Un processeur 16 bits pour implémenter une montre

Xavier Rival et Nicolas Couchoud

Table des matières

1 Structure générale et jeu d'instruction

1.1 Structure générale

Le processeur comprend les éléments suivants :
- Bus de données sur 16 bits (registres, bus vers les I/O et la mémoire).
- Adresses d'instructions et de mémoire sur 12 bits.
- ALU à 16 registres.
- Entrées-sorties : bus avec 16 bits d'entrée et 16 bits de sortie, adresses des périphérique sur 8 bits (256 périphériques possibles).

1.2 Le jeu d'instructions

1.2.1 Format général des instructions

Accès en RAM : inst (4 bits) + adresse (12 bits)

Opérations : inst (4 bits) + reg1 (4 bits) + reg2 (4 bits) + reg3 (4 bits)

Sauts : inst (4 bits) + adresse (12 bits)

I/O : inst (4 bits) + adresse I/O (8 bits) + reg (4 bits)

1.2.2 Les instructions

On distingue deux sortes d'instructions : celles dont l'exécution incombe totalement au module de contrôle (instructions de sauts, d'entrées-sorties et de gestion de l'exécution du code) et les instructions de calcul.

Instructions de contrôle :

stop (0111) : toutes les sorties sont gelées, aucune opération ne peut être effectuée sur les registres ni sur la mémoire. Par ailleurs, le compteur ordinal est remis à 0 et est stoppé : il ne redémarre qu'à la seconde suivante.

load (0000+adresse mémoire) : charge le contenu de la zone mémoire déterminée par l'adresse dans le registre principal R0.

store (0001+adresse mémoire) : sauvegarde le contenu de R0 dans la mémoire à l'adresse indiquée.

jump (0010+adresse code) : l'exécution continue à l'adresse "adresse code" (branchement non conditionnel).

cond (0011+adresse code) : comme jump, uniquement si le registre R0 vaut 0 (branchement conditionnel).

out (0101+adresse périphérique+registre) : envoie le contenu de "registre" sur le bus I/O à l'adresse indiquée.

in (0100+adresse périphérique+registre) : affecte à "registre" la valeur envoyée par le périphérique d'adresse correspondante (entrée).

instructions de calcul (instructions alu) :

copy (1000+r1+(4 bits)+r2) : affectation r2 = r1

add (1001+r1+r2+r3) : affectation r3 = r1 + r2

sub (1001+r1+r2+r3) : affectation r3 = r1 - r2

and (1001+r1+r2+r3) : affectation r3 = r1 and r2 (bit à bit)

xor (1001+r1+r2+r3) : affectation r3 = r1 xor r2 (bit à bit)

insfaible (1100+mot(8 bits)+reg) : insère "mot" dans "reg" (au niveau des 8 bits de poids faible).

insfort (1100+mot(8 bits)+reg) : insère "mot" dans "reg" (au niveau des 8 bits de poids forts).

2 Architecture du microprocesseur.

2.1 Les différents modules.

La structure générale du processeur est présentée en figure 1. Le processeur se divise en 3 modules :

L'unité de contrôle. Ce module contient le séquenceur et le compteur ordinal. Il gère donc l'accès à la ROM qui contient le code, ainsi que les modifications de valeur du compteur ordinal lors des sauts. Par ailleurs, ce module gère les accès à la RAM (chargement et stockage) ainsi que le bus I/O. L'instruction stop est également gérée par le module contrôle (retour à 0 du compteur ordinal et blocage du processeur).

