INF1600 — TP2
Architecture à deux bus et introduction à l'assembleur IA-32

Giovanni Beltrame giovanni.beltrame@polymtl.ca
Imane Hafnaoui imane.hafnaoui@polymtl.ca
Karim Keddam karim.keddam@polymtl.ca

Polytechnique Montréal
Hiver 2016

Introduction

Après un premier travail pratique qui vous a introduit à la simulation d'une architecture de processeur complète en utilisant le logiciel Electric, vous êtes maintenant appelés à écrire des petits bouts de code en assembleur IA-32 et à vous approfondir en Electric (une dernière fois !) en étudiant une architecture à deux bus.

Comme tous les travaux pratiques de ce cours, ne prenez pas celui-ci à la légére. Il peut s'avérer particulièrement long lorsq'on ne prend pas bien le temps de comprendre le concept et le fonctionnement de la simulation sur Electric, dont la méthode sera un peu différente ici. Évitez donc de commencer le travail la veille !

Remise

Voici les détails concernant la remise de ce travail pratique :

— **Méthode :** sur Moodle (une seule remise par groupe).

— **Echéance :**

Groupes B2	23 & 24 Février
Groupes B1	01 & 02 Mars

— **Format :** un seul fichier zip, dont le nom sera `<Nom1>-<Nom2>.zip`. Exemple : `Hafnaoui-Keddam.zip`. Le contenu de cette archive doit être, précisément :

```
rapport.pdf
exo1
      tp2mem.txt
      tp2opalu.txt
      tp2ucode_adr.txt
      tp2ucode.txt
exo2
      tp2_2.c
      tp2_2.s
exo3
      tp2_3.c
      tp2_3.s
```

Le rapport écrit, inclus dans l'archive zip, doit comprendre une page de titre avec les noms et matricules des membres de l'équipe.

La correction sera particuliérement sévère en ce qui concerne le format de remise : veuillez donc respecter les chemins complets des fichiers demandés ainsi que la casse des caractères.

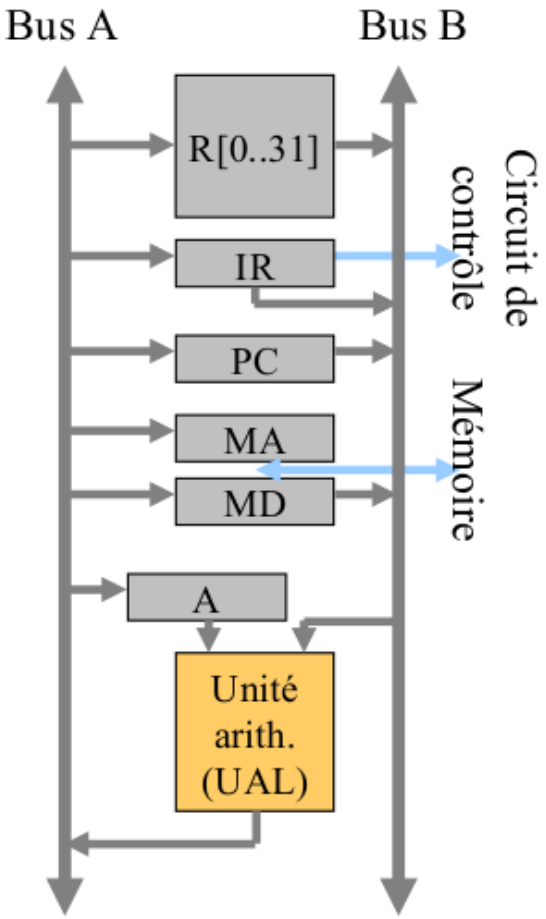
— **Langue écrite :** français.

— **Distribution :** les deux membres de l'équipe recevront la même note.

