,1        // 1
  add         r2,r1,r1    // i+1
  store       r0,r1       // i++
  jmp l11
l12:
  load        r128,r1     // s
  loadimm16   r0,2        // 2     libération de la mémoire dans la pile
  add         r128,r0,r128// r128+=2     pour s et i
  ret                     // return s
```

On peut faire diverses optimistions dans ce code : On peut remplacer l'appel à `sqr` par son code, cela économise toutes les manipulations de pile. On peut mettre les variables `n`, `t`, `s` et `i` dans les registres au lieu de les mettre en mémoire dans la pile. Enfin on peut utiliser un registre qui contient toujours 1, cela évite les `loadimm16` dans la boucle :

```
somt2: // r0  r1  r2  r3  r4   r5      *r128     r128[1]  r128[2]
       //  s   1   n   t   i    *      &retour     n        t
  xor         r0,r0,r0    // s=0
  loadimm16   r1,1        // r1=1
  add         r128,r1,r2  // &n
  add         r2,r1,r3    // &t
  load        r2,r2       // n
  load        r3,r3       // t
  xor         r4,r4,r4    // i=0
  sub         r4,r2,r5    // if(i>=n) goto l10
  jge l10
l9:
  add         r3,r4,r5    // t+i
  load        r5,r5       // t[i]
  mul         r5,r5,r5    // sqr(t[i])
  add         r0,r5,r0    // s+=sqr(t[i])
  add         r4,r1,r4    // i++
  sub         r4,r2,r5    // if(i<n) goto l9
  jl l9
l10:   ret                // return s
```

```
typedef struct{ int num, den; } frac;
frac simplifie(int a, int b) { int p=pgcd(a,b); return (frac){a/p,b/p}; }
simplifie:              //                        r0   r1  r2    r3  r4
  mov        r0,r3      // a                      a,p   b  pgcd  a   b
  mov        r1,r4      // b
  call       pgcd       // p=pgcd(a,b)
  idiv       r4,r0,r1;     idiv       r3,r0,r0   //  b/p, a/p
  ret
frac mulfrac(frac a, frac b)
{ frac c=simplifie(a.num,b.den), d=simplifie(b.num,a.den);
  return (frac){c.num*d.num,c.den*d.den}; }
mulfrac: // r0    r1    r2    r3    r4  r5    r6    r7    r8
         // a.num a.den b.num b.den  *  b.num a.den c.num c.den
  mov        r1,r6      // a.den
  mov        r2,r5      // b.num
  mov        r3,r1      // b.den
  call simplifie        // c=simplifie(a.num,b.den)
  mov        r0,r7;     mov        r1,r8      // c
  mov        r5,r0      // b.num
  mov        r6,r1      // a.den
  call simplifie        // d=simplifie(b.num,a.den)
  mul        r0,r7,r0   // c.num*d.num
  mul        r1,r8,r1   // c.den*d.den
  ret
frac addfrac(frac a, frac b)
{ int p=pgcd(a.den,b.den);
  frac c=simplifie((a.den/=p)*b.num+a.num*(b.den/=p),p);
  c.den*=a.den*b.den;   return c; }
addfrac: // r0       r1    r2    r3    r4    r5    r6
         // a.num,p  a.den b.num b.den a.num a.den b.num
  mov        r0,r4;    mov   r1,r5      // a
  mov        r2,r6      // b.num
  mov        r3,r0      // b.den
  call pgcd             // p=pgcd(b.den,a.den)
  idiv       r3,r0,r3   // b.den/=p
  idiv       r5,r0,r5   // a.den/=p
  mul        r3,r4,r4   // b.den*a.num
  mul        r5,r6,r6   // a.den*b.num
  mul        r3,r5,r5   // a.den*b.den
  mov        r0,r1      // p
  add        r4,r6,r0   // b.den*a.num+a.den*b.num
  call simplifie        // d=simplifie(b.den*a.num+a.den*b.num,p)
  mul        r1,r5,r1   // c.den*=a.den*b.den
  ret
```

```
typedef { unsigned l,h; } ll;
ll addll(ll a, ll b) { a.l+=b.l; a.h+=b.h+(a.l<b.l); return a; }
long long addll(long long a, long long b) { return a+b; }
addll:                  //                        r0 r1  r2 r3
  add       r0,r2,r0    // a.l+=b.l                 a      b
  adc       r1,r3,r1    // a.h+=b.h+CF
  ret                   // return a
long long mulll(long long a, long long b) { return a*b; }
mulll:                  //                        r0 r1  r2 r3
  mul       r0,r3,r3    // b.h*=a.l                 a      b
  mul       r2,r1,r1    // a.h*=b.l
  add       r3,r1,r1    // a.h=b.h*a.l+a.h*b.l
  imul2     r0,r2,r0    // a.l*=b.l  r2=partie haute du produit
  add       r1,r2,r1    // a.h=b.h*a.l+a.h*b.l+partie haute de a.l*b.l
  ret
```

```
int cube(int x) {return x*x*x;}
cube:
  mul r0,r0,r1 // x^2
  mul r0,r1,r0 // x^3
  ret
puiss4:
  mul r0,r0,r0 // x^2
  mul r0,r0,r0 // x^4
  ret
```

