

## TME 6 : Micro-émulateur (entraînement au partiel sur machine)

### Déroulement du TME et du partiel sur machine

Ce TME est basé sur le partiel de l'année 2017–2018.

Il a pour but de vous familiariser avec l'environnement qui sera utilisé lors de l'épreuve sur machine.

L'examen se déroulera sur machine et sera individuel. Il demandera d'écrire des classes Java. Nous conseillons, sans que cela soit obligatoire, d'utiliser l'environnement de développement Eclipse. Les fichiers fournis seront évalués automatiquement et notés grâce à une série de tests JUnit 4 exécutés sur un serveur distant. Le serveur produit un compte-rendu de la compilation et de l'exécution des tests, et calcule votre note : la compilation sans erreur puis chaque test passé avec succès vous rapportera un certain nombre de points, suivant le barème précisé.

Pendant l'examen, vous travaillerez *off-line*, sans connexion au serveur de correction, sur un compte vierge. Les fichiers Java trouvés sur votre compte seront automatiquement ramassés à la fin de l'épreuve. Le sujet propose quelques tests JUnit pour vous auto-évaluer, mais le jeu de tests servant à la notation, plus complet, ne sera pas communiqué durant l'épreuve. Le rapport de correction sera consultable uniquement après l'épreuve. Il aura valeur de copie corrigée.

Toutefois, pour ce TME d'entraînement, nous allons travailler en mode connecté. Vous soumettrez au serveur de correction, au fur et à mesure du TME, une archive **zip** de votre travail pour correction immédiate. Nous utiliserons le serveur GitLab habituel, mais uniquement pour y déposer l'archive des sources, pas pour la gestion de projet git. Par ailleurs, nous fournissons ici directement l'ensemble des tests JUnit 4 qui ont servi à la notation, en plus de ceux fournis pendant l'épreuve.

Afin d'éviter que vos réponses soient rejetées, assurez-vous de :

- respecter les noms de classe, de package et de chemin des fichiers indiqués dans l'énoncé ;
- fournir des sources qui compilent, même si elles ne répondent pas complètement à la question (les méthodes qui fonctionnent pourront vous rapporter une partie des points ; une classe dans le mauvais package ou qui ne compile pas ne rapportera aucun point, même si certaines méthodes sont correctes) ;
- implanter les interfaces et hériter des classes spécifiées dans l'énoncé ;
- respecter les consignes de visibilité des méthodes ; les attributs seront toujours privés ;
- ne pas modifier les interfaces fournies dans l'énoncé ; sinon, les tests de notation, qui sont programmés vis à vis de ces interfaces, ne compileront plus.

Lors du partiel, tous les documents papier (transparents du cours, notes personnelles, etc.) seront autorisés ; attention, cela ne sera pas le cas lors de l'examen final sur table. L'utilisation d'appareils électroniques, de moyens de communication ou d'accès à internet (téléphone, tablette, ordinateur personnel, etc.) est interdite pour tous les examens.

### Mise en place

Nous n'allons pas utiliser git dans ce TME. Le serveur GitLab habituel sera utilisé, mais uniquement pour lancer la correction automatique, pas pour héberger vos sources. Le squelette du projet sera par contre importé dans Eclipse depuis une archive ZIP, comme ce sera le cas lors du partiel.

Téléchargez l'archive **Emulator.zip** disponible sur la [page Moodle du cours](#), semaine 7. Celle-ci contient les sources des classes et interfaces fournies par l'énoncé, les tests JUnit fournis lors du partiel 2017–2018 pour vous aider pendant l'épreuve (package `pobj.tme6.test`), ainsi que les tests JUnit qui ont servi à la notation mais n'étaient pas fournis durant l'épreuve (package `pobj.tme6.notation`).

Dans Eclipse, créez un projet à partir de l'archive. Pour cela, utilisez le menu « File > Import > General > Existing Projects into Workspace », sélectionnez « Select archive file », cliquez sur « Browse » et sélectionnez le fichier archive « **Emulator.zip** ». Vérifiez que le projet « Emulator » apparaît bien dans la boîte « Projects » et cliquez sur « Finish ». Vous devriez maintenant avoir un

projet nommé « Emulator » dans le « Package Explorer » d'Eclipse. Les fichiers sont stockés dans le répertoire `~/eclipse-workspace/Emulator/`, en supposant que votre *workspace* Eclipse se trouve dans le répertoire `eclipse-workspace` à la racine de votre compte.

Le sujet suppose que toutes les classes sont créées dans le package nommé `pobj.tme6`.


Connectez-vous maintenant sur le [serveur GitLab habituel](#). Vous y trouverez un projet GitLab nommé `Emulator`. Faites-en un *fork*. Le projet GitLab ne contient pas de projet Eclipse ni de sources, mais seulement les fichiers utiles à la notation sur le serveur, il est donc inutile de l'importer localement dans Eclipse.