Barème

Contenu	**Points du cours**
Exercice 1 : a + b + c + d + e	0,4 + 0,8 + 0,8 + 0,6 + 0,4
Exercice 2	1,5
Exercice 3	1,5
Français écrit erroné	jusqu'à -0.6
Format de remise erroné (irrespect de l'arborescence ou des noms de fichiers demandés, fichiers superflus, etc.)	jusqu'à -1
Retard	-0,025 par heure

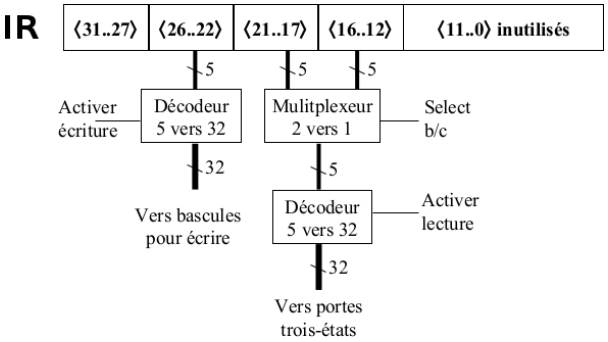
Exercice 1 Architecture avec microcodes



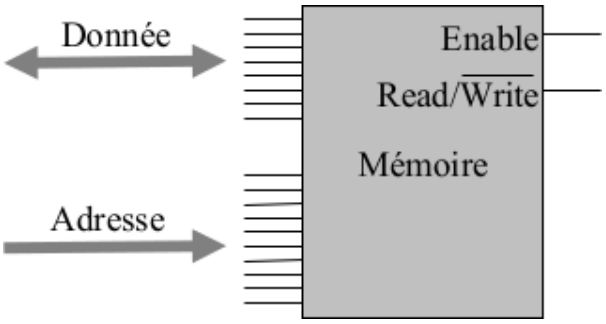
Soit l'architecture à deux bus décrite ci-contre. Les instructions 32-bit sont décomposées comme suivant :

```
op := IR<31..27>  rb := IR<21..17>
ra := IR<26..22>  rc := IR<16..12>
```

La sélection du registre dans la banque `R[0..31]` est faite comme dans cette image :



La mémoire est implémentée comme dans l'image ci-contre :



Lors d'une lecture :

— présenter l'adresse binaire à la puce (`MA`)

— mettre actif le signal de lecture

— la puce répond avec la donnée (`MD`).

Lors d'une écriture :

— présenter l'adresse binaire à la puce (`MA`)

— présenter la donnée binaire à la puce (`MD`)

— mettre actif le signal d'écriture.

L'UAL reçoit en permanence sur ses entrées le contenu du registre `A` et du bus `B`, puis reçoit comme contrôle directement le code de l'opération de l'instruction (`op`) pour savoir quelle opération effectuer. **Ce contrôle de l'UAL est différent du TP1**, où une seule instruction pouvait donner lieu à plusieurs opérations effectuées par l'UAL. Ici, une seule opération principale est contrôlée par l'instruction en soi. Le circuit de contrôle met les bits suivants en sortie sur des fils (nommés ici en ordre du plus significatif au moins significatif) :

Bit	Rôle
15	Fin du microprogramme. Si le contrôleur est dans le microprogramme d'une instruction : 1 dit d'aller à la recherche d'instruction au prochain cycle. Si le contrôleur est dans la recherche d'instruction, 1 dit que l'instruction se trouve sur le bus `A` (et l'écrit dans `IR`) et indique au contrôleur de sauter au bon microprogramme. 0 dit de passer normalement au prochain microcode.
14	Écrire `PC`.
13	Lire `PC`.
12	Écrire `MA`.
11	Source de `MD`. 0 pour que la donnée vers `MD` vienne du bus `A`, 1 pour que la donnée vers `MD` vienne de la mémoire.
10	Écrire `MD`.
9	Lire `MD`.
8	Lire/écrire mémoire. 0 pour lire la mémoire (dit à `MD` de ne pas écrire sur le bus de donnée de la mémoire), 1 pour écrire dans la mémoire (dit à `MD` d'écrire sur le bus de donnée de la mémoire).
7	Accès mémoire (doit être à 1 aussitôt qu'il y a un accès à la mémoire, que ce soit en lecture ou en écriture)
6 et 5	Contrôle de l'UAL. 0 dit de faire l'opération venant de l'instruction (selon `op`), 1 dit de faire une addition (sans tenir compte de `op`), 2 dit de faire « + 4 » sur la donnée provenant du bus `B` (sans tenir compte de `op`) et 3 dit de passer directement la valeur du bus `B` (sans tenir compte de `op`).
4	Écrire dans la banque de registres.
3	Lire dans la banque de registres.
2	Sélectionner le registre numéro `rb` ou `rc` en lecture : 0 pour `rb`, 1 pour `rc`.
1	Écrire `A`
0	Met la constante de l'instruction sur le bus `B` (`IR<11..0>`).

