Customizable ARM Designs and Linux
The SoC FPGA family from Altera

Dipl.-Math. techn. Detlev Zundel (dzu@denx.de)
DENX Software Engineering GmbH
Kirchenstr. 5
82194 Gröbenzell, Germany

June 25, 2013
DENX

Customizable ARM Designs and Linux
Today we have these technologies growing towards each other:

**General processors**
- Software programmable
- Great flexibility
- Poor power efficiency
- Few application specific features

**Application-specific**
- Hard-wired, not programmable
- Poor flexibility
- Great power efficiency
- Many contain embedded processors
Where the trends meet

These trends meet in SoC FPGAs with Hard Processor Systems (HPS)

- Hardware programmable
- Great flexibility
- Good power efficiency
- Set of commonly required interfaces (Ethernet, I\textsuperscript{2}C, SPI, CAN, ...)
ARM Cortex A9 + Altera FPGA

ARM Processor System
- Dual Core ARM Cortex-A9 MPCore Processor
- Hard Memory Controller
- Peripherals

28-nm FPGA
- Cyclone V
- Arria V

SoC
System Architecture

- Processor
  Dual-Core ARM Corex-A9 cpu
  4000 DMIPS (@800 MHz per core)
  NEON coprocessor with double-precision FPU
  32 KiB I- and 32 KiB D L1 $
  shared 512 KiB L2 $

- Multiport SDRAM controller
  Up to 533 MHz DDR3
  Up to 400 MHz DDR2

- High- bandwidth on-chip interface
  125 Gbps HPS-to-FPGA
  125 Gbps FPGA-to-SDRAM
SoC Device Family Plan

There SoCFPGA family can optimally fit a large range of requirements:

<table>
<thead>
<tr>
<th>Family</th>
<th>KLE</th>
<th>Block Memory Bits (Mb)</th>
<th>Var. Prec. Multiplier Blocks</th>
<th>Max. FPGA User I/Os</th>
<th>Max. HPS I/Os</th>
<th>Max. XCVRs (GP)</th>
<th>Per-XCVR Max. Data Rate (Gbps)</th>
<th>HPS Hard Memory Controller</th>
<th>FPGA Hard Memory Controllers</th>
<th>Hard PCIe®</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclone V SoC</td>
<td>25</td>
<td>1.4</td>
<td>36</td>
<td>145</td>
<td>188</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2 ea, Gen1</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>2.2</td>
<td>58</td>
<td>145</td>
<td>188</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2 ea, Gen1</td>
</tr>
<tr>
<td></td>
<td>85</td>
<td>4.0</td>
<td>87</td>
<td>288</td>
<td>188</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>2 ea, Gen2</td>
</tr>
<tr>
<td></td>
<td>110</td>
<td>5.1</td>
<td>112</td>
<td>288</td>
<td>188</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>2 ea, Gen2</td>
</tr>
<tr>
<td>Arria V SoC</td>
<td>350</td>
<td>17.3</td>
<td>809</td>
<td>528</td>
<td>216</td>
<td>30 / 16</td>
<td>6 / 10</td>
<td>1</td>
<td>3</td>
<td>2 ea, Gen2</td>
</tr>
<tr>
<td></td>
<td>460</td>
<td>22.8</td>
<td>1,068</td>
<td>528</td>
<td>216</td>
<td>30 / 16</td>
<td>6 / 10</td>
<td>1</td>
<td>3</td>
<td>2 ea, Gen2</td>
</tr>
</tbody>
</table>
Qsys flow

1. Create Quartus II Project (Select SoC Device)
2. Create Qsys Project
3. Add/Configure HPS IP
4. Add IP to Qsys System
5. Add Custom IP to Qsys System
6. Interconnect Components
7. Generate Qsys System
8. Perform Functional Simulation
9. Instantiate Qsys System in Quartus Project
10. Run Analysis & Elaboration
11. Invoke I/O Assignments (DDR, HPS Bank VCC)
12. Compile Quartus II Project
13. HW Verification
14. SW Handoff
HPS in Qsys

The Quartus toolchain fully supports GNU/Linux as development host.
Software Support

Even though the device can run “bare metal” applications just fine and other operating systems will eventually be supported, GNU/Linux with its already existing excellent ARM and SMP support will be the main focus of the device.