## Notation

Pour obtenir votre note, vous devez déposer à la racine de votre *fork* GitLab vos sources sous forme d'une archive nommée `upload.zip`. L'archive doit contenir un répertoire `pobj`, contenant un sous-répertoire `tme6`, contenant vos sources. Pour créer l'archive, en supposant que vous utilisez le répertoire *workspace* d'Eclipse par défaut, vous pouvez faire :

```
cd ~/eclipse-workspace/Emulator/src
zip -r upload.zip pobj
```

Pour déposer une première fois l'archive sous GitLab, vous pouvez cliquer dans l'interface Web sur le bouton `+` à droite du nom de répertoire courant (`emulator/`), puis sur « Upload file ». Pour la remplacer par une nouvelle version, vous pouvez cliquer sur le nom du fichier, `upload.zip`, dans GitLab, puis sur le bouton « Replace ». Dans ce TME, il ne sera pas nécessaire d'utiliser git ni de synchroniser votre projet Eclipse avec le serveur GitLab : seul le dépôt de l'archive est demandé.

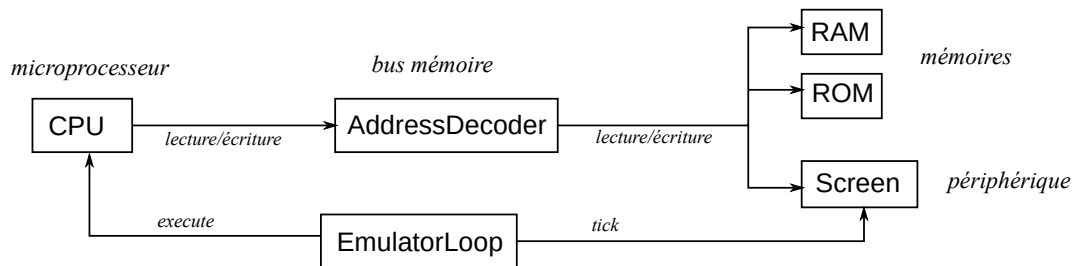
Après chaque dépôt de l'archive, le mécanisme d'intégration continue de GitLab relance le processus de correction automatique. Votre rapport de correction est disponible sous forme de page HTML dans l'onglet « Build / Pipelines » de votre projet. La ligne la plus en haut correspond à la dernière correction effectuée. Le rapport est alors disponible via l'icône « Artifacts »  dans la colonne de droite sous forme d'archive `zip` contenant un fichier `index.html` que vous pouvez ouvrir dans un navigateur. Notez que l'intégration continue indiquera toujours un succès (✓), même en cas d'erreur, il est donc important de consulter le rapport de correction généré pour savoir si vos classes sont effectivement correctes et connaître votre note.

Durant l'examen, afin que le ramassage automatique de vos sources fonctionne correctement, il sera important de respecter le nom des répertoires : le répertoire *workspace* (`~/eclipse-workspace`), le nom du projet Eclipse (`Emulator`) et les packages comme précisé dans l'énoncé (`src/pobj/tme6/...`). Par exemple, si nous étions en mode examen, au vu des consignes ci-dessus, seul le répertoire `~/eclipse-worspace/Emulator/src/pobj/tme6` (avec ses sous-répertoires) serait ramassé et corrigé.

## Introduction

Le sujet abordé dans cette épreuve est la conception d'un émulateur de système matériel, c'est à dire d'un logiciel capable de reproduire le fonctionnement d'un ordinateur, d'une console de jeu, d'une borne d'arcade, etc. et d'exécuter des logiciels conçus pour le matériel d'origine. Un émulateur doit reproduire fidèlement le comportement des différents composants matériels du système émulé : microprocesseur, mémoires, périphériques d'entrée/sortie. Nous allons en construire ici une version très simplifiée, en insistant sur la modularité et la réutilisabilité. De la même manière qu'un système est composé de plusieurs composants électroniques interconnectés, nous développerons ici un ensemble de classes réutilisables qui peuvent être connectées pour émuler différentes machines. Une boucle d'émulation se chargera alors d'orchestrer l'exécution des composants.

Voici une illustration partielle de l'architecture que nous allons créer :



## 6.1 Mémoires

Au niveau matériel, une mémoire est un circuit électronique permettant de lire et d'écrire des mots à des adresses. L'ensemble des adresses (l'espace d'adressage) est une plage contiguë d'entiers, de 0 à la taille de la mémoire moins 1. Un mot est un entier Java (`int`)<sup>1</sup>.

Les mémoires obéissent donc à l'interface `IMemory` suivante :

```

package pobj.tme6.memory;
public interface IMemory {
    int size(); // Taille de la mémoire
    int read(int addr); // Lit le mot à l'adresse addr, entre 0 et size()-1
    void write(int addr, int val); // Écrit val à l'adresse addr, entre 0 et size()-1
}
  
```

Les classes relatives aux mémoires sont définies dans le package `pobj.tme6.memory`.

### 6.1.1 Mémoire vive : RAM (1 pt)

Une mémoire vive (RAM) se comporte comme un tableau d'entiers. Écrivez une classe `RAM` dans le package `pobj.tme6.memory` qui implante l'interface `IMemory` et a un attribut de type tableau d'entiers. Le constructeur `public RAM(int size, int init)` alloue un tableau de taille `size` et initialise tous ses mots à la valeur `init`. Les méthodes `int read(int addr)` et `void write(int addr, int val)` lisent et écrivent, respectivement, la case du tableau à l'indice `addr`. On ne se préoccupera pas de vérifier les bornes d'accès dans le tableau. La méthode `int size()` retourne la taille du tableau.