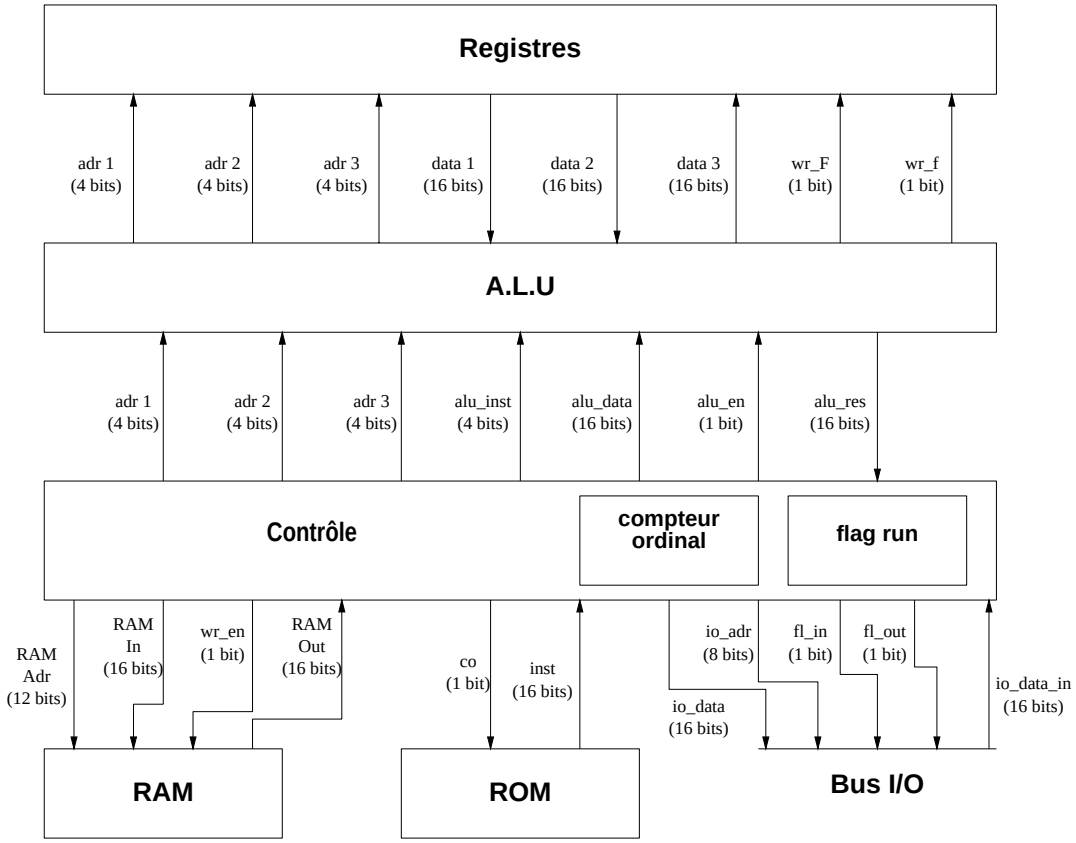

FIG. 1: Structure générale du processeur. Les noms des connexions entre modules sont explicités dans le tableau 1 et dans la section 2.3.

L'unité arithmétique et logique (ALU). Ce module exclusivement combinatoire effectue les opérations arithmétiques et logiques. Il gère aussi les accès (lectures et écritures) aux registres.

Les registres : Cette unité sous contrôle de l'ALU contient 16 registres de 16 bits. L'ALU lisant deux registres en même temps, deux séries de multiplexeurs 16 bits sont donc nécessaire en plus de la série de 16 démultiplexeurs 4 bits pour l'écriture.

2.2 Le module de contrôle.

Le tableau 1 montre ce que renvoie le module à sa sortie en fonction de l'instruction exécutée.

Compteur ordinal et flag run

Le compteur ordinal co (figure 2) stocke l'adresse de l'instruction courante. Lors

Sorties	**bits**	**stop**	**store**	**jump**	**load**
Sorties vers les mémoires					
bus adresse code	12	co	co	co	co
bus adresse mémoire (RAMAdr)	12	#	inst[4..15]	#	inst[4..15]
bus de données vers la mémoire (RAMIn)	16	#	alu_res	#	#
write enable de la RAM (wr_en)	1	0	1	0	0
Sorties vers l'ALU					
adresse du registre 1 (adr1)	4	0000	0000	0000	0000
adresse du registre 2 (adr2)	4	0000	0000	0000	0000
adresse du registre 3 (adr3)	4	0000	0000	0000	0000
bus de données vers l'ALU (alu_data)	16	#	#	#	data_mem
bus d'instruction ALU (alu_inst)	4	0000	0000	0000	0000
write enable de l'ALU (alu_en)	1	0	0	0	1
Sorties vers le bus I/O					
indicateur d'entrée (fl_in)	1	0	0	0	0
indicateur de sorite (fl_out)	1	0	0	0	0
adresse de périphérique (io_adr)	8	#	#	#	#
bus de données I/O (io_data)	16	#	#	#	#

Sorties	**bits**	**cond**	**in**	**out**	**instr. alu**
Sorties vers les mémoires					
bus adresse code	12	co	co	co	co
bus adresse mémoire (RAMAdr)	12	#	#	#	#
bus de données vers la mémoire	16	#	#	#	#
write enable de la RAM (wr_en)	1	0	0	0	0
Sorties vers l'ALU					
adresse du registre 1 (adr1)	4	0000	0000	0000	inst[4..7]
adresse du registre 2 (adr2)	4	0000	0000	0000	inst[8..11]
adresse du registre 3 (adr3)	4	0000	inst[12..15]	inst[12..15]	inst[12..15]
bus données ALU (alu_data)	16	#	data_in	#	#
bus d'instruction ALU (alu_inst)	4	0000	0000	0000	inst[0..3]
write enable de l'ALU (alu_en)	1	0	1	0	1
Sorties vers le bus I/O					
indicateur d'entrée (fl_in)	1	0	1	0	0
indicateur de sorite (fl_out)	1	0	0	1	0
adresse de périphérique (io_adr)	8	#	inst[4..11]	inst[4..11]	#
bus de données I/O (io_data)	16	#	alu_res	alu_res	#

TAB. 1: Spécification du module du point de vue des sorties. Un # signifie un comportement indifférent (dans la pratique, les sorties indifférentes sont déterminées de manière à optimiser (moins de portes, profondeur combinatoire plus faible).

