Getting Started with Patmos

Martin Schoeberl

March 8, 2016

These exercises are intended to make you familiar with the Patmos processor and the T-CREST tool chain. The main documentation for the exercise if the Patmos Reference Handbook, which is online available at http://patmos.compute.dtu.dk/ (and as source in t-crest/patmos/doc/handbook).

1 Introduction

This manual assumes that you have the virtual machine (VM) image with the T-CREST tools installed and already compiled. The T-CREST project is hosted at GitHub[1] within several git repositories. You find those repositories local in your VM under directory `t-crest`.

The Patmos processor source lives in directory `patmos`, the directory where you will do most examples.

The build instructions and most exercises for this lab are described in Chapter 6 in the Patmos handbook. As the full tool chain is already setup in your VM, you can skip Section 6.1.

2 Hello World

We start with the standard Hello World:

```
main() {
    printf("Hello Patmos!\n");
}
```

With the Patmos compiler installed and in the PATH it can be compiled to a Patmos executable and run with the simulator as follows:

```
patmos-clang hello.c
pasim a.out
```

[1] https://github.com/t-crest

However, this innocent examples is quiet challenging for an embedded system: It needs a C compiler, an implementation of the standard C library, printf itself is a challenging function, the generated ELF file needs to be understood by a tool and the individual sections downloaded, and finally a terminal (often a serial line) needs to be available on the target, and your test PC needs to have a serial line as well and a terminal program needs to run.

Therefore, we might start with a minimal assembler program and execute that in the simulator and emulator.

If you have not yet downloaded the handbook you can also build it on your VM:

```
cd t-crest/patmos/doc/handbook
make
```

3 Assembler Programming

All compilation and generation is based on `Makefile`s.

To prepare that all assembler tools are compiled and installed execute

```
make tools
```

in the `patmos` folder.

The assembler programs are located in subfolder `asm`. Take a look into `basic.s` and try to understand what this small program does. Assemble the example with:

```
make asm BOOTAPP=basic
```

You should now find a `basic.bin` in the `tmp` folder. This file is just a plain binary file containing the instructions for Patmos. You can display binary files with the Unix command `od` (e.g., with `od -t x1 tmp/basic.bin`). The first 32-bit word in the binary file is the length of the function, that number that was defined in the assembler file with `.word 40;`. The next word should be the first instruction. Look into the Patmos handbook and check if the encoding of the first instruction is correct.

Now execute this 'progam' on the simulator `pasim`. As there is nothing written to stdout, the simulator will not output much. Explore the options (with `-h`) to enable dumping of register contents. The simulator can also print statistics of instruction usage and caches. The assembler and the software simulator can be executed with one step with the help of the `Makefile`:

```
make swsim BOOTAPP=basic
```

The software simulator `pasim` is a C based simulator of the Patmos processor.

Patmos itself is written in `Chisel` a high level language for hardware design. Chisel is a language embedded in Scala. Therefore, you have the full power of Scala available. The Chisel code can generate `Verilog` code for the hardware synthesis and a C++ based emulator to simulate the hardware. The benefit of this Chisel based emulator is that it is exactly the same function as the hardware.

The emulator (the Chisel based simulator) can execute the same program with following command:

```
make hwsim BOOTAPP=basic
```

This command assembles the application, executes the Chisel based hardware construction during which the program is used to initialize the on-chip ROM, generates a C++ based emulator, compiles that emulator, and executes it. The emulator shows the register content after each instruction.

Those two Patmos simulations, the software simulator and the Chisel based emulator, are used for a co-simulation based test. In this co-simulation all available assembler programs are executed in both simulations and the register out put is compared.

You can watch the hardware details by dumping the wave form during the execution of the emulator. To enable waveform dumping you need to add the `-v` option for the call of the emulator in `hardware/Makefile`:

```
test: emulator
        $(HWBUILDDIR)/emulator -v -r -i -l 1000000 -O /dev/null; exit 0
```

Now rerun your example (with `make hwsim`) and change into the `hardware` folder. There you start the waveform viewer with:

```
make view
```

To watch signals they need to be dropped into the wave window. For example the program counter (`io_fedec_pc` from the `fetch` component) and some registers (`rf_1` and `rf_2` from the register file in component `decode/rf`). You should be able to see the same register changes as before, but now with an exact timing, i.e., with the delay between instruction fetch till register write in the last pipeline stage.