**À fournir :** Un fichier `pobj/tme6/memory/RAM.java`.

**Fourni :** Dans l'archive `Emulator.zip`, l'interface `pobj.tme6.memory.IMemory` ci-dessus et une classe de test JUnit `pobj.tme6.test.TestRAM`. Le package `pobj.tme6.notation` contient les classes de test ayant servi à la notation, qui n'étaient pas fournies lors de l'examen.

### 6.1.2 Mémoire morte : ROM (1 pt)

Une mémoire morte (ROM) se comporte comme un tableau d'entiers en lecture seule : le contenu de la mémoire est fixé à la création de la ROM. La méthode `read` permet de lire un mot, mais `write` n'a aucun effet (elle ne modifie pas le tableau, mais ne provoque pas d'erreur non plus). Puisqu'une ROM ressemble à une RAM, notre classe `ROM` héritera de `RAM`.

Écrivez une classe `ROM` dans le package `pobj.tme6.memory` qui hérite de `RAM`, a un constructeur `public ROM(int data[])` qui crée une ROM de la taille de `data` et l'initialise avec le contenu de `data`. Attention : le contenu de `data` doit être copié dans la ROM, ainsi, si l'appelant modifie `data` après

<sup>1</sup>En réalité, on stockerait un octet, `byte`, à chaque adresse mémoire. Pour simplifier, notre émulateur utilisera partout des entiers classiques : `int`.

la construction de la ROM, le contenu de la ROM reste inchangé. La classe redéfinit `write` de manière à ce que le tableau ne soit pas modifié.

**À fournir :** Un fichier `pobj/tme6/memory/ROM.java`.

**Fourni :** Une classe de test JUnit `pobj.tme6.test.TestROM`.

### 6.1.3 Masque d'adresses : `AddressMask` (1.5 pts)

L'espace d'adressage vu par le microprocesseur est parfois plus grand que la mémoire réellement disponible. Par exemple, un microprocesseur 16-bit lira et écrira dans les adresses `0000` à `ffff` (0 à 65535 en décimal), alors que la RAM n'aura qu'une taille de 256 mots, donc un espace d'adressage dans `00-ff` (0 à 255). Une solution consiste à ne considérer que les 8 bits de poids faible de l'adresse et ignorer les 8 bits de poids fort lors de chaque lecture ou écriture. Les accès aux adresses `0005`, `0105`, `...`, `ff05` par le microprocesseur accéderont alors en réalité toutes à l'adresse `05` de la RAM. Il s'agit d'un *masque d'adresses*. Il peut être implanté par un *et bit à bit*, qui correspond à l'opérateur `&` en Java. Par exemple, garder uniquement les 8 bits de poids faible de `addr` s'écrit `addr & 0xff`.

Nous souhaitons pouvoir ajouter un masque d'adresses à toutes les implantations de mémoires (RAM, ROM, ou autres). Nous allons créer une classe qui s'utilise comme une mémoire (obéissant à l'interface `IMemory`) et prend en argument un objet de type `IMemory` à qui elle délègue les lectures et écritures après avoir corrigé les adresses. Il s'agit du motif Décorateur.

Écrivez une classe `AddressMask` dans le package `pobj.tme6.memory` qui implante l'interface `IMemory` et a un constructeur `public AddressMask(IMemory mem, int size, int mask)` prenant en argument une mémoire `mem` à décorer, une taille `size` et un masque `mask` ; ces trois informations sont stockées dans des attributs. La classe délègue `read(addr)` et `write(addr, val)` à la mémoire `mem` spécifiée à la construction, en lui passant l'adresse masquée, c'est à dire `addr & mask`. La méthode `size()` renvoie la valeur `size` spécifiée lors de la construction (et pas la taille de la mémoire `mem`).

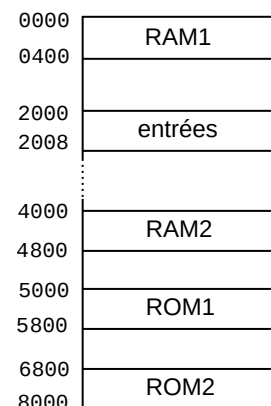
**À fournir :** Un fichier `pobj/tme6/memory/AddressMask.java`.

**Fourni :** Une classe de test JUnit `pobj.tme6.test.TestAddressMask`.

## 6.2 Décodeurs d'adresses

Un système comporte généralement plusieurs circuits mémoire (RAM, ROM, périphériques) connectés sur un bus commun à des adresses différentes. L'espace d'adressage est fragmenté : différentes plages d'adresses permettent d'accéder à ces différentes mémoires. Un circuit spécialisé, le *décodeur d'adresses*, se charge d'orienter les lectures et les écritures vers le circuit mémoire adéquat en fonction de l'adresse demandée.

