Architecture RISC-V

J.-M Friedt, 22 janvier 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` et GNU/Linux y est donc fonctionnel. Nous désirons démontrer notre capacité à rapidement appréhender une nouvelle architecture mise en œuvre dans `buildroot` pour exécuter GNU/Linux sous `qemu`.

Ressources :

- `qemu` compilé pour émuler RISC-V est disponible dans `/home/jmfriedt/qemu_riscv` avec l'exécutable dans `riscv64-softmmu/qemu-system-riscv64` (noter que `poweroff` permet de quitter `qemu` en fermant la session GNU/Linux qui s'y exécute)
- `buildroot` configuré pour RISC-V et en particulier avec les sources de GNU/Linux se trouve dans `/home/jmfriedt/buildroot-riscv/`. Une copie des images générées par `buildroot` est placée dans `/home/etudiant/buildroot-riscv/` pour exécution avec `qemu` (ne pas tenter d'exécuter `qemu` sur les images situées dans `/home/jmfriedt`)

1 Espace utilisateur

Afin de s'échauffer sur la nouvelle architecture, nous nous proposons d'y exécuter un programme trivial permettant de vérifier le bon fonctionnement de GNU/Linux et de la chaîne de compilation. Commencer par lancer `qemu` en exportant le chemin contenant l'exécutable et en lançant [1]

```
qemu-system-riscv64 -M virt -kernel output/images/bbl -append "root=/dev/vda ro console=ttyS0"
-drive file=output/images/rootfs.ext2,format=raw,id=hd0 -device virtio-blk-device,drive=hd0
-device virtio-net-device,netdev=vmnic -netdev tap,id=vmnic,ifname=vnet0 -nographic
```

où `output/images/bbl` et `output/images/rootfs.ext2` réfèrent aux images fournies par `buildroot`, qu'il faudra donc préfixer du chemin complet pour accéder à ces fichiers.

1. Quel fichier sur la cible identifie l'architecture du processeur ? Qu'indique le contenu de ce fichier pour prouver que nous sommes sur la bonne architecture de processeur ?
2. Comment se nomme l'interface de communication à un réseau compatible internet sur l'émulateur ? Configurer l'adresse IP de cette interface en 192.168.3.2
3. L'interface virtuelle de communication avec l'interface réseau de qemu se nomme `vnet0` sur le PC. Configurer cette interface de façon appropriée pour pouvoir communiquer avec `qemu` : quelle est cette configuration et comment est-elle obtenue ? Vérifier cette configuration par `ping` depuis `qemu` vers le PC et réciproquement.
4. Écrire un programme chargé d'afficher "Hello World" dans la console et le compiler pour RISC-V.

1. on évitera les fautes de frappe en copiant la ligne de commande du fichier `/home/jmfriedt/qemu.sh`

5. Copier l'exécutable généré à la question précédente et démontrer son exécution sur architecture RISC-V. Noter que nous ne pouvons pas copier du PC vers qemu en l'absence du mot de passe de l'administrateur : on pourra soit passer par un répertoire partagé par le réseau, soit par une copie depuis qemu du fichier stocké sur le PC.

2 Espace noyau

Nous étant convaincus du bon fonctionnement du compilateur, nous désirons maintenant ajouter un module noyau chargé d'afficher "Hello World" lors du chargement du module, ainsi qu'accéder aux ressources communiquant avec du matériel.

6. Identifier l'emplacement des sources du noyau et sa version : quels sont-elles ?
7. Proposer un module noyau affichant le message "Hello World" lors de son chargement, le compiler (l'archicture RISC-V s'appelle `riscv` dans le noyau Linux) et démontrer son bon-fonctionnement sur la cible RISC-V. Comment observer le message ?
8. Rappeler le rôle du *devicetree*
9. Les nœuds du *devicetree* sont décrits dans `/sys/firmware/devicetree/base/` incluant le nom de chaque nœud suivi de l'adresse de base des registres. Quel outil en espace utilisateur vous permet de lire le contenu du premier registre (offset 0x00) du périphérique `virtio_mmio` ? Quelle est cette valeur ? est-elle cohérente avec la documentation `http://docs.oasis-open.org/virtio/virtio/v1.0/csprd01/virtio-v1.0-csprd01.html`
10. Compléter le module noyau ci-dessus en ajoutant l'accès à ce registre et démontrer l'affichage de son contenu dans les messages du système.

