ITI 1521. Introduction à l'informatique II †

Marcel Turcotte
(contributions de R. Holte)

École d'ingénierie et de technologie de l'information
Université d'Ottawa

Version du 8 janvier 2012

†. Pensez-y, n'imprimez ces notes de cours que si c'est nécessaire!

Revue

Objectifs :

1. Connaître les attentes face à Java
2. Introduction à l'architecture des ordinateurs et l'exécution d'un programme

Lectures :

- Pages 597–631 de E. Koffman and P. Wolfgang.

Prérequis

Vous devez maîtriser ces concepts :

- Utilisation des types prédéfinis de Java : y compris les tableaux et les chaînes de caractères
- Structures de contrôles :
  telles que `if`, `for`, `while`... ;
- Abstraction procédural (programmation structurée) :
  c'.-à.-d. comment décomposer un problème en sous-problèmes, implémentés à l'aide de méthodes de classe (static) ;
- Comment éditer, compiler et exécuter un programmes Java.

Pourquoi Java ?

1	Java	18%
2	C	18%
3	C++	10%
4	PHP	9%
5	Basic	6%
6	C#	5%
7	Python	4%
8	Perl	3%
9	Objective-C	3%
10	Delphi	2%

⇒ TIOBE Programming Community Index

Pourquoi Java ?

Java partage le premier rang avec C des langages les plus utilisés, pourtant, je ne connais pas d'applications Java. Où touve-t-on des applications Java ?

- La partie serveur des applications et services Web
- Applications mobiles (téléphones portables)

Pourquoi Java ?

≪ According to a report from NetApplications, which has measured browser usage data since 2004, Oracle's Java Mobile Edition has surpassed Android as the #2 mobile OS on the internet at 26.80%, with iOS at 46.57% and Android at 13.44%. And the trend appears to be growing. Java ME powers hundreds of millions of low-end 'feature phones' for budget buyers. In 2011, feature phones made up 60% of the install base in the U.S. ≫

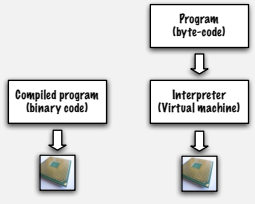
Slashdot
3 janvier 2012
http ://bit.ly/xSk5pN

Pourquoi Java ?

- La programmation en C demande une grande discipline (gestion de la mémoire, manipulation de pointeurs...)
- Java est un bon véhicule pour l'enseignement (interface, héritage simple, types génériques...)
- Lorsqu'on maîtrise Java, l'apprentissage des langages orientés objet ou impératifs est simple

Exécution d'un programme

Quels sont les deux principaux modes d'exécution ?



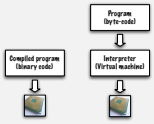
Compilation et exécution d'un programme Java

```
> javac MyProgram.java
```

Produira `MyProgram.class` (code-octet – *byte-code*)

```
> java MyProgram
```

Ici, **java** c'est la Machine Java Virtuelle (JVM).

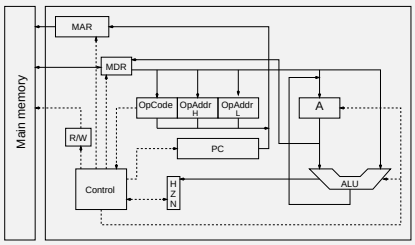


Motivation

- Sous l'ancien programme d'études, ce cours (CSI 1501) comportait une section sur l'architecture des ordinateurs (algèbre de Boole, logique des circuits, représentation des nombres, assembleur, compilation et interprétation de programmes).
- **Cette présentation a pour but de présenter un modèle simplifié de l'exécution des programmes au niveau du matériel.**
- La maîtrise du concept de variables références est essentielle à la réussite de ce cours !

TC1101

Présentation d'un modèle **simplifié** d'un microprocesseur et de son langage d'assemblage.



