DE1 SoC Overview and Configuration of Android OS
THE BOARD
Board Layout
Why so many other chips?

Switch bouncing problem

FTDI UART to USB

Provides important log data at boot up!!
Cyclone V Peripheral Connections
Inside SoC
Physical Address Mapping

ACP: Accelerator Coherency Port
Cyclone V HPS Memory Map

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x 0000 0000</td>
<td>1001 1001</td>
</tr>
<tr>
<td>0x FF704000</td>
<td>0100 1011</td>
</tr>
<tr>
<td>0x FF704001</td>
<td>1101 0010</td>
</tr>
<tr>
<td>0x FF7043FF</td>
<td>0100 1011</td>
</tr>
<tr>
<td>0x FFFFFFFF</td>
<td>1001 1001</td>
</tr>
</tbody>
</table>

Slaves via HP AXI Bridge
Slaves via LW AXI Bridge
SDMMC Module
QSPI Flash Controller Module
GPIO Module
USB OTG Controller
DMA Module
Boot ROM Module
On-Chip RAM Module
Custom Logic Bindings

1. Custom Logic can be a master or slave
2. Usually NOIS CPU (Soft IP) is Avalon-MM Master
3. Usually any custom I/O is Avalon-MM Slave
Avalon-MM Slave Interface

<table>
<thead>
<tr>
<th>Name</th>
<th>Width</th>
<th>Direction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>avs_address</td>
<td>64 bits</td>
<td>Input</td>
<td>Address of slave being accessed</td>
</tr>
<tr>
<td>avs_read</td>
<td>1 bit</td>
<td>Input</td>
<td>Read operation requested</td>
</tr>
<tr>
<td>avs_write</td>
<td>1 bit</td>
<td>Input</td>
<td>Write operation requested</td>
</tr>
<tr>
<td>avs_readdata</td>
<td>8, 16, 32, or 64 bits</td>
<td>Output</td>
<td>Data read from slave</td>
</tr>
<tr>
<td>avs_writedata</td>
<td>8, 16, 32, or 64 bits</td>
<td>Input</td>
<td>Data to be written to slave</td>
</tr>
</tbody>
</table>

Read Waveforms:

Write Waveforms:
Avalon-MM Slave Interface – Software Side
Putting it all together

**HARDWARE**

- Altera Quartus
- Quartus Programmer
- Flash Drive
- Cyclone V
- FPGA
- 7-Seg Disp.
- LEDs
- USB-Blaster II
- Audio In/Out
- VGA Out
- USB Host
- UART-to-USB
- SD Card
- SDRAM
- 2x20 GPIOs
- Push Buttons
- Switches

**SOFTWARE**

- Altera Custom Compiler
- .X Executable File
- Terminal Shell
- Serial Communication

- Altera SRAM Object File
- User Generated Software
- NIOSII Shell Generated Software
- .h Header File
- .c Source File
- .qproj Project
BOOT SEQUENCE 3
# Internal ROM Configuration

<table>
<thead>
<tr>
<th>MSEL[4:0]</th>
<th>Configuration Scheme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10010</td>
<td>AS</td>
<td>FPGA configured from EPCQ (default)</td>
</tr>
<tr>
<td>01010</td>
<td>FPPx32</td>
<td>FPGA configured from HPS software: Linux</td>
</tr>
<tr>
<td>00000</td>
<td>FPPx16</td>
<td>FPGA configured from HPS software: U-Boot, with image stored on the SD card, like LXDE Desktop or console Linux with frame buffer edition</td>
</tr>
</tbody>
</table>
Boot Stages

1. Reset
Boot Stages

1. Reset
2. Boot from ROM
3. Setup boot source from BSEL pins
**Boot Stages**

1. Reset
2. Boot from ROM
3. Setup boot source from BSEL pins
4. Copy Pre-loader into Onchip RAM
Boot Stages

1. Reset
2. Boot from ROM
3. Setup boot source from BSEL pins
4. Copy Pre-loader into Onchip RAM
5. Run pre-loader
6. Setup HPS I/O and DDR
Boot Stages

1. Reset
2. Boot from ROM
3. Setup boot source from BSEL pins
4. Copy Pre-loader into Onchip RAM
5. Run pre-loader
6. Setup HPS I/O and DDR
7. Configure FPGA (Optional)
Boot Stages

1. Reset
2. Boot from ROM
3. Setup boot source from BSEL pins
4. Copy Pre-loader into Onchip RAM
5. Run pre-loader
6. Setup HPS I/O and DDR
7. Configure FPGA (Optional)
8. Copy User Bootloader into DDR RAM
Boot Stages