La figure ci-contre donne un exemple. Un accès aux adresses `0000` à `03ff` accédera à RAM1, tandis qu'un accès aux adresses `5000` à `57ff` accédera à ROM1. Notez qu'un accès à l'adresse `5000` accédera à l'élément 0 de ROM1, et un accès à l'adresse `57ff` à l'élément `7ff` (`57ff - 5000`). Nous appellerons `5000` l'*adresse de base* de ROM1, et `i - 5000` sera l'adresse dans la ROM correspondant à l'adresse absolue `i`. Certaines adresses ne sont pas affectées, par exemple celles de `0400` à `1fff` : le décodeur d'adresses retournera une erreur en cas d'accès dans cette zone.



Nous allons programmer une classe `AddressDecoder` pour décrire cette organisation. Le décodeur d'adresses maintient une liste de paires associant une adresse de base et une mémoire. Vu de l'extérieur, le décodeur d'adresses se comporte comme une mémoire ; en interne, il redirige chaque accès `read` ou `write` vers la mémoire correspondante dans la liste, en ajustant l'adresse par soustraction de l'adresse de base. Il s'agit du motif Composite.

### 6.2.1 Mémoire avec adresse de base : `MemorySlot` (1 pt)

Programmez d'abord dans le package `pobj.tme6.memory` une classe `MemorySlot` qui est un simple conteneur pour une paire adresse de base (entière) et mémoire (de type `IMemory`). Elle contiendra : un constructeur `MemorySlot(int, IMemory)` et des getters publics associés `int getAddress()` et `IMemory getMemory()`. Comme les objets de cette classe ont vocation à être stockés dans une liste, il sera nécessaire de redéfinir la méthode `boolean equals(Object)` : deux `MemorySlot` sont égaux s'ils ont la même adresse de base et s'ils référencent la même mémoire `IMemory`.

**À fournir** : Un fichier `pobj/tme6/memory/MemorySlot.java`.

**Fourni** : Une classe de test JUnit `pobj.tme6.test.TestMemorySlot`.

### 6.2.2 Décodeur d'adresses : `AddressDecoder` (2.5 pts)

Programmez maintenant dans le package `pobj.tme6.memory` une classe `AddressDecoder` qui implante `IMemory`, a un constructeur `public AddressDecoder(int size)` spécifiant la taille de la mémoire et un attribut contenant une liste, initialement vide, de `MemorySlot`. La classe a des méthodes publiques `void add(MemorySlot)` et `void remove(MemorySlot)` pour respectivement ajouter une mémoire au décodeur d'adresses et l'enlever. Sa méthode `size` retourne la valeur spécifiée à la construction, tandis que `read` et `write` recherchent dans la liste de `MemorySlot` la mémoire correspondante, puis lui délèguent l'opération demandée après avoir ajusté l'adresse. Si toutefois aucun `MemorySlot` de la liste ne prend en charge l'adresse demandée, une exception `IllegalArgumentException` doit être signalée. On supposera, sans chercher à le vérifier, que les plages d'adresses des `MemorySlot` sont disjointes, donc un accès `read` ou `write` ne peut correspondre qu'à un seul `MemorySlot` au plus.

**À fournir** : Un fichier `pobj/tme6/memory/AddressDecoder.java`.

**Fourni** : Une classe de test JUnit `pobj.tme6.test.TestAddressDecoder`.

## 6.3 Duplication de l'état de la mémoire (3 pts)

Une fonctionnalité utile d'un émulateur est de sauvegarder l'état courant du système émulé complet, afin de reprendre plus tard l'exécution à ce point. Pour simplifier, nous nous intéressons ici uniquement à dupliquer l'état mémoire (similaire à l'opération *clone*).

Nous étendons l'interface `IMemory` modélisant la mémoire de manière suivante :

```
package pobj.tme6.memory;
public interface ICopyableMemory extends IMemory {
    ICopyableMemory copy();
}
```

1  
2  
3  
4

Modifiez les classes `RAM` et `AddressDecoder` pour qu'elles implantent `ICopyableMemory` : `RAM` doit dupliquer le tableau, tandis que `AddressDecoder` doit dupliquer récursivement les mémoires référencées par le décodeur.

**Important** : nous souhaitons garder la compatibilité avec les classes qui implantent uniquement `IMemory`. Il est donc interdit de modifier la classe `MemorySlot` ; elle garde un attribut de type `IMemory`. La méthode `copy` de `AddressDecoder` devra s'assurer que les mémoires sont en réalité des instances de `ICopyableMemory` avant d'appeler leur méthode `copy`. Si certaines mémoires ne sont pas `ICopyableMemory`, elle signalera une exception `UnsupportedOperationException`. Ainsi, tous les émulateurs continueront à compiler et à fonctionner du moment que la fonction de duplication n'est pas utilisée. Si celle-ci est utilisée, alors soit tous les composants de la machine supportent cette fonctionnalité, et la fonction réussit, soit une exception est levée.

