Howard Mao

Exploring the Arrow SoCKit Part I - Blinking LEDs

In September, I bought one of the new Arrow SoCKits. They are development boards for the Altera Cyclone V, a system-on-chip with an ARM processor and FPGA. Now that I'm out of school, I finally have some time to play around with it. I've decided to document the things I do with the SoCKit as a series of tutorials. Figuring out how to work with FPGA boards can be confusing and requires poring through a bunch of vendor datasheets and example code, so hopefully someone finds these tutorials helpful. For these tutorials, I will assume that you have programmed before and have some rudimentary understanding of digital electronics. If you have never programmed before, there are literally tons of free resources online. To be honest, though, FPGA programming and embedded systems might be a bit too challenging for first-time programmers, so I'd recommend getting more programming experience before coming back to these articles. If you have never studied digital electronics, ASIC World has a pretty good tutorial. I will be using the Verilog and SystemVerilog hardware description languages for these tutorials. I don't expect you to be familiar with either. I will explain the syntax as I go along. All of the hardware descriptions can be found on Github.

Getting Started

The first thing to do when trying out a new FPGA dev board is to get the LEDs to blink back-and-forth, Knight Rider style. This is a pretty simple circuit to create and gets you familiar with using the design tools and programming the chip. To make things more interesting, let's also design our circuit so that the speed at which the LEDs sweep back-and-forth can be controlled using the push-buttons on the board.

Installing the Software

For programming the FPGA, you will need Altera's Quartus II Web Edition design software. There are versions for Windows and Linux. Altera officially supports only Red Hat Enterprise Linux with their Linux version, but you can install it on other distributions. If you are using Arch Linux like I am, there is a pretty good article on how to install it on the Arch Wiki. Those instructions may also be applicable to other distros. When downloading the files for installation, do not download the "Combined Files" package. It is 4.5 GB and contains some device families that you will not need. Instead, go to the "Individual Files"

section and download "Quartus II Software", "ModelSim-Altera Edition", and "Cyclone V device support". This should give you files named like "QuartusSetupWeb-*w.x.y.z*.run", "ModelSimSetup-*w.x.y.z*.run", and "cyclonev-*w.x.y.z*.qdz". Change the .run files to be executable and then run the "QuartusSetupWeb-*w.x.y.z*.run" file. Follow the installation instructions given.

Creating the Project

Start Quartus and click on the big button labeled "New Project Wizard". On the first screen, enter the directory you'd like to put the project files in (you'll probably want to create a new directory for this). Give your project a name (I called mine "sockit_test"). Skip page 2, as you have no files to add. On page 3, set the device family to Cyclone V and choose 5CSXFC6D6F31C8 as the specific device. On page 4, pick ModelSim-Altera as the Simulation tools and choose "SystemVerilog HDL" as the format. This is not important for this part, since you won't be doing simulation just yet, but it will come in handy later. Once you get to page 5, you can press Finish.

Top-Level File and Pin Assignment

Now that you've created the project, you can set up the top-level file and assign the pins you need to it. In Quartus, create a Verilog file by clicking "File" -> "New" -> "Verilog HDL file". Save the file with the same name as your project (so if you called your project "sockit_test", save it as "sockit_test.v". I recommend putting your HDL files in a separate subdirectory in your project folder called "rtl".

Put the following Verilog code in "sockit_test.v".

```
module sockit_test (
    input CLOCK_50,
    input [3:0] KEY,
    output [3:0] LED
);

endmodule
```

If you're not familiar with Verilog, a "module" is a hardware block. In this code, we are simply specifying what the inputs and outputs of the block are. For the top-level module, the inputs and outputs are the pins of the FPGA. For our example, we only need the 4 push button keys, 4 LEDs, and the 50 MHz clock. You can assign the pins on the FPGA to your inputs and outputs by going to "Assignments" -> "Pin Planner" and entering in the

following assignments.

Node Name	Location
CLOCK_50	PIN_K14
KEY[3]	PIN_AD11
KEY[2]	PIN_AD9
KEY[1]	PIN_AE12
KEY[0]	PIN_AE9
LED[3]	PIN_AD7
LED[2]	PIN_AE11
LED[1]	PIN_AD10
LED[0]	PIN_AF10

Controlling the LEDs

Now that the pins have been assigned, you can start putting together the different modules to make the circuit work. The first module we will make is the one driving the LEDs. We will call it “blinker.v”.

```
module blinker (
    input clk,
    input [3:0] delay,
    output reg [3:0] led,
    input reset,
    input pause
);

reg [23:0] count = 24'b0;
reg [2:0] pos = 3'b000;
reg running = 1'b1;

always @(pos) begin
    case (pos)
        3'b000: led <= 4'b0001;
        3'b001: led <= 4'b0010;
        3'b010: led <= 4'b0100;
        3'b011: led <= 4'b1000;
        3'b100: led <= 4'b0100;
        3'b101: led <= 4'b0010;
```

```
            default: led <= 4'b0000;
        endcase
    end

    always @(posedge clk) begin
        if (reset) begin
            count <= 24'b0;
            pos <= 3'b000;
            running <= 1'b1;
        end else if (pause) begin
            running <= !running;
        end else if (running) begin
            if (count == 24'b0) begin
                count <= {delay, 20'b0};
                if (pos == 3'b101)
                    pos <= 3'b000;
                else
                    pos <= pos + 1'b1;
            end else begin
                count <= count - 1'b1;
            end
        end
    end

    endmodule
```

Warning! To everyone reading this who is primarily a "software person" and does not have much experience with digital logic, take heed that though Verilog looks superficially like software code with its "case" and "if" statements, it is actually describing hardware blocks. One particular difference is that there is no concept of order in Verilog. Statements on subsequent lines are all "running" at the same time. To give an "ordering" to computation, you must explicitly design state machines as the previous code does. If you don't keep these things in mind, you might end up writing completely valid Verilog that is impossible to synthesize.

And now back to our regularly scheduled blog post ...

The first part of this module should look familiar to you. I am stating that this module takes as input a clock `clk`, a four-bit `delay` signal (we treat it as a 4-bit unsigned integer), a `reset` signal, and a `pause` signal. The `reset` and `pause` signals correspond to two of the push buttons on the board. The output is the 4-bit led output. I have

declared this as `reg`, which stands for register. Verilog requires you to declare as `reg` anything that could hold state. It turns out that the output won't actually hold any state, but the Verilog compiler is not clever enough to figure this out. If this confuses you, don't worry. It will make more sense once I explain the `always` blocks.

The second part of the module are some internal registers `count`, `pos`, and `running`, which are initialized to 0, 0, and 1, respectively. These registers actually will hold state, unlike the output `led` register.

The third part of the module are the `always` blocks. These constructs tell our hardware to perform some operation whenever the signals in the sensitivity list (the stuff inside the parenthesis after the @ sign) change. In the first `always` block, the sensitivity list is the signal `pos`, so the operations inside the `always` block will occur whenever `pos` changes. Inside this `always` block is a `case` statement that maps certain values of pos to certain values of `led`. You will notice that in everything except the default case, exactly one bit in `led` is high, corresponding to a lit led. With increasing `pos`, the lit led goes to the left and then back to the right. Since every possible case of pos has been covered, this `always` block functions as a combinational circuit. If we had left out a case (say, by getting rid of the default case), the compiler would warn us about inferring a latch. That is, if `pos` happened to be in a case where the behavior was unspecified, the value of `led` would keeps its previous value. You can see now why `led` had to be declared a register even though it isn't one.

The second `always` block contains in its sensitivity list, `posedge clk`. This means that the `always` block is triggered by the rising edge of `clk`. Inside the `always` block is a large nested if statement. Here, we state what values each of the internal registers will take at each cycle. If `reset` is triggered, we change `count`, `pos`, and `running` back to their original values. If `pause` is triggered, we toggle the value of `running`. If we are running, then we are under normal operation, during which we want to regularly increment `pos` until it reaches 5, at which point we wrap around back to 0. However, we want the incrementing of `pos` to happen slowly at a controlled speed. We accomplish by initially setting the `count` variable to the value of `delay` multiplied by 2^{20}, decrementing until it reaches 0, and then resetting it. The value of `pos` is then only updated when `delay` is reset. There is no logic specified for when `running` is false, so in that case all

Setting the delay

So how do we set the `delay` variable that the `blinker` module needs? We'll have to create a different module. Call it "delay_ctrl.v".

```
module delay_ctrl (
    input clk,
    input faster,
    input slower,
    output [3:0] delay,
    input reset
);

reg [3:0] delay_intern = 4'b1000;

assign delay = delay_intern;

always @(posedge clk) begin
    if (reset)
        delay_intern <= 4'b1000;
    else if (faster && delay_intern != 4'b0001)
        delay_intern <= delay_intern - 1'b1;
    else if (slower && delay_intern != 4'b1111)
        delay_intern <= delay_intern + 1'b1;
end

endmodule
```

This module takes as input the clock, two control signals `faster` and `slower`, as well as a `reset` signal. The control signals correspond to push buttons. The output is the 4-bit `delay` which will feed into the blinker. We declare an internal register `delay_intern` and initialize it to 8, which is the halfway point. This internal register is then assigned to the `delay` output. In our positive-edge triggered `always` block, we first check to see if a reset is triggered, in which case we set `delay_intern` back to 8. If `faster` is pressed, we reduce the delay. If `slower` is pressed we increase it. If none of the control signals are high, we maintain state.

