TP 5 : Interruptions et périphériques

Le but de cette séance est de comprendre le mécanisme d'interruption d'un processeur en réalisant des traitants d'interruption en langage d'assemblage.

Les exercices sont organisés comme dans les TP précédents : pour chaque fonction, il y a un fichier `exo.c` qui contient le programme principal, un fichier `fct_exo.s` à remplir, et une règle de génération dans le Makefile (`make exo`) pour générer l'exécutable.

Ex. 1 : Traitant d'interruption minimaliste : it

Le but de cet exercice est d'explorer le mécanisme d'interruption à partir d'un système très simple. Le code vous est donné, et permet de faire clignoter les LEDS à chaque appui sur un bouton poussoir. Il s'agit du programme `it`. Le traitant d'interruption est directement le code décrit dans `fct_it.s` à partir de l'étiquette `mon_vecteur`.

Concrètement, un clic sur les boutons poussoirs déclenche une interruption. Le processeur achève l'instruction en cours mais, au lieu d'exécuter l'instruction qui suit, saute à l'adresse du traitant d'interruption qui est positionné dans le registre `mtvec` lors du boot.

Dans cet exercice, on étudiera aussi une séquence de boot minimale qui est donnée dans le fichier `crt.s`. Pour le bon fonctionnement de la suite, il faut dès à présent faire `make crt.o` sur la ligne de commande. Ceci permet de substituer au boot par défaut (le `crt.o` présent dans la directive `INPUT` du script de l'éditeur de liens) le contenu du fichier `crt.o` local.

Question 1 Compilez et testez l'exécutable `it` sur le simulateur QEMU en remplaçant l'option `-nographic` par `-display default,show-cursor=on -serial mon:stdio` qui vous permet d'utiliser la souris pour cliquer sur les boutons de la plateforme. Attention, le simulateur ouvre deux fenêtres, celle qui nous intéresse est cachée sous l'autre (qui est complètement noire). Pour fermer le simulateur, vous pouvez simplement cliquer sur la croix dans la barre de titre de l'une de ses fenêtres.

Question 2 Ouvrez le fichier `fct_it.s`, et répondez aux questions suivantes :

1. Quels sont les registres sauvegardés et restaurés lors d'une interruption dans le cas présent, et pourquoi ?
2. Pourquoi la routine se termine par `mret` et non `jr ra` (pseudo-instruction `ret`) comme habituellement pour une fonction ?
3. À quoi sert l'instruction `lw t2, 0(t1)`, que se passe-t-il si on l'enlève (tester) ?
4. Comment écrit-on sur les LEDS ?

Question 3 À présent on va s'intéresser au mécanisme d'interruption. Placez vous en mode debug (en ajoutant les options -s et -S sur QEMU et en lançant en parallèle GDB). Lancez l'exécutable `it` sur le simulateur. Commencez l'exécution en pas-à-pas et vérifiez que le registre `mtvec` est chargé avec l'adresse du traitant d'interruption. Placez ensuite un point d'arrêt à

l'adresse du symbole `mon_vecteur`, et continuez l'exécution. Appuyez avec la souris sur un bouton poussoir de la plateforme dessinée dans le simulateur et exécutez en pas à pas le programme d'interruption et notez les étapes nécessaires à la gestion de l'interruption.

Question 4 Modifiez le programme présent dans `fct_it.s` pour qu'à chaque appui sur un bouton poussoir, `blink` soit incrémenté de 1 et non complémenté comme c'est le cas dans ce qui vous est fourni. Testez.

Question 5 Validez la question précédente à l'aide de l'infrastructure d'évaluation. Attention, au contraire des TP précédents, les programmes que l'on exécute ici ne s'arrêtent pas. Ainsi, générer la sortie avec une simple redirection ne suffit pas, il faut de plus interrompre le programme au bout d'un certain temps. Ici, on choisit un timeout de 10 secondes, donc la commande à utiliser est : `timeout 10 qemu-system-riscv32 -machine cep -bios none -nographic -kernel it > test/it.sortie`. Notez qu'on laisse l'option `-nographic` car on ne s'intéresse qu'à la sortie.