**À fournir** : Des versions à jour de `pobj/tme6/memory/RAM.java` et de `AddressDecoder.java`.

**Fourni** : L'interface `pobj.tme6.memory.ICopyableMemory` et une classe de test `pobj.tme6.test.TestCopy`.

## 6.4 Périphériques

Un périphérique est un matériel qui s'exécute en même temps que le microprocesseur et communique avec lui. Nous allons implanter ici un périphérique de sortie (de type écran) qui possède une mémoire vive (la mémoire vidéo) dans laquelle le microprocesseur peut lire et écrire. Par ailleurs, indépendamment de l'action du microprocesseur, l'écran affiche régulièrement sur la console une ligne de texte correspondant au contenu de cette mémoire. Afin d'implanter une action périodique, la boucle principale d'émulation appellera régulièrement une méthode `tick(time)` du périphérique. Le paramètre `time` indique le temps écoulé depuis le dernier appel à `tick`. Tout périphérique qui doit effectuer une action régulière obéira donc à la signature `IDevice` suivante :

```
package pobj.tme6.device;
public interface IDevice {
    void tick(int time);
}
```

1  
2  
3  
4

Toutes les classes des périphériques seront dans le package `pobj.tme6.device`.

### 6.4.1 Périphérique périodique : `PeriodicDevice` (1 pt)

Souvent, un périphérique n'effectue pas une action à chaque appel à `tick`, mais attend qu'un laps de temps suffisant soit écoulé. Nous allons encapsuler cette fonction dans une implantation abstraite.

Programmez la classe abstraite `PeriodicDevice` dans le package `pobj.tme6.device` qui implante `IDevice`. Elle aura un constructeur `public PeriodicDevice(int period)` qui spécifie une période, ainsi qu'un attribut entier servant de compteur et initialisé à `period`. La classe a également une méthode `public void action()` qui sera laissée abstraite : elle sera définie dans les sous-classes. La classe implante `void tick(int time)` de la manière suivante : à chaque appel, le compteur est diminué de la valeur de `time` ; si le compteur devient négatif ou nul, alors la méthode `action` est appelée et le compteur est augmenté de `period`. Ceci a pour effet d'exécuter `action` toutes les `period` unités de temps.

**À fournir :** Un fichier `pobj/tme6/device/PeriodicDevice.java`.

**Fourni :** L'interface `pobj.tme6.device.IDevice` et une classe de test `pobj.tme6.test.TestPeriodicDevice`.

### 6.4.2 Écran : `Screen` (1.5 pts)

Un écran a deux fonctions :

- il comporte une mémoire de 10 mots où le microprocesseur peut composer une ligne de texte ;
- il affiche ce texte (avec `System.out.println`) toutes les 100 unités de temps.

C'est à la fois une mémoire et un périphérique, il obéira donc aux interfaces `IMemory` et `IDevice`.

Programmez la classe `pobj.tme6.device.Screen` qui implante `IMemory` et hérite de `PeriodicDevice`. Elle a un constructeur sans argument qui précisera au constructeur de `PeriodicDevice` une période de 100. Elle comporte une mémoire de 10 mots initialisés à 32 (caractère espace). Vous pourrez déléguer à un attribut de type `RAM` la gestion de cette mémoire, ainsi que les méthodes `read`, `write` et `size` demandées par l'interface `IMemory`. La méthode `action`, qui était laissée abstraite dans `PeriodicDevice`, est implantée ici : elle se chargera de l'affichage du texte avec `System.out.println`. L'affichage sera composée des mots de la mémoire vidéo, où chaque mot `m` est converti en un caractère par `(char)m`. Les caractères sont collés, sans espace ni retour à la ligne ; il n'y a un retour à la ligne qu'après les 10 caractères affichés. Ainsi, si la mémoire vidéo contient « 66 79 78 74 79 85 82 32 33 32 », le message « BONJOUR ! » sera affiché.

**À fournir :** Un fichier `pobj/tme6/device/Screen.java`.

**Fourni :** Une classe `pobj.tme6.test.ScreenMain` contenant un point d'entrée `main` qui affiche « BON-JOUR ! » deux fois sur la console (attention, ce n'est pas un test JUnit).

## 6.5 Microprocesseurs

Un microprocesseur exécute des instructions stockées en mémoire. Il obéit à l'interface suivante :

```
package pobj.tme6.cpu;
public interface ICPU {
    void reset(); // Réinitialise le CPU
    int execute(); // Exécute la prochaine instruction et retourne sa durée
}
```

où `reset` réinitialise l'état du microprocesseur et `execute` exécute une instruction et retourne un entier indiquant le temps d'exécution de l'instruction. L'émulateur (développé en partie 6.6) commencera par appeler `reset`, puis appellera `execute` dans une boucle qui s'arrêtera quand le temps total écoulé dépasse un seuil prédéfini.

Nous allons programmer une classe `CPU` implantant un microprocesseur très simple. Afin de rester modulaire, nous allons isoler dans des classes séparées la gestion de l'état du microprocesseur, l'exécution de chaque type d'instruction et le microprocesseur lui-même qui connecte ces classes.

Toutes les classes du microprocesseur seront dans le package `pobj.tme6.cpu`.

### 6.5.1 État du microprocesseur : `CPUState` (0.5 pt)

La classe `CPUState` gèrera l'état du microprocesseur. Ses attributs sont :

- une mémoire, de type `IMemory` ;
- un entier `PC` (le compteur de programme) indiquant l'adresse de la prochaine instruction ;
- un entier `A` (l'accumulateur) qui servira aux calculs numériques.