Handling the Buttons

In our previous two modules, we assumed that our control signals would be high for exactly one cycle after the keys are pressed. Given the speed of the human finger, this would obviously be impossible if the control signals were tied directly to the keys.

Therefore, we need a unit to detect when each key is pressed and set the corresponding control signal high for one cycle. We will call it "oneshot.v".

```
module oneshot (
    input clk,
    input [3:0] edge_sig,
    output [3:0] level_sig
);

reg [3:0] cur_value;
reg [3:0] last_value;

assign level_sig = ~cur_value & last_value;

always @(posedge clk) begin
    cur_value <= edge_sig;
    last_value <= cur_value;
end

endmodule
```

Here, `edge_sig` is the input from our keys and `level_sig` is the output for our control signals. The trick here is that we keep two 4-bit registers `cur_value` and `last_value`. On each cycle, we read the values of the keys into `cur_value` and the previous value of `cur_value` into `last_value`. The signals from the keys are 0 when pressed and 1 when unpressed, so we want each bit of our output to be high when the current value is 0 and the last value was 1, which is what the `assign` statement is doing. You may think that I have mixed up the order of `cur_value` and `last_value` in the `always` block. But actually, order does not matter when using the non-blocking <= assignment operator. When using <=, the values being read will always be the values from the previous clock cycle.

Tying it All Together

Finally, we must tie our three components together in our top-level module.

```
module sockit_test (
    input CLOCK_50,
    input [3:0] KEY,
```

```
    output [3:0] LED
);

wire [3:0] key_os;
wire [3:0] delay;
wire main_clk = CLOCK_50;

oneshot os (
    .clk (main_clk),
    .edge_sig (KEY),
    .level_sig (key_os)
);

delay_ctrl dc (
    .clk (main_clk),
    .faster (key_os[1]),
    .slower (key_os[0]),
    .delay (delay),
    .reset (key_os[3])
);

blinker b (
    .clk (main_clk),
    .delay (delay),
    .led (LED),
    .reset (key_os[3]),
    .pause (key_os[2])
);

endmodule
```

We've now expanded our original "sockit_test.v" to connect everything together. We have three internal signals, `key_os`, `delay`, and `main_clk`. These signals are marked `wire`, which is the opposite of `reg`. Our modules are tied to these signals using "port mappings", which are statements of the following form.

```
module_name instance_name (
    .internal_signal (external_signal),
    .internal_signal2 (external_signal2)
);
```

The `module_name` is the name we gave to the module, and `instance_name` is the name we give to this instance of the module. The instance name doesn't really matter as long as they are unique within a module. The `internal_signal` name is just the input/output name given inside the port-mapped module. The `external_signal` name of the wire in the outer module.

Compiling and Programming

Now that we've finished our circuit, we can compile our hardware description and program it onto the FPGA. First, though, let's do a quick static check to make sure we didn't screw up somewhere. Run "Analysis and Synthesis" by clicking on the icon with a purple triangle and blue check mark in the tool bar (third from left in below image). Wait for the action to complete. You can watch its progress in the "Tasks" window at the middle left. After it's finished, inspect the log output in the "Messages" window at the bottom. Make sure it doesn't have any errors or warnings beyond "Parallel compilation is not licensed and has been disabled". If you do see warnings or errors, look back at your hardware descriptions and make sure there isn't a typo. "Analysis and Synthesis" will be useful later on for catching syntax mistakes.

Now that you've checked the descriptions, you can run a full compilation by clicking the icon with the purple triangle (second from left in the image). Be prepared to wait a little while. Once the compilation is complete, a file called "sockit_test.sof" should be generated in the "output_files" subdirectory of your project folder. This file contains the configuration you will program into the FPGA.

Before you try to program the FPGA, make sure that the USB Blaster drivers are installed correctly (check the Arch Wiki article at the top for Linux, or this article for Windows). Make sure you have the SoCKit connected to your computer correctly. The USB Blaster port is the microUSB port farthest to the right if the ports are facing toward you.

Careful! Early versions of the SoCKit board had surface-mounted microUSB ports with no reinforcement. The microUSB ports on these boards are likely to break off if you push too hard when inserting the USB cable. Later versions of the board came with a reinforcing metal plate to fix this problem. If you have one of the earlier boards, be very careful when plugging in the microUSB cable.

Once you are sure the drivers are working, open up the programmer by double-clicking on "Program Device" in the "Tasks" window, by clicking on "Tools" -> "Programmer" in the

menu, or the Programmer button in the toolbar (second from right in the above image). In the new window, go to "Hardware Setup" and make sure "Currently selected hardware" is set to something like "CV SoCKit". If you cannot find this selection in the dropdown menu, you may want to check that the board is on, the USB blaster is connected, and the drivers are installed properly. Once you've selected the correct hardware, press the "Auto Detect" button in the programmer window. It may ask you to choose your device. Choose "5CSXFC6D6".

You should now see two devices, 5CSXFC6D6F31 and SOCVHPS. The former is the FPGA, the latter is the ARM processor. Right click on the entry for the FPGA and select "Change File". Pick the "sockit_test.sof" file that was generated during compilation. Now press start, and the .sof file will be programmed onto the FPGA. If you are successful, the SoCKit will look like the following.

Conclusion

Congratulations, you just programmed an FPGA! In my next post, I will take a look at the ARM processor on the Cyclone V and how to install an operating system on it.

Part 2 ->

Howard Mao

Exploring the Arrow SoCKit Part II - Installing Linux

In my last post, I showed you how to get a simple FPGA example working on the Arrow SoCKit dev board. In this post, we will explore the CPU side of the Cyclone V SoC and install a custom Linux kernel and root filesystem using Buildroot. This tutorial will focus on booting via SD card, so make sure you have a microSDHC card handy. If your computer doesn't have a microSD port, you will also need a microSD - SD and possibly an SD - USB adapter. Also, these instructions assume you are running some form of Linux. It should be possible to do all these things on Windows or OSX as well, but you'll have to figure that out yourself (or switch to a *real* operating system).

RocketBoards.org has some articles on booting Linux on the SoCKit, but they are focused on booting Altera's Yocto Linux distribution. If you don't feel like going through compiling everything yourself, you can always just follow their instructions for booting from a pre-built image. However, if you're a Linux nerd like me and want to get the educational benefits of doing things the hard way, read on!

Warning! Here there be dragons! Getting Linux running on an embedded system can be something of a dark art. One semester in my days as an undergraduate, I took on a research project that involved booting Android on an Arndale development board. It took two months of banging my head against the problem before I finally gave up and asked for help on a kernel dev mailing list. Turns out I was just using the wrong versions of the kernel and bootloader. So what am I trying to say here? I'm saying that you shouldn't get discouraged if things don't work and **for the love of all that is holy, ask for help if you get stuck!!!!** Don't do the same thing I did and try the same things over and over again. If it's not working, there's probably some information you're missing. Go open an issue on the Github repo for this site with your question and I'll try to answer it. If that doesn't work (and it probably won't, since I'm just figuring all of this out as I go along), you can try asking your question on the RocketBoards.org mailing lists.

Step 1 - Configuring the Board

To boot Linux from the SD Card, you will first need to set the jumpers and switches on the

board correctly. Follow the instructions on the RocketBoards.org article to do this.

Step 2 - Partitioning the SD Card

To partition the SD card, use the fdisk program. Plug the SD card into your computer and delete any existing partitions if there are any. Then, run the command `fdisk /dev/sdX`. Replace `/dev/sdX` with whatever device file your OS recognizes the SD card as (it'll probably be `/dev/sdb`). This will bring you into the fdisk command prompt. Enter the following commands (note: when I type `<enter>` it means just hit enter without entering a command).

```
n
p
3
2048
+2048
t
a2
n
p
<enter>
<enter>
+256M
t
1
0b
n
p
<enter>
<enter>
<enter>
w
```

Once you enter "w", the partition table will be written to the SD card.

