IFT6223
Systèmes reconfigurables

Un processeur rudimentaire

Jean Pierre DAVID

david@iro.umontreal.ca

Université de Montréal

Plan

- Le processeur : un ASM particulier
- La partie opérative intuitive
- La partie de contrôle
- Les instructions
  - ASM (micro-instructions)
  - Assembleur
- Le programme interpréteur
  - Version C
  - Version ASM
- L'utilisation d'une mémoire synchrone
- L'ASM parallélisé
- La partie opérative finale

Le processeur : un ASM particulier

- Les registres « obligatoires »
  - IR (Instruction Register)
  - MAR (Memory Address Register)
  - PC (Program Counter)
  - F (Flags)
  - Registres « Assembleur » et registres « cachés »
- La partie de contrôle
  - Utilise les F ET IR comme variables de Test
- La partie opérative
  - Gère des accès avec l'extérieur (mémoire et E/S)

Exemple d'un jeu d'instructions assembleur

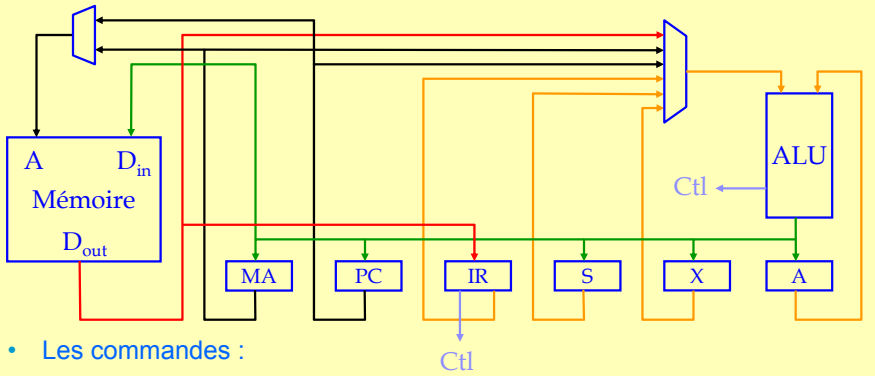
- Soit une instruction par mot de la ROM externe: *ROM[15..0]*
- 4 types d'instruction: *ROM[1..0]*
  - Chargement de la mémoire vers un registre: R=M[Adresse] *$ROM_{1..0}$= 0*
  - Transfert d'un registre vers la mémoire: M[Adresse]=R *$ROM_{1..0}$= 1*
  - Exécution d'une opération: R=$ALU_{ROM[5..3]}$(A,M[Adresse]) *$ROM_{1..0}$= 2*
  - Branchement: PC= ($flag_{ROM[5..3]}$) ? Adresse : PC++ *$ROM_{1..0}$= 3*
- 2 Registres possibles: *ROM[2]*
  - Registre A *ROM_2=0*
  - Registre X *ROM_2=1*
- ROM[5..3] code l'instruction de l'ALU ou le flag testé *ROM[5..3]*
- *4* types de mode d'adressage: *ROM[7..6]*
  - Absolu: Adresse=D *$ROM_{7..6}$=0*
  - Indexé: Adresse=D+X *$ROM_{7..6}$=1*
  - Relatif (avant): Adresse= PC+1+D *$ROM_{7..6}$=2*
  - Relatif (arrière): Adresse= PC+1-D *$ROM_{7..6}$=3*

L'ALU et les Flags

Com.	ALU	Détail
0	COPY	Recopie l'entrée
1	ADD	Addition
2	SUB	Soustraction
3	LSL	Décalage à gauche
4	LSR	Décalage à droite
5	AND	ET logique
6	OR	OU logique
7	XOR	OU exclusif logique

Com	Flag	Détail
0	Z	Nul
1	NZ	Non nul
2	C	A-B>=0 (NS)
3	NC	A-B<0 (NS)
4	POS	>=0 (S)
5	NEG	<0 (S)
6	OV	Retenue (S)
7	'1'	Toujours