```
puiss21:
  mul r0,r0,r1 // x^2
  mul r1,r1,r1 // x^4
  mul r0,r1,r1 // x^5
  mul r1,r1,r1 // x^10
  mul r1,r1,r1 // x^20
  mul r0,r1,r0 // x^21
  ret
```

```
int puiss(int x, unsigned n) // x élevé à la puissance n
{ int y=1;
  while(n)            // y*(x^n)  est un invariant de boucle
   if(n&1) --n,y*=x;  // y*(x^n) = (y*x)*(x^(n-1))
   else   n/=2,x*=x;  // y*(x^n) = y*((x^2)^(n/2))
  return y;
}
int puiss(int x,unsigned n)
{ int y=1;
  do
  { if(n&1) y*=x;
    x*=x;
  } while(n/=2);

  return y;
}
```

```
puiss:                      // r0  r1  r2  r3
  loadimm16 1,r2 // y=1;    // x   n   y   z
l1:              // do
  mul r0,r2,r3   // { z=x*y
  mul r0,r0,r0   //   x*=x
  shrimm r1,1,r1 //   n/=2
  movccc r3,r2   //   if(n était impair) y=z;
  jnz l1         // } while(n);
  mov r2,r0      // return y;
  ret
```

```
void fusion(int*t, int*v, int*u, int n, int m) // On fusionne u[n] et v[m]
{ while(n && m)                                // dans t[n+m].
  if(*u<*v) *t++=*u++,n--;
  else      *t++=*v++,m--;
  while(n)  *t++=*u++,n--;
  while(m)  *t++=*v++,m--;
}
```

La procédure `fusion` peut être simplifiée car on a toujours `n>0` et `m>0` et `v=t+n`.

```
void fusion(int*t, int*v, int*u, int n, int m)
{ for(;;)
  if(*u<*v) { *t++=*u++;   if(!--n) return; }
  else      { *t++=*v++;   if(!--m)  break; }
  do          *t++=*u++; while(--n);
}
fusion: // r0 r1 r2 r3 r4 r5 r6   r7   r8
        // t  v  u  n  m  1  x=*u y=*v x-y
  loadimm16 r5,1
l1:  load  r2,r6     // x=*u;
l2:  load  r1,r7     // y=*v;
     sub   r6,r7,r8  // x-y;
     jge   l3        // if(x<y)
     store r0,r6     // { *t=x;
     add   r0,r5,r0  //   t++;
     add   r2,r5,r2  //   u++;
     sub   r3,r5,r3  //   --n;
     jnz   l1        //   if(!n) return;
     ret             // } else
l3:  store r0,r7     // { *t=y;
     add   r0,r5,r0  //   t++;
     add   r1,r5,r1  //   v++;
     sub   r4,r5,r4  //   --m;
     jnz   l2        //   if(!m) break; }
l4:  load  r2,r6     // do { x=*u;
     store r0,r6     //      *t=x;
     add   r2,r5,r2  //      u++;
     add   r0,r5,r0  //      t++;
     sub   r3,r5,r3  //      --n;
     jnz   l4        //    } while(n):
     ret
```

```
void tri_fusion(int*t, int nm, int*u)
{ int n=nm/2, m=nm-n, *v=t+n, i;
  if(!n) return;
  tri_fusion(t,n,u);
  tri_fusion(v,m,u);
  for(i=n;i--;) u[i]=t[i];
  fusion(t,v,u,n,m);
}
tri_fusion: // r0  r1  r2  r3  r4  r5  r6    r7  r8
            // t   nm  u   n   m   1   p,u+i t+i i
  shrimm  r1,1,r3  // n=nm/2
  jz    l7         // if(!n) return;
  loadimm16 r5,1
  sub r128,r5,r128;  store r128,r0  // push t
  sub r128,r5,r128;  store r128,r1  // push nm
  sub r128,r5,r128;  store r128,r2  // push u
  mov r3,r1        // n
  call tri_fusion  // tri_fusion(t,n,u);
  load r128,r2                 // u
  add r128,r5,r6;  load r6,r1  // nm
  add r6  ,r5,r6;  load r6,r0  // t
  shrimm  r1,1,r3  // n=nm/2
  add r0,r3,r0     // v=t+n
  sub r1,r3,r1     // m=nm-n
  call tri_fusion  // tri_fusion(v,m,u);
  load r128,r2; add r128,r5,r128    // pop u
  load r128,r1; add r128,r5,r128    // pop nm
  load r128,r0; add r128,r5,r128    // pop t
  shrimm  r1,1,r3  // n=nm/2
  sub r1,r3,r4     // m=nm-n
  mov r3,r8        // i=n
l5:
  sub r8,r5,r8     // --i
  jc  l6           // while(i--)
  add r0,r8,r7     // t+i
  add r2,r8,r6     // u+i
  load r7,r7       // t[i]
  store r8,r7      // u[u]=t[i]
  jmp l5
l6:
  add r0,r3,r1     // v=t+n
  jmp fusion       // fusion(t,v,u,n,m)
l7:
  ret
```