Step 3 - Installing the bootloader

The bootloader used for embedded ARM systems is called u-boot. Unfortunately, this is the one part that I couldn't figure out how to install from source. There seems to be a problem with the stage 1 bootloader in the rocketboards.org git repo. Fortunately there is a (rather hacky) way around this. Simply extract the bootloader from the pre-built image.

I've uploaded the bootloader image for you so that you don't have to go and download a 1.9 GB image in order to extract a single megabyte from it. Once you've downloaded it, you can write it to the SD card like so.

```
sudo dd if=bootloader.img of=/dev/sdX3 bs=512
sudo sync
```

Don't forget to replace the "X".

Step 4 - Installing the kernel

Clone the sources of the kernel for the SoCKit from RocketBoards.org.

```
git clone git://git.rocketboards.org/linux-socfpga.git
cd linux-socfpga
git checkout -b sockit ACDS13.1_REL_GSRD_PR
```

You will also need to download the Linaro ARM toolchain to get the C cross-compiler. Find the latest release tarball for your system here. If you are on Linux, you are looking for the tarball named something like "gcc-linaro-arm-linux-gnueabihf-*some-version-number*_linux.tar.xz"

Extract the tarball anywhere on your system and then add the "bin" subdirectory of the extracted folder to your path. If you've done this properly, you should be able to run the following command.

```
arm-linux-gnueabihf-gcc --version
```

Now that you have the cross-compiler, you can start building the kernel. First, make sure to choose the correct configuration.

```
make CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm socfpga_defconfig
```

You'll probably want to add an alias for the first part of the command to your .bashrc or .zshrc. For instance, I have

```
alias armmake='make CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm'
```

Now we can build the kernel image itself. To do this, you will first need to install the U-boot "mkimage" tool. On Ubuntu and Debian, just install the "uboot-mkimage" package. On Arch, install the "uboot-mkimage" package from the AUR. The Linux kernel takes a

long time to build. You can speed things up a bit by using multiple threads. The command I use is...

```
armmake -j4 uImage LOADADDR=0x8000
```

You should replace 4 with the number of cores you have on your computer.

You also need to build the device tree for the SoCKit board.

```
armmake dtbs
```

Now that the kernel and device tree are built, you need to copy them onto the boot partition. First, format the partition as FAT32 and mount it.

```
sudo mkfs.vfat /dev/sdX1
sudo mount /dev/sdX1 /mnt
```

If you don't have the `mkfs.vfat` program on your computer, you can probably get it by installing the "dosfstools" package from your distribution's package repository.

Then copy kernel and device tree onto the filesystem and unmount it.

```
sudo cp arch/arm/boot/uImage /mnt
sudo cp arch/arm/boot/dts/socfpga_cyclone5.dtb /mnt/socfpga.dtb
sudo umount /mnt
```

Step 5 - Installing the Root Filesystem

For our root filesystem, we will use Buildroot. Buildroot is a set of buildscripts that can produce a Linux filesystem for an embedded platform. First, download and extract the tarball. Go into the extracted folder and run `make menuconfig`. Here, you can configure the filesystem image that Buildroot will produce. I suggest you choose the following options.

First, go to "Target options" and set the following options

- Target Architecture: ARM (little endian)
- Target Architecture Variant: cortex-A9
- Target ABI: EABIhf
- Enable NEON SIMD extension support

- Floating point strategy: VFPv2
- ARM instruction set: ARM

Then, go to "Build Options" and change "Number of jobs to run simultaneously" to the number of cores you have.

Go to "Toolchain" and do the following

- Set "Toolchain Type" to "External toolchain"
- Set "Toolchain" to "Custom toolchain"
- Set "Toolchain origin" to "Pre-installed toolchain"
- Set "Toolchain path" to the folder you extracted the Linaro toolchain to
- Set "Toolchain prefix" to "arm-linux-gnueabihf"
- Set "External toolchain header series" to "3.1.x"
- Change "External toolchain C library" to "glibc/eglibc"
- Select "Toolchain has RPC support"
- Select "Toolchain has C++ support"
- Make sure "Enable MMU support" is selected

Go to "System configuration" and change the following.

- Change the hostname, if you wish
- Set a root password, if you prefer (if you do not, there will be no root password)
- Select "Run a getty after boot"
- Go to "getty options"
  - Change the baudrate to 57600
- Make sure "remount root filesystem during boot" is selected

Go to "Target packages" and select any extra packages you want. I suggest you select "Package managers" -> "opkg". This will allow you to install packages later if you want.

Go to "Filesystem images" and make sure "tar the root filesystem" is selected.

Save and exit menuconfig, then run `make` to build everything. If everything works out, you will see the root filesystem image generated under "output/images/rootfs.tar". You will need to untar this onto the root partition.

```
sudo mkfs.ext2 /dev/sdX2
sudo mount /dev/sdX2 /mnt
sudo tar xf output/images/rootfs.tar -C /mnt
```

This will format the root partition and extract the files onto it. Don't unmount the partition just yet, we'll need to add some more files.

Note - Recent releases of the linaro toolchain can cause an issue in which `init` cannot find libc.so.6. If you get a problem like this, follow the workarounds described in this Github issue. To summarize, create symlinks on the partition from /lib to /lib/arm-linux-gnueabihf and /usr/lib to /usr/lib/arm-linux-gnueabihf. Running the following commands should do the trick.

```
sudo ln -s /lib /mnt/lib/arm-linux-gnueabihf
sudo ln -s /usr/lib /mnt/usr/lib/arm-linux-gnueabihf
```

Step 6 - Install the kernel modules

The kernel has been installed on the boot partition, but you will also need to build the kernel modules and install them on the root partition. Go back to your kernel folder and run the following.

```
armmake -j4 modules
sudo make ARCH=arm INSTALL_MOD_PATH=/mnt modules_install
sudo umount /mnt
```

Now, finally, your SD card is ready. You can now try booting it. But to see what the OS is doing, you will need to use the serial port.

Step 7 - Setup the serial port

You will use a USB serial connection to communicate with the board. First, you should install "minicom", a serial terminal program for Linux. Then, you'll have to configure minicom with the correct settings for the SoCKit's serial line. To enter the configuration menu, run `sudo minicom -s`. Go to "Serial port setup". Change the "Serial Device" to "/dev/ttyUSB0". In the same submenu, open "Bps/Par/Bits". Set the speed to 57600. You will probably have to hit "<next>" or "<prev>" a bunch of times. Set parity to "None", data to 8, and stop bits to 1. In the end, "Current" should be "57600 8N1". Go back to the "Serial port setup" menu and make sure "Hardware Flow Control" and "Software Flow Control" are both off. Then exit this menu by hitting enter and hit "Save setup as dfl", followed by "Exit from minicom".

Now, plug your microSD card into the microSD slot on the board (it's on the bottom underneath the push buttons). Plug the microUSB cable into the USB UART port (it's the microUSB port farthest to the left), and connect it to your computer. Press the red button to switch on the board and then run "minicom" without arguments in your terminal. If you leave it for a while, the kernel will start booting. If everything is successful, you will eventually be presented with a login prompt.

If you hit some buttons when the board was first booting up, you may see a prompt reading "SOCFPGA_CYCLONE5". This is the U-boot console. To boot the kernel, just type in "boot" and hit enter.

Conclusion

Congrats, you just booted Linux on the SoCKit! If this is your first time booting Linux on an ARM development board, this is actually a pretty major accomplishment. You've now learned the general flow of how to get Linux working on an embedded platform (excluding building and installing U-boot). Next time, we'll look at how the CPU and FPGA can communicate.

<- Part 1 Part 3 ->

Howard Mao

Exploring the Arrow SoCKit Part III - Controlling FPGA from Software

In part I, I showed you how to load a simple LED example onto the FPGA. In part II, I showed you how to install Linux onto the ARM processor. Now, in part III, I will show you how to connect the two together so that you can control the speed of the blinking LEDs from software. To do this, we will use Qsys, a system integration tool from Altera that can automatically generate interconnect logic to hook up different hardware modules.

Hardware descriptions and C code can be found on Github.

A Brief Aside on Memory-Mapped IO

Before we begin, it's useful to go over exactly how software running on the CPU interacts with hardware peripherals. If you are already familiar with the concept of memory-mapped IO, feel free to skip this section. Otherwise, read on.