La partie opérative



- Les commandes :
  - Un « enable » par registre (et un « reset » sur le PC)
  - Un RW sur la mémoire
  - Le type d'opération de l'ALU
  - Les positions des MUX

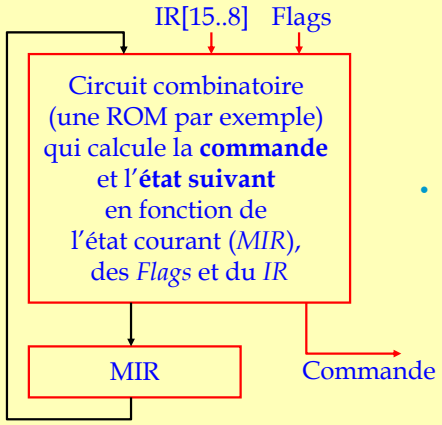
Les micro-instructions (ASM)

n_i: if (Test) { Reg_{if} = ALU_{if} (A, $Data_{if}$); goto $n_{i,if}$; }
else { Reg_{else} = ALU_{else} (A, $Data_{else}$); goto $n_{i,else}$; }

n_i: switch (Test[...])
case 0...0: { $Reg_{0...0}$ = $ALU_{0...0}$ (A, $Data_{0...0}$); goto $n_{i,0...0}$; }
...
case 1...1: { $Reg_{1...1}$ = $ALU_{1...1}$ (A, $Data_{1...1}$); goto $n_{i,1...1}$; }

Test: Bits de $flags_{7..0}$ ou de $IR_{7..0}$
Reg: MA, IR, PC, X, S ou A
Data: RAM[MA ou PC], MA, IR, PC, X, S, A
ALU: Une opération de l'ALU parmi les 8 disponibles

La partie de contrôle



- Les micro-instructions sont les états de l'ASM. Elles sont contenues dans la partie de contrôle. La micro-instructions courrant est contenue dans le MIR.
- Les instructions assembleur sont contenues dans la mémoire extérne (voir partie opérative) et doivent être interprétées par l'ASM. L'instruction assembleur courante est contenue dans le IR.

Le programme interpréteur (comportemental)

```
interpreter() {
    Initialiser le processeur (PC=0)
    while (TRUE) {
        Lire l'instruction assembleur à l'adresse PC; et augmenter PC de 1;
        Calculer l'adresse de la mémoire concernée par l'instruction;
        Exécuter l'instruction;
    }
}
```

L'exécution d'une instruction assembleur demande donc plusieurs micro-instructions (ASM) pour être finalisée.

Le degré de complexité de la partie opérative permet de faire varier le nombre de cycles nécessaires à la réalisation d'une instruction assembleur.

L'architecture RISC, qui utilise plusieurs ASM en parallèle, permet d'obtenir une instruction ASM/cycle dans les meilleurs cas (voir plus loin)

Le programme interpréteur (C)

```
interpreter() {
PC=0;
while (TRUE) {
   IR=M[PC];
   PC++;
   S=A;
   A=DISPL;
   switch (IR[7..6]) {
      case 0: MA=A;          break;
      case 1: MA=A+X;        break;
      case 2: MA=PC+A;       break;
      case 3: MA=PC-A;       break;
   }
   A=S;
   switch (IR[1..0]) {
     case 0: if (IR[2]) X=M[MA];
             else       A=M[MA];
             break;
     case 1: if (IR[2]) M[MA]=X;
             else       M[MA]=A;
             break;
     case 2: if (IR[2]) X=ALU_IR[5..3]
   (A,M[MA]);
             else
   A=ALU_IR[5..3](A,M[MA]);
   break;
     case 3: if (Flag_IR[5..3]) PC=MA;
             break;
   }
}}
```


L'ASM du processeur (PO intuitive) (1/2)

```
N0:    if (true)          {PC=0; goto N1a;}
N1a:   if (true)          {IR=M[PC]; goto N1b;}
N1b:   if (true)          {PC:++; goto N1c;}
N1c:   if (true)          {S=A; goto N1d;}
N1d:   if (true)          {A=DISPL; goto N2a;}

N2a:switch (IR[7..6])     {
       case 0:            {MA=A; goto N2b;}
       case 1:            {MA=A+X; goto N2b;}
       case 2:            {MA=PC+A; goto N2b;}
       case 3:            {MA=PC-A; goto N2b;}
       }
N2b:   if (true)          {A=S; goto N3;}
```