Références

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

Correction

1. `/proc/cpuinfo` indique `rv64imafdcsu`
2. `eth0` se configure par `ifconfig eth0 192.168.3.2`
3. il est judicieux de configurer l'interface `vnet0` sur le même sous réseau, par exemple `sudo ifconfig vnet0 192.168.3.1` qui se traduit par

```
$ ping 192.168.3.2
PING 192.168.3.2 (192.168.3.2) 56(84) bytes of data.
64 bytes from 192.168.3.2: icmp_seq=1 ttl=64 time=0.595 ms
64 bytes from 192.168.3.2: icmp_seq=2 ttl=64 time=0.696 ms
```

et

```
# ping 192.168.3.1
PING 192.168.3.1 (192.168.3.1): 56 data bytes
64 bytes from 192.168.3.1: seq=0 ttl=64 time=3.537 ms
64 bytes from 192.168.3.1: seq=1 ttl=64 time=0.792 ms
```

4. `int main() {printf("Hello World\n");}` se compile par `riscv64-linux-gcc -o binaire source.c` après avoir exporté le PATH pour inclure `/home/jmfriedt/buildroot-riscv/output/host/usr/bin` (on peut omettre `usr` dans ce chemin).
5. `scp root@192.168.3.1:/repertoire/binaire` puis exécution par `./binaire`, ou montage NFS :

```
# mount -o nolock 192.168.3.1:/home/etudiant/nfs /mnt
# scp 192.168.1.1:/tmp/*ko .
```

6. `/home/jmfriedt/buildroot-riscv/output/build/linux-fe92d7905c6ea0ebeabeb725b8040754ede7c220`
7. voir programme ci-dessous
8. liste des périphériques, leur configuration (en particulier plage d'adresses) et ressources associées
9. `devmem 0x10001000` renvoit `0x74726976` qui est évidemment cohérent avec la documentation
10. `ioremap` pour convertir l'adresse physique en adresse virtuelle.

```
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */
#include <linux/ioport.h>       // request_mem
#include <linux/io.h>           // ioremap

#define IO_BASE1 0x10001000

static void __iomem *jmf_gpio;   //int jmf_gpio;

int hello_start(void);
void hello_end(void);

int hello_start()
{int delay = 1;
 unsigned int stat;
 printk(KERN_INFO "Hello World\n");

 if (request_mem_region(IO_BASE1,0x12c,"GPIO cfg")==NULL)
     printk(KERN_ALERT "mem request failed");
 else
  {jmf_gpio=(void __iomem*)ioremap(IO_BASE1, 0x4);
   stat=readl(jmf_gpio+0x0);
   printk("stat=%x\n",stat);
   release_mem_region(IO_BASE1, 0x12c);
  }
 return 0;
}

void hello_end()
{printk(KERN_INFO "Goodbye\n");
}

module_init(hello_start);
module_exit(hello_end);

MODULE_LICENSE("GPL");
```

se compile par

```
obj-m +=hello.o

all:
   make ARCH=riscv CROSS_COMPILE=riscv64-buildroot-linux-gnu- -C \
/home/jmfriedt/buildroot-riscv/output/build/linux-8fe28cb58bcb235034b64cbbb7550a8a43fd88be/ \
M=$(PWD) modules

clean:
   rm *.o *.ko *.mod.c .*.cmd
```

pour donner

```
# insmod hello.ko
[  284.804000] hello: loading out-of-tree module taints kernel.
[  284.816000] Hello World
[  284.816000] stat=74726976
```