MaSoCist 'bertram' reference

Martin Strubel

November 6, 2017

Revision:
`v0.3-proprietary`

1 Overview

The cCAP – configurable Custom Application Processor – is a CPU design which allows to implement custom functionality 'inline', meaning, that specific calculations or acceleration co-routines can be embedded into the program code. Application specific hard IP cores can be implemented as independent coprocessor as well.
The cCAP family is heavily based on the MaSoCist environment – a build and configure environment for CPUs using the Linux kernel config utility and GNU make, enhanced by a proprietary tool for the SoC (System on Chip) architecture generation.
The cCAP architecture is by default technology independent, i.e. configureable and portable to many FPGA architectures. The main development however is focused on Xilinx and Lattice FPGAs.

1.1 Target applications

The cCAP do not intend to replace any of the middle end solutions provided by most FPGA vendors. It tries to match a niche where resources are sparse and where IP core configuration and debugging as well as optimum code density is more important than speed. Apart from the PyPS core, the system is not intended to run any major operating systems, but rather simple, bare metal code running with a somewhat deterministic sequence.
Reference applications:

- Smart video I/O processor for HDMI I/O and RGB/YUV conversion
- JPEG encoder with direct Bayer to YUV ('Cottonpicken' engine)
- Simple multi channel LED PWM controller (LED matrix)
- netpp communication stack on FPGA for IoT applications
- Special Coprocessor DSP solutions (custom purpose accelerators)

A few more application scenarios are listed below:

The core is too slow for some operations, for example realtime PWM control
Enhance CPU core by special engine 'code window'

You need a multitasking OS to serve several tasks
Instance another core and set up shared memory or inter-core FIFO for communication

You are tired of hardware and software mismatches
Let configuration tools do the versioning and synchronization **automatically**.

You require increased security or you would like to prevent cloning
Lock up your intellectual property behind obscurity or even encryption, and bake your algorithms into hardened logic

1.2 cCAP Design family

The cCAP main line is based on the proprietary big endian ZPUng core that is fully binary compatible with the classic ZPU architecture. However, it can have vendor specific code extensions. It is also possible for the 'a' family to change the CPU against the OpenSource Zealot variant.
The cCAP family includes the following base peripherals:

(TAP)
Test Access Port via native JTAG interface for debugging

UART
UART console for interactive I/O

SPI Simple polled SPI master interface

TIMER/PWM
Rudimentary timer and PWM controller for synchronous pulse width output

Name	Extra Peripherals	Application
agathe	8x PWM	OpenSource config processor demo first generation
anselm	"	", for MACHXO[2,3] platforms
agneta	EFB	Configuration processor for Lattice MACHXO[2,3] platforms
beatrix	LCDIO, SCACHE, SPI32	embedded netpp processor
bertram	PWMplus, SCACHE, SPI32	embedded netpp processor with PWMplus extensions
cranach	SCACHE, SPI32, MAC, DMA	networked netpp/UDP processor
(cordula ZPUng)	DMA, JPEG, CPK, VPI	Video processor
dombert	DMAA, JPEG, MAC, VPI	JPEG encoder SoC (*dorothea* successor)
dagobert	DMAA, MAC, VPI	netpp/UDP 'industrial' processor

Table 1.1: cCAP overview

Table 1.2 shows the standard reference design chip family and its typical logic resource usage in a set of supported FPGAs. These designs are considered mature and stable. The more complex the peripherals are, the higher ranks the naming in the alphabet.
The given usage percentage is based on a pessimistic maximum of the number of SLICEs used and the LUT usage.
The extra peripherals are specific to the configuration:

SCACHE
Virtual ROM (SPI cache). Enhances program memory up to several MB of ROM code for program overlay or data storage

LCDIO
Custom LCD driver engine

SPI32
32 bit word capable SPI I/O, DMA capable

Name	CPU / FPGA >	MACHXO?-7000	SP3-250	SP6-LX9	ECP3-45	ECP5-45
agathe	Zealot	N/A	15%	-	-	-
anselm	Zealot	25%	N/A	N/A	-	-
agneta	ZPUng	34%	N/A	-	-	-
beatrix	ZPUng	-	50%	-	-	-
bertram	ZPUng	40%	90%	20%	-	-
cordula	PyPS	-	-	-	20%	-
cranach	ZPUng	-	-	30%	15%	14%
dombert	ZPUng	-	-	N/A	N/A	29%
dagobert	ZPUng	-	-	60%	N/A	N/A

Table 1.2: FPGA usage overview

I2C proprietary i2c peripheral (no support for clock stretching) or optional OpenCores implementation

PWMPLUS
Improved PWM for realtime pulse width control

FX2FIFO
Cypress FX2 FIFO interface for fast isochronous data transfer

MAC
Ethernet MAC interface for RGMII or GMII capable GigE Phy

DMA/DMAA
DMA engine with optional autobuffer configuration (high speed peripheral streaming)

JPEG
JPEG hardware encoder

CPK
'Cottonpicken' proprietary image processing pipeline

VPI
Parallel port peripheral video interface

SPORT
Fast serial port for audio codec I/O, DMA capable

1.3 MaSoCist build environment

A short feature overview:

- IP core instanciation and SoC configuration via XML description
- Automatic synchronization of HDL and software
- Wishbone support
- Optional Co-Simulation support

- CPU and IP cores mostly open source

The MaSoCist environment is featured by a few number of CPU cores as shown in Table 1.3. The SoC configuration families have specific names, like agathe, beatrix, etc. Typically, the configuration is independent from the CPU core, however, the address map of the CPU core must correspond to the configuration, therefore the core variants may have specific extensions.