Linux support was developed from the beginning in the regular community context and - with some help from DENX - fed into the upstream projects, i.e. U-Boot and the mainline Linux kernel.
Booting from non-directly addressable devices

In the “good old days” a CPU started by fetching instructions from a specific address (reset vector) from directly attached memory (usually NOR flash).
Nowadays directly attached memory is the exception. More often we have mass storage devices that need a protocol themselves:

- NAND Flash
- SPI NOR Flash
- SD/MMC Card
In order to support booting from non-directly addressable memory, the CPU will execute *Boot ROM* code that will usually transfer a fixed size of code into internal RAM. For such setups, the *SPL framework* was started. SPL U-Boot is designed to be a very small loader (sharing code with U-Boot) that itself gets loaded from a boot ROM. It’s main purpose is to load and start the “payload”. Initially, the payload was always the (full) U-Boot but today we have SPL U-Boots capable of booting a linux kernel directly (“Falcon Mode”).
HPS Boot

The *Boot ROM* copies the *Preloader* from a device (BSEL and CSEL) into internal memory (64 KiB, but boot ROM uses top 4 KiB, so we have a limit of 60 KiB) and jumps to it. A typical boot flow then is

- Reset
- Boot ROM
- Preloader (SPL U-Boot)
- Bootloader (U-Boot)
- Operating system (Linux)
- Application (Userspace)
As described, the Preloader depends on the actual design. U-Boot is thus split into the `u-boot-spl.bin` and the `u-boot.bin` parts. The latter is quite generic but the former depends on the boot medium and on the system design. Auto-generated files are fed into an U-Boot compilation process for this to work.
The Linux kernel needs drivers for the hardware of the system. In our case these may be:

- HPS subsystem blocks like Ethernet and CAN (fixed)
- Functional blocks in the FPGA part (variable)

The first group of drivers is already mostly in mainline, whereas the second group will always be very project specific. A key concept for both however, are *Device Trees* to describe the hardware to driver software.
Device Trees

The idea of having an OS independent description of the hardware has been around for a while. IBM, Sun and Apple had “Open Firmware Trees” for this purpose. Beginning with the arch/ppc64 and arch/ppc unification, “flattened device trees” were introduced in the Linux kernel. Other architectures then started to replace ad-hoc mechanisms with this approach. The ARM kernel adopted the device tree instead of the old ATAGs around 3.0.
Device Tree excerpt 1

Specifying the CPU cores:

```c
cpus {
    #address-cells = <1>;
    #size-cells = <0>;

    cpu@0 {
        compatible = "arm,cortex-a9";
        device_type = "cpu";
        reg = <0>;
        next-level-cache = <&L2>;
    }

    cpu@1 {
        compatible = "arm,cortex-a9";
        device_type = "cpu";
        reg = <1>;
        next-level-cache = <&L2>;
    }
}
```
Some peripherals:

uart0: serial0@ffc02000 {
    compatible = "snps,dw-apb-uart";
    reg = <0xffc02000 0x1000>;
    interrupts = <0 162 4>;
    reg-shift = <2>;
    reg-io-width = <4>;
};

uart1: serial1@ffc03000 {
    compatible = "snps,dw-apb-uart";
    reg = <0xffc03000 0x1000>;
    interrupts = <0 163 4>;
    reg-shift = <2>;
    reg-io-width = <4>;
};
Short excursion to device driver design

In order to support a new peripheral that is included in the FPGA part, there are three general approaches:

- Implement a full Linux driver representing the device at its functional level
- Use the uio framework to export the memory mapped registers into userspace.
- Bypass the whole operating system by using mmap on /dev/mem. This is fraught with peril and can only be a quick hack in early project phases.
Doing `mmap` on `/dev/mem` is of course the anti-thesis of using an operating system by bypassing it completely. One step better is to abstract only the relevant part of the address space into its own simple “device”:

```c
uio-gpio@ff2000000 {  
    compatible = "generic-uio";
    reg = <0xff200000 0x20>;
};
```
A uio “driver” isn’t really a driver but only a very small shim in the linux kernel to export a memory area. Introducing interrupts complicates the picture somewhat - we don’t like userspace being in control of acking IRQs. Of course userspace has no control of cache flushing, so the memory regions exported in this way are not cached and do not allow DMA in any way.
Worst of all an uio driver in truth violates the design consideration of an operating system. Knowledge of the implementation of the actual device “escapes” into userspace and thus has to be taken care of in the application.
A real device driver