Question 6 Modifiez le programme présent dans `fct_it_incr_decr.s` pour qu'à chaque appui sur le bouton poussoir BTN0, `blink` soit incrémenté de 1 et à chaque appui sur le bouton poussoir BTN1, `blink` soit décrémenté. Le bouton numéro i correspond au bit d'indice $16 + i$ à l'adresse `REG_PINS_ADDR` et ce bit vaut 1 si le bouton a été pressé, 0 sinon. Testez.

Ex. 2 : Timer : interruption d'horloge

Dans cet exercice, nous allons exploiter plus de périphériques de la plateforme, et également utiliser un gestionnaire d'interruption beaucoup plus complet permettant de gérer plusieurs sources d'interruptions (`cep_exp.s`).

Le programme utilisera l'horloge intégrée dans le Core Local Interruptor (`CLINT`), qui est une horloge accessible par adresses (comme une mémoire), à l'aide des instructions `lw` et `sw`. Le `CLINT` possède 2 registres de 64 bits : `CLINT_TIMER` en lecture seule, à l'adresse `0x0200bff8`, et `CLINT_TIMER_CMP` à l'adresse `0x02004000`. Ils sont accessibles en 2 fois 32 bits, à l'adresse de base pour les poids faibles, et à l'adresse de base plus 4 pour les poids forts. Leur principe est que le registre `CLINT_TIMER` donne l'heure courante tandis que `CLINT_TIMER_CMP` donne l'heure à laquelle la prochaine interruption devra être levée. Ainsi, comme on peut le voir sur le schéma en annexe, l'interruption est déclenchée dès que l'heure courante dépasse l'heure prévue. Dans la spécification privilégiée du RISCV, le bout de code suivant nous est donné pour éviter de générer une interruption « fausse » qui serait due à la mise à jour par deux écritures successives (on ne peut écrire 64 bits d'un coup avec une machine 32 bits) de la valeur à laquelle se comparer :

```
# New comparand is in a1:a0.
li t0, -1
la t1, mtimecmp # Address of memory mapped register
sw t0, 0(t1)    # No smaller than old value.
sw a1, 4(t1)    # No smaller than new value.
sw a0, 0(t1)    # New value.
```

Notez que pour nous, l'adresse `mtimecmp` sera `CLINT_TIMER_CMP`.

Le gestionnaire d'interruption pour le timer va essentiellement mettre à jour un compteur. Le fichier `timer.c` fourni le code permettant de gérer le fonctionnement de ce compteur.

Question 1 Dans le fichier `fct_timer.s`, écrivez la fonction `reveil` qui écrit la date courante plus la valeur spécifiée en argument dans `CLINT_TIMER_CMP`, de sorte de reprogrammer une

interruption du timer au bout de l'intervalle de temps passé en argument. Attention, en RISCV la date courante et la date future sont des valeurs sur 64 bits, aussi la mise à jour n'est pas aussi triviale qu'elle n'y paraît : l'addition des poids faibles avec `delta_t` peut déborder.

Question 2 Compilez, puis exécutez votre code avec la commande :

```
qemu-system-riscv32 -machine cep -bios none -nographic -serial mon:stdio -kernel timer
```

De l'observation du résultat de l'exécution sur la sortie standard et du fichier `timer.c`, que pouvez-vous dire de l'invocation de la fonction `mon_vecteur_horloge` ? Vous pouvez arrêter la simulation depuis le terminal avec la combinaision de touches C-a x.

Question 3 Pour s'en assurer, relancez l'exécution en mode debogage (`-s -S`). Dans GDB, posez un point d'arrêt à l'adresse d'entrée des interruptions (`br mon_vecteur`), puis continuez (`c`) jusqu'au premier point d'arrêt. Observez et notez alors ce qu'il se passe jusqu'à la fin de ce traitant d'interruption (achevé par l'instruction `mret`). La commande GDB `list` peut vous aider à observer le code parcouru et la commande `s` vous permet d'avancer pas à pas [1].