Elle obéira à l'interface `ICPUState` suivante :

```
package pobj.tme6.cpu;
public interface ICPUState {
    // getters et setters
    int getPC();
    void setPC(int pc);
    int getA();
    void setA(int a);
    IMemory getMemory();

    int fetch(); // Retourne le mot à l'adresse PC et incrémente PC
}
```

qui comporte notamment :

- des getters et setters pour `A` et `PC` ;
- un getter pour la mémoire (sans setter, celle-ci étant fixée à la construction) ;
- une méthode utilitaire `fetch` qui renvoie le mot stocké en mémoire à l'adresse `PC` courante et incrémente `PC` pour qu'il indique l'adresse du mot suivant en mémoire.

Programmez une classe `pobj.tme6.cpu.CPUState` qui implante `ICPUState` et a un constructeur publique `CPUState(IMemory)` qui fixe la mémoire et initialise `A` et `PC` à 0.

**À fournir :** Un fichier `pobj/tme6/cpu/CPUState.java`.

**Fourni :** L'interface `pobj.tme6.cpu.ICPUState` et une classe de test `pobj.tme6.test.TestCPUState`.



### 6.5.2 Opcodes (2.5 pts)

Une instruction du microprocesseur occupe normalement deux mots en mémoire : un mot indiquant le type d'instruction (le code d'opération, ou *opcode*), et le mot suivant indiquant un argument. La méthode `execute` de `CPU` va appeler `fetch` pour lire l'opcode à l'adresse `PC` ; puis déléguer l'exécution de l'instruction (dont le `fetch` de l'argument) à une classe dédiée.

Nous avons cinq types d'instructions, avec les opcodes suivants :

- 0 : **SET** : stocke l'argument dans `A` ;
- 1 : **ADD** : ajoute l'argument à `A` ;
- 2 : **LOAD** : stocke dans `A` la valeur en mémoire à l'adresse indiquée par l'argument ;
- 3 : **STORE** : stocke la valeur de `A` à l'adresse indiquée par l'argument ;
- 4 : **JUMP** : si `A > 0`, stocke l'argument dans `PC` (il s'agit donc d'un saut conditionné par la valeur de `A`).

Par exemple, les deux mots successifs 1 et 9 signifient « **ADD 9** », donc « ajouter 9 à `A` » avant de passer à l'instruction suivante. Par ailleurs, 4 et 20 aux adresses 10 et 11 signifient « **JUMP 20** », donc « aller à l'instruction d'adresse 20 si `A > 0` ; sinon, aller à l'instruction suivante, donc à l'adresse 12 ». Nous supposons que les instructions ont toutes un temps d'exécution de 2 (i.e., `execute` renvoie 2) : une unité de temps par mot lu en mémoire.

Les instructions obéissent à l'interface `IOpCode<T>` suivante du package `pobj.tme6.cpu.op` :

|  |                  |
|--|------------------|
| <pre>package pobj.tme6.cpu.op; public interface IOpCode&lt;T&gt; {     int execute(T state); }</pre> | 1<br>2<br>3<br>4 |
|--|------------------|

L'exécution d'une instruction doit accéder à l'état du microprocesseur (`A`, `PC`, mémoire) et le modifier. Nous le passons donc en argument *via* `state`. Nous utilisons un type générique, `IOpCode<T>`, afin de pouvoir réutiliser notre interface pour plusieurs microprocesseurs, ayant des définitions différentes pour leur état. Dans notre cas, il sera instancié avec `T = ICPUState`. Nous aurons donc une classe implantant `IOpCode<ICPUState>` par type d'instruction : **SET**, **ADD**, **LOAD**, **STORE** et **JUMP**.

La méthode `execute` de chaque classe fonctionne sur le même principe : elle appelle `state.fetch()` pour lire l'argument et incrémenter `PC`, elle modifie éventuellement `A`, `PC` ou la mémoire puis elle retourne 2 (i.e., le temps d'exécution). Notez qu'à la fin de l'exécution, `PC` donnera toujours l'adresse de l'opcode de la prochaine instruction à exécuter (généralement l'instruction suivant immédiatement celle qui vient d'être exécutée, sauf en cas de saut quand `A > 0`).

Programmez dans le package `pobj.tme6.cpu.op` des classes `OpSet`, `OpAdd`, `OpLoad`, `OpStore` et `OpJump` qui implantent `IOpCode<ICPUState>`, ont un constructeur publique sans argument et une méthode `int execute(ICPUState state)` qui exécute l'instruction comme décrit ci-dessus.

**À fournir** : Des fichiers `pobj/tme6/cpu/op/OpSet.java`, `OpAdd.java`, `OpLoad.java`, `OpStore.java` et `OpJump.java`.