Questions

1. **Recherche d'instruction :**
   Donnez la séquence de microcodes qui correspond à la recherche d'une instruction. Donnez la réponse sous la forme d'un tableau. Le tableau suivant montre un exemple de ce qui est attendu (la première ligne étant déjà écrite pour vous) :

RTN concret	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	**hexa**
`MA <- PC;`	0	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0x3060
`MD <- M[MA] :` `PC <- PC + 4;`																	
`IR <- MD;`																	

2. **Execution d'une instruction générique :**
   Sous la même forme qu'en Question.1., écrivez la séquence de microcodes permettant d'exécuter l'instruction d'opérations arithmétiques/logiques typiques décrite par le RTN abstrait :

```
(IR<31..27> = opcode) ->
  R[IR<26..22>] <- R[IR<21..17>] oper M[R[IR<16..12>] + IR<11..0>];
```

   où `opcode` correspond au code d'opération requis pour exécuter l'opération arithmétique/-

logique `oper`.
N'oubliez pas d'inclure le RTN concret à chaque ligne du tableau pour justifier vos choix de signaux de contrôle.

3. **Simulation :**
Pour la simulation, nous allons encore utiliser Electric. La façon de simuler sera toutefois assez différente de ce que vous avez fait au TP1. Vous n'aurez pas à contrôler les signaux manuellement dans la fenêtre de simulation ; ces signaux seront plutôt le fruit de l'exécution d'un microprogramme.
Si vous avez bien compris l'architecture, vous voyez que chaque instruction possède un opcode différent et que chaque opcode donne lieu à un microprogramme. Plusieurs opcodes différents peuvent pointer vers le même microprogramme. Chaque microprogramme est constitué d'une série de microcodes de 16 bits (parfois seulement un). Les 16 bits en question sont ceux des tableaux que vous avez écrits en 1. et 2.

   — Depuis un terminal, créez un sous-répertoire dans votre répertoire personnel :

   ```
   $ cd ~
   $ mkdir inf1600_tp2
   ```

   — Depuis Moodle, téléchargez les fichiers nécessaires dans ce dossier nouvellement créé : `inf1600_tp2.zip` et `inf1600_tp2_config.zip`.

   — Depuis votre répertoire personnel, décompressez les fichiers de configuration Java :

   ```
   $ unzip inf1600_tp2_config.zip -d ~
   ```

   — Depuis le dossier `inf1600_tp2`, décompressez les fichiers de l'exercice :

   ```
   $ cd inf1600_tp2
   $ unzip inf1600_tp2.zip
   ```

   — Rendez-vous dans le dossier `exo1` qui vient d'être décompressé :