In order for software to control hardware peripherals, the processor must have a way to communicate with the peripherals. This communication method must also be extensible without changing the CPU hardware, since one CPU model could be used in many types of systems with different sets of peripherals. The method used by most modern processors is memory-mapped IO, in which the "memory" that a processor sees is actually a bus or some other kind of interconnect, and different parts of the address space are mapped either to actual RAM or to hardware peripherals.

MMIO

The processor can then send commands to a peripheral by writing to the peripheral's address space and get information back by reading from the peripheral's address space.

The Avalon MM Interface

So now you know that we'll need to hook up our peripherals to a memory bus in order for the CPU to communicate with them. But what exactly is the interface for connecting a hardware unit to the bus?

On Altera's FPGAs, the easiest bus interface to use is the Avalon MM interface. Avalon MM is a master-slave protocol, with a CPU being the master and the peripherals being the slaves.

Avalon memory-mapped slaves can have the following signals

Name	Direction	Width	Description
address	input	up to 64	the address on the slave being accessed
read	input	1	indicates whether a read operation is requested
readdata	output	8, 16, 32, or 64	the data that will be read
write	input	1	indicates whether a write operation is requested
writedata	input	8, 16, 32, or 64	the data to be written

byteenable	input	2, 4, or 8	for multi-byte writedata, indicates which bytes are valid

This is not an exhaustive list of course, but these are the ones that are likely to be of concern unless you are doing something fancy.

These signals are also optional, so you can, say, leave out "read" and "readdata" if you don't care about reading from the peripheral. In fact, that's exactly what we're going to do for our delay controller.

Delay Control as Avalon Slave

We will modify our `delay_ctrl` module from part 1 so that it is an Avalon slave. This will allow us to set the delay from the CPU.

```
module delay_ctrl (
    input clk,
    input reset,

    input faster,
    input slower,
    output [3:0] delay,

    input write,
    input [7:0] writedata
);

reg [3:0] delay_intern = 4'b1000;

assign delay = delay_intern;

always @(posedge clk) begin
    if (reset)
        delay_intern <= 4'b1000;
    else if (write)
        delay_intern <= writedata[3:0];
    else if (faster && delay_intern != 4'b0001)
        delay_intern <= delay_intern - 1'b1;
    else if (slower && delay_intern != 4'b1111)
        delay_intern <= delay_intern + 1'b1;
end

endmodule
```

You'll see that I have added a "write" and 8-bit "writedata" input. When "write" is asserted high, the stored delay will take the value of the lower 4 bits of "writedata". This will allow us to set the delay by writing it to this peripheral's memory. Notice that, since this peripheral only has one thing that can be written, it does not need an address input.

Building a System in Qsys

Now that you have an Avalon peripheral, we can hook it up to the processor. For this, we will need to use Altera's Qsys tool. You can open Qsys from Quartus by going to "Tools" -> "Qsys". You can also click the Qsys icon, which is the farthest on the right in our trusty Quartus toolbar screenshot.

When you first start Qsys, the only component in place will be the clock and reset controller. We will need to add our processor to this system.

Adding the HPS

Open the "Embedded Processors" section in the "Library" window at the top left. Then, select "Hard Processor System" and click the "Add" button. This will open up a menu where you can select the options for the hard processor. You will need to make the following changes.

1. Under the "General" section of the "FPGA Interfaces" tab, deselect "Enable MPU standby and event signals".
2. In the "AXI Bridges" section, change "FPGA-to-HPS interface width" and "HPS-to-FPGA interface width" to "Unused". We will only need the lightweight HPS-to-FPGA interface for this project.
3. Delete the entry in the "FPGA-to-HPS SDRAM interface" section.
4. Go to the "SDRAM" tab and click on the "Memory Parameters" subtab.
5. In the "Memory Initialization Options" section, change "ODT Rtt nominal value" under "Mode Register 1" to "RZQ/6".

Once you've made all these changes, you can click "Finish" to add the HPS to the system.

Creating and Adding the Delay Controller

Now you will need to add the delay controller to the system. Since this is a custom module, you will first need to create a new qsys component for it. Go to the "Library" window and double-click on "New Component". In the newly opened window, select the following options.

1. Under the "Component Type" tab, change "Name" and "Display name" to "delay_ctrl".
2. Go to the "Files" tab and click the "+" button under "Synthesis Files" to add a new file to this component. Choose the "delay_ctrl.v" file.
3. Click "Analyze Synthesis Files" to check the file for syntax errors and pull out the signals.
4. Go to the "Signals" tab, where you will indicate the purpose of the signals in the module.
5. Make sure that the "write" and "writedata" signals are on an avalon slave interface called "avalon_slave_0" and that the signal types are "write" and "writedata", respectively.
6. Make sure "clk" and "reset" are on "clock" and "reset" interfaces with signal types "clock" and "reset" respectively.
7. Change the interface for "faster" to "new Conduit". This will create an interface called "conduit_end".
8. Assign "slower" and "delay" to also be on the "conduit_end" interface. The conduit interface type means that the signals will not be used internally by the Qsys interconnect and will instead be exported out to the top-level.
9. Change the signal type for all of the conduit signals to "export".
10. Go to the "Interfaces" tab. Make sure there are four interfaces: "clock", "reset", "conduit_end", and "avalon_slave_0". If there are others, you can remove then using "Remove Interfaces with no Signals".
11. Make sure "reset" has "clock" as its associated clock.
12. Make sure that "avalon_slave_0" has "clock" as its associated clock and "reset" as its associated reset.

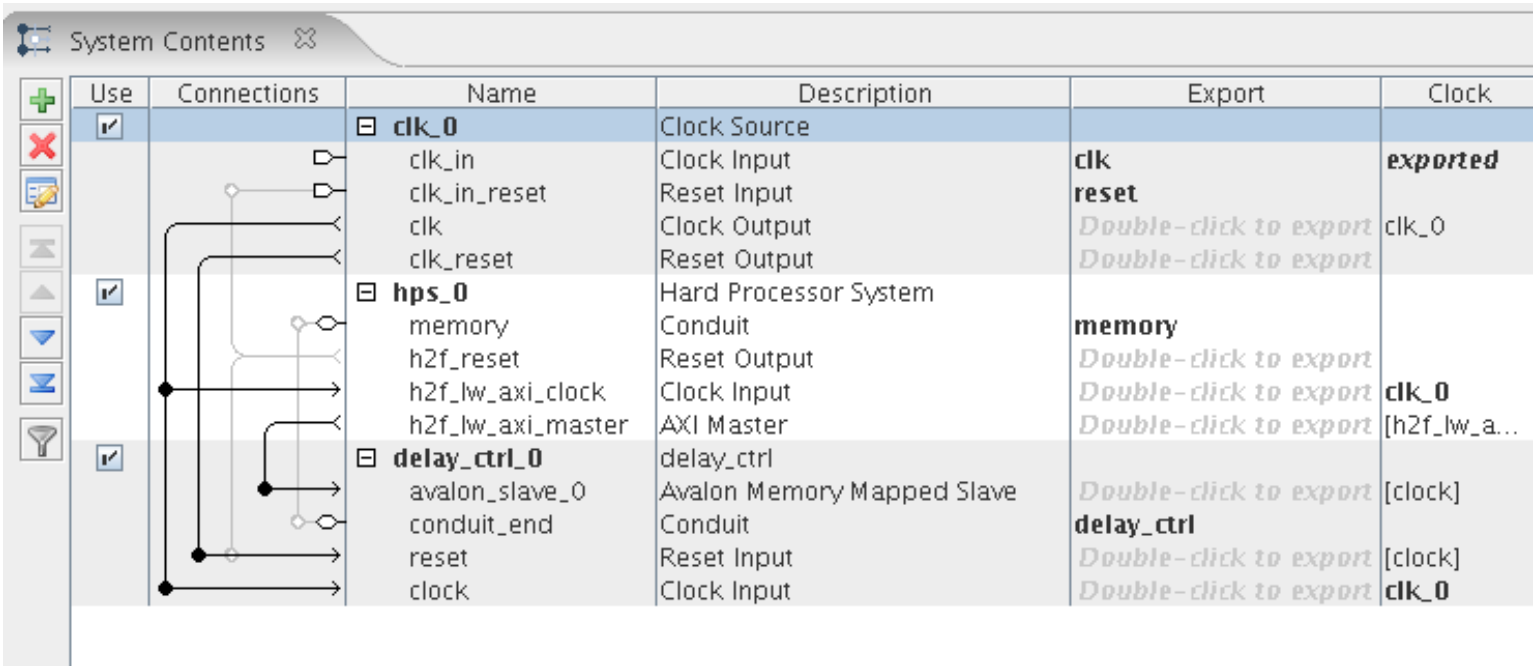
Press "Finish" and save this component. You should see a new file called "delay_ctrl_hw.tcl" in your project directory and a component named "delay_ctrl" under "Project" in the library window. Add this component to your system. You can just press "Finish" in the add dialog as there are no options.

