INF1900

Projet initial de système embarqué

Travail pratique No. 2

Machines à états finis logicielles

Objectif: Introduction aux machines à états finis logicielles

Date de remise: avant vendredi 25 janvier 17h00

Travail préparatoire: Aucun

Documents à remettre: Ces programmes seront corrigés (un par équipe de deux) et devront être enregistrés sous Git

Présentation en classe: fichier LibreOffice Impress

> *«Le monde que nous avons créé est le résultat de notre niveau de réflexion, mais les problèmes qu'il engendre ne sauraient être résolus à ce même niveau.»*
>
> - Albert Einstein

Dans le laboratoire précédent, vous avez été initiés à la programmation d'un processeur en utilisant un environnement de développement particulier.

Vous devriez maintenant:

- être à l'aise avec Geany, Visual Studio Code ou autre éditeur et AVR-GCC sous Linux
- utiliser correctement DDRx, PORTx et PINx
- utiliser adéquatement les fonctions de délais.
- Connaître certains opérateurs agissant sur des bits.

Introduction

En général, les programmes sont complexes et exigent une méthode de spécification adéquate et claire. Pour cette seconde séance de programmation, la méthode utilisée est la machine séquentielle. Elle peut être représentée sous la forme d'un graphe ou d'une table donnant les transitions. Les machines séquentielles sont aussi appelées machines à états finis ou automates d'états finis. Un exemple est illustré ci-dessous.

État présent	**A**	**B**	**État suivant**	**Sortie Z**
INIT	0	X	INIT	0
INIT	X	0	INIT	0
INIT	1	1	EA	0
EA	0	X	EA	0
EA	X	0	EA	0
EA	1	1	EB	0
EB	0	X	EB	0
EB	X	0	EB	0
EB	1	1	EC	0
EC	X	X	INIT	1

Table 1: Tableau des états

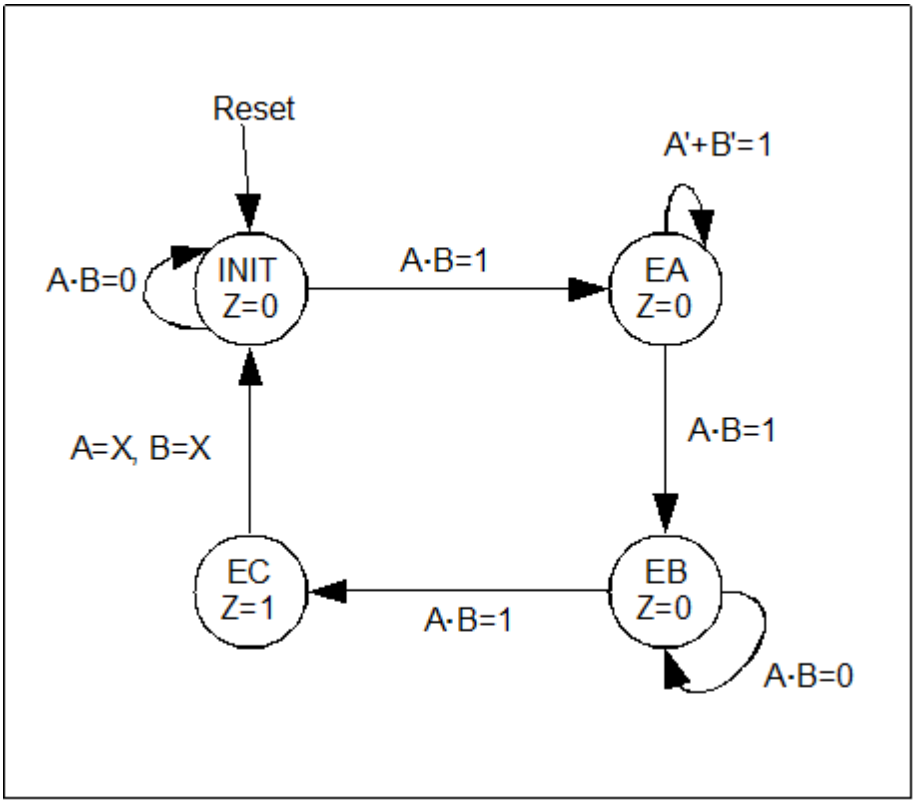

Figure 1: Exemple de diagramme d'états

Les diagrammes d'états peuvent également représenter un fonctionnement global. Chaque état ou «bulle» peut représenter une partie d'un mode de fonctionnement d'un plus grand système. À l'inverse, il peut aussi représenter un fonctionnement interne localisé et précis. Le début de cet article sur Wikipédia donne un aperçu du sujet.