Name	Description
ZPU small core (Zealot)	OpenSource ZPU small, non pipelined
ZPUng	Proprietary, pipelined ZPU variant, configureable

Table 1.3: Default SoC core CPU options

Other cores (such as vendor specific CPU designs) can be integrated on request. Basically, the MaSoCist environment can be customized to use any other third party core with Wishbone support at possible sacrifices with respect to In Circuit Emulation debugging.
The cCAP supports integration of third party, Wishbone compatible IP cores, such as the popular I2C core from opencores.org. These will need a specific description in the I/O peripheral map description.
This overview does not cover the details of SoC architecture implementations using the devdesc XML device description. Please refer to the generic netpp/SoC manual (available as separate support package).

1.4 Licensing

To avoid too much complicated language, the licensing scheme is simplified in this paragraph. There are several licenses and other parties' rights involved in the MaSoCist:

1. MaSoCist components license
2. ZPU Opensource license
3. Third party (typically FPGA vendor) licenses, such as for generated IP cores

The licensing terms may turn out complex when further developments fall under different licenses, or may even collide with licenses of a third party module. The original MaSoCist module license rules, apart from the ZPU/Zealot licensing, are summarized as follows:

- You do have a license to simulate, build, and use the synthesized image on the hardware whose board supply image distribution you have received
- The software tools and scripts received with the MaSoCist are subject to dual licensing. For the OpenSource version, you may use the tools for other projects, but you will have to publish modifications and disclose their source. If you wish to use the tools in a completely different project that can not be opensourced, you will have to acquire a separate non-Opensource license.
- If you have received the opensource eval distribution (to be recognized by the '-opensource-$VERSION' suffix of the distribution directory, all modifications you make must end up in the official distribution again
- If you received a non-opensource distribution, you are not allowed to make it OpenSource without signed consent of section5.

The dual licensing scheme does no longer apply, once a third party module using a license similar to a GNU Public License is introduced.

You are responsible for the licensing terms with regards to your end product. section5 will not take responsability or liability for license violations caused by other parties

1.5 How to read this documentation

This SoC documentation is set up in a modular way, where parts may be shared by different architectures and SoC configurations. Therefore you may find references to other SoC setups. However, the register map listed in this document applies specifically to the configuration you have obtained. Since the same configuration can run on different platforms, there are platform specific sections listed below.
There may be reference to configuration variables like CONFIG_foo. These are not explicitely documented in this documentation, rather, the kconfig tool mentioned in Section 3.1 contains built-in help functionality about each documentation variable.
For quick system specific reference, you may want to use the following short links:

- Peripheral map of this SoC: Table B.1
- Platform specific memory and pin maps: Appendix A

1.6 Memory map and register architecture

The SoC variants agathe and anselm use a 16 bit wide addressing space. Configurations starting with 'b' have an extended address range.
The address map is typically split into a memory range and memory mapped register (MMR) for peripheral configuration, plus there is a range specifically reserved for the stack.
The precise mapping is depending on the used core configuration. Please refer to Appendix A for the detailed address map of your CPU variant.

1.6.1 Addressing

When accessing a memory mapped register, the address base of the MMR range must be added within the physical addressing routines. This is also regarded as "MMR offset" below. Note that the MMR space is a fully linear address space, indirect addressing is not used.
For the C header generation, the address offset calculation for each register is taken care of in the code. It should just be noted that there are up to four offsets specified for the effective I/O address calculation of a register:

1. The **MMR offset**
2. The **register map offset**
3. An **index offset** when several units of one map are instanced (e.g. UART0, UART1, ...)
4. Finally, the **register address offset**

Detailed address generation details are covered in the generic netpp/SoC manual.

1.6.2 Register architecture

Typically, a register map is defined per peripheral unit. Per register map, a register table can be emitted, such as Table 4.2.
In this SoC design, registers have the following access properties:

RO Read only, writing to this register has no effect unless a WO register is specified for the same address

WO
Write only, reading of this register returns undefined values

RW
Read/write, instanced as a real hardware register inside the MMR map

These properties are shown in the table under the `Access` header. For the documentation, they are rendered as shown in Fig. 1.1. The MixedAccess register type is not used in this architecture. The single bits are typically marked with a code, when relevant, denoting the default value after power up, or:

x or X
Undefined, do not probe this bit for old HW revisions without checking for the HWVersion register

s Sticky status bit, clears when you write a '1' at this position ('W1C')

S Sticky status bit, clears when a device reset occurs

- Reserved bits, do not use for own purpose or alter when RW

A soft reset does not necessarily reset to the default values. See next section for details.