Question 4 Générer la sortie de votre programme sans oublier de placer `timeout 10` en tête de votre commande. Commiter puis pousser vos modifications sur votre dépôt pour lancer l'évaluation.

Ex. 3 : Compteur commandé par les boutons poussoirs

L'exercice précédent vous a permis de voir que le traitant fourni délaisse pour l'instant les interruptions externes, par exemple les boutons poussoirs. Nous allons y remédier.

Pour aller plus loin...
Question 1 Dans le fichier `fct_timer_raz.s`, implantez à partir de l'étiquette `interruption_externe` le code permettant de mettre à zéro le champ `tics` de la structure `param`. Compilez le programme avec `make timer_raz` et vérifiez le résultat dans le simulateur sans l'option `-nographic` afin d'avoir accès aux boutons simulés (il ne se passe rien dans la fenêtre graphique, c'est normal) :

```
qemu-system-riscv32 -serial mon:stdio -display default,show-cursor=on -machine cep -kernel timer
```

Notez l'ajout de l'option `-serial mon:stdio` qui permet d'avoir la sortie textuelle sur le terminal et de l'option `-display default,show-cursor=on` qui permet d'avoir la souris du PC visible sur la pseudo-carte.

Pour aller plus loin...
Question 2 Adapter votre fonction dans le fichier `fct_timer_raz_btn0.s` de sorte que seul le premier bouton fasse la remise à zéro. Compilez (avec `make timer_raz_btn0`) puis testez à l'aide du simulateur.

Pour aller plus loin...
Question 3 Adaptez-le encore dans le fichier `fct_timer_papl.s` pour que :

- l'appui sur le deuxième bouton fasse que l'on incrémente les tics (cas par défaut) ;
- l'appui sur le troisième bouton fasse que l'on décrémente les tics ;
- l'appui sur le quatrième bouton fasse que l'on ne change pas les tics.

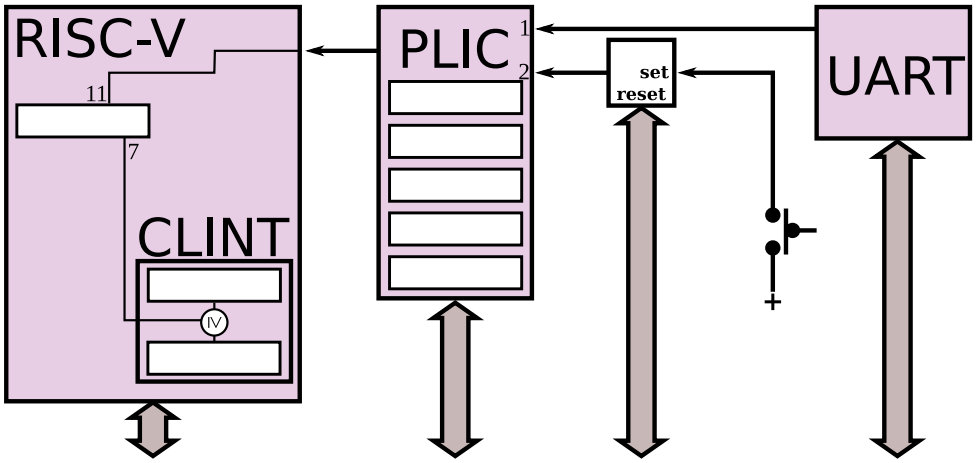
Utilisez le champ `sens` de la structure définie dans le fichier C pour réaliser ces différents cas. La pseudo-instruction `la` (*load address*) pourra vous servir pour récupérer dans un registre l'adresse

1. Notez que pour avancer instruction assembleur par instruction assembleur dans du code C, il faut faire un `display/i $pc` pour afficher l'instruction assembleur, puis `si`, pour avancer par instruction.

d’un symbole.

Annexes

Le schéma de connexion des lignes d'interruptions sur la plateforme QEMU :



Se référer aux `#define`s qui sont dans `cep_platform.h` pour les adresses des périphériques, au texte du TP pour ce qui concerne l'accès aux leds et à l'état des boutons-poussoirs, et à la page 24 du document `s51_core_complex_manual_v19_02` pour le CLINT.