L'ASM du processeur (PO intuitive) (2/2)

```
N3:    switch (IR[2..0]) {
       case 0:            {A=M[MA]; goto N1a;}
       case 1:            {M[MA]=A; goto N1a;}
       case 2:            {A=ALU_IR[5..3] (A,M[MA]); goto N1a;}
       case 3:            {goto N4;}
       case 4:            {X=M[MA]; goto N1a;}
       case 5:            {M[MA]=X; goto N1a;}
       case 6:            {X=ALU_IR[5..3](A,M[MA]); goto N1a;}
       case 7:            {goto N4;}

N4:    if (Flag_IR[5..3])  {PC=MA;   goto N1a;}
       else                {goto N1a;}
```

Compromis espace - temps

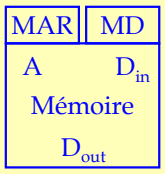
- Il n'est pas possible de réaliser l'ASM parallélisé sur la partie opérative intuitive car celle-ci ne permet de réaliser qu'une opération à la fois (en passant chaque fois par l'ALU).
- Il y a deux alternatives :
  - Modifier l'ASM pour qu'il ne comporte qu'une opération à chaque cycle
    - ✓ C'est la version la plus "économique".
    - ✓ Un nombre de cycles plus important sera nécessaire pour exécuter une instruction assembleur.
  - Modifier la partie opérative pour pouvoir implémenter les opérations avec un maximum de parallélisme
    - ✓ C'est la version la plus rapide.
    - ✓ Un certain nombre de ressources (connexions – opérateurs) devront être ajoutés et consommeront donc plus d'espace (de cellules logiques)
- La solution la plus intéressante se trouve peut-être entre les deux extrêmes

L'ASM du processeur (parallélisé)

```
N0:   if (true)                {PC=0; goto N1;}
N1:   if (true)                {IR=M[PC]; PC:++; S=A; A=DISPL; goto N2;}
N2:   switch (IR[7..6]) {
        case 0:                {MA=A; A=S; goto N3;}
        case 1:                {MA=A+X; A=S; goto N3;}
        case 2:                {MA=PC+A; A=S; goto N3;}
        case 3:                {MA=PC-A; A=S; goto N3;}}
N3:   switch (IR[2..0]) {
        case 0:                {A=M[MA]; goto N1;}
        case 1:                {M[MA]=A; goto N1;}
        case 2:                {A=ALU_IR[5..3] (A,M[MA]); goto N1;}
        case 3:                {goto N4;}
        case 4:                {X=M[MA]; goto N1;}
        case 5:                {M[MA]=X; goto N1;}
        case 6:                {X=ALU_IR[5..3](A,M[MA]); goto N1;}
        case 7:                {goto N4;}}
N4:   if (Flag_IR[5..3])       {PC=MA;    goto N1;}
      else                     {goto N1;}
```