Connecting the Components

Now that you've placed all of the components, you must connect all the interfaces together. All of the possible connections are indicated by light grey lines. To make an actual connection, simply click on the empty bubbles at the intersections of lines. A connection which is actually made will turn black and the bubble will be filled in.

1. Double-click in the "Export" column for the "clk_in" signal under the "clk_0" component and export it as "clk".
2. Double-click to export the "clk_in_reset" signal as "reset".
3. Connect the "clk" output of the "clk_0" component to the "h2f_lw_axi_clock" input in "hps_0" and to the "clock" input of "delay_ctrl_0".
4. Connect the "clk_reset" output of "clk_0" to "h2f_reset" of "hps_0" and to "reset" of "delay_ctrl_0".
5. Connect "h2f_lw_axi_master" of "hps_0" to "avalon_slave_0" of "delay_ctrl_0".
6. Export "memory" of "hps_0" as "memory"
7. Export "conduit_end" of "delay_ctrl_0" as "delay_ctrl".

In the end, your "System contents" window should look something like this.



You have now finished the system, so save it as "soc_system.qsys". You can now generate the system by clicking "Generate" -> "Generate" from the menu. In the "Generation" dialog, make sure "Create HDL design files for synthesis" is set to Verilog. You can also change the "Output Directory" to a directory of your choosing. By default, it will be a subdirectory of your project directory called "soc_system". Press the "Generate" button, and Qsys will begin producing Verilog files for this system. Once the system finishes generation successfully, you can close Qsys.

Adding Qsys System to Quartus Project

Now that we have a generated Qsys system, we will need to add it to our Quartus project so that it can be compiled into the .sof. Since we have included "delay_ctrl.v" in the system, we can remove it from the project. In its place, we will add the system, which has been generated at "soc_system/synthesis/soc_system.qip". You can add this file to your project by going to the "Files" tab of the "Project Navigator" window on the left, right-clicking on the "Files" folder icon, and choosing "Add/Remove Files in Project".

Once you have added "soc_system" to the project, you must add it to the top-level file, "sockit_test". First, the top-level inputs will have to change in order to accomodate the exported "memory" interface of the system. Change the `module` declaration of sockit_test to the following.

```
module sockit_test (
    input         CLOCK_50,
    input  [3:0]  KEY,
    output [3:0]  LED,

    output [14:0] hps_memory_mem_a,
    output [2:0]  hps_memory_mem_ba,
    output        hps_memory_mem_ck,
    output        hps_memory_mem_ck_n,
    output        hps_memory_mem_cke,
    output        hps_memory_mem_cs_n,
    output        hps_memory_mem_ras_n,
    output        hps_memory_mem_cas_n,
    output        hps_memory_mem_we_n,
    output        hps_memory_mem_reset_n,
    inout  [39:0] hps_memory_mem_dq,
    inout  [4:0]  hps_memory_mem_dqs,
    inout  [4:0]  hps_memory_mem_dqs_n,
    output        hps_memory_mem_odt,
    output [4:0]  hps_memory_mem_dm,
    input         hps_memory_oct_rzqin
);
```

Then, delete the `delay_ctrl` port mapping from the body of `sockit_test` and replace it with a port mapping for `soc_system`.

```
soc_system soc (
    .delay_ctrl_delay (delay),
    .delay_ctrl_slower (key_os[0]),
    .delay_ctrl_faster (key_os[1]),
    .memory_mem_a        (hps_memory_mem_a),
    .memory_mem_ba       (hps_memory_mem_ba),
    .memory_mem_ck       (hps_memory_mem_ck),
    .memory_mem_ck_n     (hps_memory_mem_ck_n),
    .memory_mem_cke      (hps_memory_mem_cke),
    .memory_mem_cs_n     (hps_memory_mem_cs_n),
    .memory_mem_ras_n    (hps_memory_mem_ras_n),
    .memory_mem_cas_n    (hps_memory_mem_cas_n),
    .memory_mem_we_n     (hps_memory_mem_we_n),
    .memory_mem_reset_n  (hps_memory_mem_reset_n),
    .memory_mem_dq       (hps_memory_mem_dq),
    .memory_mem_dqs      (hps_memory_mem_dqs),
    .memory_mem_dqs_n    (hps_memory_mem_dqs_n),
    .memory_mem_odt      (hps_memory_mem_odt),
    .memory_mem_dm       (hps_memory_mem_dm),
    .memory_oct_rzqin    (hps_memory_oct_rzqin),

    .clk_clk (main_clk),
    .reset_reset_n (!key_os[3])
);
```

Your final `sockit_test.v` file should look like this

Adding Pin Assignments and Compiling Project

Now that you've added the system, you have to make the pin assignments for the new inputs. Fortunately, Qsys generates a Tcl script which can add these assignments automatically. First, run Analysis and Synthesis so that Quartus can determine what the new pins are. Once this is done, open "Tools" -> "Tcl Scripts" in the Quartus menu. The script you need is at "soc_system/synthesis/submodules/hps_sdram_p0_pin_assignments.tcl". Once the script has run, you can run the full compilation.

Programming the FPGA from HPS

Since we will be using the HPS a lot, it's useful to know how to program the FPGA from the HPS. This way, you won't have to keep switching the USB cable between the UART to the USB Blaster.

Programming from the HPS requires a slightly different board configuration. The MSEL switches should be set to 00000, so make sure all the switches are in the '0' position. This configuration should still allow you to program from the USB Blaster.

Programming the FPGA from the HPS requires a raw binary file (.rbf) instead of a .sof file. You can convert the .sof file to a .rbf file using the `quartus_cpf` tool. Run the following command from your project directory.

```
quartus_cpf -c output_files/sockit_test.sof output_files/sockit_test.rbf
```

Copy the "sockit_test.rbf" file to the "/root" folder of the Linux partition on your SD card. You can then run the following command from the HPS to program the FPGA.

```
dd if=sockit_test.rbf of=/dev/fpga0
```

You should see the FPGA LEDs begin to blink at this point. Be careful that you do not program the FPGA in this way when the FPGA-to-HPS or HPS-to-FPGA bridges are enabled. When you first boot up the board, the bridges are disabled by default, but later we will switch one of the bridges on. You should always make sure to disable any bridges you've enabled before you program the FPGA again. The following sequence of commands will disable all the bridges on the Cyclone V.

```
echo 0 > /sys/class/fpga-bridge/fpga2hps/enable
echo 0 > /sys/class/fpga-bridge/hps2fpga/enable
echo 0 > /sys/class/fpga-bridge/lwhps2fpga/enable
```

Echoing 1 to the sysfs files will re-enable the bridges. You can also run this script to disable the bridges, program the fpga, and re-enable the bridges.

Setting the Delay from the HPS

And now, the final step: controlling the delay from software. To do this, you will have to write to the base address of the "delay_ctrl" peripheral. This peripheral is connected to the lightweight HPS-to-FPGA bridge. The lightweight bridge's region of memory begins at address 0xff200000, so to find the address of an FPGA peripheral, simply add the peripheral's offset as shown by Qsys to that address. In our case, the "delay_ctrl" peripheral was assigned the offset 0x00000000, so the full address is simply 0xff200000.