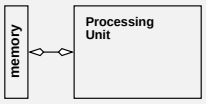
modèle de von Neumann

L'architecture des ordinateurs modernes suit le modèle proposé par (John) von Neumann (1945).

mémoire : contient les instructions et les données

unité de traitement : effectue les opérations arithmétiques et logiques

unité de contrôle : interprète les instructions

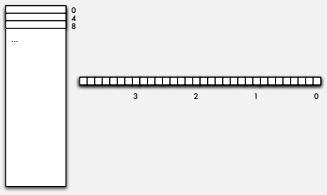


Modèle de la mémoire

- Peut-être vue comme un immense tableau, où chaque cellule du tableau peut contenir zéro ou un (*binary digit* — bits) ;

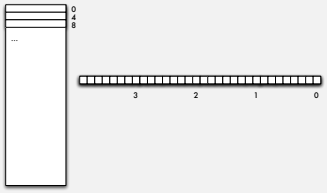
Modèle de la mémoire

- Chaque octet (groupement de 8 bits) possède une adresse unique (distincte)
- Les octets sont regroupés en mots
- Certains types de données nécessitent plus d'un octet



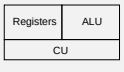
Modèle de la mémoire

- Ce type de mémoire est dit à accès direct (*Random Access Memory*)
- **Le temps d'accès aux cellules de la mémoire est uniforme et constant,**
  De l'ordre de 5 à 70 nano secondes (nano $= 10^{-9}$)



Central Processing Unit (CPU), μ-processeur

- Exécute les instructions une à la fois



UAL Unité Arithmétique Logique (*Arithmetics/Logic Unit* — ALU), contient des circuits spéciaux afin d'exécuter les instructions supportées

UC Unité de Contrôle (*Control Unit*) transfert les instructions depuis la mémoire principale et détermine leur type

Les **registres** sont des unités de mémoire spécialisées situées dans le processeur servant au traitement des données

Mnémoniques, opCodes, description

LDA	91	load x
STA	39	store x
CLA	08	clear (a=0, z=vrai, n=faux)
INC	10	incrémente accumulateur (modifie z et n)
ADD	99	ajoute x à l'accumulaeur (modifie z et n)
SUB	61	retranche x de l'accumulateur (modifie z et n)
JMP	15	branchement inconditionnel vers x
JZ	17	branchement sur x si z==vrai
JN	19	branchement sur x si n==vrai
DSP	01	affiche la valeur se trouvant à l'adresse x
HLT	64	fin

Compilation

Les programmes, suites d'énoncés d'un langage de programmation de haut niveau, sont traduits (compilés), en langage de bas niveau (assembleur, code machine), directement interprétable par le matériel.

L'expression $y = x + 1$ est traduite en assembleur :

```
LDA X
INC
STA Y
HLT
```

qui est ensuite traduit en code machine :

91 00 08	10	39 00 09	64	10	99

Division par soustractions successives

```
[1]   CLA
      STA Quot
[2]   LDA X
[3]   SUB Y
[4]   JN  [7]
[5]   STA Temp
      LDA Quot
      INC
      STA Quot
      LDA Temp
[6]   JMP [3]
[7]   ADD Y
[8]   STA Rem
[9]   DSP Quot
[10]  DSP Rem
[11]  HLT
X     BYTE 25
Y     BYTE 07
Quot  BYTE 00
Rem   BYTE 00
Temp  BYTE 00
```


