Architecture RISC-V

J.-M Friedt, 16 mai 2019

Tous documents autorisés, connexion internet autorisée pour rechercher les documents cités dans le sujet, communications et téléphones portables proscrits.

Alors qu'il semblait acquis que l'architecure ARM allait dominer le monde de l'électronique numérique embarquée avec sa panoplie de processeurs répondant aux besoins allant du petit automate aux téléphones mobiles, 2018 s'avère peut être marquer la fin de cette hégémonie avec la mise sur le marché d'une architecture qui couvait depuis 8 ans à Berkeley. Cette architecture libre (sous licence BSD), initialement proposée comme *softcore* sur FPGA, a été présentée dans une puce silicium contrôlant un ordinateur pour la première fois au FOSDEM en 2018 (`https://archive.fosdem.org/2018/schedule/event/riscv/`) par SiFive, et donnera lieu à une session dédiée en 2019 (`https://fosdem.org/2019/schedule/track/risc_v/`). Non content de remettre en cause l'hégémonie d'ARM en lui imposant de libérer l'implémentation de certains de ses cœurs (`https://www.arm.com/resources/designstart/designstart-fpga`), cette nouvelle architecture libre impose à d'autre architectures de suivre la tendance, en particulier MIPS (`https://wavecomp.ai/mipsopen`). Les années à venir promettent donc une compétition excitante entre les "anciennes" architectures (ARM, MIPS, SPARC) et les nouveaux venus de la gamme RISC-V (Fig. 1) [1], avec laquelle il est certainement judicieux de se familiariser, d'autant plus qu'une version combinant CPU et FPGA, dans la lignée du Zynq, est proposée par Microsemi (ex-Lattice : `https://www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga`).

Figure 1: Logo du projet RISC-V

En l'absence de circuit matériel implémentant un des cœurs RISC-V, nous allons nous familiariser avec le travail sur cette architecture sur l'émulateur `qemu`. RISC-V est évidemment supporté par `gcc`. Bien que GNU/Linux soit supporté sur cette architecture, nous nous focaliserons ici sur sa programmation en C (*baremetal*) et au moyen de FreeRTOS [1].

Ressources :

- `qemu` compilé pour émuler RISC-V est disponible dans `/home/jmfriedt/qemu_riscv` avec les exécutables pour les diverses architectures dans `riscv64-softmmu/qemu-system-riscv64` (version 64 bits) ou `riscv32-softmmu/qemu-system-riscv32` pour la version 32 bits. La machine `-M sifive_e` est appropriée pour nos tests. On rappelle que `qemu` définit l'exécutable par `-kernel`.
- une chaîne de compilation croisée exécutée sur le processeur Intel x86 de l'ordinateur personnel et générant un code à destination d'un RISC-V dans le répertoire `/home/jmfriedt/toolchain-riscv`. Le préfixe de la chaîne de compilation est `riscv64-unknown-elf-` et les exécutables se trouvent dans le répertoire `bin`. On prendra soin de compléter la liste des répertoires consultés lors de la recherche d'un exécutable du chemin approprié pour accéder à ces ressources.

1 Programmation en C

Afin de s'échauffer sur la nouvelle architecture, nous nous proposons d'y exécuter un programme trivial permettant de vérifier le bon fonctionnement de la chaîne de compilation et son adéquation avec l'émulateur `qemu`. Nous avons vu que l'accès aux périphériques tels que les terminaux pour afficher des messages ne sont pas inclus dans le langage C natif mais doit être inclus sous forme d'appel à des bibliothèques externes. Pour RISC-V, la bibliothèque `libfemto` mise à disposition à `https://github.com/michaeljclark/riscv-probe` fournit les fonctions nécessaires à l'émulation d'un terminal.

1. Cloner le dépôt `riscv-probe` et le compiler. Démontrer votre capacité à exécuter dans `qemu` l'exemple `hello` disponible dans `riscv-probe/build/bin/rv32imac/qemu-sifive_e` en fournissant comme format de machine à `qemu` l'option `-M sifive_e` et en prenant soin de demander `-nographic` afin que l'affichage s'effectue sur le terminal courant. On tuera `qemu` après la démonstration en se déconnectant du terminal créé lors de la simulation en appuyant sur "CTRL-a" puis "x" (comme nous le ferions pour quitter `minicom`).
2. Proposer un programme trivial permettant d'afficher les nombres de 0 à 9 sur le terminal, le compiler et l'exécuter dans `qemu`. Afin de compiler, on pourra s'inspirer des commandes utilisées pour compiler les exemples de `riscv-probe` : ces commandes sont explicitées par `make V=2` pour rendre Makefile verbeux. On prendra soin notamment de faire pointer les dépendances vers ls bons répertoires.