**Fourni** : L'interface `pobj.tme6.cpu.op.IOpCode` et une classe de test `pobj.tme6.test.TestOpCode`.

### 6.5.3 Microprocesseur : CPU (1 pt)

Le microprocesseur `CPU` implante l'interface `ICPU` à l'aide des classes précédentes. En particulier, il délègue à `ICPUState` la gestion de l'état (`A`, `PC`, mémoire), et à une table de `IOpCode` l'exécution des instructions. Il possède donc :

- un attribut `state` de type `ICPUState` avec getter public `ICPUState getState()` (ce getter est indispensable pour les tests de notation) ;
- un attribut `ops` de type `List<IOpCode<ICPUState>>`, fixé par le constructeur, associant à chaque



entier de 0 à 4 une instance de `IOpCode` permettant d'exécuter l'instruction d'opcode donné (i.e., `OpSet` à l'indice 0, `OpAdd` à l'indice 1, etc.) ;

- un constructeur `public CPU(IMemory memory)` qui construit l'état `state` et initialise la liste `ops` ;
- une méthode `public void reset()` qui réinitialise `A` et `PC` à 0 ;
- une méthode `public int execute()` qui lit l'opcode à l'adresse `PC` avec `fetch`, retrouve l'instruction correspondante dans la table `ops`, et lui délègue l'exécution de l'instruction ; la valeur retournée par `execute` est celle retournée par la méthode `execute` de l'instruction ; si l'opcode lu n'est pas dans l'ensemble des valeurs autorisées, 0 à 4, alors la méthode se contente de retourner 1 sans lancer d'exception (ceci correspond à une instruction `NOP` (*no operation*) qui occupe un seul mot, sans argument, et dont l'exécution a une durée de 1).

Par exemple, si `PC=2` et `A=10`, et que l'on trouve en mémoire aux adresses 2 et 3 les mots 1 et 9, alors `execute` donnera `A=19` (l'instruction ajoute 9), `PC=4` (l'instruction occupe deux mots) et retournera 2 (le coût d'une addition est 2 unités de temps).

Voici un autre exemple : la séquence 0 10 1 -1 99 4 2, que l'on peut interpréter par `SET 10; ADD -1; NOP; JUMP 2`, est une boucle qui décrémente `A` à partir de 10 jusqu'à 0.

Programmez dans le package `pobj.tme6.cpu` la classe `CPU` décrite ci-dessus.

**À fournir :** Un fichier `pobj/tme6/cpu/CPU.java`.

**Fourni :** L'interface `pobj.tme6.cpu.ICPU` et une classe de test `pobj.tme6.test.TestCPU`.

## 6.6 Émulateur

Nous allons utiliser les classes des questions précédentes pour programmer un émulateur, en séparant l'étape de création de l'émulateur de son exécution proprement dite. La création (question 6.6.2) consiste à créer des instances du microprocesseur, des mémoires et des périphériques. L'exécution (question 6.6.1) consiste à appeler les méthodes `execute` du microprocesseur et `tick` des périphériques dans une boucle. La boucle d'exécution est indépendante du système émulé, et pourra être réutilisée avec d'autres microprocesseurs et périphériques.

### 6.6.1 Boucle d'émulation : `EmulatorLoop` (1 pt)

La boucle d'émulation est une classe qui a comme attributs un microprocesseur (sur lequel `execute` sera appelé) et une liste de périphériques (sur lesquels `tick` sera appelé). Elle obéit à l'interface `pobj.tme6.IEmulatorLoop` suivante :

```
package pobj.emu;
public interface IEmulatorLoop {
    void run(int time);
    void addDevice(IDevice t);
}
```

Programmez dans le package `pobj.emu` la classe `EmulatorLoop` qui implante l'interface `IEmulatorLoop`. Elle a un constructeur `public EmulatorLoop(ICPU cpu)` qui précise le microprocesseur. Elle a pour attribut une liste de périphériques `IDevice` initialement vide, et une méthode publique `void addDevice(IDevice)` pour ajouter des périphériques après la construction. Elle a une méthode publique `void run(int time)` qui implante la boucle d'émulation : à chaque tour de boucle, elle appelle d'abord `execute`, puis les méthodes `tick` de chaque périphérique en leur passant en argument la valeur retournée par `execute` (i.e., le temps écoulé par l'exécution de l'instruction). La boucle compte le temps total écoulé (i.e., la somme des valeurs retournées par `execute` depuis le début de la boucle) et se termine quand ce temps dépasse strictement `time`.

**À fournir :** Un fichier `pobj/tme6/EmulatorLoop.java`.

**Fourni :** L'interface `pobj.tme6.IEmulatorLoop` et une classe de test `pobj.tme6.test.TestEmulatorLoop`.

### 6.6.2 Création d'une machine émulée : **Emulator** (1 pt)

Nous allons maintenant créer un émulateur concret en connectant les classes définies aux questions précédentes. La machine émulée possède un microprocesseur **CPU**, un écran **Screen**, une ROM de 100 mots (initialisés par le constructeur) et une RAM de 100 mots (initialisés à 0). Son espace d'adressage est de 210 mots : 100 mots de ROM aux adresses 0 à 99, 100 mots de RAM aux adresses 100 à 199, et les 10 mots de mémoire vidéo aux adresses 200 à 209.

Programmez dans le package `pobj.emu` la classe **Emulator** qui a :

- un constructeur `public Emulator(int[] rom)` prenant en argument un tableau de 100 mots donnant le contenu de la ROM ; il crée une instance de **EmulatorLoop** en instanciant les classes **RAM**, **ROM**, **AddressDecoder**, **CPU** et **Screen** des questions précédentes ; attention : l'écran est vu à la fois comme une mémoire et un périphérique, il doit donc être à la fois connecté au décodeur d'adresses (avec `add`) et enregistré comme périphérique auprès de la boucle d'émulation (avec `addDevice`) ;
- un getter public `CPU getCPU()` pour retrouver le microprocesseur émulé (la présence de ce getter est indispensable pour les tests de notation) ;
- une méthode `public void run(int time)` qui appelle d'abord `reset` sur le microprocesseur, puis lance l'émulation pour `time` pas de temps par délégation à **EmulatorLoop**.