La gestion d'une mémoire synchrône

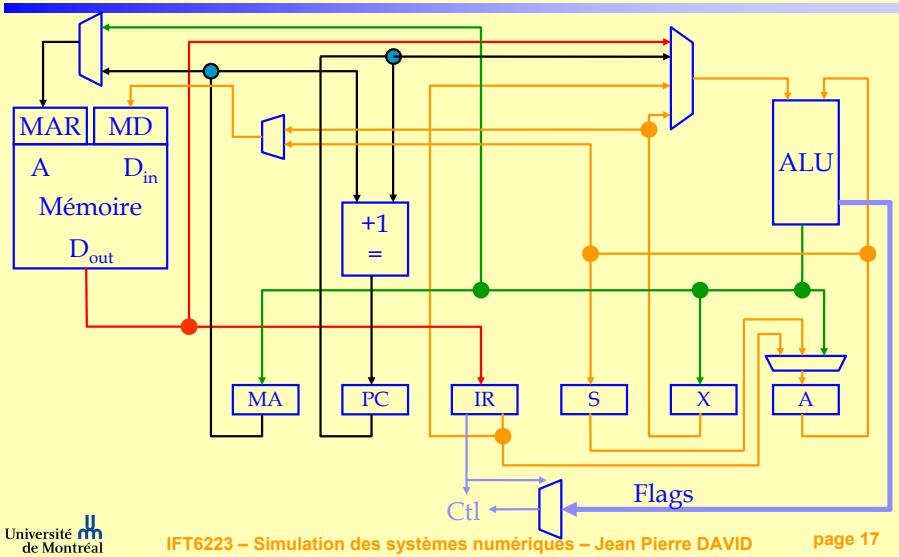
- Les accès en lecture (2 cycles)
  - Cycle 1: Transférer l'adresse dans le registre d'adresse de la mémoire
  - Cycle 2: Transférer le contenu de la mémoire (signal R/W=1)
  - Exemple : Cycle 1 : MAR=Adresse
  - Cycle 2 : X=MEM[MAR];
- Les accès en écriture (1 cycle + 1 cycle d'exécution)
  - Cycle 1: Transférer l'adresse dans MAR et la donnée dans le MD (R/W=0)
  - Cycle 2: La donnée est physiquement écrite en mémoire
  - Exemple : Cycle 1 : MAR=Adresse; MD=X
  - Cycle 2 : *MEM[MAR]=MD; "implicite"*
- Pipeline : Ecrire D_0,D_1 et lire D_2 aux adresse A_0,A_1 et A_2 :
  - Cycle 1 : MAR=A_0; MD=D_0; WE=1;
  - Cycle 2 : MAR=A_1; *MEM[MAR]=MD;*MD=D; WE=1;
  - Cycle 3 : MAR=A_2; *MEM[MAR]=MD;*
  - Cycle 4 : Data=MEM[MAR];



L'ASM du processeur (avec l'accès mémoire)

```
N0:if (true)                   {PC=0; WE=0; goto N1a}
N1a:if (true)                  {MAR=PC; WE=0; goto N1;}
N1:if (true)                   {IR=M[MAR]; PC:++; S=A; A=DISPL; goto N2;}
N2:switch (IR[7..6]) {
        case 0:                {MA=MAR=A; A=S; goto N3;}
        case 1:                {MA=MAR=A+X; A=S; goto N3;}
        case 2:                {MA=MAR=PC+A; A=S; goto N3;}
        case 3:                {MA=MAR=PC-A; A=S; goto N3;}}
N3:switch (IR[2..0]) {
        case 0:                {A=M[MAR]; MAR=PC; goto N1;}
        case 1:                {MAR=MA; MD=A; WE=1; goto N1a;}
        case 2:                {A=ALU_IR[5..3] (A,M[MAR]); MAR=PC; goto N1;}
        case 3:                {goto N4;}
        case 4:                {X=M[MAR]; MAR=PC; goto N1;}
        case 5:                {MAR=MA; MD=X; WE=1; goto N1a;}
        case 6:                {X=ALU_IR[5..3] (A,M[MAR]); MAR=PC; goto N1;}
        case 7:                {goto N4;}
N4:     if (Flag_IR[5..3])     {PC=MAR=MA; goto N1;}
        else                   {MAR=PC; goto N1;}
```

La partie opérative



Fibonacci en assembleur

```
0   A = M[128]          ; 0x8000
1   A = A + M[129]      ; 0x810A
2   X = M[129]          ; 0x8104
3   M[128] = X          ; 0x8005
4   M[129] = A          ; 0x8101
5   GOTO 0              ; 0x003B
```