Division : code machine

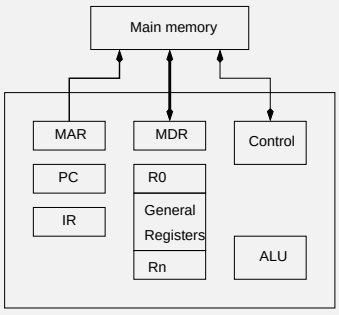
```
[1]   CLA                  08
      STA Quot             39      00 44
[2]   LDA X                91      00 42
[3]   SUB Y                61      00 43
[4]   JN  [7]              19      00 29
[5]   STA Temp             39      00 46
      LDA Quot             91      00 44
      INC                  10
      STA Quot             39      00 44
      LDA Temp             91      00 46
[6]   JMP [3]        ⇒     15      00 07
[7]   ADD Y                99      00 43
[8]   STA Rem              39      00 45
[9]   DSP Quot             01      00 44
[10]  DSP Rem              01      00 45
[11]  HLT                  64
X     BYTE 25              25
Y     BYTE 07              07
Quot  BYTE 00              00
Rem   BYTE 00              00
Temp  BYTE 00              00
```


Division : code machine

08 39 00 44 91 00 42 61 00 43 19 00 29 39 00 46 91 00 44 10 39 00 44 91 00 46 15 00 07 99 00 43 39 00 45 01 00 44 01 00 45 64 25 07 00 00 00

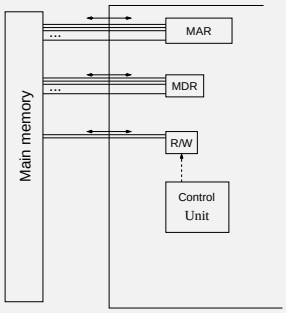
Registres

- Résident dans le processeur, très grande vitesse d'accès
- On y accède à l'aide d'un nom logique plutôt qu'une adresse (MAR, MDR, ...)
- Chaque registre a une fonction spécifique ; au contraire pour la mémoire, vous vous souvenez nous avons dit qu'il n'y a pas de distinction entre les données et les programmes ;
- Leur taille varie selon leur fonction.



Interface entre la mémoire et l'UCT

- Les bits ne sont pas transférés un à la fois, mais en parallèle
- On appelle *bus*, l'ensemble des fils (lignes) qui permettent la communication entre les unités
- **On distingue 3 types de bus : bus de données, bus d'adressage et bus de contrôle**
- Le nombre de lignes (fils) détermine la largeur du bus.



Bus d'adressage

- La largeur du bus détermine la taille maximale de la mémoire
- Si la largeur du bus est de 16 lignes, les adresses sont constituées de 16 bits, il y a donc $2^{16} = 65536$ adresses différentes, s'il y a 32 lignes, les adresses sont constituées de 32 bits, il y donc $2^{32} \simeq 4 \times 10^9$ adresses différentes
- Le registre adresse mémoire (MAR) aura 32 bits afin de stocker une adresse

Bus de données

La taille du bus détermine le nombre de bits transférés en un seul accès (depuis/vers la mémoire).

Bus de contrôle (lignes)

Par exemple, indique le sens du transfert :

R (read) l'élément est transféré de la mémoire au processeur

W (write) l'élément est transféré du processeur à la mémoire

Transfert vers la mémoire

Afin de transférer une valeur v du processeur vers l'adresse x de la mémoire :

1. mettre v dans le registre de donnée (MDR),
2. mettre x dans le registre d'adresse (MAR),
3. mettre le bit de statut RW à faux,
4. activer la ligne de contrôle ≪access_memory≫.

Transfert depuis la mémoire

Afin de transférer une valeur de l'adresse x (en mémoire) vers le processeur :

1. mettre la valeur x dans le registre d'adresse (MAR),
2. mettre le bit de statut RW à vrai,
3. activer la ligne de contrôle ≪access_memory≫
4. le registre de donnée (MDR) contient maintenant une copie de la valeur se trouvant à l'adresse x de la mémoire.

Cycle ≪transfert-exécute≫

1. transfert :
   1.1 transfert de l'OPCODE,
   1.2 incrémente PC,
2. selon OPCODE alors transfert opérande :
   2.1 transfert premier octet,
   2.2 incrémente PC,
   2.3 transfert second octet,
   2.4 incrémente PC,
3. exécute.

⇒ *Fetch-Execute Cycle*.