If the bit is not marked but named, it is:

- Not initialized in a R/W register
- Volatile in a Read-Only register

A special case is, when a RO and a WO register definition map to the same register address. This is the standard solution for a 'W1C' (Write one to clear) scenario. In this case, the WO register is not explicitely documented, rather, the 's' notation from above will be used on the RO register.

Registers also have a specific size, in this design up to the maximum possible atomic indirect addressable data width of 32 bits (4 bytes). The size of the register is listed in the `Span` property of the register table.

1.6.3 Register initialization

One important topic is the reset value initialization of registers with specified defaults. There are two implementation variants:

1. Register is initialized at FPGA boot time (default)
2. Register is initialized using a hard/soft reset (See `CONFIG_GENSOC_RESET_DEFAULTS`)

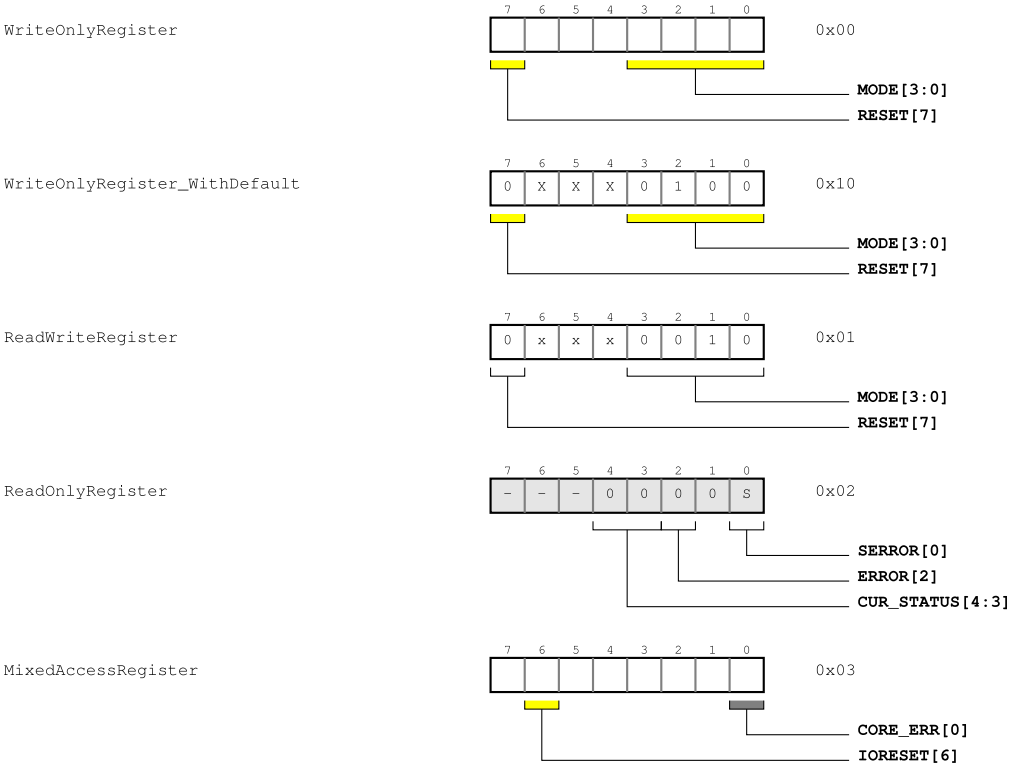

Figure 1.1: Register example legend

Depending on the FPGA architecture, the explicit reset pin initialization might be preferred. Registers that do not contain a default specification are not initialized, therefore their value can be undefined.

Generally, it is considered good programming practise to only use defaults where necessary and initialize registers within the hardware driver routines.

Application examples

One example where defaults are necessary is the SIC_IMASK register. If it is undefined, spurious interrupts may trigger an IRQ while the event vector registers are yet uninitialized. The CPU will then jump into undefined code at startup, which can be tricky to debug. Likewise, it makes sense to initialize specific hardware enable bits for crucial peripherals to zero so that they do not consume power when the system resets, for example due to a watchdog event. In this case, it may be reasonable to enable `CONFIG_GENSOC_RESET_DEFAULTS`.

1.7 Register naming and namespaces

The gensoc translator utility generates source code from the device description XML following roughly this scheme:

- In C, Registers are prefixed by 'Reg_' and the register name. For example, the `Magic` register in the SysCtrl register map becomes `Reg_Magic` when the map prefix is not enabled (default).

- Likewise in VHDL, however, registers are prefixed by 'R_' and bit field definitions by 'B_'. Bit fields are always spelled in capitals.

Automatic conversion of XML to source is not covered in this documentation.