Ne connaissant pas l'organisation de la mémoire lors de son utilisation par un cœur RISC-V, nous nous proposons de vérifier dans quel emplacement mémoire se trouve l'octet de poids le plus faible lors du stockage d'une variable codée sur 32 bits.

1. `https://fosdem.org/2019/schedule/event/riscvfreertos/attachments/slides/3235/export/events/attachments/riscvfreertos/slides/3235/1901_FreeRTOS_On_RISC_V_FOSDDEM.pdf`

3. Modifier le programme précédent afin d'afficher le nombre d'octets occupés en mémoire par une variable de type `long`. Quelle est cette valeur ?
4. Proposer un programme qui permette de comprendre comment ces octets sont organisés en mémoire, i.e. quel octet du mot est placé à l'adresse la plus faible et quel octet est placé à l'adresse la plus élevée. Quelle est cette organisation ? Pour ce faire, on pourra faire pointer un pointeur de caractères sur l'emplacement de l'entier à analyser, et observer l'ordre du contenu de ce tableau.
5. Compiler ce même programme sur PC (processeur Intel compatible x86) et fournir l'organisation de la mémoire par la même méthode. Est-ce qu'un processeur RISC-V peut échanger directement des données codées sur plus de 8 bits avec un PC architecturé autour d'un processeur Intel x86 ou faut-il manipuler ces octets pour que les deux interlocuteurs se comprennent ?
6. Reproduire l'expérience sur la version 64 bits du processeur. Est-ce que le format `long` est toujours codé sur le même nombre d'octets ? Reprendre les questions précédentes avec ce nouveau programme.

2 Programmation sous FreeRTOS

La version V10.2.0 de FreeRTOS [2] publiée le 25 Février 2019 supporte officiellement l'architecture RISC-V de processeur, et en particulier sa déclinaison 32 bits.

Télécharger les soures de FreeRTOS [3] à `https://sourceforge.net/projects/freertos/files/latest/download` et aller dans le répertoire `FreeRTOS/Demo/RISC-V-Qemu-sifive_e-FreedomStudio`. Nous y trouvons les scripts nécessaires pour compiler un exemple qu'il faut cependant modifier un peu pour s'adapter à la chaîne de compilation fournie par `https://github.com/riscv/riscv-gnu-toolchain` : nous remplaçons dans `BuildEnvironment.mk` l'appel à `riscv32-unknown-elf` par `riscv64-unknown-elf` et ajoutons dans `Makefile` l'ordre de lier le binaire sur la version 32 bits de la bibliothèque `libc` en ajoutant `LDFLAGS += -march=rv32imac -mabi=ilp32` après la première occurence de la variable d'environnement `LDFLAGS` autour de la ligne 128.

7. Démontrer votre capacité à compiler l'exemple `FreeRTOS-simple.elf` fourni comme exemple, et exécuter cet exemple dans `qemu` comme nous venons de le faire avec l'application en C. Quel est le fichier exécutable généré lors de la compilation ? comment vérifier que cet exécutable est compilé pour RISC-V : quelle commande fournit quel résultat permettant de conclure sur la nature de ce fichier exécutable ? Que fait ce programme lors de son exécution ?
8. en s'inspirant des programmes vus en cours, proposer une émulation de deux GPIOs dont l'allumage et l'extinction sont indiqués par des messages appropriés sur le terminal, et proposer un programme comportant trois tâches, deux dont la fonction est de faire clignoter un GPIO chacun au rythme de 2 et 3 Hz, et une troisième tâche affichant "Hello World" au rythme de 1 Hz. En cas d'échec de l'exécution, on se rappellera que le RISC-V est équipé d'un volume conséquent de mémoire sur laquelle il n'est pas nécessaire de trop se limiter.
9. Ajouter une tâche supplémentaire chargée d'afficher un second messages "Bonjour le monde" et garantir par le mécanisme approprié (quel est-il ?) que les deux messages n'interfèreront pas au cours de leur affichage.
10. Afficher la liste des tâches et leur statut au cours de l'exécution du programme par FreeRTOS. Commenter sur l'occupation en mémoire de la pile allouée à chaque tâche.

