HE-IG VD | REDS Institut Reconfigurable and Embedded Digital Systems

Programmation assembleur (ASM)

Prof. Daniel Rossier
Assistants : Thomas Rieder, Bruno Da Rocha Carvalho

Directives de compilation et instructions de traitement

lab03 (22.03.2023)

Objectifs de laboratoire

L'objectif de ce laboratoire est de se familiariser avec les directives de compilation ainsi que les instructions de traitement x86 grâce à la réalisation de programmes assembleur. Par ailleurs, le laboratoire permettra d'examiner les différences entre le code résultant d'une compilation C et un code assembleur écrit à la main.

Echéance

- Le laboratoire sera rendu au plus tard la veille du prochain laboratoire (**durée : 2 semaines**)

Validation du laboratoire

Un script de rendu vous est fourni afin de packager les fichiers sources modifiés sans inclure tous les fichiers générés (code objet, exécutables, etc.). Ce script s'appelle ***rendu.sh*** et sera fourni pour tous les laboratoires.

```
reds@reds2022:~/asm/asm_lab03$ ./rendu.sh
```

Cette commande va générer un fichier *asm_lab03_rendu.tar.gz* et c'est cette archive qui sera à remettre sur Cyberlearn. Il est demandé de toujours préparer l'archive du laboratoire grâce au script de rendu fourni.

Étape 1 - Récupération des sources & environnement *Eclipse*

a) La récupération du dépôt s'effectue à l'aide de la commande suivante :

```
reds@reds2022:~$ retrieve_lab asm lab03
```

⇒ Ceci va créer un dossier « **asm_lab03** » qui contiendra les fichiers nécessaires au laboratoire.

Étape 2 – Utilisation des directives de compilations

Le répertoire «***x86/directives/*** » contient un fichier « *main.c* » ainsi qu'une série de fichiers assembleur (.S et .inc). Le fichier « *string_asm.S* » contient un programme qui permet de transformer les lettres d'une chaîne de caractères (*string*) en majuscules. La fonction « *to_upper_case()* » est appelée dans « *main.c* ». Les directives de compilation **sont manquantes** et le programme ne compile pas.

a) Compiler le code avec la commande « *make* ». Le programme ne compile pas et des erreurs devraient s'afficher dans le terminal.

⇒ Le programme a besoin des valeurs définies dans le fichier « *string_asm.inc* ».
⇒ Seul le fichier « ***string_asm.S*** » est à modifier.
⇒ Ajouter seulement des directives de compilation. **Le code ne doit pas être modifié**.

b) En tenant compte des erreurs affichées à la compilation, éditer le fichier « *string_asm.S* » et ajouter la directive qui permet au programme « *main.c* » d'utiliser la fonction « *to_upper_case()* ».
c) Ajouter la directive qui permet d'utiliser la fonction « *copy_string*() » du fichier « *copy_string_c.c* ».
d) Déclarer la constante « MAX_STRING_SIZE » avec la valeur 30 et les « variables » (espaces mémoire) manquant(e)s.

⇒ La longueur maximale du tableau est donnée par MAX_STRING_SIZE.

e) Ajouter les déclarations de section afin que le code soit dans la section « ***text*** » et les variables dans une section « ***data*** ».
f) Tester le programme et contrôler qu'il fonctionne correctement en utilisant le debugger dans *Eclipse*.

⇒ Comment l'algorithme fonctionne-t-il ? Qu'est-ce qui aurait pu être optimisé ? Est-il possible de transformer la chaîne de caractères « *in-place* » ?

Étape 3 – Opérations arithmétiques et logiques élémentaires

Le code se trouve dans le dossier « ***x86/gpio/*** » pour l'architecture x86. Le fichier « *gpio_c.c* » contient une fonction qui fait des opérations sur des « **GPIOs** » (*General Purpose Input/Output*)[1].

a) Compléter le fichier « *gpio_asm.S* » afin qu'il reproduise, en assembleur x86, la fonction « *config_register()* » du fichier « *gpio_c.c* ».

⇒ Pour x86, la valeur d'entrée est passée à la fonction dans le registre **eax** et la valeur de sortie doit être stockée dans **eax**. Chaque assignation de *gpio_reg* doit être écrite dans **eax**. Ce

[1] https://en.wikipedia.org/wiki/General-purpose_input/output

registre **ne doit pas** être utilisé pour stocker des valeurs intermédiaires (opérations à droite de l'assignation C « = »).

b) Compiler et désassembler le code avec l'utilitaire *objdump* de la *toolchain* « ***i686-linux*** » et comparer le résultat avec le code assembleur généré pour la fonction « *config_register()* ».