The Linux kernel we are running uses virtual memory, so we cannot directly write to address 0xff200000 from

a userspace process, since that physical address is not mapped into the process's address space. The proper way to expose the "delay_ctrl" peripheral is to write a kernel module, which I will discuss in my next post. For now, we will use a simpler method, which is to use the `mmap` system call on the "/dev/mem" device file, which represents physical memory, to map the HPS-to-FPGA bridge's memory into the process memory.

```
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>

#define PAGE_SIZE 4096
#define LWHPS2FPGA_BRIDGE_BASE 0xff200000
#define BLINK_OFFSET 0x0

volatile unsigned char *blink_mem;
void *bridge_map;

int main(int argc, char *argv[])
{
        int fd, ret = EXIT_FAILURE;
        unsigned char value;
        off_t blink_base = LWHPS2FPGA_BRIDGE_BASE;

        if (argc < 2) {
                fprintf(stderr, "Usage: %s number\n", argv[0]);
                exit(EXIT_FAILURE);
        }

        /* check the bounds of the value being set */
        value = atoi(argv[1]);
        if (value < 1 || value > 15) {
                fprintf(stderr, "Invalid delay setting."
                                "Delay must be between 1 and 15, inclusive.\n");
                exit(EXIT_FAILURE);
        }

        /* open the memory device file */
        fd = open("/dev/mem", O_RDWR|O_SYNC);
        if (fd < 0) {
                perror("open");
                exit(EXIT_FAILURE);
        }

        /* map the LWHPS2FPGA bridge into process memory */
        bridge_map = mmap(NULL, PAGE_SIZE, PROT_WRITE, MAP_SHARED,
                                fd, blink_base);
        if (bridge_map == MAP_FAILED) {
                perror("mmap");
                goto cleanup;
        }
```

```
        /* get the delay_ctrl peripheral's base address */
        blink_mem = (unsigned char *) (bridge_map + BLINK_OFFSET);

        /* write the value */
        *blink_mem = value;

        if (munmap(bridge_map, PAGE_SIZE) < 0) {
                perror("munmap");
                goto cleanup;
        }

        ret = 0;

cleanup:
        close(fd);
        return ret;
}
```

Ignoring all of the error-handling and setup code, the important parts of the program are the following.

```
bridge_map = mmap(NULL, PAGE_SIZE, PROT_WRITE, MAP_SHARED,
                        fd, blink_base);
blink_mem = (unsigned char *) (bridge_map + BLINK_OFFSET);
*blink_mem = value;
```

The `mmap` call maps a single page of memory beginning at 0xff200000 into the process's memory space. The first argument to `mmap` is the virtual memory address we want the mapped memory to start at. By leaving it null, we allow the kernel to use the next memory address available. The second argument is the size of the region we want mapped. The size will always be a multiple of the page size (on Linux, this is 4 kB or 4096 bytes), so we specify the size of a single page even though we only need a byte.

The second line calculates the base address of the "delay_ctrl" peripheral. In this case, `BLINK_OFFSET` is 0, so the addition isn't really necessary, but it's good to use named constants.

Finally, the coup-de-grace, the third line writes to the memory address, setting the value of the `delay_intern` signal in the "delay_ctrl" module. Notice that `blink_mem` is declared with the `volatile` keyword. This tells the compiler that the value stored at this memory address can change without being written to from software. This disables certain compiler optimizations that can cause incorrect behavior.

You can find the source code and a Makefile in the [software/blinker_us](#) folder in the git repository. Compiling it will produce a "blinker" ARM executable, which can be copied to the SD card. The program can be run like so...

```
# enable the lwhps2fpga bridge
echo 1 > /sys/class/fpga-bridge/lwhps2fpga/enable
# make it blink fast
./blinker 1
# make it blink slow
./blinker 15
```

Conclusion

And now you've seen it all, from hardware to software. In my next post, we'll clean things up a bit and write a kernel module to handle the writes to the HPS-to-FPGA bridge.

<- Part 2 Part 4 ->

Howard Mao

Exploring the Arrow SoCKit Part IV - Writing a Linux Device Driver

Now that we are able to control our blinker module from software, we should write a device driver that sets up an interface between our userspace code and the hardware. This allows us to avoid having to mmap "/dev/mem", which is hacky and unsafe.

Ideally, we would like our driver to export a file in sysfs (the /sys filesystem) that we can write a number to and have that number set as the delay value in our hardware.

So here is the code. We will go through it bit by bit in this post.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/device.h>
#include <linux/platform_device.h>
#include <linux/uaccess.h>
#include <linux/ioport.h>
#include <linux/io.h>

#define BLINKER_BASE 0xff200000
#define BLINKER_SIZE PAGE_SIZE

void *blink_mem;

static struct device_driver blinker_driver = {
        .name = "blinker",
        .bus = &platform_bus_type,
};

ssize_t blinker_show(struct device_driver *drv, char *buf)
{
        return 0;
}

ssize_t blinker_store(struct device_driver *drv, const char *buf, size_t count)
{
        u8 delay;

        if (buf == NULL) {
                pr_err("Error, string must not be NULL\n");
                return -EINVAL;
```

```
        }

        if (kstrtou8(buf, 10, &delay) < 0) {
                pr_err("Could not convert string to integer\n");
                return -EINVAL;
        }

        if (delay < 1 || delay > 15) {
                pr_err("Invalid delay %d\n", delay);
                return -EINVAL;
        }

        iowrite8(delay, blink_mem);

        return count;
}

static DRIVER_ATTR(blinker, S_IWUSR, blinker_show, blinker_store);

MODULE_LICENSE("Dual BSD/GPL");

static int __init blinker_init(void)
{
        int ret;
        struct resource *res;

        ret = driver_register(&blinker_driver);
        if (ret < 0)
                return ret;

        ret = driver_create_file(&blinker_driver, &driver_attr_blinker);
        if (ret < 0) {
                driver_unregister(&blinker_driver);
                return ret;
        }

        res = request_mem_region(BLINKER_BASE, BLINKER_SIZE, "blinker");
        if (res == NULL) {
                driver_remove_file(&blinker_driver, &driver_attr_blinker);
                driver_unregister(&blinker_driver);
                return -EBUSY;
        }

        blink_mem = ioremap(BLINKER_BASE, BLINKER_SIZE);
        if (blink_mem == NULL) {
                driver_remove_file(&blinker_driver, &driver_attr_blinker);
                driver_unregister(&blinker_driver);
                release_mem_region(BLINKER_BASE, BLINKER_SIZE);
                return -EFAULT;
        }
```

```
        return 0;
}

static void __exit blinker_exit(void)
{
        driver_remove_file(&blinker_driver, &driver_attr_blinker);
        driver_unregister(&blinker_driver);
        release_mem_region(BLINKER_BASE, BLINKER_SIZE);
        iounmap(blink_mem);
}

module_init(blinker_init);
module_exit(blinker_exit);
```

You can find the module code and Makefile on Github. This code was based off of material in Linux Device Drivers, 3rd Edition, specifically chapters 9 and 14. This is a really great book on writing Linux device drivers written by core kernel maintainers. I highly recommend looking at it if you're interested in learning more.

Setting up the Module

When creating a Linux kernel module, we first need to register init and exit functions, which are run when the module is loaded and unloaded, respectively. In our module, the functions are called `blinker_init` and `blinker_exit`.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("Dual BSD/GPL");

static int __init blinker_init(void)
{
    /* ... */
    return 0;
}

static void __exit blinker_exit(void)
{
    /* ... */
}

module_init(blinker_init);
module_exit(blinker_exit);
```

We register the init and exit functions using the `module_init` and `module_exit` macros. We also need the `MODULE_LICENSE` module to tell the kernel what license we wish to put our module under.

Just the above code would give you a valid kernel module, albeit one that does absolutely nothing. But how

do we build a kernel module? We have to create a Makefile compatible with the Linux kernel's build system. Such a Makefile, assuming you have named your file blinker.c as I have, looks like this.

```
obj-m := blinker.o
```

To compile it, you'd run something like

```
armmake -C ~/path/to/linux-socfpga M=$PWD modules
```

You should replace the path after the "-C" flag with the path to which you cloned the Linux kernel sources. This will run make in the kernel source folder and tell it to build a module in your current directory. You can add a command to your Makefile to run this command for you.

```
KERNEL_SRC_DIR=/home/zhehao/programs/others/linux-socfpga
PWD=$(shell pwd)

all:
	make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C $(KERNEL_SRC_DIR) \
		M=$(PWD) modules
```

The output of the build will be a "kernel object" file called "blinker.ko". You can copy this over to your SD card and load it into the running kernel using the following command.

```
insmod blinker.ko
```

You can then unload it using

```
rmmod blinker
```

Now let's add code to our module to make it do something useful.

Exporting Sysfs File

Linux, being a UNIX-like operating system, subscribes to the philosophy of "everything is a file". That is, the standard way for userspace to communicate with drivers is through file IO operations. For reading and writing small bits of configuration information to driver modules, the Linux kernel provides a filesystem called Sysfs, which is mounted at "/sys" in your filesystem tree.

To get a driver entry in Sysfs, we need to declare and register a `device_driver` struct.

```
#include <linux/device.h>
#include <linux/platform_device.h>

static struct device_driver blinker_driver = {
	.name = "blinker",
	.bus = &platform_bus_type,
```

```
};
```

Device drivers must have a name and a bus. The bus is what connects the device to the CPU. This could be PCI, USB, or some other method. Since our blinker module can be accessed directly from system memory, we will use the generic platform bus type.