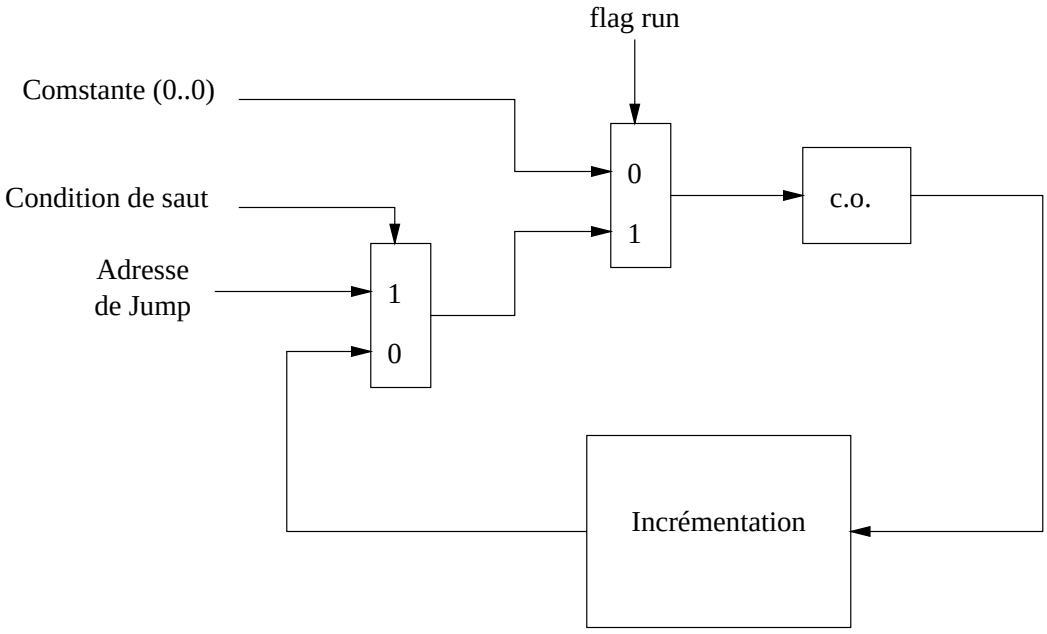

FIG. 2: Le compteur ordinal.

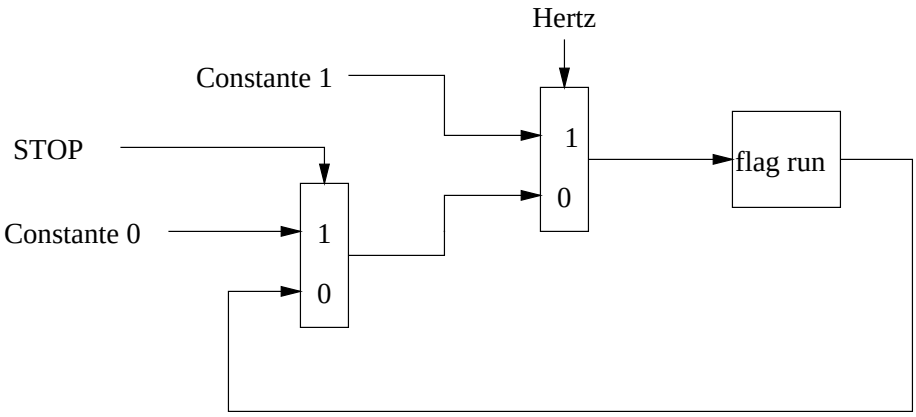

FIG. 3: Le flag run.

d'une instruction ordinaire (sans saut), sa valeur est simplement incrémentée ; par ailleurs, sa valeur est affectée dans le cas d'une instruction de saut, jump ou cond (dans ce dernier cas, la modification de la valeur du co ne doit être modifiée que dans le cas où la valeur contenue dans le registre 0 est égale à 0...0).

Le registre flag run (figure 3) contient la valeur 1 dans le cas où le processeur est en fonctionnement normal (exécution du code), sa valeur est affectée à 0 lorsque l'exécution du programme se termine. Sa valeur est ensuite remise à 1 lorsque le processeur reçoît un signal extérieur (dans le cas de la montre, ce bit de reset arrive une fois chaque seconde (net Hertz)).