Exemple

Afin d'ajouter 1 à la valeur x et de sauver le résultat dans y ($y = x + 1$).

- Charger la valeur de x dans l'accumulateur, registre A.
- Incrémenter la valeur de l'accumulateur,
- Sauver la valeur qui se trouve dans l'accumulateur à l'adresse y.

Ceci implique 3 instructions machines :

91 : load

10 : increment

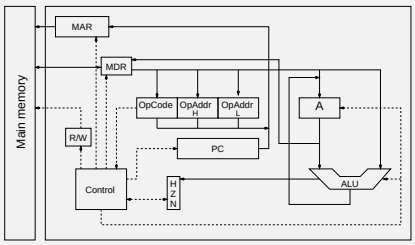
39 : store

Si x désigne l'adresse 00 08 et y désigne l'adresse 00 09, alors $y = x + 1$ peut s'écrire en langage machine comme suit :

91 00 08	10	39 00 09	64	10	99

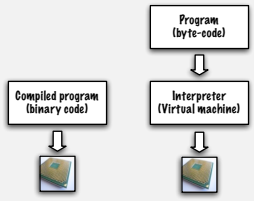
Simulateur Java TC1101

Cette section contient des informations supplémentaires au sujet de ce microprocesseur.



Simulateur Java TC1101

Le Simulateur Java TC1101 joue un rôle semblable à la Machine Java Virtuelle (JVM) — l'interpréteur à droite — il partage aussi plusieurs caractéristiques.



Mnemoniques, opCodes, descriptions

LDA	91	load x
STA	39	store x
CLA	08	clear (a=0, z=vrai, n=faux)
INC	10	incrémente accumulateur (modifie z et n)
ADD	99	ajoute x à l'accumulaeur (modifie z et n)
SUB	61	retranche x de l'accumulateur (modifie z et n)
JMP	15	branchement inconditionnel vers x
JZ	17	branchement sur x si z==vrai
JN	19	branchement sur x si n==vrai
DSP	01	affiche la valeur se trouvant à l'adresse x
HLT	64	fin

Description des unités fonctionnelles du TC-1101

PC (mot) : Compteur de programme (*Program Counter*), registre d'un mot mémoire contenant l'adresse de la prochaine instruction à exécuter ;

OPCODE (octet) : registre d'instruction (parfois dénoté IR), contient l'OPCODE de l'instruction en cours ;

opAddr (mot) : L'opérande de l'instruction en cours d'exécution. L'opérande est toujours une adresse (pour le TC-1101 seulement). Certaines instructions nécessitent la valeur contenue à l'adresse indiquée ; cette valeur n'est pas transférée par le cycle de transfert de base, mais doit être transférée par l'opération elle-même lors de l'exécution (voir étape 3 du cycle de transfert ainsi que la description de chacune des instructions ci-bas) ;

Description des unités fonctionnelles du TC-1101

MDR (octet) : registre donnée mémoire (*Memory Data Register*). Une donnée transférée (*read/loaded*) de la mémoire vers le processeur est toujours placée dans le registre de donnée. De même, une donnée transférée du processeur vers la mémoire (*stored*) doit être placée dans le registre de donnée ;

MAR (mot) : registre adresse mémoire (*Memory Address Register*). Ce registre contient l'adresse mémoire à partir de laquelle une donnée doit être lue ou vers laquelle elle doit être écrite ;

A (octet) : Accumulateur. Toutes les opérations arithmétiques utilisent ce registre comme opérande et aussi pour sauver leur résultat ;

Description des unités fonctionnelles du TC-1101

H (bit) : le bit de statut ≪Halt≫. Ce bit est mis-à-jour par l'instruction ≪halt≫ (hlt). Si ce bit est vrai, le processeur s'arrête lorsque l'opération courante est terminée ;