We will also need to declare a `driver_attribute` struct, which has function pointers to "show" and "store" functions that are run when userspace reads from or writes to the sysfs file, respectively.

```
ssize_t blinker_show(struct device_driver *drv, char *buf)
{
        return 0;
}

ssize_t blinker_store(struct device_driver *drv, const char *buf, size_t count)
{
    /* ... */
    return count;
}

static DRIVER_ATTR(blinker, S_IWUSR, blinker_show, blinker_store);
```

Since our blinker module is write-only, we don't need to do anything in `blinker_show`. The `DRIVER_ATTR` macro helps us declare a `driver_attr` struct. The arguments to the macro are name, permissions mode, show function, and store function. This will declare a struct called `driver_attr_blinker`. The mode can be any combination of `S_IWUSR`, meaning the user has write access, and `S_IRUGO`, meaning everyone has read access. Again, we want our sysfs file to be write-only, so we only give `S_IWUSR`.

We register our driver in the init function like so ...

```
ret = driver_register(&blinker_driver);
/* error handling ... */
ret = driver_create_file(&blinker_driver, &driver_attr_blinker);
/* error handling ... */
```

Later, in the module exit function, we will unregister the driver.

```
driver_remove_file(&blinker_driver, &driver_attr_blinker);
driver_unregister(&blinker_driver);
```

Now, when the kernel module is loaded, a file will be created at "/sys/bus/platform/drivers/blinker/blinker". Writing to this file will trigger the `blinker_store` function. But how do we make this function do what we want it to?

Accessing IO Memory

As in the previous post, we will set the delay by writing a byte to physical memory at address 0xff200000. However, this address is not yet mapped into the kernel's address space, so we will have to that first. Fortunately, the kernel provides functions for properly mapping and accessing the memory space for peripherals, which is termed IO memory.

First, we will need to request exclusive access to the memory region we want to write to.

```
#define BLINKER_BASE 0xff200000
#define BLINKER_SIZE PAGE_SIZE

res = request_mem_region(BLINKER_BASE, BLINKER_SIZE, "blinker");
if (res == NULL) {
    /* do some error handling */
}
```

`BLINKER_BASE` is set to the base address we want, and `BLINKER_SIZE` is set to the page size. As with the `mmap` system call, we can only get memory a page at a time, so it makes sense to just request a whole page. Now that we know we have exclusive access, we need to map the address into virtual memory.

```
void *blink_mem;

blink_mem = ioremap(BLINKER_BASE, BLINKER_SIZE);
if (blink_mem == NULL) {
        /* error handling */
}
```

We can now write to `blink_mem` to set the hardware delay. Of course, it's not considered proper to just do `*blink_mem = delay`. Instead, we should use the `iowrite*` functions. In our case, we are writing a single byte, so we use `iowrite8`.

```
u8 delay;
if (kstrtou8(buf, 10, &delay) < 0) {
    /* error handling if buf isn't a number */
}
if (delay < 1 || delay > 15) {
    /* error handling if delay out of bounds */
}
iowrite8(delay, blink_mem);
```

Now with the full module, we can write a number between 1 and 15 to "/sys/bus/platform/drivers/blinker/blinker" and set the delay in the FPGA module.

Conclusion

So now you know how to write a basic device driver. There are a lot more things that come into play when developing a driver, and I recommend reading Linux Device Drivers for reference on how to accomplish

certain things.

So far, we have been working with a rather trivial example of what the FPGA can do. If you're interesting in FPGAs, you are probably more interested in getting them to do efficient parallel computation. In my next post, I will introduce a more complex hardware module that will perform such computation.

<- Part 3 Part 5 ->

Howard Mao

Exploring the Arrow SoCKit Part X - Sending and Handling Interrupts

Hi everyone! It's been a long time, but here is another Cyclone V tutorial blog post. This time, we will look at how to send interrupts from the FPGA to the HPS and handle the interrupt in software on the HPS. All hardware descriptions and software programs can be found on Github.

What is an interrupt?

Until now, all of our communication between HPS and FPGA has been initiated by the HPS. In order to detect changes in the state of the FPGA peripherals, the HPS has had to continuously poll the FPGA over the bus. If the state changes infrequently, but we want software to get notified of the change quickly, polling can be rather inefficient. In this case, it would be better if the FPGA could asynchronously notify the HPS of a change.

The way the FPGA can do this is through interrupt. Interrupts are essentially signals going from the FPGA to an interrupt controller on the HPS. The FPGA can make an interrupt request (IRQ) by asserting the interrupt signal high. When an IRQ reaches the HPS, it saves its current state and jumps to an interrupt service routine (ISR). The ISR should service the IRQ by reading or writing some data from the peripheral. Once the ISR has returned, the processor jumps back to its original state.

Creating an Avalon Interrupt Interface

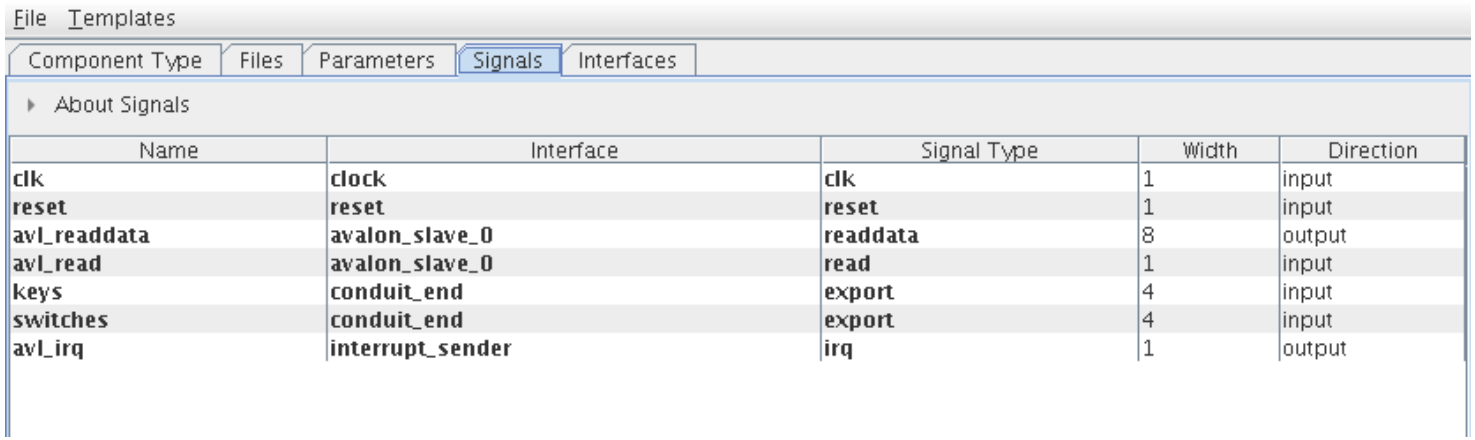
We will create an FPGA peripheral from which we can read the state of the keys and switches attached to the FPGA. The peripheral should send an IRQ when the state changes.

As with other signals sent between FPGA and HPS on the Cyclone V, interrupt signals go through an Avalon interface. The interrupt interface is quite simple, only a single one-bit irq signal is required. However, we also put in a memory-mapped interface so that the state of the inputs can be read.

```
module user_input_device (
    input clk,
    input reset,
    input [3:0] keys,
    input [3:0] switches,

    output avl_irq,
    input  avl_read,
    output [7:0] avl_readdata
);

reg [7:0] cur_inputs;
reg [7:0] last_inputs;
```

```
wire [7:0] changed_inputs = cur_inputs ^ last_inputs;

reg irq;

assign avl_irq = irq;
assign avl_readdata = last_inputs;

always @(posedge clk) begin
    if (reset) begin
        cur_inputs <= 8'd0;
        last_inputs <= 8'd0;
        irq <= 1'b0;
    end else begin
        cur_inputs <= {keys, switches};
        last_inputs <= cur_inputs;
        if (changed_inputs != 8'd0)
            irq <= 1'b1;
        else if (avl_read)
            irq <= 1'b0;
    end
end

endmodule
```