Remarque 1 : Cette architecture présente le "défaut" suivant : lors d'une réinitialisation de la valeur de flag run, la première instruction exécutée est l'instruction numéro 1 dans le code (et non l'instruction numéro 0) car la valeur de flag run uitilisée dans l'ensemble du module de contrôle est la valeur sortant du registre "flag run" d'où un décalage de un cycle. Cet inconvénient apparu assez tardivement à la conception pourrait être supprimé assez simplement en utilisant la valeur d'entrée du registre. Par ailleurs ce problème ne gêne nullement la programmation à condition de ne pas commencer le code assembleur à l'instruction 0.

Remarque 2 : L'implémentation des sauts pourrait être améliorée : en effet, l'ajout d'un additionneur permettrait de gérer les sauts relatifs (cf partie sur les améliorations possibles).

2.3 L'ALU (Unité Arithmétique et Logique)

L'ALU est chargée du décodage des instructions arithmétiques et logiques et de l'accès aux registres. C'est un circuit purement combinatoire (sans registres), le décodage étant fait avec des mux.

Le module de contrôle lui envoie les numéros des trois registres qu'elle doit manipuler (adr1 à adr3), l'instruction à exécuter (inst_alu), une éventuelle donnée à écrire dans un registre pour les instructions load et in (alu_data) et un flag disant si l'ALU doit modifier le registre dont le numéro est dans adr3 (mis si une instruction arithmétique ou logique, load ou in est exécutée) ; il reçoit d'elle une donnée à écrire dans la mémoire (instruction store) ou sur un périphérique (instruction out).

L'ALU transmet au circuit de registres les numéros de registres contenus dans adr1, adr2 (opérandes) et adr3 (résultat de l'opération) ; elle reçoit de lui le contenu des registres d'adresse adr1 et adr2 (data1 et data2) et lui envoie le résultat de son calcul (data3), ainsi que deux bits wr_F et wr_f lui indiquant s'il doit effectivement écrire dans le registre d'adresse adr3, respectivement dans les bits de poids fort et de poids faible. Il est nécessaire d'avoir ces deux bits à cause des instructions insfaible et insfort, qui n'écrivent que dans la moitié d'un registre.

2.4 Les registres

Le circuit de registres contient les 16 registres 16 bits du processeur, sous la forme de 256 registres synchrones avec enable.

L'entrée de ces registres est connectée directement à data3.

Le bit d'enable est mis à 1 pour le registre numéro adr3 si wr_F (pour les bits de poids fort) ou wr_f (pour les bits de poids faible) est à 1. Ceci est fait à l'aide de démultiplexeurs 4 bits, c'est-à-dire des circuits à 4 entrées et 16 sorties qui

prennent un nombre sur 4 bits à leur entrée et mettent à 1 la sortie correspondante et à 0 les autres. Ceci permet d'écrire data3 uniquement dans le bon registre (donné par adr3).

La sortie des registres est connectée à des multiplexeurs 16 entrées qui envoient sur data1 et data2 le contenu des registres de numéro adr1 et adr2.

3 Améliorations possibles

Adressage variable :

Il serait intéressant de permettre l'accés à des zones mémoires dont l'adresse serait placée dans des registres (et pouvant donc être calculée) et non dans les instructions. En effet, cela rendrait des chargements de données extérieures plus efficaces. Dans le processeur tel qu'il a été réalisé (cela suffit largement pour l'éxécution du programme "montre" !), chaque écriture ou lecture en mémoire requiert une ligne de code, ce qui ne permet pas vraiment d'envisager une utilisation complète de la mémoire (les tailles de la RAM et de la ROM sont égales).

Cette adaptation n'exigerait qu'une légère modification du module controle et de l'ALU (ajout d'un second bus de données entre l'ALU et le module controle, servant à transmettre les addresses) avec la création de deux instructions :

Load2 + reg_adr + reg_cible

Store2 + reg_adr + reg_cible

On peut noter que ces instructions permettraient également d'affecter le résultat d'un chargement à un registre quelconque (pas forcément R0) ou de sauvegarder le contenu d'un registre quelconque.

Sauts relatifs :

On pourrait envisager de remplacer les sauts absolus par des sauts relatifs. Il suffirait pour cela d'ajouter un additionneur 12 bits ayant pour entrées la sortie du registre 12 bits "compteur ordinal" et l'adresse relative de saut et d'en connecter la sortie au niveau des multiplexeurs en amont de co (de manière à ce que le résultat soit placé dans le registre co dans le cas où un saut doit être effectué).