Optional: Tinker with the Patmos Hardware

You can find the hardware description of Patmos in `hardware/src/patmos`. Each of the 5 pipeline stages is in its own Chisel class (and file). For example, change some instructions in the `Execute` stage by manipulating `Execute.scala`. You could change the addition to a subtract operation and test it with the `basic.s` program, or your own assembler test program.

Don't forget to undo your changes for the next exercises. The Patmos repository is a `git` repository. Therefore, undo is easily done with:

```
git checkout Execute.scala
```

4 I/O Programming

4.1 Hello World in Assembler

To communicate with the external world, Patmos contains a UART (or serial line) as a minimal I/O interface. In the real hardware that UART is then connected to the PC for text output and

for program download as well. In the simulator the UART output is just echoed to stdout of the host.

The I/O devices are memory mapped, which means they can be accessed with load and store instructions. However, Patmos has typed load and store instructions. Therefore, I/O devices are also mapped into a type. In our case I/O devices are mapped into the local memory areas. Therefore, use `swl` as instruction, like:

```
swl [r7+0] = r9;
```

This above instruction writes the content of register `r9` into a data location at address of register `r7`. Find the address of the UART device in the handbook and write a single character (e.g., '*') to it. The UART is described in the Memory and I/O Subsystem chapter. You can find a short I/O example in `asm/hello.s`.

Optional: The Real Hello World

Transmission of characters takes some time and the processor needs to wait till the next character can be sent. Waiting can be done with a busy loop polling the status register of the UART (the Transmit ready bit).

4.2 Embedded Hello World in C

Embedded systems are often built bare-bone, that means without an operating system and maybe even without a standard library. In this example you shall write a the Hello World example without using `printf`. That means you access the UART with load and store instructions, like you did in the assembler example. Remember, the I/O devices are mapped into local memory space. The Patmos compiler needs to be informed that we do want to access local memory. This is performed with the help of a little macro:

```
#include <machine/spm.h>

int main() {

        volatile _SPM int *uart_status = (volatile _SPM int *) 0xF0080000;
        volatile _SPM int *uart_data = (volatile _SPM int *) 0xF0080004;
```

Emulator and elf File The emulator can read a standard ELF file. Therefore, we use the prebuilt emulator of Patmos and compile only C programs. A barebone C program (e.g., `myhello.c` placed in folder `c`) for the emulator (and the hardware) is compiled with:

```
make comp APP=myhello
```

We execute this .elf program with the emulator:

```
patmos-emulator tmp/myhello.elf
```

or with `pasim`.

Now start similar to the assembler based Hello World and write a short program to write a single character to the UART.

As a next step write out a longer string of characters. However, transmission of characters takes some time and the processor needs to wait till the next character can be sent. Waiting can be done with a busy loop polling the status register of the UART (the Transmit ready bit).

4.3 Periodic Tasks

Real-time tasks are usually periodic tasks. Therefore, we will program a small example that uses the Patmos time to execute periodic tasks. First we start with polling of the timer/counter to generate periodic event. Write out a character about every second. For this polling use the timer counter and wait until some time elapsed. As we run in a simulation, time elapses way slower. Therefore, start with short waiting times and increase with error and retry.

With this example you can explore the simulation time difference between the SW simulator `pasim` and the hardware generated emulator. Which one is faster? And by how much?

Optional: Periodic Task as Interrupt Handler

Polling consumes computing resources and is only a solution for single tasks. Better is to use a time interrupt and an interrupt handler for the periodic task. Reprogram the above example as a timer interrupt handler. You can find an example for interrupt handlers in `c/intrs.c`.

Having the timer interrupt under control is almost half of a scheduler for a real-time operating system!

5 Further Steps

After this exercise you master the T-CREST tool flow for the Patmos processor. Next step is to get an FPGA board, such as the Altera DE2-115, and see the processor executing in real hardware. From this on you can proceed to extend the processor with your own ideas, explore the multicore version of Patmos with the real-time network on chip Argo, write your own operating systems, do WCET analysis with `aiT` and/or `platin`, ...

Contributions are always welcome and easy to do with a GitHub pull request. You can ask questions to the Patmos community via the Patmos mailing list. See: http://patmos.compute.dtu.dk/.