   ```
   $ cd exo1
   ```

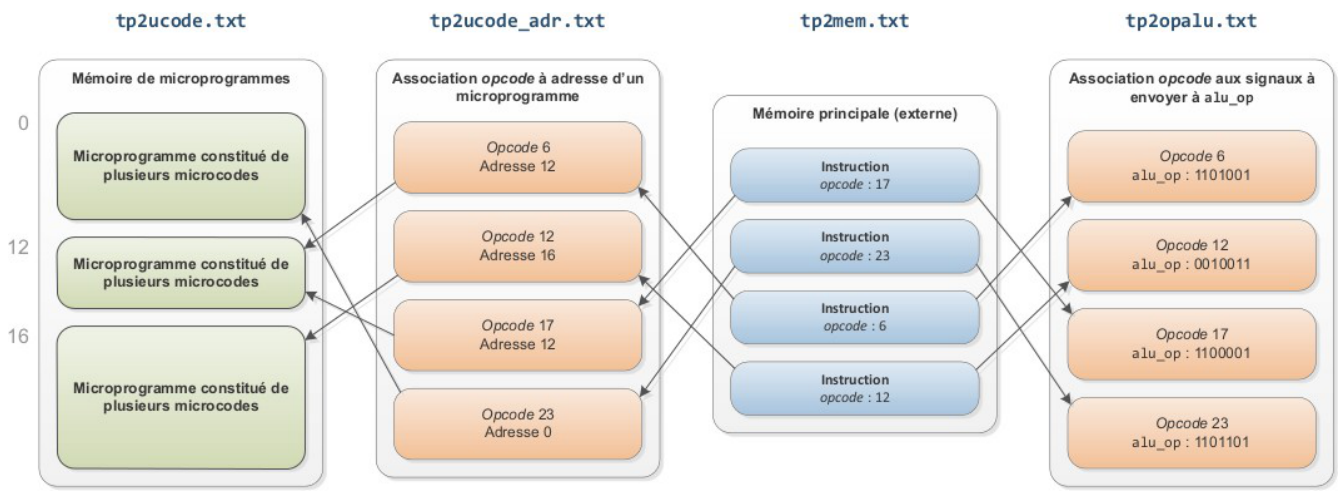
   — Enfin, depuis Moodle, téléchargez Electric dans le dossier `exo1`.

Le résultat de la commande `ls` devrait vous afficher tous les fichiers présents dans `exo1` :

```
$ ls
arch_2bus.vec          testLogique2.jelib  tp2opalu.txt      tp2ucode.txt
electric-8.05frb.jar   tp2mem.txt          tp2ucode_adr.txt
```

Vous voyez les quatre fichiers textes que vous devrez remettre. Ceux-ci interagissent ensemble pendant la simulation selon le schéma qui suit.

La mémoire principale (`tp2mem.txt`), qui n'est pas contenue dans le processeur, contient une séquence d'instructions. Un microprogramme se fera exécuter par le processeur pour chaque instruction. Pour trouver quel microprogramme est associé à quelle instruction, une table d'association est placée entre les deux. Si vous observez le fichier `tp2ucode_adr.txt`, vous verrez que les deux instructions dont les opcodes sont 12 et 14 sont associées au même microprogramme, tout comme les opcodes fictifs 6 et 17 dans le schéma ci-haut.
L'instruction détermine aussi l'opération principale effectuée par l'UAL. Chaque opcode est donc associé à un vecteur précis pour (qui contrôle, comme dans le TP1, l'opération de l'UAL). Le fichier `tp2opalu.txt` est responsable de cette association.



On vous demande donc de simuler ce que vous avez trouvé en 1. et en 2. Pour ce faire, entrez les séquences de microcodes aux endroits appropriés dans le fichier `tp2ucode.txt`. **Attention :** cette mémoire de microprogrammes (tout comme les autres) est lue en little-endian, alors assurez-vous d'écrire vos octets dans le sens inverse. Assurez-vous aussi de ne pas laisser de caractères invalides dans le fichier `tp2ucode.txt` modifié. Il est également conseillé d'utiliser `gedit` plutôt que `Kate` pour l'édition des fichiers textes.
Comme le fichier représentant la mémoire principale, `tp2mem.txt`, contient déjà une séquence d'instructions à simuler, vous n'avez pas à y toucher. Ces instructions vous serviront à vérifier si vos microprogrammes sont justes.
Une fois le fichier texte modifié et sauvegardé, lancez Electric comme d'habitude :