Extensions possibles du jeu d'instruction :

On pourrait envisager l'ajout de nouvelles instructions de manipulation bit à bit (décalages) ou de sauts (en particulier un saut conditionnel ne s'executant que si la valeur de R0 n'est pas 0..0, ce qui serait utile pour des boucles.)

A Code du processeur en Jazz

```
import jazz.circuit.*;
import jazz.circuit.Net.*;
import jazz.circuit.blues.*;

device Control {
  // Entrée : coup d'horloge
  input hertz : Net;

  // Entrée : bus io :
  input io_in : Net[16];

  // Entrée : résultat retourné par l'alu :
  input alu_res : Net[16];

  // Entrées : données depuis mémoires (prog et datas) :
  var inst : Net[16];
  var dt : Net[16];

  // Sorties - mémoires (addresses et data) :
  var bus_addr_cache_prog : Net[12];
  var bus_addr_cache_data : Net[12];
  var bus_data_ram_wr : Net[16];
  var fl_ram_wr_en : Net;

  // Sorties vers l'alu :
  output nreg1 : Net[4];
  output nreg2 : Net[4];
  output nreg3 : Net[4];
  output alu_inst : Net[4];
  output alu_aff : Net[16];
  output fl_wr : Net;

  // flag run
  var flag_run : Net;

  // compteur ordinal :
  var co : Net[12];
  var co_inc : Net[12];

  // Sorties vers le bus io :
  output fl_in : Net;
```

```
output fl_out : Net;
output io_addr : Net[8];
output io_data : Net[16];
output flstop : Net;

// drapeaux d instructions :
var Load : Net;
var Store : Net;
var IAlu_ins : Net;
var Stop : Net;
var Jump : Net;
var Cond : Net;
var In : Net;
var Out : Net;
var condition : Net;

// memoires cache rom et ram :
cache_prog=new ROM(naddr=12, ndata=16, content="rom.dat",
                   addr=bus_addr_cache_prog);
cache_data=new RAM(naddr=12, ndata=16,
                   read_addr=bus_addr_cache_data,
                   write_addr=bus_addr_cache_data,
                   write_enable=fl_ram_wr_en,
                   data_in=bus_data_ram_wr);
inst=cache_prog.data;
dt=cache_data.data_out;

Stop=~inst[0] & inst[1] & inst[2] & inst[3];
Store=~inst[0] & ~inst[1] & ~inst[2] & inst[3];
Load=~inst[0] & ~inst[1] & ~inst[2] & ~inst[3];
Jump=~inst[0] & ~inst[1] & inst[2] & ~inst[3];
Cond=~inst[0] & ~inst[1] & inst[2] & inst[3];
In=~inst[0] & inst[1] & ~inst[2] & ~inst[3];
Out=~inst[0] & inst[1] & ~inst[2] & inst[3];
IAlu_ins=inst[0];

nv_flag_run=mux(hertz,
                constant(#(1)),
                mux(Stop, constant(#(0)), flag_run)
                );
flag_run=reg(nv_flag_run);
flstop=~flag_run;
```

```
// Gestion du compteur ordinal :
co_inc[0]=~co[0];
vp[0]=co[0];
condition=(((~alu_res[0]&~alu_res[1])&(~alu_res[2]&~alu_res[3]))
  &((~alu_res[4]&~alu_res[5])&(~alu_res[6]&~alu_res[7])))
  &(((~alu_res[8]&~alu_res[9])&(~alu_res[10]&~alu_res[11]))
  &((~alu_res[12]&~alu_res[13])&(~alu_res[14]&~alu_res[15]));
for (i<11) {
  co_inc[i+1]=co[i+1]^vp[i];
  vp[i+1]=co[i+1]&vp[i];
}
for (i<12) {
  jp_co[i]=inst[i+4];
  nv_co[i]=mux(flag_run & (Jump | (Cond&condition)),
               jp_co[i],
               mux(flag_run, co_inc[i], constant(#0))
               );
  co[i]=reg(nv_co[i]);
}

// Calcul des valeurs des sorties :

// Valeur des sorties ->> mémoires
bus_addr_cache_prog=co;
for (i<12) {
  bus_addr_cache_data[i]=inst[i+4];
}
for (i<16) {
  bus_data_ram_wr[i]=alu_res[i];
}
fl_ram_wr_en=Store;

// Valeur des sorties ->> Alu
fl_wr=mux(flag_run, (Load | In | IAlu_ins), constant(#(0)));
for (i<4) {
  alu_inst[i]=mux(flag_run & inst[0],
                  inst[i], constant(#(0))
                  );
}
for (i<4) {
    nreg1[i]=mux(flag_run,
                 mux(IAlu_ins,
                     inst[i+4],
```

```
                    mux(Out, inst[i+12], constant(#(0)))