Suite des carrés en C

```
x=0; sx=0;
While (x!=100) {
  sx=sx+2*x+1;
  mem[x+101]=sx;
  x=x+1;
}
Return;

x est placé en mem[64]
sx est placé en mem[65]
100 est placé en mem[66]
1 est placé en mem[67]
```

```
0   A = M[64]           ; 0x4000
1   X = A-M[66]         ; 0x4216
2   GOTO Z,2            ; 0x0203
3   A=A<<1;             ; 0x001A
4   A = A + M[67]       ; 0x430A
5   A = A + M[65]       ; 0x410A
6   M[65] = A           ; 0x4101
7   X = M[64]           ; 0x4004
8   M[X+101]=A          ; 0x6541
9   A = M[64]           ; 0x4000
A   A = A + M[67]       ; 0x430A
B   M[64] = A           ; 0x4001
C   GOTO 0              ; 0x003B
```


Suite des carrés en assembleur

```
0   A = M[64]           ; 0x4000    START: LDR   D#64,ABS,A
1   X = A-M[66]         ; 0x4216           ALU   D#66,ABS,SBT,X
2   GOTO Z,2            ; 0x0203    FIN:   JP    FIN,ABS,Z
3   A=A<<1;             ; 0x001A           ALU   D#0,ABS, LSL,A
4   A = A + M[67]       ; 0x430A           ALU   D#67,ABS,ADD,A
5   A = A + M[65]       ; 0x410A           ALU   D#65,ABS,ADD,A
6   M[65] = A           ; 0x4101           STR   D#65,ABS,A
7   X = M[64]           ; 0x4004           LDR   D#64,ABS,X
8   M[X+101]=A          ; 0x6541           STR   D#101,IND,A
9   A = M[64]           ; 0x4000           LDR   D#64,ABS,A
A   A = A + M[67]       ; 0x430A           ALU   D#67,ABS,ADD,A
B   M[64] = A           ; 0x4001           STR   D#64,ABS,A
C   GOTO 0              ; 0x003B           JP    START,ABS,AL
```

Configuration du meta-assembleur

ABS:	EQU B#00	Z:	EQU B#000	COPY:	EQU B#000
IND:	EQU B#01	NZ:	EQU B#001	ADD:	EQU B#001
REP:	EQU B#10	C:	EQU B#010	SBT:	EQU B#010
REM:	EQU B#11	NC:	EQU B#011	LSL:	EQU B#011
		POS:	EQU B#100	LSR:	EQU B#100
A:	EQU B#0	NEG:	EQU B#101	AND:	EQU B#101
X:	EQU B#1	OV:	EQU B#110	OR:	EQU B#110
		AL:	EQU B#111	XOR:	EQU B#111

LDR: DEF 8VH#00,2VB#00,B#000,1VB#0,B#00

STR: DEF 8VH#00,2VB#00,B#000,1VB#0,B#01

ALU: DEF 8VH#00,2VB#00,3VB#000,1VB#0,B#10

JP: DEF 8VH#00,2VB#00,3VB#000,B#0,B#11

Fichier de configuration .mif

```
Depth = 256;Width = 16;Address_radix = HEX;Data_radix = HEX;Content
Begin
    [00..FF]: 00000000; -- nop
00: 4000;   -- START:     LDR      D#64,ABS,A
    01: 4216;   --        ALU      D#66,ABS,SBT,X
    02: 0203;   -- FIN:   JP       FIN,ABS,Z
    03: 001A;   --        ALU      D#0,ABS, LSL,A
    04: 430A;   --        ALU      D#67,ABS,ADD,A
    05: 410A;   --        ALU      D#65,ABS,ADD,A
    06: 4101;   --        STR      D#65,ABS,A
    07: 4004;   --        LDR      D#64,ABS,X
    08: 6541;   --        STR      D#101,IND,A
    09: 4000;   --        LDR      D#64,ABS,A
    0A: 430A;   --        ALU      D#67,ABS,ADD,A
    0B: 4001;   --        STR      D#64,ABS,A
    0C: 003B;   --        JP       START,ABS,AL
    40: 0000;   --  VAR X     MANUALLY ADDED
    41: 0000;   --  VAR SX    MANUALLY ADDED
    42: 0064;   --  CST 100   MANUALLY ADDED
    43: 0001;   --  CST 1     MANUALLY ADDED
End;
```