Méthodologie

Voici les étapes que nous vous demandons de suivre pour construire précisément un diagramme d'états. Nous utilisons l'exemple de la table 1 pour préciser les éléments concernés:

1. Énumérer toutes les entrées. Dans l'exemple de la table 1, il s'agit de A et B.

2. Énumérer toutes les sorties. Z dans le cas de la table 1. Attention, une sortie n'est pas un état!

3. Énumérer tous les états possibles. Nous avons pour la table 1 quatre états (INIT, EA, EB et EC).

4. Normalement, en matériel, il faudrait encoder les états sur le nombre nécessaire de bits. En logiciel, on ne peut pas réduire le nombre de bascules utilisées. On travaille avec les registres fixes du processeur. Un simple registre de 8 bits peut encoder 256 états ce qui est beaucoup. On peut donc généralement éliminer cette étape, à moins de se trouver dans une situation très particulière (comme le déverminage avec un oscilloscope, par exemple, en suivant les changements d'état qui sont assignés sur un port en sortie). On pourra plus souvent laisser le compilateur choisir les zéros et les uns de chacun des états sans s'en soucier. On voudra plutôt que chaque état ait un nom bien identifié pour faciliter la lecture du programme.

5. Assigner un état initial au système. Le graphe de la figure 1 montre que cet état est INIT. Autrement dit, le signal Reset de l'exemple donné est associé directement à l'activation du bouton-poussoir du même nom sur votre carte mère. Son activation doit placer le système dans un état connu au départ. C'est un peu aussi pour cette raison qu'on ne considère pas le signal de reset comme étant une entrée. Généralement, dans une machine à états finis matérielle, il y a également un signal reset asynchrone disponible pour tout le circuit.

Ce qui est mentionné précédemment est vrai tant pour les systèmes matériels que logiciels. Cependant, pour l'aspect logiciel, nous devons ajouter quelques remarques supplémentaires pour tenir compte de certaines réalités:

6. Souvent, en programmation, pour bien identifier les états, on préfère définir un type énuméré (en langage C/C++, le type enum). La variable de ce type qui mémorisera l'état ne pourra donc prendre que ces valeurs et ressortira clairement dans le code. Ce principe est important, car autrement, on peut rapidement créer un trop grand nombre de variables «pour se souvenir de quelque chose» dans le code. Le résultat est souvent désastreux, car il n'y a

plus moyen de relire le code et de percevoir tous les états et transitions que peut traverser le système.

7. Il faut souvent créer des variables (au moins temporaires) pour représenter les entrées et les sorties. Quelques fois aussi, on voudra tout simplement avoir une variable temporaire qui identifie une combinaison particulière des entrées pour faciliter la programmation. Dans l'exemple plus haut, le produit de A et B revient souvent et on pourrait vouloir donner un nom à cette combinaison.

8. Coder les instructions qui permettent le changement d'un état à un autre en fonction de l'état présent et des entrées. Généralement, cette partie est écrite avec une instruction «switch-case» telle que vue en INF1005C.
9. Programmer le comportement des sorties en fonction de l'état présent seulement (machine de Moore) ou de l'état présent et des entrées (machine de Mealy). On pourrait se retrouver avec un deuxième «switch-case» pour couvrir les cas à décoder.

Les diagrammes d'états font aussi partie de la notation UML (*UML state diagram*) (ou en français et ici) qui est abordée dans le cours de LOG1000 et dans d'autres plus tard également.

Problèmes à résoudre

Vous devez faire allumer la DEL bicolore d'une manière précise, en utilisant uniquement la carte mère.