                    ),
                constant(#(0))
                );
    nreg2[i]=mux(flag_run & IAlu_ins,
                inst[i+8], constant(#(0))
                );
    nreg3[i]=mux(flag_run & (IAlu_ins | In),
                inst[i+12], constant(#(0))
                );
  }
  for (i<16) {
    alu_aff[i]=mux(In, io_in[i], dt[i]);
  }

  // Valeur des sorties ->> io :
  fl_in=In & flag_run;
  fl_out=Out & flag_run;
  for (i<8) {
    io_addr[i]=inst[i+4];
  }
  for (i<16) {
    io_data[i]=alu_res[i];
  }
}

fun full_add_s(a: Net, b: Net, c: Net)=(s: Net) {
  s=(a & b & c) | (~a & ~b & c) | (~a & b & ~c) | (a & ~b & ~c);
}

fun full_add_r(a: Net, b: Net, c: Net)=(r: Net) {
  r=(a & b) | (a & c) | (b & c);
}

fun full_sub_s(a: Net, b: Net, c: Net)=(s: Net) {
  s=~((~a & b & c) | (a & ~b & c) | (a & b & ~c) | (~a & ~b & ~c));
}

fun full_sub_r(a: Net, b: Net, c: Net)=(r: Net) {
  r=(~a & b) | (~a & c) | (b & c);
}

fun fun_add(a: Net[16], b: Net[16])=(c: Net[16]) {
```

```
  var r: Net[16];
  c[0]=full_add_s(a[0], b[0], constant(#(0)));
  r[0]=full_add_r(a[0], b[0], constant(#(0)));
  for (i<15) {
    c[i+1]=full_add_s(a[i+1], b[i+1], r[i]);
    r[i+1]=full_add_r(a[i+1], b[i+1], r[i]);
  }
}

fun fun_sub(a: Net[16], b: Net[16])=(c: Net[16]) {
  var r: Net[16];
  c[0]=full_sub_s(a[0], b[0], constant(#(0)));
  r[0]=full_sub_r(a[0], b[0], constant(#(0)));
  for (i<15) {
    c[i+1]=full_sub_s(a[i+1], b[i+1], r[i]);
    r[i+1]=full_sub_r(a[i+1], b[i+1], r[i]);
  }
}

device Alu {
  // Entrées depuis Control
  input reg1       : Net[4];
  input reg2       : Net[4];
  input reg3       : Net[4];
  input ddata      : Net[16];
  input alu_inst   : Net[4];
  input fl_wr      : Net;

  // Entrées depuis les registres
  input rdata1   : Net[16];
  input rdata2   : Net[16];

  // Sorties vers Control
  output cdataout   : Net[16];

  // Sorties vers les registres :
  output rin1     : Net[4];
  output rin2     : Net[4];
  output rout     : Net[4];
  output rdataout : Net[16];
  output Wfort    : Net;
  output Wfaible  : Net;
```

```
  // Calcul de Wfort et Wfaible
  x=(~alu_inst[1])|alu_inst[2];
  Wfort=fl_wr&(x|~alu_inst[3]);
  Wfaible=fl_wr&(x|alu_inst[3]);

  // Calcul des opérations :
  for (i<4) {
    insertion[i]=reg1[i];
    insertion[i+4]=reg2[i];
    insertion[i+8]=reg1[i];
    insertion[i+12]=reg2[i];
  }
  addition=fun_add(rdata1, rdata2);
  soustraction=fun_sub(rdata1, rdata2);

  // Valeur des sorties vers les registres :
  rin1=reg1; rin2=reg2; rout=reg3;
  rdataout=[i->mux(alu_inst[0],
                   mux(alu_inst[1],
                       mux(alu_inst[2],
                           rdata1[i] & rdata2[i],
                           insertion[i]
                           ),
                       mux(alu_inst[2],
                           mux(alu_inst[3],
                               rdata1[i] ^ rdata2[i],
                               soustraction[i]
                               ),
                           mux(alu_inst[3],
                               addition[i],
                               rdata1[i]
                               )
                           )
                       ),
                   ddata[i])
  ];
  cdataout=rdata1;
}

// Multiplexeur 16 canaux, défini avec multiplexeur 4 canaux

fun M4(i0: Net, i1: Net, c0: Net, c1: Net, c2: Net, c3: Net)
     =(s: Net) {
```

```
  s=mux(i1,
        mux(i0, c3, c2),
        mux(i0, c1, c0)
        );
}

fun M16(i: Net[4], c: Net[16])=(s: Net) {
  s=M4(i[2], i[3],
       M4(i[0], i[1], c[0] , c[1] , c[2] , c[3]),