1. Reset
2. Boot from ROM
3. Setup boot source from BSEL pins
4. Copy Pre-loader into Onchip RAM
5. Run pre-loader
6. Setup HPS I/O and DDR
7. Configure FPGA (Optional)
8. Copy User Bootloader into DDR RAM
9. Run User Bootloader
Boot Stages

1. Reset
2. Boot from ROM
3. Setup boot source from BSEL pins
4. Copy Pre-loader into Onchip RAM
5. Run pre-loader
6. Setup HPS I/O and DDR
7. Configure FPGA (Optional)
8. Copy User Bootloader into DDR RAM
9. Run User Bootloader
10. Configure FPGA (optional)
Boot Stages

1. Reset
2. Boot from ROM
3. Setup boot source from BSEL pins
4. Copy Pre-loader into Onchip RAM
5. Run pre-loader
6. Setup HPS I/O and DDR
7. Configure FPGA (Optional)
8. Copy User Bootloader into DDR RAM
9. Run User Bootloader
10. Configure FPGA (optional)
11. Copy OS into DDR RAM
Boot Stages

1. Reset
2. Boot from ROM
3. Setup boot source from BSEL pins
4. Copy Pre-loader into Onchip RAM
5. Run pre-loader
6. Setup HPS I/O and DDR
7. Configure FPGA (Optional)
8. Copy User Bootloader into DDR RAM
9. Run User Bootloader
10. Configure FPGA (optional)
11. Copy OS into DDR RAM
12. Run OS
13. Run Applications
Boot Stages

1. Reset
2. Boot from ROM
3. Setup boot source from BSEL pins
4. Copy Pre-loader into Onchip RAM
5. Run pre-loader
6. Setup HPS I/O and DDR
7. Configure FPGA (Optional)
8. Copy User Bootloader into DDR RAM
9. Run User Bootloader
10. Configure FPGA (optional)
11. Copy OS into DDR RAM
12. Run OS
13. Run Applications
14. Configure FPGA (optional)
ANDROID ON DE1 SoC
What is Android?
• Code is available online and managed by Google Android Open Source Project (AOSP).
• Based on Linux.
• Designed for embedded devices.
• Suited to low power devices
• Android is Everywhere
Android Boot Sequence

Bootloader
- Initialize RAM
- Put basic HW in quiescent state
- Load kernel and RAM disk
- Jump to kernel

Kernel
- Initialize environment to run C code
- Initialize kernel subsystems
- Initialize all drivers
- Mount root File System
- Start "init" process

Native Daemons
- servicemanager
- vold
- netd
- debuggerd
- rild
- app_process-X Zygote
- mediaserver

Init
- Set up env. variables
- Create mount points
- Mount File System (FS)
- Set up FS permissions
- Set OOM adj.
- Start native daemons

Android Runtime (ART)
- Start a Dalvik VM
- Call Zygote's main()

Zygote
- Register Zygote socket
- Preload all Java classes
- Preload resources
- Start System Server
  - Open socket
  - Listen for connections

App

System Server
For each service:
- Init service
- Register it with ServiceManager

Activity Manager
- Init itself
- Register onClick() handlers

Launcher
- Init itself
- Register onClick() handlers

CPU

Bootloader

Kernel

Init

Native Daemons

Android

Linux

App

Activity Manager

Flutter

StartActivity()
LITERATURE REVIEW
Books Used
Difference Between Linux and Android

Linux (Embedded System)
- Power Up Device
- Internal ROM
- Preloader (BL1)
- Bootloader (U-Boot)
- Kernel
- INIT
- Embedded Linux (Root File System)

Linux
- Power Up Device
- System Startup (BIOS)
- Preloader (MBR)
- Bootloader (GRUB, LILO, etc)
- Kernel
- INIT
- Linux (Root File System)

Android (Embedded System)
- Power Up Device
- Internal ROM
- Preloader (BL1)
- Bootloader (U-Boot)
- Kernel
- INIT
- Zygote
- Dalvik VM
- Android (Root File System)
Linux Device Files

The application finds the device file based on device file name

Device file finds the device driver based on the major number

Device driver finds device based on the minor number

Java/C/C++...
Application
A File
Device File
Executable
Device Driver
Hardware
Device

/dev
/bin
/etc
mnt

ttyS0
ttyS1
ttyS2
ttyUSB0
socfpga
enable
hps_to_fpga
fpga_to_hps
Raw Binary Files (.rbf)

- Write hardware code in VHDL/Verilog/System Verilog
- Compile and verify the design using Quartus to get SRAM Object File (.sof)
- Convert .sof file to .rbf
- Copy the generated .rbf file to SD card’s FAT partition
- Decide on how to load the configuration bit stream (Choose from following)
  - Pre-loader script
  - U-Boot source code
  - U-Boot script
  - Linux init
  - Linux application (runtime)
Altera SoC Linux Hardware/Software Handoff