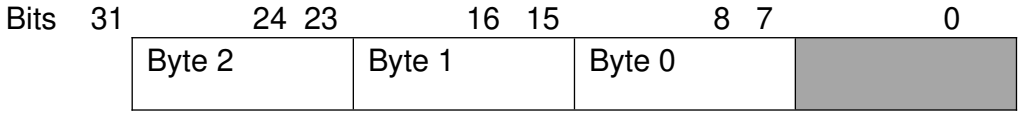
⇒ Quelles sont les différences entre le code assembleur généré suite à la compilation de « *gpios.c* » et le code assembleur créé ?

Étape 4 – Somme de contrôle par mot de parité

Le code se trouve dans le dossier « **x86/checksum/** ». Le fichier « *checksum.S* » contient la fonction « *parity_word()* » qui permet de calculer une somme de contrôle (*checksum*) pour des valeurs fournies en entrée avec la méthode de « mot de parité » (*parity word*)[2].

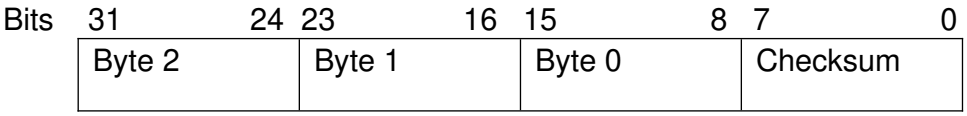
La fonction « *parity_word()* » reçoit 3 bytes de données dans le registre **eax** sous la forme suivante :

Registre **eax** :



Il calcule le *checksum* des trois bytes en faisant un « xor » des trois valeurs et retourne le résultat dans **eax** sous la forme suivante :

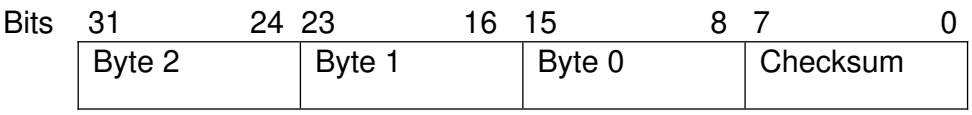
Registre **eax** :



a) Compléter la fonction « *parity_word()* » du fichier « *checksum.S* » et le tester au moyen du debugger.

La fonction « *check_parity()* » du fichier « *checksum.S* » permet de contrôler que le checksum soit correct pour les 3 bytes reçus. Il prend en entrée, dans le registre **eax** les trois bytes et le checksum sous la forme suivante :

Registre **eax** :



[2] https://en.wikipedia.org/wiki/Checksum#Parity_byte_or_parity_word

et retourne dans le registre **eax** la valeur 0 si aucune erreur n'est détectée et une valeur différente de 0 si une erreur est détectée.

b) Compléter la fonction « *check_parity()* » du fichier « *checksum.S* » et le tester au moyen du debugger.

Conseil : Regardez aussi le code C qui appelle le code assembleur.

Etape 5 – Conversion d'un caractère chaîne ASCII hexadécimal.

Le code se trouve dans le dossier «***x86/hex/*** ».
Pour cette étape il s'agira de convertir un caractère (ASCII[3]) en sa représentation hexadécimale. La fonction « char_to_hex*()* » du fichier « *convert.S* » devra convertir le caractère passé en sa valeur hexadécimale textuelle. Par exemple le caractère d'entrée 'Z' reçu dans le registre **eax** sous la forme suivante :
Registre eax:

Bits	31 24	23 16	15 8	7 0
	0	0	0	'Z' = 0x5A

Devra être converti en sa représentation hexadécimale sous forme de caractères "0x5A". Le résultat sera retourné sous la forme de 4 caractères dans le même registre 32-bits **eax**.

Bits	31 24	23 16	15 8	7 0
	'0'	'x'	'5'	'A'

a) Compléter le fichier « *convert.S* » et le tester au moyen du debugger.
Les deux octets de poids forts auront toujours la valeur '0' et 'x'
Exemples : 'A' devra donner "0x41", 'R' devra donner "0x52" et 'M' devra donner "0x4D"
Conseil :

- Utiliser des constantes (directives « *.equiv* »).
- Regardez aussi le code C qui appelle le code assembleur.
- La valeur est retournée au code C en passant par le registre **eax**.
- Utilisez l'instruction de comparaison « *cmp* » x86.
- Après une comparaison vous devez utiliser une instruction de branchement conditionnel (présentent dans la «Carte de référence rapide x86»). Ci-dessous un exemple d'utilisation du sant conditionnel :

```
movl $0x1, %edx
loop:
   cmpl $0x5, %edx
   incl %edx
   jl loop
other_stuff:
```

[3] https://en.wikipedia.org/wiki/ASCII