Problème 1

Les compteurs sont une forme de machines à états. On veut ici simplement que la DEL soit éteinte au départ. On doit appuyer et relâcher 5 fois le bouton-poussoir avant que la DEL

tourne au rouge pendant exactement 1 seconde. Par la suite, on revient au départ pour pouvoir recommencer.

Problème 2

Quand la carte mère démarre, la DEL libre doit s'allumer en rouge. Si le bouton-poussoir noir est pesé, la DEL affiche la couleur ambre. Quand le bouton-poussoir est relâché, la DEL devient verte. Si le bouton est de nouveau pesé, la DEL prend la couleur rouge encore. Quand il est relâché, la DEL s'éteint. Si le bouton est de nouveau pesé, la DEL affiche la couleur verte. Quand il est relâché, la DEL tourne au rouge ce qui fait que la carte mère est de retour à son état initial et tout peut recommencer.

Méthode de résolution de problèmes à employer

Dessiner les diagrammes d'états clairement sur papier avant de coder quoi que ce soit. Inclure la table des états (pas le diagramme) en commentaire dans l'en-tête de votre programme. Cette table sera corrigée avec votre programme.

Questions intéressantes à vous poser durant votre réflexion...

- Avez-vous besoin de faire un «antirebond» pour le problème 1? Et le 2? Expliquez.
- Combien de cycles de délai avons-nous besoin pour faire 1 seconde sur unmicrocontrôleur qui roule à 16 MHz? Et pour 8.192 MHz?

Soumission du programme (bien lire ce qui suit...)

Ces deux programmes seront corrigés. Ils devront être placés dans votre entrepôt Git. Pour faciliter la correction, les programmes devront être dans un répertoire précis, portés le nom exigé et accompagnés de Makefile. XY est votre numéro d'équipe (qui est aussi celui à

l'extérieur de votre boîte de robot). Cet entrepôt Git est différent de celui utilisé en LOG1000 et a le chemin suivant:

```
https://githost.gi.polymtl.ca/git/inf1900-XY/
```

Un fois votre entrepôt cloné, vous pouvez nommer les fichiers de la façon que vous le désirez à l'intérieur des deux répertoires suivant, tant que le code peut être compilé avec la commande «make». Par contre, les chemins dans l'entrepôt devront être les suivant:

```
<votre_entrepôt_local>/tp/tp2/pb1
<votre_entrepôt_local>/tp/tp2/pb2
```

Ceci a pour but de faciliter la correction par un script qui prendra les programmes de toutes les équipes dans les différents entrepôts pour les compiler automatiquement. Cette étape de la correction se fait donc rapidement tout en vérifiant si les instructions ont été suivies.
Il est fort probable que vos fichiers sources ne seront pas directement annotés par les chargés de laboratoire durant la correction. Un fichier `correction.txt` sera plutôt placé sous Git dans votre répertoire tp2 par le correcteur pour donner des remarques et la note. Les programmes seront corrigés selon le barème de la section évaluation du site web.

Attention! Linux est sensible à la casse (*case sensitive*, en anglais). En d'autres termes, les lettres majuscules et minuscules sont considérées comme étant distinctes.

Vous devez évidemment vous référer aux concepts enseignés en LOG1000 pour approfondir votre utilisation de Git. Ce petit guide n'est pas parfait et est même un peu trop complexe pour un premier usage mais constitue tout de même une bonne référence.

De plus, ce script bash vous sera utile pour vérifier la soumission de vos travaux par Git. Des instructions se trouvent à l'intérieur du fichier texte lui-même pour apprendre comment l'utiliser. Ce n'est pas une procédure de vérification absolue mais ça permet de gagner en confiance avec l’utilisation du système.

Enfin, il peut être intéressant de consulter la section évaluation du site web de cours. La liste des équipes et leur numéro, de même que la notation des évaluations de l'École y sont présentés. Si une erreur dans les numéros d'équipe, il faut en avertir le responsable du cours dans les plus brefs délais pour éviter les problèmes avec Git et la correction des travaux pratiques.