Lab #4

Objective: Using the U-Boot embedded bootloader
Outcomes:
After this lab, you will be able to
- Execute bare-metal application on QEMU ARM using U-Boot
- Boot Linux with U-Boot on QEMU ARM
- Understand U-Boot startup

U-Boot on bare-metal ARM

Download U-Boot from ftp://ftp.denx.de/pub/u-boot/ I use u-boot-2010.03 (Stick with this version)
Untar the tarball.

Check to see if the environment variables are ARCH=arm and CROSS_COMPILE=arm-none-eabi-

In the U-Boot directory, configure and compile the source
$ make CROSS_COMPILE=arm-none-eabi- versatilepb_config
$ make CROSS_COMPILE=arm-none-eabi- all
This creates the u-boot.bin binary image

To simulate,
$ arm-softmmu/qemu-system-arm -M versatilepb -m 128M -serial stdio -kernel u-boot.bin
This brings up the U-Boot prompt
(Note: the patched version of U-boot described in the next section does not work on bare metal arm)

The U-boot bootm command is used to boot a program that is loaded in memory as a special U-Boot image, that can be created with the tool mkimage. This program is usually an operating system kernel, but instead of running a full-blown Linux kernel, we can instead run the simple "Hello world" program from Lab #1. Since QEMU places the U-boot.bin binary (about 100 KB size) at 0x10000, we place our binary sufficiently far away at 1 MB (0x100000). Use the files HelloWorld.c, startup.s and HelloWorld.ld from Lab #1 with the address in the linker script changed to load HelloWorld.bin at 1MB (0x100000). Following the steps from Lab #1, create a Hello_World.bin binary.

Install uboot-mkimage package (sudo aptitude install uboot-mkimage)

Create the uboot Hello_World image
$ mkimage -A arm -C none -O linux -T kernel -d HelloWorld.bin -a 0x00100000 -e 0x00100000 HelloWorld.uimg

With these options we affirm that the image is for ARM architecture, is not compressed, is meant to be loaded at address 0x100000 and the entry point is at the same address. Using "linux" as operating system and "kernel" as image type causes U-Boot to clean the environment before passing the control to our image: this means disabling interrupts, caches and MMU. The -a and -e options sets the load address and entry points to 1MB.

Combine the Hello_World and U-Boot binaries into a single image
$ cat u-boot-2010.03/u-boot.bin HelloWorld.uimg > flash_hello.bin

To simulate,
$ ./arm-softmmu/qemu-system-arm -M versatilepb -m 128M -serial stdio -kernel flash_hello.bin

To calculate the address of HelloWorld.bin, take the size of u-boot.bin and sum the initial address where flash.bin is mapped. To do this, from the directory that has u-boot.bin execute the following script-

$ printf "bootm 0x%X\n" $(expr $(stat -c%s u-boot.bin) + 65536)
The Linux command 'stat' is used for displaying status information of Linux files and file systems. The Linux command 'expr' evaluates an expression.

On my system this prints 0x250e0

Now, at the U-Boot prompt type bootm 0x250e0 (substitute your memory address). This command copies Hello_World.bin into the address 0x100000 as specified in the U-Boot image, and then jumps to the entry point. This should print the “Hello World” message.

Booting Linux with U-Boot

On real physical boards the boot process usually involves a non-volatile memory (e.g. a Flash) containing a boot-loader and the operating system. On power on, the core loads and runs the boot-loader, that in turn loads and runs the operating system. QEMU can emulate Flash memory on many platforms, but not on the VersatilePB. QEMU can load a Linux kernel using the -kernel and -initrd options; at a low level, these options have the effect of loading two binary files into the emulated memory: the kernel binary at address 0x10000 (64KiB) and the ramdisk binary at address 0x800000 (8MiB). Then QEMU prepares the kernel arguments and jumps at 0x10000 (64KiB) to execute Linux.