1.8 Firmware

The bare metal software running embedded on the SoC is considered to be the 'firmware', although this term might be used for the HDL as well. The firmware is basically the main program, converted into ROM such that it is starting immediately after the FPGA is powered up.
The firmware is built during the general synthesis 'make' procedure. Prior to building, the target must be configured. This is elaborated in Section 3.1.

1.8.1 ROM generation

The Makefile in the firmware directory creates a VHDL ROM instanciation for synthesis such that the program code is run at power up time just like when started from GDB (see Section 3.3.3). The Makefile evaluates a `SIMULATION=yes` option from the command line to optionally build for simulation. If the SIMULATION flag is set for the firmware that is built into the synthesized ROM, the system will not start up correctly and likely hang.

Make sure to fully rebuild the firmware (**make clean all**) before running synthesis to be sure everything is in sync.

More details about the ROM generation in Section 3.2.3.

2

Quickstart

2.1 Prerequisites

You need a development environment. If you have signed up for a board supply package agreement, you may have received a virtual machine image (Linux container or VirtualBox image file). In this case you do not need to worry about the prerequisites, because everything has been installed "ready to go". If you have obtained the tar file distribution, the whole package may not be complete. In this case you have to make sure that the following tools are installed on your system in order to run the simulation or generate code. The tools are typically found as a package for Debian and other systems with their corresponding name.

1. Linux kernel config 'kconfig'
2. GNU full native toolchain with make, gcc, cpp, etc.
3. Target toolchain, zpu-elf-gcc, mips-elf-gcc, etc.
4. python 2.x, IntelHex module
5. xsltproc

To compile and run the simulation, the following packages are required:

1. GHDL v0.30 or greater, GHDLex v0.051 or greater
2. netpp (/usr/lib/libslave.so)
3. gtkwave

2.2 Quick board selection

The supported board supply packages, 'out of the box':

- Papilio One platform
- MachXO2-7000 Breakout board

To configure for one of these boards, change your working directory to the MaSoCist top level dir, run
make papilio_config
or
make breakout_config
Then you are ready to rebuild the source code and HDL files.
For available custom or third party configurations, see vendor/$(VENDOR)/defconfig_* or check your custom SoC documentation.

2.3 Simulation

The entire SoC is simulated by GHDL with a few extensions to support virtual interfaces. Not all boards can be simulated by default, see below.
To build the simulation, GHDL and the GHDLex library must be present. Then you execute
make sim
in the MaSoCist directory. Possibly you will have to call **make -C sim clean** after a reconfiguration.

2.3.1 Running the simulation

Once the simulation was built correctly, the resulting tb_<platform> will be created in the `sim/` directory. Some configurations are based on a virtual UART instance, see `sim/virtualuart.vhdl`. This is started using the init-pty.sh script. Then you can connect to the running simulation using a terminal program, such as minicom:

```
minicom -o -D /tmp/virtualcom
```

When running inside the linux container (LXC), no manual setup using init-pty.sh is required. The virtual COM port is set up by the LXC.

Note that this UART is a full UART emulation, so the baud rate defined by the divider value passed to the VirtualUART must match the UART peripherals baud rate, unlike the virtual console (`CONFIG_VIRTUAL_CONSOLE`), which does not depend on a baud rate.
Finally, the simulation is run by starting the tb_<platform> executable from the command line. For interactive waveform tracing using GTKwave, there is a script `run.sh`, taking the platform name as argument, e.g.

```
./run.sh papilio
```

2.3.2 Advanced interactive simulation

Enter the sim/ directory and type "make run". You should see the GTKwave window popping up, showing a slowly progressing wave display due to the enabled throttle. To disable the throttle, run the following command inside the virtual machine (or on your host)
netpp localhost TapThrottle 0
If you run netpp from your host to access the Virtual Machine or LXC (Linux Container), you need to determine the IP address of its virtual bridge interface (or what you have configured your VM with). For example, **ifconfig** on Linux may display:

```
vboxnet0  Link encap:Ethernet  HWaddr 0a:00:27:00:00:00
          inet addr:192.168.56.1  Bcast:192.168.56.255  Mask:255.255.255.0
```

Then you simply access the simulation via
netpp 192.168.56.1 TapThrottle 0

This option only applies when the simulation is configured with a virtual TAP, like `CONFIG_VTAP`

2.3.3 Vendor specific simulation issues

Simulation of some boards may require libraries that are not included in the MaSoCist, because they are vendor specific. There may be several solutions:

- Obtain necessary files from your local FPGA tool installation and create a GHDL library. Use the -P option to GHDL to specify the search path to the GHDL config file.
- Try the `CONFIG_EMULATE_PLATFORM_IP` option
- Use a virtual board config that is fully vendor IP independent