N (bit) : le bit de statut ≪Negative≫. Les opérations arithmétiques affectent la valeur vrai à ce bit lorsqu'elles produisent un résultat négatif. Certaines opérations n'affectent pas la valeur de ce bit, ainsi sa valeur ne reflète pas toujours l'état courant de l'accumulateur ;

Z (bit) : le bit de statut ≪Zero≫. Les opérations arithmétiques affectent la valeur vrai à ce bit lorsqu'elles produisent le résultat zéro. Certaines opérations n'affectent pas la valeur de ce bit, ainsi sa valeur ne reflète pas toujours l'état courant de l'accumulateur ;

Description des unités fonctionnelles du TC-1101

RW (bit) : le bit de statut ≪READ/WRITE≫. Si sa valeur est vrai, signifie qu'une donnée sera lue (*fetched*) de la mémoire et transférée dans le registre MDR de l'unité centrale. Si faux, signifie qu'une donnée sera transférée du registre MDR vers la mémoire.

Simulateur TC1101

- utilise des constantes afin de representer les OPCODES ;
- des variables de classe représentent la mémoire et les registres ;
- la méthode accessMemory() simule le transfert des données entre le processeur et la mémoire ;
- la méthode centrale est run() qui simule le cycle ≪fetch-execute≫ : lire l'OPCODE, transfert de l'opérande et exécution de l'instruction courante.

Constantes pour représenter les OPCODES

```
import java.io.*;
import SimIO; // read data from a file
class Sim {
    // addresses are 2 bytes
    public static final int  MAX_ADDRESS = 9999;

    // load from memory to the accumulator
    public static final int  LDA = 91;
    // save accumulator to memory
    public static final int  STA = 39;
    // set accumulator to 0
    public static final int  CLA =  8;
    // increment accumulator by 1
    public static final int  INC = 10;
    // add to the accumulator
    public static final int  ADD = 99;
    // subtract from the accumulator
    public static final int  SUB = 61;
    // unconditional branch "go to"
    public static final int  JMP = 15;
    // branch to address if Z
    public static final int  JZ  = 17;
    // branch to address if N
    public static final int  JN  = 19;
    // display to screen
    public static final int  DSP =  1;
    // "halt"
    public static final int  HLT = 64;

    // ...
```

Modéliser la mémoire et les registres

```
    private static final int[] memory = new int[MAX_ADDRESS + 1];

    // program counter
    private static int pc;

    // accumulator
    private static int a;

    // opcode of the current instruction
    private static int opCode;

    // address of the operand
    private static int opAddr;

    // status bit "Zero"
    private static boolean z;

    // status bit "Negative"
    private static boolean n;

    // status bit "Halt"
    private static boolean h;

    // memory address register
    private static int mar;

    // memory data register
    private static int mdr;

    // bit Read/Write.  Read = True; Write = False
    private static boolean rw;
```

Chargement du programme en langage machine

```
    public static void load(String filename)
    throws IOException {
        int[] values;
        int i;
        int address = 0;
        SimIO.setInputFile(filename);
        while (!SimIO.eof()) {
            values = SimIO.readCommentedIntegerLine();
            for (i = 0; i < values.length; i++) {
                memory[address] = values[i];
                address = address + 1;
            }
        }
    }
```

Accès mémoire

```
    // the method simulates the effect of activating the access
    // control line.

    private static void accessMemory() {


        if (rw) {

            // rw=True signifies "read"
            // copy the value from memory to processor

            mdr = memory[mar];

        } else {

            // rw=False signifies "write"
            // copy a value from the processor to the memory

            memory[mar] = mdr;

        }
    }

    // ...
```

FETCH-EXECUTE

```
// ''FETCH-EXECUTE'' cycle simulation starts
// at the address 00 00

public static void run() {

pc = 0;        // always starts at zero
h  = false;    // re-initialize the status bit halt

while (h == false) {

    // load opCode

    mar = pc;
    pc  = pc + 1;    // pc is incremented
    rw  = true;
    accessMemory ();
    opCode = mdr;
```