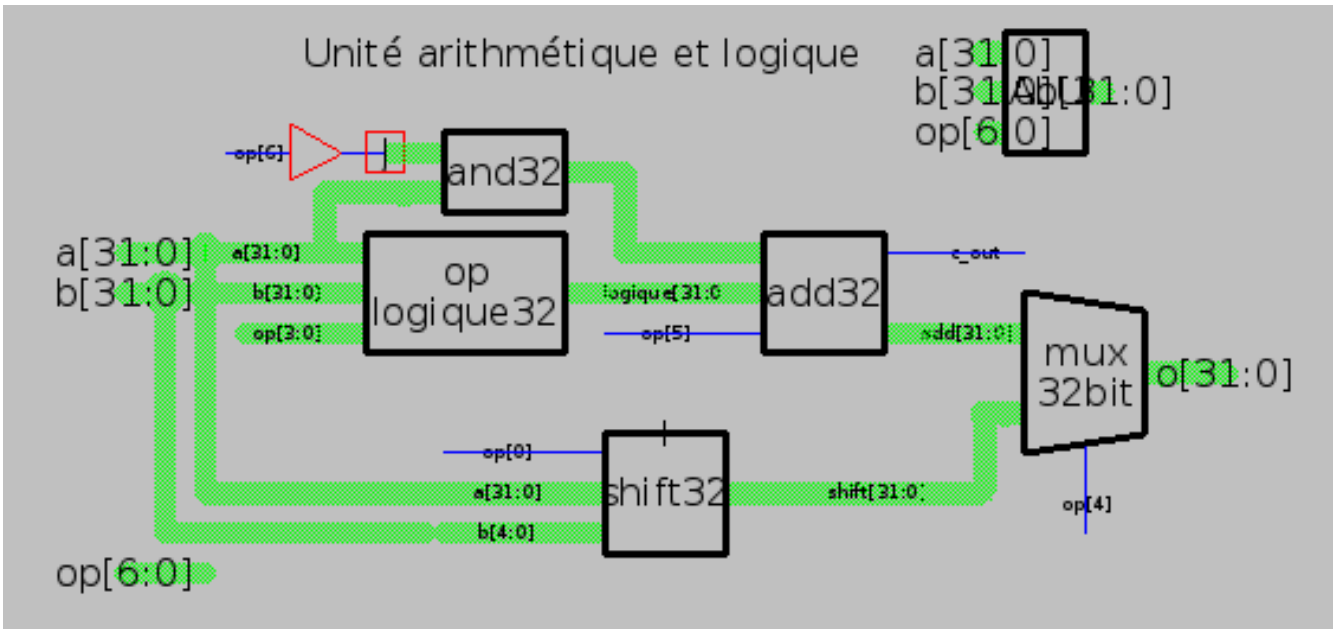
```
$ java -jar electric-8.05frb.jar &
```

Simulez de la même façon qu'au TP1 :

- Choisissez *File* > *Open Library...* (ou CTRL+O), puis ouvrez `testLogique2.jelib`.
- Dans l'onglet *Explorer* de la fenêtre, choisissez `arch_2bus/arch_2bus{sch}`.
- Pour voir le contenu d'un bloc, sélectionnez-le et appuyez sur CTRL+D (pour down). Pour en ressortir, appuyez sur CTRL+U (pour up). Vous pouvez ainsi naviguer dans l'architecture afin de bien comprendre son fonctionnement.
- Pour simuler, choisissez *Tool* > *Simulation (Built-in)* > *ALS : Simulate current cell*, dans la nouvelle fenêtre, sélectionnez le menu *Tool* > *Simulation (Built-in)* > *Restore Stimuli from Disk...* et choisissez `arch_2bus.vec`.