Références

[1] D.A. Patterson & J.L. Hennessy, *Computer organization and design – the hardware/software interface, RISC-V Edition*, Elesevier-Morgan Kaufmann (2018)

2. `https://www.freertos.org/History.txt`
3. version 10.2.0 à la date de rédaction de ce document

Corrections

1. `export PATH=/home/jmfriedt/enseignement/ufr/platforms/riscv/toolchain/bin/:$PATH`

   ```
   riscv-probe$ make
   ...
   riscv-probe$ [...]/riscv-qemu/riscv32-softmmu/qemu-system-riscv32 -M sifive_e -nographic \
     -kernel build/bin/rv32imac/qemu-sifive_e/hello
   ```

2. le programme de la forme

```
#include <stdio.h>

int main()
{int k;
 for (k=0;k<9;k++) printf("%d␣",k);
}
```

   donne après compilation par

   ```
   FEMTO=[...]/riscv-probe/

   riscv64-unknown-elf-gcc -Os -march=rv32imac -mabi=ilp32 -mcmodel=medany -c hello.c
   riscv64-unknown-elf-gcc -Os -march=rv32imac -mabi=ilp32 -mcmodel=medany -nostartfiles \
     -nostdlib -nostdinc -static -lgcc -T $(FEMTO)/env/qemu-sifive_e/default.lds \
     $(FEMTO)/build/obj/rv32imac/env/qemu-sifive_e/crt.o \
     $(FEMTO)/build/obj/rv32imac/env/qemu-sifive_e/setup.o hello.o \
     $(FEMTO)/build/lib/rv32imac/libfemto.a -o hello32
   ```

   le résultat

   ```
   [...]/riscv-qemu/riscv32-softmmu/qemu-system-riscv32 -M sifive_e -nographic -kernel hello32
   0 1 2 3 4 5 6 7 8
   ```

3. en architecture 32 bits, le long occupe 4 octets, tel que démontré par le code ci-dessous

```
#include <stdio.h>

int main()
{printf("\n%d\n",sizeof(long));}
```

4. L'affichage des octets individuels compris dans un mot codé sur plusieurs octets s'obtient par

```
#include <stdio.h>

int main()
{long x=0x12345678;
 char *c=(char*)&x;
 printf("%hhx␣%hhx␣%hhx␣%hhx\n",c[0],c[1],c[2],c[3]);
}
```

   0x78 apparaît en premier donc l'octet de poids faible est à l'adresse la plus faible : nous sommes dans un modèle *little endian*

5. L'organisation est la même sur PC : le processeur Intel x86 est lui aussi *little endian*. Les deux processeurs organisant leurs données de la même façon, ils peuvent échanger des données sans modifier leur *endianness*.

6. En compilant en 64 bits par (noter les options `-march=` et `-mabi`

   ```
   riscv64-unknown-elf-gcc -Os -march=rv64imac -mabi=lp64 -mcmodel=medany -c hello.c
   riscv64-unknown-elf-gcc -Os -march=rv64imac -mabi=lp64 -mcmodel=medany -nostartfiles \
     -nostdlib -nostdinc -static -lgcc -T $(FEMTO)/env/qemu-sifive_e/default.lds \
     $(FEMTO)/build/obj/rv64imac/env/qemu-sifive_e/crt.o \
     $(FEMTO)/build/obj/rv64imac/env/qemu-sifive_e/setup.o hello.o
     $(FEMTO)/build/lib/rv64imac/libfemto.a -o hello64
   ```

   nous obtenons sous la version 64 bits de qemu le résultat

   ```
   [...]/riscv-qemu/riscv64-softmmu/qemu-system-riscv64 -M sifive_e -nographic -kernel hello64
   hello world 8
   ```

   Cette fois un long occupe 8 octets (64 bits) mais l'organisation reste évidemment *little endian*.