For example, when you receive an error message like this:
`../hdl/plat/breakout_top.vhdl:16:9:  cannot find resource library ''machxo2''`
You have to create a file lattice/machxo2-obj93.cf somewhere using the rule:

```
MACHXO2_VHDL = $(wildcard $(LATTICE_SIM)/machxo2/src/*.vhd)
lattice/machxo2-obj93.cf: $(MACHXO2_VHDL)
    [ -e lattice ] || mkdir lattice
    ghdl -i --workdir=lattice --work=machxo2 $(MACHXO2_VHDL)
```

where LATTICE_SIM is the directory of your simulation VHDL files, like
`/usr/local/diamond/3.1_x64/cae_library/simulation/vhdl/`
Then set the LIBGHDL variable in `vendor/default/local_config.mk` to the directory where you created `lattice/machxo2-obj93.cf`, like:

```
LIBGHDL = $(HOME)/src/vhdl/lib/ghdl
```

For full 'model in the loop' simulation, it will be necessary to acquire an additional, non-free Simulation package by section5

2.4 Prepare synthesis

If you have configured something using the menuconfig utility, run either "make syn" from the top level or "make" inside the `syn/` directory to update your files. Then run your synthesis tool and open up the corresponding project file in syn/<FPGA_VENDOR>/<PLATFORM>, for example:

breakout
syn/lattice/breakout/breakout-opensource.ldf

papilio
syn/xilinx/papilio/zpu/zpu-opensource.xise

If the project files are not present for your platform, see below on how to quickly import the files required for synthesis.

2.4.1 Porting to new platform

When synthesizing for a new FPGA platform, you will have to create a new project first. The MaSoCist environment helps you with importing all the necessary files by creating TCL scripts for the supported synthesis tools (Xilinx ISE 13.4, Lattice Diamond 3.2 at this time).

1. Create the new project in syn/<fpga_vendor>/<platform_name>/<project_name>
2. To import the project files, open a TCL shell inside your IDE and run source ../../proj_<platform>.tcl
3. Run Synthesis to check if all files referenced are imported

FPGA platform specific IP cores may have to be manually imported into the project

2.5 Configure and build

For usage of the kconfig environment, see SoC specific documentation.
Basically, you execute the kconfig menu by running
make menuconfig
from the top level MaSoCist directory.
Before synthesis or simulation, it is required to rebuild the corresponding system files:
make syn

In the evaluation version (no gensoc included), the hardware peripherals can not be configured. You can only choose among supplied board configurations in `CONFIG_SOCDESC`.

2.6 Target download

2.6.1 Papilio

First, you have to set the `XILINX_ISE_DIR` variable in order to assemble the full firmware image for the SPI flash.

- Define XILINX_ISE_DIR in vendor/opensource/local_config.mk
- Export XILINX_ISE_DIR in .bashrc or alternative shell startup file

Example:

```
XILINX_ISE_DIR = /media/sandbox/Xilinx/13.4/ISE_DS
```

The Xilinx tools allow to recompile the software without the full ROM synthetization. See `syn/xilinx/papilio/Makefile` rules:

$(PLATFORM)_fw.bit:
Build rule to merge program data with existing firmware bit file

download:
Downloads bare SRAM image into target

flash:
Merges extended memory (cacheable) space into overlay SPI flash image and programs it into the target

For download to the target, the **papilio-prog** application is required. See Papilio homepage for details.
The full firmware download is run by **make flash**. If no SPI code cache and program overlay is used, you can run **make download** instead. Otherwise, you have to make sure that all addresses are in sync.

2.6.2 MachXO2 Breakout

The MachXO2 variant requires a full synthetization of the generated HDL when the boot rom was altered. For download to the target, please use the Lattice Diamond Programmer.
Project files for target download exist:

breakout.xcf
Download into SRAM (volatile)

flash.xcf
Flash permanently onto the target

2.6.3 Other boards

All other boards such as proprietary / custom development can be programmed using an ICEbearPlus adapter, if the JTAG pins are accessible. Some boards may have embedded USB JTAG controllers. See Table 2.1 for details.

Board name	Programming	JTAG adapter
HDR60	Diamond Programmer	Integrated USB JTAG
versa ECP5 eval kit	Diamond Programmer	Integrated USB JTAG
gözcü	xc3sprog	ICEbearPlus
EFM01	Proprietary Cesys tool	ICEbearPlus (debug only)
Digilent Spartan-3 Board	xc3sprog	ICEbearPlus
denver2/3	xc3sprog	Embedded ICEbear

Table 2.1: Programming method

The Impact Cableserver for ICEbear is no longer supported.

3 Software

3.1 Configure and build

The configure system is borrowed from the Linux kernel project and uses a `.config` file for the target configuration. The simulation build system fully relies on Gnu Make. Synthesis is partially (for specific platforms) supported by Makefiles as well.
The Makefile structure of the SoC project is somewhat modular and described in more detail below.

3.1.1 Kconfig

The target configuration is done like in the Linux project, however, the kconfig frontend package needs to be installed. Then, the target can be configured by running the command
make menuconfig
from the top level MaSoCist directory.
When the kconfig package is set up correctly, a dialog should display as shown in Fig. 3.1.
The configuration is saved in the file `.config` (again in the top level MaSoCist directory). This is a simple text file with typical configuration statements like CONFIG_FOO=y. To verify a configuration variable, the text file can be opened directly. It is also possible to manually edit the file and change variables, however care needs to be taken about the syntax and possile configuration variable dependencies. kconfig is able to fix most mistakes, so when the configuration turns out to be invalid, simply run menuconfig again.