       M4(i[0], i[1], c[4] , c[5] , c[6] , c[7]),
       M4(i[0], i[1], c[8] , c[9] , c[10], c[11]),
       M4(i[0], i[1], c[12], c[13], c[14], c[15])
       );
}

// Sélecteur 16 canaux, défini avec sélecteur 4 canaux

fun S4(i0: Net, i1: Net)=(s: Net[4]) {
  ni0=~i0; ni1=~i1;
  s[0]=ni0&ni1;
  s[1]=i0 &ni1;
  s[2]=ni0&i1;
  s[3]=i0 &i1;
}

fun S16(i: Net[4])=(s: Net[16]) {
  l=S4(i[0], i[1]);
  h=S4(i[2], i[3]);
  for (i<4) {
    for (j<4) {
      s[4*i+j]=h[i]&l[j];
    }
  }
}

device Registres
{
  // Entrées depuis l'alu
  input ad_reg1 : Net[4],
    ad_reg2 : Net[4],
    ad_reg3 : Net[4],
    data_in : Net[16],
    Wfort   : Net,
```

```
    Wfaible : Net;

  // Sorties vers l'alu
  output data_out1 : Net[16],
    data_out2 : Net[16];

  // Entrée des registres
  wreg=[j->data_in]; // wreg[numéro de registre][numéro de bit]
  // Autorisation d'écriture dans un registre
  WF=[j->Wfort&S16(ad_reg3)[j]];
  Wf=[j->Wfaible&S16(ad_reg3)[j]];
  // Les registres eux-mêmes
  // regs[numéro de bit][numéro de registre], ordre inverse de wreg
  for (k<8) {
    // Octet de poids faible
    regs[k]  =[j->enable(reg(wreg[j][k]), Wf[j])];
    // Octet de poids fort
    regs[k+8]=[j->enable(reg(wreg[j][k+8]), WF[j])];
  }

  // Lecture de registres par l'ALU
  data_out1=[k->M16(ad_reg1,regs[k])];
  data_out2=[k->M16(ad_reg2,regs[k])];
}

device Process {
  // Entrées du microprocesseur
  input bus_io_in : Net[16];
  input horloge   : Net;

  // Sorties du microprocesseur :
  output bus_io_flagin  : Net;
  output bus_io_flagout : Net;
  output bus_io_addr    : Net[8];
  output bus_io_out     : Net[16];
  output fls            : Net;

  // nets liants alu et control
  var addr_reg1_alu     : Net[4];
  var addr_reg2_alu     : Net[4];
  var addr_reg3_alu     : Net[4];
  var data_alu          : Net[16];
  var fl_alu            : Net;
```

```
var data_control      : Net[16];
var instruct_alu      : Net[4];

// nets liants alu et registres :
var addr_reg1_reg     : Net[4];
var addr_reg2_reg     : Net[4];
var addr_reg3_reg     : Net[4];
var ext_data1_alu     : Net[16];
var ext_data2_alu     : Net[16];
var ins_data_reg      : Net[16];
var fl_fort_reg       : Net;
var fl_faible_reg     : Net;

// declaration des devices :
mp_control=new Control(hertz=horloge, io_in=bus_io_in,
                       alu_res=data_control);
mp_alu=new Alu(reg1=addr_reg1_alu,
               reg2=addr_reg2_alu,
               reg3=addr_reg3_alu,
               ddata=data_alu,
               alu_inst=instruction_alu,
               fl_wr=fl_alu,
               rdata1=ext_data1_alu,
               rdata2=ext_data2_alu);
mp_reg=new Registres(ad_reg1=addr_reg1_reg,
                     ad_reg2=addr_reg2_reg,
                     ad_reg3=addr_reg3_reg,
                     data_in=ins_data_reg,
                     Wfort=fl_fort_reg,
                     Wfaible=fl_faible_reg);

// connexions internes au microprocesseur :
addr_reg1_alu=mp_control.nreg1;
addr_reg2_alu=mp_control.nreg2;
addr_reg3_alu=mp_control.nreg3;
instruction_alu=mp_control.alu_inst;
data_alu=mp_control.alu_aff;
fl_alu=mp_control.fl_wr;
bus_io_flagin=mp_control.fl_in;
bus_io_flagout=mp_control.fl_out;
bus_io_addr=mp_control.io_addr;
bus_io_out=mp_control.io_data;
fls=mp_control.flstop;
```

```
  data_control=mp_alu.cdataout;
  addr_reg1_reg=mp_alu.rin1;
  addr_reg2_reg=mp_alu.rin2;
  addr_reg3_reg=mp_alu.rout;
  ins_data_reg=mp_alu.rdataout;
  fl_fort_reg=mp_alu.Wfort;
  fl_faible_reg=mp_alu.Wfaible;

  ext_data1_alu=mp_reg.data_out1;
  ext_data2_alu=mp_reg.data_out2;
}

export Process();
```