We recreate this same situation using U-Boot, and to keep the situation similar to a real one create a single binary image containing the whole system, just like having a Flash on board. The -kernel option in QEMU will be used to load the Flash binary into the emulated memory, and this means the starting address of the binary image will be 0x10000 (64KiB).
One feature of U-Boot is self-relocation, which means that on execution the code copies itself into another address, which by default is 0x1000000 (16MiB). This feature comes handy in our scenario because it frees lower memory space in order to copy the Linux kernel. The compressed kernel image size is about 1.5MiB, so the first 1.5MiB from the start address must be free and usable when U-Boot copies the kernel.

At the beginning we have three binary images together: U-Boot (about 80KiB), Linux kernel (about 1.5MiB) and the root file system ramdisk (about 1.1MiB). The images are placed at a distance of 2MiB, starting from address 0x10000. At run-time U-boot relocates itself to address 0x1000000, thus freeing 2MiB of memory from the start address. The U-Boot command bootm then copies the kernel image into 0x10000 and the root filesystem into 0x800000; after that then jumps at the beginning of the kernel, thus creating the same situation as when QEMU starts with the -kernel and -initrd options.

U-Boot configured to be build for VersatilePB does not support ramdisk usage. From Moodle Lab #4 download the patch file u-boot-2010.03.patch

From the U-Boot directory apply the patch

$ patch -p1 < u-boot-2010.03.patch
Build U-Boot as described earlier

Now we create a flash image with 3 binary images (u-boot, kernel, and rootfs) placed 2 MB apart using the dd command.

$ mkimage -A arm -C none -O linux -T kernel -d zImage -a 0x00010000 -e 0x00010000 zImage.uimg
$ mkimage -A arm -C none -O linux -T ramdisk -d rootfs.img.gz -a 0x00800000 -e 0x00800000 rootfs.uimg
$ dd if=/dev/zero of=flash.bin bs=1 count=7M
$ dd if=u-boot.bin of=flash.bin conv=notrunc bs=1
$ dd if=zImage.uimg of=flash.bin conv=notrunc bs=1 seek=2M
$ dd if=rootfs.uimg of=flash.bin conv=notrunc bs=1 seek=5M

These commands do the following:
1. create the two U-Boot images, zImage.uimg and rootfs.uimg, that contain also information on where to relocate them
2. create a 7MB empty file called flash.bin
3. copy the content of u-boot.bin at the beginning of flash.bin
4. copy the content of zImage.uimg at 2MB from the beginning of flash.bin
5. copy the content of rootfs.uimg at 5MB from the beginning of flash.bin

See Lab #3 on generating zImage (Linux kernel) and rootfs.img.gz (Busybox-based file system)
Finally, to boot Linux,

From the qemu-0.15.0 directory
$ ./arm-softmmu/qemu-system-arm -M versatilepb -m 128M -kernel flash.bin -serial stdio

To do:

Examine the startup code for arm926ej-s cpu core in the file u-boot-2010.03/cpu/arm926ejs/start.S.

1. What does the U-boot startup do initially? (Hint: see the section _start)

2. U-boot then sets the processor to the supervisor mode. The following assembly code sets the processor to the supervisor mode, disables the interrupts, and sets the processor toARM state.

```
mrs     r0,cpsr
bic     r0,r0
orr     r0,r0,
msr     cpsr,r0
```

Explain what each instruction does.

3. The startup code then branches to cpu_init_crit. What is done in this section?

4. Next the U-boot startup and initialized data is relocated to RAM. Which instructions do the copying? What label is used to specify the RAM base address?

5. U-boot startup then sets up the stack.

sp = _TEXT_BASE - glb data - heap - irq_stack - fiq_stack – abrt_stack

Identify how this is done in the startup.

6. U-boot startup then clears the BSS and calls start_armboot() (lib_arm/board.c) Identify the different actions performed by start_armboot()

7. Finally the main_loop() is called (common/main.c). The main_loop sets up the U-boot shell to process user input U-boot commands. We used the U-boot bootm command. Lookup some more U-boot commands.