Default configurations

The vendor specific directories (vendor/$VENDOR) contain default configurations for one or more board supply packages, recognized by the prefix `defconfig_`. To use this configuration, which is basically copying them over the local .config, type "**make <board>_config**" where <board> is among the supplied defconfig_<board> files.
To store a valid, changed configuration, type **make defconfig**. The make rule script will check for changes using the diff tool and ask you if you want to overwrite the previous configuration. When denying the overwrite, it will call an external diff tool (default: meld) to interactively display the changes and let you edit them.
Note that the defconfig rule will choose the default board name to generate a defconfig_board file in your default vendor directory. If you have different variants for one board, use the `CONFIG_PLAT_EXTENSION` variable to name your specific configuration variant.

3.1.2 Makefiles

Every subdirectory that contains configureable modules typically contains a Makefile which is included from a superiour Makefile. Plus, several auxiliary make files with the *.mk suffix are included from the top level build Makefiles.
The list of supporting Makefiles:

config.mk
: Contains system environment specific settings. Edit this file to configure your toolchain paths.

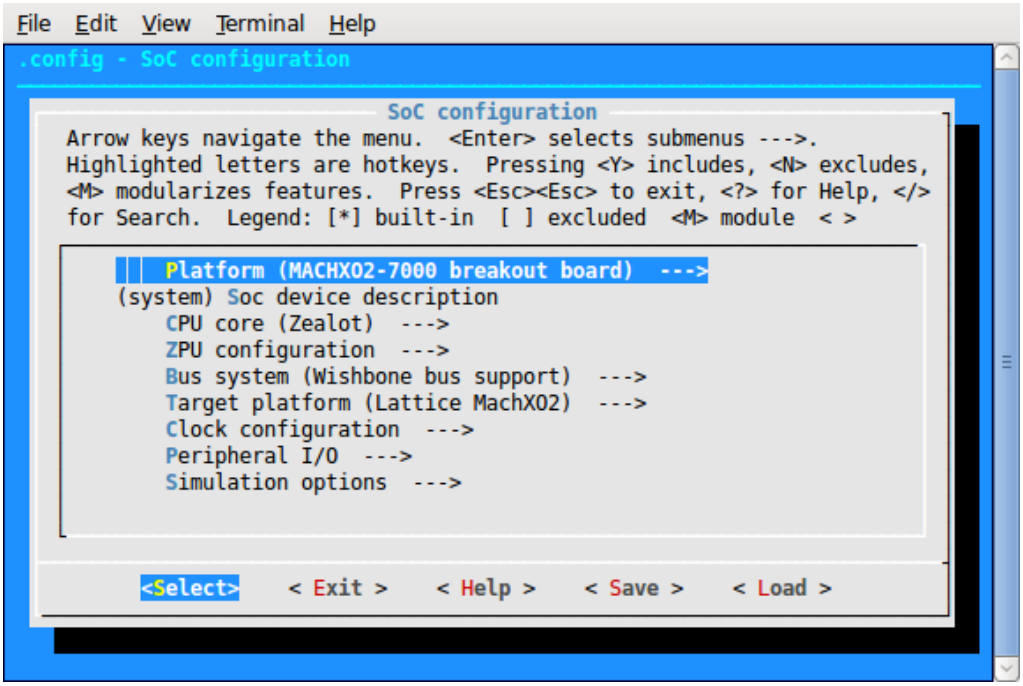

Figure 3.1: Typical menuconfig dialog

platform.mk
: Contains platform specific configuration settings

gensoc.mk
: Gensoc specific build rules

generate.mk
: All rules for VHDL file generation

hdl/vhdlconfig.mk
: Generates a VHDL configuration package from the Kconfig configuration

hdl/core/zpu.mk
: ZPU core specific build rules

sim/libsim.mk
: Simulation library specific build rules

syn/lattice/diamond_proj.mk
: Makefile to create TCL helper scripts to import files to a Diamond project

syn/xilinx/ise_proj.mk
: Makefile to create TCL helper scripts to import files to a ISE project

Some build processes depend on external tools whose location must be customized to the local machine settings. These settings are typically made in `vendor/$VENDOR/local_config.mk`. For example, to configure the diff tool to be the 'meld' application, you'll find a line like

```
DIFF = meld
```

Typical make rules

A make rule defines how things should be built and are executed by "**make** <rule>" They should be only executed from the top level or inside the `sim/` or `syn/` directory. All other Makefiles, especially those in lower directory hierarchies, require environment settings.

sim
: Builds the top level simulation entity

syn
: For supported platforms, the synthesis is run and a bit file is typically generated. If automated build is not supported, a TCL script is generated to ease the import into an existing project.

run
: The simulation is run for a certain time. See `sim/Makefile` for the exact parameters and output files.