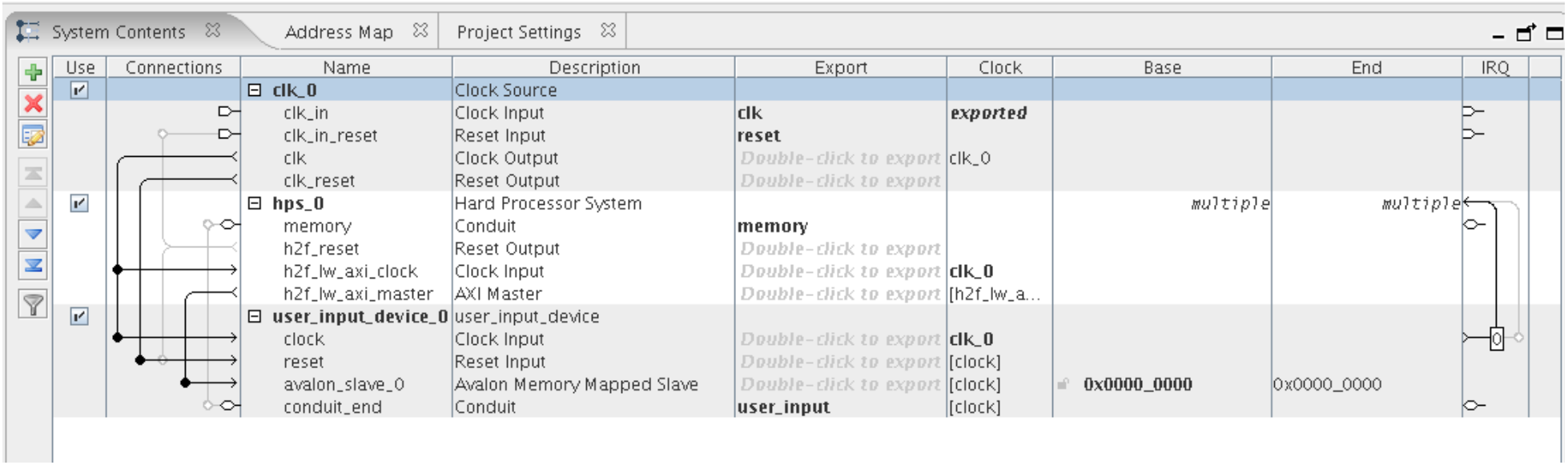
We pull the state of the keys and switches through two stages of registers. If `cur_inputs` and `last_inputs` are different, we set the `avl_irq` signal to high. According to the Avalon interrupt interface specification. The IRQ signal should not be deasserted until the slave has determined that it has been serviced. In this case, we consider the IRQ serviced once the input state is read, so we set `avl_irq` back down to 0 if `avl_read` is asserted.



Name	Interface	Signal Type	Width	Direction
clk	clock	clk	1	input
reset	reset	reset	1	input
avl_readdata	avalon_slave_0	readdata	8	output
avl_read	avalon_slave_0	read	1	input
keys	conduit_end	export	4	input
switches	conduit_end	export	4	input
avl_irq	interrupt_sender	irq	1	output

We can attach this peripheral to the HPS using Qsys. In Qsys, create a new component using the verilog module. Make sure to assign `avl_irq` to an "Interrupt Sender" interface and set the signal type to "irq". Add this component to the system.

When adding the HPS to the system, make sure to check "Enable FPGA-to-HPS interrupts" in the "Interrupts" section of the "FPGA Interfaces" tab. Connect the clock, reset, and avalon slave interfaces as usual. Then, connect the interrupt line by clicking on the path from FPGA peripheral to HPS in the "IRQ" column. Your final system should look something like the following.



Note the "0" on the interrupt line. This is the interrupt number assigned to this IRQ. It is important, as it determines what interrupt number on the HPS corresponds to this interrupt signal. On the Cyclone V, FPGA interrupts start at IRQ number 72, so our interrupt 0 corresponds to IRQ 72.

At this point you should generate your Qsys system. You will see some warnings about not being able to connect clock or reset for "irq_mapper.sender". Do not worry about these warnings. The interrupts will still work.

The Linux Kernel Module

In order to be able to handle these interrupts in software, we need to write a linux kernel module which registers an ISR for our interrupt. A basic module would register an ISR that simply reads the input state and returns. Such a module would look something like this.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/ioport.h>
#include <linux/io.h>
#include <linux/interrupt.h>

void *fpga_uinput_mem;
static uint8_t input_state;

static irqreturn_t fpga_uinput_interrupt(int irq, void *dev_id)
{
```

```
        if (irq != UINPUT_INT_NUM)
                return IRQ_NONE;

        input_state = ioread8(fpga_uinput_mem);

        return IRQ_HANDLED;
}

static int __init fpga_uinput_init(void)
{
        int ret;
        struct resource *res;

        res = request_mem_region(UINPUT_BASE, UINPUT_SIZE, "fpga_uinput");
        if (res == NULL) {
                ret = -EBUSY;
                goto fail_request_mem;
        }

        fpga_uinput_mem = ioremap(UINPUT_BASE, UINPUT_SIZE);
        if (fpga_uinput_mem == NULL) {
                ret = -EFAULT;
                goto fail_ioremap;
        }

        ret = request_irq(UINPUT_INT_NUM, fpga_uinput_interrupt,
                          0, "fpga_uinput", NULL);
        if (ret < 0)
                goto fail_request_irq;

        return 0;

fail_request_irq:
        iounmap(fpga_uinput_mem);
fail_ioremap:
        release_mem_region(UINPUT_BASE, UINPUT_SIZE);
fail_request_mem:
        return ret;
}

static void __exit fpga_uinput_exit(void)
{
        free_irq(UINPUT_INT_NUM, NULL);
        iounmap(fpga_uinput_mem);
        release_mem_region(UINPUT_BASE, UINPUT_SIZE);
        driver_remove_file(&fpga_uinput_driver, &driver_attr_fpga_uinput);
        driver_unregister(&fpga_uinput_driver);
}

MODULE_LICENSE("Dual BSD/GPL");
```

```
module_init(fpga_uinput_init);
module_exit(fpga_uinput_exit);
```

This isn't particularly useful, since there is no way to notify userspace of the state changes. In order to do that, we'll add a read-only sysfs device. Reads on the sysfs file will block until an interrupt occurs. Once this happens, the current state of the inputs is sent to the user.

How do you block the read call? We use a data structure in the kernel called a wait queue. A wait queue can defined like so.

```
static DECLARE_WAIT_QUEUE_HEAD(interrupt_wq);
```

In the "show" function for our sysfs device, we wait until a flag is set by the interrupt controller.

```
static int interrupt_flag = 0;

static ssize_t fpga_uinput_show(struct device_driver *drv, char *buf)
{
        if (wait_event_interruptible(interrupt_wq, interrupt_flag != 0)) {
                ret = -ERESTART;
                goto release_and_exit;
        }

        interrupt_flag = 0;

        buf[0] = input_state;
        ret = 1;

release_and_exit:
        return ret;
}
```

The `wait_event_interruptible` call is what pauses execution of `fpga_uinput_show` until an interrupt occurs. If the wait is interrupted (not by the interrupt we want, but by something like a `SIGINT`), it returns a non-zero value, and we must therefore do some error handling.

If the wait ends successfully, we unset the interrupt flag and copy the input state read from the peripheral to the user.

In our ISR, we must add some code to set the interrupt flag and wake up the processes waiting on the wait queue.

```
interrupt_flag = 1;
wake_up_interruptible(&interrupt_wq);
```

You can find the full code for this kernel module in the Github Repo.

Userspace program

Our userspace program is then pretty simple. All it has to do repeatedly open and read the sysfs file.

```
#define SYSFS_FILE "/sys/bus/platform/drivers/fpga_uinput/fpga_uinput"
#define NUM_SWITCHES 4
#define NUM_KEYS 4

void print_state_change(uint8_t cur_state, uint8_t last_state)
{
        uint8_t changed = cur_state ^ last_state;
        int i;

        for (i = 0; i < NUM_SWITCHES; i++) {
                if (!((changed >> i) & 1))
                        continue;
                if ((cur_state >> i) & 1)
                        printf("switch %d flipped up\n", i);
                else
                        printf("switch %d flipped down\n", i);
        }

        for (i = 0; i < NUM_KEYS; i++) {
                int shift = NUM_SWITCHES + i;

                if (!((changed >> shift) & 1))
                        continue;
                if ((cur_state >> shift) & 1)
                        printf("key %d released\n", i);
                else
                        printf("key %d pushed\n", i);
        }
}

int main(void) {
        FILE *f;
        uint8_t last_state = 0xf0;
        int ret;

        for (;;) {
                uint8_t cur_state;
                f = fopen(SYSFS_FILE, "r");
                if (f == NULL) {
                        perror("fopen");
                        return EXIT_FAILURE;
                }
                ret = fread(&cur_state, 1, 1, f);
                fclose(f);
                if (ret != 1) {
                        if (errno == EAGAIN)
                                continue;
```

```
                        return EXIT_FAILURE;
                }
                print_state_change(cur_state, last_state);
                last_state = cur_state;
        }

        return 0;
}
```

Once the userspace code reads the current state, it compares it to the previous state to determine which of the inputs has changed.

Conclusion

So now you know how to handle FPGA interrupts. This will allow you to design much more efficient interfaces between your FPGA hardware peripherals and the CPU.

<- Part 9