**À fournir** : Un fichier `pobj/tme6/Emulator.java`.

**Fourni** : Une classe de test `pobj.tme6.test.TestEmulator` et une classe `pobj.tme6.test.EmulatorMain` contenant un point d'entrée `main` qui affiche « ABC » sur la console.

## 6.7 Horloges avec alarmes (1.5 pts)

Une horloge est un périphérique permettant de programmer des alarmes, c'est à dire l'exécution d'une action à une date précise. Une action obéit à l'interface **IAction** suivante :

|  |                  |
|--|------------------|
| <pre>package pobj.tme6.device; public interface IAction {     void action(); }</pre> | 1<br>2<br>3<br>4 |
|--|------------------|

La date est un entier. L'horloge maintient une liste d'alarmes (date et action à exécuter) qui sera triée par ordre décroissant de date pour plus d'efficacité (la prochaine alarme se trouve en fin de liste). Programmez dans le package `pobj.tme6.device` une classe **Alarm** obéissant à l'interface **Comparable<Alarm>** (pour être triable) et possédant :

- un constructeur `public Alarm(int, IAction)` qui précise la date et l'action ;
- un getter public pour la date `public int getDate()` ;
- une méthode `public void action()` qui se contente d'appeler la méthode `action` de l'objet **IAction** spécifié à la construction ;
- une méthode de comparaison `public int compareTo(Alarm x)` qui renvoie 1 si la date de `x` est postérieure à celle de `this`, -1 si elle est antérieure, et 0 si elle est égale ; elle correspond donc à un ordre anti-chronologique.

L'horloge, quant à elle, est une classe **Clock** obéissant à l'interface **IDevice** pour être informée du temps qui s'écoule. Elle possède un entier `date` indiquant la date courante et une liste d'alarmes, triées par date. Chaque appel à `tick(time)` aura deux effets :

- mettre à jour la date courante ; il s'agira d'ajouter `time` à `date` ;
- rechercher dans la liste d'alarmes celles qui ont une date inférieure ou égale à la nouvelle date courante, exécuter leur action, et les retirer de la liste ; on bénéficie ici du fait que la liste est triée par ordre décroissant de date pour éviter de la parcourir intégralement : il suffira de l'explorer en partant de sa fin si le `compareTo` est correctement écrit.

Par ailleurs, la classe `Clock` offre une méthode `public void addAlarm(Alarm a)` permettant d'ajouter à tout moment une nouvelle alarme. Pour maintenir la liste des alarmes triée, on pourra utiliser la méthode statique `sort` de `java.util.Collections` après chaque ajout.

Programmez une classe `Clock` obéissant à l'interface `IDevice`, ayant un constructeur sans argument qui spécifie une date nulle et une liste d'alarmes vide, et des méthodes `addAlarm` et `tick` comme indiqué ci-dessus.

**À fournir :** Des fichiers `pobj/tme6/device/Alarm.java` et `Clock.java`.

**Fourni :** L'interface `pobj.tme6.device.IAction` et des classes de test `pobj.tme6.test.TestClock` et `pobj.tme6.test.TestAlarm`.

## 6.8 Bonus non noté : un émulateur réaliste

Cette partie, qui n'est pas notée, vous propose, si vous avez le temps, d'utiliser vos classes pour compléter un émulateur d'une machine réelle : le jeu d'arcade *Asteroids* (Atari, 1979).

Vous trouverez dans l'archive **Emulator.zip** un package `pobj.tme6.extra`.

La classe `pobj.tme6.extra.cpu.M6502` contient notamment l'implantation d'un microprocesseur MOS 6502 (1975), obéissant à l'interface `ICPU`. La classe `pobj.tme6.extra.dvg.DVG` contient l'implantation d'un circuit *Digital Vector Generator* d'Atari pour générer des graphismes vectoriels. Le répertoire `data` contient les ROM de la borne d'arcade. Le point d'entrée `main` de l'émulateur se trouve dans `pobj.tme6.extra.AsteroidsMain`. L'émulateur utilisant la bibliothèque JavaFX avec des modules, une version de Java supérieure ou égale à 11 est nécessaire.

Les classes suivantes de votre examen seront utilisées par l'émulateur : `RAM`, `ROM`, `AddressMask`, `AddressDecoder`, `EmulatorLoop`. Si vous avez correctement répondu à ces questions, alors la classe `AsteroidsMain` devrait compiler et donner un émulateur exécutable.

## Rendu de TME (OBLIGATOIRE)

Pour le rendu de ce TME particulier, nous demandons de placer sur le serveur GitLab uniquement le fichier `upload.zip`. Faites ensuite, comme d'habitude, une *release*. Assurez-vous que vous respectez bien les chemins de fichiers prescrits, et que la notation automatique tient bien compte de vos classes.

Dans la partie « Release notes » de votre *release*, vous indiquerez la note, sur 20, attribuée par la correction automatique.