Comme la simulation prend beaucoup de cycles du processeur étudié, vous aurez à redessiner l'affichage si vous vous déplacez horizontalement. Pour ce faire, sélectionnez l'item de menu *Tool* > *Simulation (Built-in)* > *Update Simulation Window* après vous être déplacé vers la droite (Ctrl+6 sur le clavier numérique).
Prenez une capture d'écran et intégrez-la dans votre rapport. Cette capture doit bien montrer le résultat placé dans `R[1]`(`ECX` dans Electric) après l'exécution de l'instruction à l'adresse 8 (dans `tp2mem.txt`, donc après la 3ième instruction). Justifiez également le résultat obtenu dans votre rapport.
Pour rappel, l'adresse présentement exécutée est contenue dans le registre `PC`. Votre capture d'écran doit donc montrer les cycles pendant lesquels `PC` vaut 0xC .

4. **L'operation XOR :**
   Soit l'UAL définie dans la cellule ALU de la librairie du TP2 sur Electric. Quelle doit être la valeur de `op[6:0]` pour que l'opération finale soit un XOR (c'est-à-dire un OU exclusive logique bit à bit) ?
   Vous devez naviguer dans Electric (CTRL+D et CTRL+U) pour répondre à cette question.



   Écrivez cette valeur à l'endroit approprié du fichier `tp2opalu.txt` et relancez la simulation (redémarrez Electric pour être certain) afin de tester l'instruction XOR de l'adresse 0xc dans `tp2mem.txt`.
   Donnez aussi cette valeur dans votre rapport.

5. **Comprehension :**
   (a) Pour l'instruction 0x98765432 du processeur étudié dans ce TP, à quoi servent les données des deux derniers octets (0x5432) ? Donnez une autre instruction (sur 32 bits) qui aurait exactement le même effet d'exécution.
   (b) Nommez un avantage d'avoir une architecture à deux bus. Vous êtes -vous servi de cet avantage dans votre microprogramme développé en 2. ?
   (c) Diriez-vous que les instructions de cette architecture peuvent être aussi flexibles, en terme d'opérations arithmétiques/logiques, que celles du processeur étudié à l'exercice 4 du TP1 ? Pourquoi ?

Exercice 2 assembleur avec processeur à pile

Pour cet exercice, nous «simulons» un processeur utilisant une pile en se servant exclusivement de la partie FPU (unité à virgule flottante) du processeur Intel. Il s'agit d'une pile dédiée au calcul flottant (différente de la pile d'appel), de grandeur 8 (elle peut contenir jusqu'à 8 entrées de type *float*), mais il est rarement nécessaire de dépasser 2 ou 3 entrées. Les quelques instructions agissent toujours sur le premier et le deuxième éléments de la pile (`st[0]` et `st[1]`).

Voici ces instructions :

Instruction	**Rôle**
`flds x`	Ajoute au dessus de la pile l'entier à l'adresse mémoire `x` (`st[1]` prend la valeur de `st[0]` et `st[0]` devient cette nouvelle valeur chargée de la mémoire).
`fstps x`	Retire l'élément `st[0]` pour le mettre en mémoire principale à l'adresse `x`. `st[1]` devient `st[0]`.
`faddp`	`st[0]` est additioné à `st[1]` et le resultat remplace ces deux éléments.
`fsubp`	`st[1]` est soustrait de `st[0]` et le resultat remplace ces deux éléments.
`fsubrp`	`st[0]` est soustrait de `st[1]` et le resultat remplace ces deux éléments.
`fmulp`	`st[0]` est multiplié avec `st[1]` et le resultat remplace ces deux éléments.
`fdivp`	`st[0]` est divisé par `st[1]` et le resultat remplace ces deux éléments.
`fdivrp`	`st[1]` est divisé par `st[0]` et le resultat remplace ces deux éléments.

À l'aide ce ces instructions, écrivez l'expression suivante en assembleur :

$$a = \frac{g+d}{b \cdot c} \cdot \frac{e}{f-c} - b \cdot (g+d)$$

où a, b, c, d, e, f et g sont des variables de type *float* (IEEE-754 sur 32 bits) et leurs adresses en assembleur sont respectivement les symboles `a`, `b`, `c`, `d`, `e`, `f` et `g` (il s'agit ici de noms donnés aux adresses ; en assembleur, écrire `a` directement est équivalent à écrire la constante entière qui est l'adresse de la variable globale `a`).