3.1.3 C preprocessing

The C preprocessor **cpp** is used for conditional code generation according to the CONFIG variables generated by the kconfig tool. This is also used for some VHDL files. The source files are given the .chdl extension. If you edit the generated VHDL files, keep in mind the changes will not be permanent, as they will be overwritten by the next build procedure. A warning in the file header will typically remind you again of this detail.

3.1.4 Large programs - function overlay

If the `CONFIG_SCACHE_INSN` option is present in the board supply package, larger programs than the actual block ram size allows can be created. Depending on the board configuration, functions may be placed into the extended program area by default, or a special attribute might be required to decorate the function. See also Section 4.2.2.
Which of these applies, is easily determined by the first letter of the board supply package name. If it begins with 'a', then you will need to use the decorator attribute. If it begins with 'c' or later, the linker script will put all non-decorated program code into extended memory. For details, please look at the generated map files or linker script of your board supply package. Also, the SoC specific section might describe implementation specific details.

Up to the ZPUng v1 architecture, program code in extended memory can not be single step debugged using ICE. When single stepping inside exception code, the debugger may freeze.

Linker scripting

Where a program section is placed, is determined by the linker and a memory map specified in the linker script that is specific to each board supply package and memory configuration.
By default, code ends up in the `.text` segment. If the special attribute define `EXTERN_PROG` is used, code is placed in the `.ext.text` section. The linker script snippet below defines the placement of functions from this segment in the `xprog` section. Likewise, read-only data that is no program code is allocated into the `xdata` segment.

```
  /* external cached data area: */
  .ext.text           :
  {
    _ext_program_memory = .;
    KEEP (*(.ext.text))
  } > xprog

  . = 0x10000;
  .ext.rodata         :
  {
    _ext_data_memory = .;
    /* Put bitmap files into ext ROM */
    bm*.o(.rodata)
    *(.ext.rodata)
  } > xdata
```

The sections are typically defined in a board layout specific separate linker script that is included or in the header of the active linker script. The section memory layout must specifically match the memory map of the SoC, as shown in the SoC specific memory map Section A.1. If the cache is not enabled at all, the `l1cache` area can be used for normal program/data storage. The snippet below shows an example MEMORY specification.

```
MEMORY
{
  l1ram(rwx): ORIGIN = 0x0000, LENGTH = 0x2000
  l1cache(rwx): ORIGIN = 0x2000, LENGTH = 0x2000
  xprog(r): ORIGIN = 0x08000, LENGTH = 0x8000
  xdata(r): ORIGIN = 0x20000, LENGTH = 0x20000
}
```

Program layout and timing issues

Note that loading from SPI flash takes some time. It is therefore recommended, to make sure that frequently used code is not spread across different pages, unless code in l1ram is called.

Currently, functions can not cross a page boundary. Make sure your functions are completely inside one cache page!

To examine the precise memory layout of the program, a map file is typically output during compilation. More advanced linker scripts may assert that no code inside a function can overlap the page boundaries.
Also important to know is that interrupt routines must not be in extended program memory, since the exception handlers are NON-reentrant. Even if reentrancy would be implemented, it would massively slow down execution due to frequent external memory access.
During an exception handler, no other interrupts are allowed from ZPUng v1.1 on, unlike v1.0. This change was necessary to address safety issues with repeated spurious interrupts that could under certain conditions cause unwanted reentrancy.

3.2 Firmware

The software running on the FPGA is named ‘firmware’ in this reference. It is cross compiled using the target toolchain within the general HDL build process.

3.2.1 Shell

The standard board supply packages are featured by a simple UART shell and a few test functions. Users wishing to test their hardware on a low level can do so easily by extending the `exec_cmd()` function of their board supply test code (naming scheme `test_$PLATFORM.c`)

3.2.2 Register access

The firmware source code is written in C using a GNU cross compiler toolchain. Direct register access is supported through a few macros by using generated register definitions (`soc_register.h`). Programmers who wish to write their own drivers are encouraged to include the mid level auxiliary header `driver.h` and access registers as follows:

```
#include "driver.h"

...
// Set GPIO port 0 bits (14..15) to out:
MMR(Reg_GPIO_GPIO_OUT) = 0xc000;
```

Note that some CPU cores do not care about the effective register width, plus big endian (default) and little endian aspects become relevant. On the default MaSoCist SoC distribution, all registers are accessed using the MMR() macro. Other SoC variants using the PyPS core may use macros of the sort MMR16(), etc.
When accessing a register of width < 32 bits, the upper bits are undefined and can be garbage.

The most effective code is produced on the Zealot when accessing the bus using a 32 bit wide access. If you assign the result of an MMR query to a variable with higher width than the actual register you were reading, you have to mask out the undefined MSBs before comparison!

