INFO-F105 – Langages de programmation 1
Projet – Phase 3

1 Introduction

Lors de cette troisième et dernière phase, vous adapterez votre code pour y intégrer des types plus complexes et ainsi rendre votre code plus professionnel (mais aussi plus simple).

Cette phase est aussi une dernière occasion de vous assurer que vous comprenez bien votre code et la façon dont il s'exécute.

2 Consignes

2.1 Registres

Vos registres doivent désormais être représentés par une classe :

```
class Register {
 public:
  constexpr void operator=(uint16_t value);
  constexpr void operator+=(uint16_t value);
  constexpr void operator-=(uint16_t value);
  constexpr operator uint16_t() const;
};
```

Cette classe ainsi que les types de ses champs doivent être implémentés dans un fichier **séparé** que vous nommerez adéquatement.

Vous êtes libres d'ajouter les membres **privés** que vous jugerez nécessaires, mais vous ne pouvez pas définir d'autres membres publics.

Attention : n'oubliez pas qu'un registre contient un entier non signé et saturé de 16 bits.

Question : à quoi sert le dernier opérateur ? Essayez de le retirer puis recompilez votre programme. Y a-t-il quelque chose qui a changé ? N'oubliez pas de le remettre ensuite.

2.2 Instructions

Vos instructions doivent désormais être représentées par une structure :

```
struct Instruction {
  const Opcode opcode;
  const Operand* operands[2];

  Instruction(const std::string& raw);
  inline ~Instruction();
};
```

Cette structure ainsi que les types de ses champs doivent être implémentés dans un fichier **séparé** que vous nommerez adéquatement.

Chaque opérande peut être à `nullptr` si l'instruction ne contient pas cette opérande. Vous verrez de meilleures façons d'implémenter cela l'année prochaine lorsque vous apprendrez les concepts de la *programmation orientée objet*.

Question : que cela changerait-il si `Instruction` était une `class` plutôt qu'une `struct` ?

Opcode

L'opcode est représenté à l'aide d'une `enum` dont les valeurs sont uniques à chaque instruction.

Pour vous simplifier la tâche, nous considérons que les instructions similaires ayant des paramètres différents ont maintenant des opcodes différents :

Variante `NUMERIC`	Variante `REGISTER`
`SETv r n`	`SETr r1 r2`
`ADDv r n`	`ADDr r1 r2`
`SUBv r n`	`SUBr r1 r2`

Opérandes

Les opérandes sont des `struct` de 2 champs :

Champ	Type	Description
`type`	`OperandType`	le type d'opérande
`parsed`	`uint16_t`	La valeur de l'opérande

Le type d'opérande est représenté à l'aide d'une `enum`, cette fois-ci à deux valeurs :

- `NUMERIC`
- `REGISTER`

La valeur `parsed` d'une opérande `REGISTER` doit être unique pour chaque registre.

Exécution

Maintenant que vos instructions sont représentées avec une `enum`, modifiez la condition de votre fonction `exec` pour utiliser un `switch` plutôt qu'un `if`.

Cette nouvelle implémentation est normalement plus propre que la précédente, mais elle n'est toujours pas idéale. Vous verrez l'année prochaine comment remplacer ce gros `switch` par du *polymorphisme*, un concept essentiel de la programmation orientée objet.

2.3 Mémoire

Pour terminer, votre mémoire doit maintenant être implémentée à l'aide d'une classe `Memory`, dont la définition minimale se trouve ci-dessous. Le paramètre `nbits` indique la taille des adresses (en bits). Pour une mémoire de $256 = 2^8$ bytes, les adresses font 8 bits.

```
class Memory {
 private:
  uint8_t* MEM;

 public:
  inline Memory(uint8_t nbits);
  inline ~Memory();

  // TODO surcharge de l'opérateur []

  uint16_t pop();
  void push(uint16_t value);
};
```

Les fonctions `read` et `write` doivent être remplacées par une surcharge de l'opérateur `[]` :

```
Memory mem = Memory(8);  // taille = 2^8 bytes
mem[100] = 1234;
std::cout << mem[100] << std::endl;  // 1234
```

Notez que l'attribut `MEM` est maintenant un pointeur vers un `uint8_t`. Cela ne devrait pas trop vous perturber si vous gardez en tête que les tableaux (*arrays*) ne sont qu'une façon plus pratique de manipuler des pointeurs.

Conseil : pour la surcharge de l'opérateur, commencez par écrire la signature puis essayez petit à petit de convertir les informations que vous avez en celles que vous recherchez. Il est possible de le faire de façon très concise, mais il est important de bien visualiser le type de chaque valeur dans votre code pour y parvenir.

3 Exemple

L'exemple de la phase 2 est toujours applicable si on lui ajoute les suffixes d'instructions.

```
python3 tests3.py chemin/vers/votre/executable
```

4 Makefile

Votre `Makefile` doit contenir une entrée pour compiler votre programme. Par exemple:

```
EXEC := bin/exec

$(EXEC): main.cpp  # NOTE : cet exemple est incomplet
  g++ $^ -o $@
```

Pour simplifier la correction, il vous est également demandé d'ajouter une entrée à votre `Makefile` qui lancera le fichier `tests3.py` situé **à la racine** de votre projet.

```
test: $(EXEC)
    python3 tests3.py ./$(EXEC)
```

Important : vérifiez bien que tout fonctionne correctement avant de remettre votre projet.

5 Rapport

Outre le code, il vous est demandé de remettre un rapport de maximum 2 pages qui développera très brièvement l'organisation de votre code et vos choix d'implémentation.

Vous y réponderez notamment aux questions présentes dans les consignes et y décrirez les types de toutes les valeurs intermédiaires dans la fonction `Memory::operator[]`.

6 Remise

Comme pour les autres phases :

1. La **librairie standard** est la seule librairie autorisée.
2. Le code doit être **commenté**.
3. Le code doit compiler sans **warning**, à moins qu'un commentaire ne justifie sa présence.
4. Tous les fichiers sources (`.cpp`, `.hpp`) et `Makefile` doivent être remis.

Pour la dernière phase, veillez à bien respecter les points suivants :

1. Prenez soin d'appliquer les conseils donnés dans les **feedbacks** des phases précédentes.
2. Assurez-vous d'avoir une entrée `test` fonctionnelle dans votre `Makefile`.
3. Le programme doit passer tous les **tests** du fichier `tests3.py`
4. Les **cas limites** doivent être pris en compte.
5. Il ne peut y avoir **aucun** *memory leak* ni *double free*, ni aucune erreur liée aux pointeurs.

Consignes de remise

1. Mettez le tout dans un fichier `<matricule>.zip`
2. Remise sur l'UV
3. Date limite : le **vendredi 9 mai** avant **22h**