Un programme écrit en langage C (`tp2_2.c`) ainsi qu'un squelette de fonction en assembleur (`tp2_2.s`) pour cet exercice sont fournis dans l'archive du TP afin de tester vos instructions. Vous n'avez qu'à remplir le squelette, puis compiler les deux et effectuer une édition de liens afin d'obtenir un exécutable. Utilisez ainsi la commande :

```
$ gcc -m32 -gdwarf-2 -o tp2_2 tp2_2.c tp2_2.s
```

afin d'obtenir l'exécutable `tp2_2` que vous pouvez lancer comme ceci :

```
$ ./tp2_2
```

à partir du répertoire dans lequel il se trouve. Le fichier `tp2_2.s` doit être complet et fonctionnel.

Exercice 3 conditions et branchements

Cet exercice vous entraînera à utiliser des instructions de type *jump* en assembleur IA-32. Toutes les instructions de branchements ne prennent qu'une opérande : l'adresse du branchement. L'adresse utilisée est généralement une étiquette placée dans le code.

Voici un exemple de branchement conditionnel qui permettra de sauter (à la *goto*) à l'étiquette nommée condition, si le contenu de `EBX` est strictement supérieur à celui de `EAX` :

```
    cmp %eax, %ebx
    ja  condition
    ...
condition:
    ...
```

En vulgarisant, les instructions de branchement conditionnel prennent le résultat de la dernière instruction arithmétique/logique afin de déterminer s'ils doivent effectuer le branchement ou non. Par contre, vous verrez plus tard dans le cours que ceci n'est pas exactement ce qui se passe.

Pour simplifier la notation RTN, nous regardons si `flags` est plus grand ou plus petit que 0 ; dans la réalité, un encodage sur 2 bits est utilisé dans le registre `flags` pour distinguer entre les 6 conditions indiquées ci-dessous. De plus, la notation `(R[a], flags) <- ...` est utilisée ici pour spécifier qu'à la fois `R[a]` prend le résultat de l'opération et `flags` est affecté selon ce résultat. Une description plus complète vous sera présentée en cours, mais n'est pas nécessaire pour cet exercice.

Instruction IA-32	**RTN abstrait**
`mov x, %a`	`R[a] <- M[x]`
`mov %a, x`	`M[x] <- R[a]`
`add %b, %a`	`(R[a], flags) <- R[a] + R[b]`
`add $x, %a`	`(R[a], flags) <- R[a] + x`
`cmp %b, %a`	`flags <- R[a] - R[b]`
`jmp x`	`PC <- x`
`ja x`	`(flags > 0) -> PC <- x`
`jna x`	`(flags <= 0) -> PC <- x`
`jae x`	`(flags >= 0) -> PC <- x`
`jnae x`	`(flags < 0) -> PC <- x`
`je x`	`(flags = 0) -> PC <- x`
`jne x`	`(flags != 0) -> PC <- x`

Les noms des registres disponibles sont `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi` et `%edi`.

Écrivez à l'aide de ces instructions la séquence suivante décrite en langage C :

```
for (i=0; i<=10; i++){
            if ((i<=e)||(b==c)) {
                    a=c+a;
                    b=b+2;
            } else {
                    a=a+d;
                    c=c-1;
            }
    }
```

où `a`, `b`, `c`, `d` et `e` sont des entiers signés (type *int* en langage C) sur 32 bits. Vous pouvez utilisez directement ces symboles pour représenter leurs adresses en assembleur, comme à l'exercice 2.

Un fichier en langage C vous est également fourni pour cet exercice (`tp2_3.c`), ainsi qu'un squelette de fonction en assembleur (`tp2_3.s`). Procédez comme à l'exercice 2 pour compiler et exécuter ce programme. Le fichier `tp2_3.s` doit être complet et fonctionnel.