FETCH-EXECUTE (suite)

```
    // if the opCode is odd, this instruction
    // necessitates an operand

    if ((opCode % 2) == 1) {
        mar = pc;
        pc  = pc + 1;     // increment pc
        rw  = true;
        accessMemory (); // reading the high part of
        opAddr = mdr;     // this address
        mar = pc;
        pc  = pc + 1;     // increment pc
        rw  = true;
        accessMemory (); // read low part of this address
        opAddr = 100 * opAddr + mdr;  // put high+low together
    }
```

FETCH-EXECUTE (suite)

```
    // execute the instruction

    switch (opCode) {

    case LDA: {
        mar = opAddr;     // read value designated by operand
        rw  = true;
        accessMemory ();
        a   = mdr;        // put this value into accumulator
        break;
    }
```

FETCH-EXECUTE (suite)

```
    case STA: {
        mdr = a;          // put content of the accumulator
        mar = opAddr;     // at address designated by opAddr
        rw  = false;
        accessMemory ();
        break ;
    }

    case CLA: {
        a   = 0;          // clear = set accumulator to zero
        z   = true;       // also sets status bit Z and N
        n   = false;
        break ;
    }
```

FETCH-EXECUTE (suite)

```
case INC: {
    a   = (a + 1) % 100; // increment = add 1 to accumulator
    z   = (a == 0);      // affect the status bits
    n   = (a < 0);
    break ;
}

case ADD: {
    mar = opAddr;          // read value designated by operand
    rw  = true;
    accessMemory();
    a   = (a + mdr) % 100; // add this value to accumulator
    z   = (a == 0);        // update the status bits
    n   = (a < 0);
    break ;
}
```


FETCH-EXECUTE (suite)

```
case SUB: {
    mar = opAddr;          // read value designated by operand
    rw  = true;
    accessMemory();
    a   = (a - mdr) % 100; // subtract from the accumulator
    z   = (a == 0);        // update the status bits
    n   = (a < 0);
    break;
}

case JMP: {
    pc = opAddr;           // the operand contains the address
    break;                 // of next instruction to be executed
}
```

FETCH-EXECUTE (suite)

```
case JZ : {
    if (z) {               // branch if Z
        pc = opAddr;
    }
    break;
}

case JN : {                // branch if N
    if (n) {
        pc = opAddr;
    }
    break;
}

case HLT: {
    h = true;              // sets H to true
    break;
}
```

FETCH-EXECUTE (suite)

```
case DSP: {
    mar = opAddr;          // read value designated by operand
    rw  = true;
    accessMemory();

    // in order to produce a clean output add zeros to the left
    // if necessary

    String smar = "" + mar ;
    while (smar.length() < 4) {
        smar = "0" + smar;
    }

    String smdr = "" + mdr ;
    if (mdr < 10) {
        smdr = " " + smdr ;
    }

    System.out.println("memory location " + smar +
                       " contains " + smdr);
    break;
}
default: System.out.println ("Error - unknown opCode: " + opCode) ;
}
}
}
}
```

Résumé

- Vous devez
  - Comprendre les types primitifs et références
  - Maîtriser les structures de contrôles
- J'ai présenté un modèle simplifié de l'architecture d'un ordinateur
- J'ai simulé l'exécution d'un programme
- Vous devriez comprendre la notion de variable dans le contexte de cette architecture simplifiée

Prochain cours

- Retour sur les types primitifs et références
- Appel par valeur
- Portée

Ressources I

E. B. Koffman and Wolfgang P. A. T.
Data Structures : Abstraction and Design Using Java.
John Wiley & Sons, 2e edition, 2010.

P. Sestoft.
Java Precisely.
The MIT Press, second edition edition, August 2005.

D. J. Barnes and M. Kölling.
Objects First with Java : A Practical Introduction Using BlueJ.

Prentice Hall, 4e edition, 2009.

Pensez-y, n'imprimez ces notes de cours que si c'est nécessaire !