B Code de la montre.

Nous présentons ci-dessous le code tel qu'il a été conçu avant d'être transformé en code binaire.

Ce code se découpe en trois principales sections : dans la première, les constantes utiles sont placées dans les registres (instructions 0 à 3). Dans une seconde phase (instructions 4 à 67), le calcul de la nouvelle heure est effectué (l'heure est stockée en mémoire RAM aux adresses présentées dans le tableau ci-dessous). Dans la dernière partie, la nouvelle heure est adressée vers les afficheurs 7 segments.

Adresses mémoire utilisées :

variable	adresse
secondes (unités)	0
secondes (dizaines)	1
minutes (unités)	2
minutes (dizaines)	3
heures (unités)	4
heures (dizaines)	5

Adresses de périphériques :

afficheur	numéro de périphérique
secondes (unités)	5
secondes (dizaines)	4
minutes (unités)	3
minutes (dizaines)	2
heures (unités)	1
heures (dizaines)	0

Le code :

```
conventions :
RX X sur 4 bits (registres)
iX X sur 8 bits
MX X sur 12 bits
sX X sur 4 bits (non registres)


instruction nulle
```

```
insfaible i1 R15
insfaible i10 R14
insfaible i6 R13
load  M0   //chargement de sec_unit//
add  R0 R15 R1 //R1<-R0+R15//
xor  R1 R14 R0 //R0<-R1@R14//
cond  sec_diz
copy  R1 s0 R0 //R0<-R1//
store  M0   //sauvegarde de sec_unit//
goto  Affichage
insfaible i0 R0  //point sec_diz//
store  M0   //sec_unit remis à 0//
load  M1   //chargement de sec_diz//
add  R0 R15 R1 //R1<-R0+R15//
xor  R1 R13 R0 //R0<-R1@R13//
cond  min_unit
copy  R1 s0 R0 //R0<-R1//
store  M1   //sauvegarde de sec_diz//
goto  affichage
insfaible i0 R0  //point min_unit//
store  M1   //sec_diz remis à 0//
load  M2   //chargement de min_unit//
add  R0 R15 R1 //R1<-R0+R15//
xor  R1 R14 R0 //R0<-R1@R14//
cond  min_diz
copy  R1 s0 R0 //R0<-R1//
store  M2   //sauvegarde de min_unit//
goto  affichage
insfaible i0 R0  //point min_diz//
store  M2   //min_unit remis à 0//
load  M3   //chargement de min_diz//
add  R0 R15 R1 //R1<-R0+R15//
xor  R1 R13 R0 //R0<-R1@R13//
cond  hr_unit
copy  R1 s0 R0 //R0<-R1//
store  M3   //sauvegarde de min_diz//
goto  affichage
insfaible i0 R0  //point hr_unit//
store  M3   //min_diz remis à 0//
load  M4   //chargement de hr_unit//
add  R0 R15 R1 //R1<-R0+R15//
xor  R1 R14 R0 //R0<-R1@R14//
cond  hr_diz
```

```
copy  R1 s0 R0 //R0<-R1//
store  M4   //sauvegarde de hr_unit//
goto  affichage
insfaible i0 R0  //point hr_diz//
store  M4   //hr_unit remis à 0//
load  M5   //chargement de hr_diz//
add  R0 R15 R1 //R1<-R0+R15//
load  M4   //chargement de hr_unit//
add R1 R1 R3 // R3 contient 2*diz
add R3 R3 R4 // R4 contient 4*diz
add R4 R4 R5 // R5 contient 8*diz
add R5 R3 R2 // R2 contient 10*diz
add  R0 R2 R2 //hr calculé dans R2//
insfaible i24 R12
xor  R2 R12 R0 //R0<-hr==24//
cond  hr_zero
copy  R1 s0 R0 //R0<-R1//
store  M5   //sauvegarde de hr_diz//
goto  affichage
insfaible i0 R0  //point hr_zero//
store  M4   //hr_unit remis à 0//
store  M5   //hr_diz remis à 0//
load  M5   //point affichage//
out  i0 R0  //affichage de hr_diz//
load  M4
out  i1 R0  //affichage de hr_unit//
load  M3
out  i2 R0  //affichage de min_diz//
load  M2
out  i3 R0  //affichage de min_unit//
load  M1
out  i4 R0  //affichage de sec_diz//
load  M0
out  i5 R0  //affichage de sec_unit//
stop
```