Legend:
- Provided by Altera
- Open Source
- Input File
- Intermediate File
- Output File

Hardware Project → Quartus II → Handoff Folder

Board Information → Device Tree Generator: sopc2dts

Bootloader Generator: bsp-editor

Bootloader DT Source → DTC → Bootloader DT Blob

Regenerate when hardware project is recompiled

Make

Regenerate only when user options (boot source etc.) change

mkpimage → u-boot Binary

Legend:
- Provided by Altera
- Open Source
- Input File
- Intermediate File
- Output File

Linux DT Blob
## Step 2: Finding Resources

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Component</th>
<th>Repository Name</th>
<th>GitHub URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Angstrom Scripts</td>
<td>angstrom-socfpga</td>
<td><a href="https://github.com/altera-opensource/angstrom-socfpga">https://github.com/altera-opensource/angstrom-socfpga</a></td>
</tr>
<tr>
<td>3.</td>
<td>Device Tree Generator</td>
<td>sopc2dts</td>
<td><a href="https://github.com/altera-opensource/sopc2dts">https://github.com/altera-opensource/sopc2dts</a></td>
</tr>
</tbody>
</table>
Step 4: Generating Preloader

[Diagram showing the process of generating a preloader.]
Step 5: Generating U-Boot

$ git clone https://github.com/altera-opensource/u-boot-socfpga.git
$ make mrproper
$ make socfpga_cyclone5_config
$ make

Android (DE1-SoC)
- Power Up Device
- Internal ROM
- Preloader (BL1)
- Bootloader (U-Boot)
- Kernel
- INIT
- Zygote
- Dalvik VM
- Android (Root File System)

Preloader (Unknown)
- U-Boot (W95 FAT32)
- Linux File System (Linux)

Micro SD
- 1 MB
- 256 MB
- ~ MB

.copy
uboot.img

.copy
Step 6: Generating U-Boot Script

- echo -- Programming FPGA –
  fatload mmc 0:1 $fpgadata soc_config.rbf;
  fpga load 0 $fpgadata $filesize;
  run bridge.enable_handoff;

- echo -- Setting Env Variables –
  setenv fdtimage soc_system.dtb;
  setenv mmcroot /dev/ram;
  setenv mmcmd 'mmc rescan;
  ${mmcmd} mmc 0:${mmcmdpart} ${loadaddr}
  ${bootimage};
  ${mmcmd} mmc 0:${mmcmdpart} ${fdtaddr}
  ${fdtimage};
  setenv mmcmdboot 'setenv bootargs
  console=ttyS0,115200 root=${mmcroot} rw
  rootwait;
  bootz ${loadaddr} - ${fdtaddr}';

  run mmcmd;
  run mmcmdboot;

---

Diagram:
- U-Boot (W95 FAT32)
- Preloader (Unknown)
- Linux File System (Linux)
- Micro SD
- Android (Root File System)
- Android (DE1-SoC)
- Power Up Device
- Internal ROM
- Preloader (BL1)
- Bootloader (U-Boot)
- Kernel
- INIT
- Zygote
- Dalvik VM
- Android (Root File System)

Notes:
- copy uboot.scr
- 1 MB
- 256 MB
- ~ MB
Step 7: Configuring Linux

$ git clone https://github.com/altera-opensource/linux-socfpga.git
$ make ARCH=arm socfpga_custom_defconfig
$ make ARCH=arm menuconfig

Specify the file system and RAM disk to load
Step 8: Compiling Linux

$ make ARCH=arm LOCALVERSION= zImage
Step 8: Generating Device Tree Blob

$ make ARCH=arm CONFIG_DTB_SOURCE=arch/arm/boot/dts/baseTree.dts <your-dev-board>.dtb
**Step 9: Android Bindings**

Developing your device drivers is similar to developing a typical Linux device driver. Android uses a version of the Linux kernel with a few special additions such as wake locks (a memory management system that is more aggressive in preserving memory), the Binder IPC driver, and other features important for a mobile embedded platform. These additions are primarily for system functionality and do not affect driver development.

You can use any version of the kernel as long as it supports the required features (such as the binder driver).
Step 10: Android Application

- `/`
- `dev`
- `socfpga`
  - `enable`
  - `hps_to_fpga`
  - `fpga_to_hps`

write
write
read
Raw Binary Files (.rbf) in Android Application Package (.apk)

- Load .rbf file on activity create event onto the fpga for acceleration
- Remove .rbf file on activity close event
- Android Init() should configure the fpga with base configuration file and several PRRs (Partial Reconfiguration Regions).
CONCLUSION
App-based Hardware Acceleration
People who are really serious about software should make their own hardware.

~ Alan Kay (Computer Scientist)
References

THANKS!

Any questions?
You can find me at
ENG 402 / mobaidullah@ryerson.ca