7. `RISC-V-Qemu-sifive_e-FreedomStudio$ file build/FreeRTOS-simple.elf`
   `build/FreeRTOS-simple.elf: ELF 32-bit LSB executable, UCB RISC-V, version 1 (SYSV),`
   `statically linked, with debug_info, not stripped`
   Lors de son exécution sous `qemu`, ce programme affiche périodiquement le même message :

   ```
   [...]/RISC-V-Qemu-sifive_e-FreedomStudio$ riscv-qemu/riscv32-softmmu/qemu-system-riscv32 \
     -M sifive_e -nographic -kernel build/FreeRTOS-simple.elf
   core freq at 8628832 Hz
   Blink
   Blink
   Blink
   [...]
   ```

8. Nous pouvons utiliser en l'état `common.c` et `common.h` vus en cours puisqu'ils sont indépendants de la plateforme (émulation des GPIOs par des messages affichés sur le terminal). De ce fait, le programme se résume à

```c
#include <FreeRTOS.h>
#include <task.h>

void vApplicationMallocFailedHook( void );
void vApplicationIdleHook( void );
void vApplicationStackOverflowHook( TaskHandle_t pxTask, char *pcTaskName );
void vApplicationTickHook( void );

void vLedsFloat(void* dummy)
{while(1){
  Led_Hi1();
  vTaskDelay(501/portTICK_RATE_MS);
  Led_Lo1();
  vTaskDelay(501/portTICK_RATE_MS);
 }
}

void vLedsFlash(void* dummy)
{while(1){
  Led_Hi2();
  vTaskDelay(301/portTICK_RATE_MS);
  Led_Lo2();
  vTaskDelay(301/portTICK_RATE_MS);
 }
}

/* Writes each 500 ms */
void vPrintUart(void* dummy)
{portTickType last_wakeup_time;
 last_wakeup_time = xTaskGetTickCount();
 while(1){uart_puts("Hello World\r\n");
     vTaskDelayUntil(&last_wakeup_time, 500/portTICK_RATE_MS);
          // vTaskDelay(1001/portTICK_RATE_MS);
   }
}

int main(void){
 volatile int i;
 Usart1_Init(); // inits clock as well
 Led_Init();
 Led_Hi1();

 if (!(pdPASS == xTaskCreate( vLedsFloat, (signed char*) "LedFloat",192,NULL,1,NULL ))) goto hell;
 if (!(pdPASS == xTaskCreate( vLedsFlash, (signed char*) "LedFlash",192,NULL,2,NULL ))) goto hell;
 if (!(pdPASS == xTaskCreate( vPrintUart, (signed char*) "Uart",    192,NULL,3,NULL ))) goto hell;

 vTaskStartScheduler();
hell: while(1);
    return 0;
}

void vApplicationStackOverflowHook( TaskHandle_t pxTask, char *pcTaskName )
```

```
{taskDISABLE_INTERRUPTS();
 for( ;; );
}

void vAssertCalled( void )
{volatile uint32_t ulSetTo1ToExitFunction = 0;
 taskDISABLE_INTERRUPTS();
 while( ulSetTo1ToExitFunction != 1 ) __asm volatile( "NOP" );
}

void vApplicationMallocFailedHook( void ) { for( ;; ); }
void vApplicationIdleHook( void ) { }
void vApplicationTickHook( void ) { }
```

9. Nous modifions le programme suivant selon le listing qui suit, dans lequel les fonction inchangées ont été omises, afin d'utiliser un mutex pour protéger l'affichage :

```
#include <FreeRTOS.h>
#include <task.h>
#include "semphr.h"
#include "common.h"
#include "stdlib.h" // rand

xSemaphoreHandle xMutex;

void vApplicationMallocFailedHook( void );
void vApplicationIdleHook( void );
void vApplicationStackOverflowHook( TaskHandle_t pxTask, char *pcTaskName );
void vApplicationTickHook( void );

void vPrintUart(void* p)
{char *t=(char*)p;
 while (1) {
    xSemaphoreTake( xMutex, portMAX_DELAY );
    uart_puts(t);
    xSemaphoreGive( xMutex );
    vTaskDelay( ( rand() & 0x5 ) );
    }
}

int main(void){
 volatile int i;
 srand( 567 );
 xMutex = xSemaphoreCreateMutex();
 Usart1_Init(); // inits clock as well
 Led_Init();
 Led_Hi1();

 xTaskCreate( vLedsFloat, (signed char*) "LedFloat",192,NULL,1,NULL );
 xTaskCreate( vLedsFlash, (signed char*) "LedFlash",192,NULL,2,NULL );
 xTaskCreate(vPrintUart, (signed char*)"t1", 192, "1111111111111111111111111111111111111111111111\r\n\0", 1, 0);
 xTaskCreate(vPrintUart, (signed char*)"t2", 192, "2222222222222222222222222222222222222222222222\r\n\0", 1, 0);
 vTaskStartScheduler();
 while(1);
 return 0;
}
```

```
11111111111111111111111111111111111111111111
1111111111111111111111111111111111111112222222222222222222222222222222222222222222222

111
22222222222222222222222222222222222222222
```