Operating systems on the other hand should isolate devices at a *functional level* to allow an application to work with very different devices implementing the same functionality. For example, an application using the *Socket CAN API* can work on a PCI CAN card, a SoC with a CAN block or even a microcontroller hooked up via SPI having multiple CAN interfaces. This works as long as they all implement a Socket CAN driver.
The Linux kernel needs a root filesystem where it will find the programs to execute. Usually the process with PID 1 will be the `/sbin/init` process. Apart from that this filesystem contains all the other binaries and libraries. Even “simple” root filesystems consist of hundreds of files.
How to generate a Root-FS

Unfortunately there is no “embedded GNU/Linux distribution” that fits the very tight constraints of embedded systems. Assembling a root filesystem by hand is also not really feasible. Taking a desktop GNU/Linux distribution like Ubuntu or Debian and “cutting it down” also is not a very promising direction.
Yocto

Yocto is meant to design, develop, build, debug, simulate, and test the complete software stack using Linux, the X Window System, GNOME Mobile-based application frameworks, and Qt frameworks. From the beginning it supports ARM, PowerPC, MIPS and x86 systems. With its open development model backed by the Linux Foundation, it is vendor independent and thus future proof.
Getting a head start with ELDK

Yocto will be the reproducible “all encompassing” build procedure. During development it is perfectly ok to work with ad-hoc methods instead. So at the beginning of the project one can for example use the pre-compiled ELDK toolchain together with one of its pre-compiled rootfilesystems. Installation of a toolchain and root-fs (usable over NFS) is a matter of minutes (see http://www.denx.de/wiki/ELDK-5):

$ ./install.sh -s gmae -r lsb-sdk armv7a-hf

Using it is also very easy:

$ eldk-switch -r 5.3 armv7a-hf
Altere SoC Development Board
EBV SoCrates board

- **Altera Cyclone V SoC device:**
  - 5CSEBA6U23C7N
  - 110 K LEs
  - 112 DSP Blocks
  - 5.1 Mbit RAM

- **Interfaces:**
  - 1GiB Ethernet
  - USB 2.0 OTG
  - CAN
  - SPI
  - I²C

- **Memory:**
  - 128MiBx32 DDR3 Memory
  - µSD Card Slot
  - 2x EPCQ256 Configuration Device
Welcome to RocketBoards.org

Accelerate development by exchanging ideas with the open source community. Discover the right resources for your embedded solutions. Use one of the community-oriented development boards for the SoC, targeted for people interested in exploring and prototyping their applications.

Get started now

Getting started
- Altera SoC Development Board
- Arrow SoCKit Evaluation Board
- EBV SoCrapes Evaluation Board
- Manuel Heis SoC Evaluation Kit

Projects
- Register a Project
- Project Hosting
- List All Projects
- Search Projects

Community
- Mailing Lists
- Git Web
- Get Involved
- Register on RocketBoards.org

What's new
- Fixed USB peripheral driver
  Posted 14 Jun 2013 - 18:20
- Upgrade to kernel version 3.9
  Posted 14 Jun 2013 - 18:20
- Expand Your Open Source Embedded Universe with RocketBoards.org
  Posted 02 May 2013 - 21:56
- Upgrade to kernel version v3.8
  1
Further reading

- [http://rocketboards.org](http://rocketboards.org) Online Community
- [http://rocketboards.org/gitweb](http://rocketboards.org/gitweb) Git Repositories (U-Boot, Linux, Poky)
- [http://www.denx.de/wiki/ELDK-5](http://www.denx.de/wiki/ELDK-5) The ELDK 5 toolchain and target distribution
- [http://www.denx.de/wiki/U-Boot](http://www.denx.de/wiki/U-Boot) das U-Boot
- [http://www.denx.de/](http://www.denx.de/) our homepage