10. On pense à activer les fonctions d'affichage de la liste des fonctions par

```
#define configUSE_TRACE_FACILITY        1
#define configUSE_STATS_FORMATTING_FUNCTIONS    1
```

en pensant à retirer l'option par défaut de désactiver ces fonctions (`#define configUSE_TRACE_FACILITY 0`) et une fois avoir configuré FreeRTOS de cette façon, nous reprenons le code vu en cours

```
#include <FreeRTOS.h>
#include <task.h>
#include "semphr.h"
#include "stdlib.h" // rand

xSemaphoreHandle xMutex;

void vApplicationMallocFailedHook( void );
void vApplicationIdleHook( void );
void vApplicationStackOverflowHook( TaskHandle_t pxTask, char *pcTaskName );
void vApplicationTickHook( void );

void vLedsFloat(void* dummy)
{[...]
}

void vLedsFlash(void* dummy)
{[...]
}

void vPrintUart(void* p)
{[...]
}

void ps(void* dummy)
{char c[256];
 portTickType last_wakeup_time;
 last_wakeup_time = xTaskGetTickCount();
 while(1){uart_puts("\nHello World\r\n");
//#define configUSE_TRACE_FACILITY        1
//#define configUSE_STATS_FORMATTING_FUNCTIONS    1
          vTaskList(c);
          uart_puts(c);
     vTaskDelayUntil(&last_wakeup_time, 500/portTICK_RATE_MS);
   }
}

int main(void){
 volatile int i;
 srand( 567 );
 xMutex = xSemaphoreCreateMutex();
 Usart1_Init(); // inits clock as well
 Led_Init();
 Led_Hi1();

 xTaskCreate( vLedsFloat, (signed char*) "LedFloat",192,NULL,1,NULL );
 xTaskCreate( vLedsFlash, (signed char*) "LedFlash",192,NULL,2,NULL );
 xTaskCreate(vPrintUart, (signed char*)"t1", 256, "1111111111111111111111111111111111111111111111\r\n\0", 1, 0);
 xTaskCreate(vPrintUart, (signed char*)"t2", 256, "2222222222222222222222222222222222222222222222\r\n\0", 1, 0);
 xTaskCreate(ps, (signed char*)"ps", 512, "", 1, 0);
 vTaskStartScheduler();
 while(1);
 return 0;
}

void vApplicationStackOverflowHook( TaskHandle_t pxTask, char *pcTaskName )
{taskDISABLE_INTERRUPTS();
 for( ;; );
}

void vAssertCalled( void )
{volatile uint32_t ulSetTo1ToExitFunction = 0;
 taskDISABLE_INTERRUPTS();
 while( ulSetTo1ToExitFunction != 1 ) __asm volatile( "NOP" );
```

```
65 }
66
67 void vApplicationMallocFailedHook( void ) { for( ;; ); }
68 void vApplicationIdleHook( void ) { }
69 void vApplicationTickHook( void ) { }
```

pour afficher la liste des fonctions et leur statut :

```
Led_Lo2
1111111111111111111111111111111111111111
2222222222222222222222222222222222222222
1111111111111111111111111111111111111111
2222222222222222222222222222222222222222

Hello World
ps              X       1       284     5
IDLE            R       0       22      6
t1              B       1       117     3
t2              B       1       117     4
LedFlash        B       2       73      2
LedFloat        B       1       73      1
Tmr Svc         B       6       20      7
Led_Hi1
1111111111111111111111111111111111111111
2222222222222222222222222222222222222222
Led_Hi2
```