3.2.3 ROM file generation

The ROM builder utility `utils/buildrom.py` is a Python script converting from an ELF format into a set of VHDL files that can be used to generate a preinitialized DRAM emulating a Boot ROM. It currently supports ZPU and MIPS architectures. It relies on an open source elf.py module, also included in the utils directory. The generation rules are found in generate.mk. buildrom.py relies on the external 'intelhex' python module. You may have to obtain this module separately.
The Makefile will build two files from your program code:

rom_generated.vhd
: The built in boot loader code, used to initialized the block RAM

flashdata.bin
: The overlay program and data for the SPI flash user partition

The synthesis tool will typically build a `.bit` file from your design. An extra script will append the binary image `flashdata.bin` to this bit file which will be used to program the flash. There are also options to use in-system upgrade functionality for a user program. See SoC specific section Appendix A whether these options are present.

3.3 gdb - the GNU debugger

The following sections describe how to access the SoC ZPU core through the GNU debugger for non-intrusive debugging (to the most possible extent on this architecture).

3.3.1 uniproxy - the SoC debug agent

The uniproxy debug server (former gdbproxy/rproxy) is a small debug agent to allow remote debugging of a target featured by StdTAP/JTAG. It listens on a TCP port (standard: 2000) for incoming connections from a GNU debugger. It is making use of the uniemu TAP library that supports several processor architectures through a uniform test access port implementation (StdTAP IP core). It also supports various JTAG TAPs of different FPGA manufacturers. It is typically started from a linux console as follows:
gdbproxy zpu
and responds with the following output upon successful detection of the SoC:

```
Remote proxy for GDB, v0.9.302, Copyright (C) 1999 Quality Quorum Inc.
Blackfin target adaption (C) 2005-2012 and
StdTap generic interface [ZPU, MIPS, plm32, FLiX]
            (c) 2013 by www.section5.ch

notice:    Found FTDI unbranded adapter device
notice:    Setting clock to 6 MHz (wait cycle: 0)
IR size: 8
Trying Lattice ECP3 35EA, got id c2048080
Found Lattice ECP3 35EA
Got user id (cmd 17): f77db57b, reverse: deadbeef
notice:    Detected 1 device(s)
notice:    Selecting CPU 0
notice:    gdbproxy: waiting on TCP port 2000
```

Now, connection to the target SoC can be made.

From the v1.0 package release on, the proxy is called 'uniproxy' to avoid conflicts with other gdbproxy distributions

If the device is not found, you might have to use a different port using the `--port` option.

Virtual gdbproxy

As part of a Co-Simulation extension to the SoC project, a special proxy version is supplied to connect to the running simulation via a virtual JTAG adapter. This allows to live-debug the simulated CPU and SoC over gdb.

3.3.2 Connecting to the Target

Start GDB using the command zpu-elf-gdb and provide the firmware executable of the SoC as argument, for example:
zpu-elf-gdb main
Then make a target connection using
(gdb) **target remote :2000**
Once connected, gdb shows the current position the program was interrupted in, for example:

```
0x0000030b in _neqbranch ()
(gdb)
```

The program can then be resumed using the **continue** command and interrupted again by using **Ctrl-C**.

3.3.3 Program download

To load the program onto the target at runtime, it is important to know that the SoC core is a stack machine and requires a valid stack pointer address in order to operate correctly by In Circuit Emulation. Therefore the core must be reset using **monitor reset** before download. It is no longer required to explicitely set the stack pointer.

```
define init
        monitor reset
        load
end
```

To load the current main program, only type **init**:

```
Loading section .fixed_vectors, size 0x400 lma 0x0
Loading section .init, size 0x7 lma 0x400
Loading section .text, size 0x16e8 lma 0x407
...
Start address 0x0, load size 8857
Transfer rate: 11809 bits/sec, 98 bytes/write.
```

For all other gdb commands, please type **help** or use the various gdb documentation resources from the internet.

monitor reset may not fully reset all logic, depending on your peripheral configuration. If peripheral logic prevents the CPU from booting correctly, like for example interrupts, you may have implement further reset scripts to restore default register values. See Section 1.6.3 for specific reset initialization options.

When downloading a large program using function overlays the jump addresses may have changed. Therefore, functions that call code from non-updated ROM will likely crash or produce unexpected results. When testing programs via RAM downloads, make sure your ROM is up to date or not called (CONFIG_SCACHE_INSN disabled). Another possibility is to avoid relative jump addresses by using the `--mno-callpcrel` flag during compilation (`CFLAGS` variable in the Makefile).
If the startup procedure of your program code is not depending on any global variables that can be altered, you can use **monitor reset** to restart. If your program may have been altered however, be it a malfunction or a feature, you will have to reload it into the block RAM.

3.3.4 Debugging and interactive register dumps

Native microcode debugging

The ZPU small core provides an emulation feature for a few assembly instructions that are not implemented in hardware. Therefore, when interrupting the program, the CPU may be stuck in the emulation routines ('native microcode') outside the firmware program. In this case, the backtrace (**bt**) may not show the correct stack trace. To return to a CPU state where the trace is valid, use the **next** command (or just **n**) to continue until exit:

```
(gdb) n
Single stepping until exit from function _neqbranch,
which has no line number information.
0x00000577 in delay (i=32620) at main.c:59
59            while (j--) { asm("nop"); }
```

It is recommended to use breakpoints as far as possible to stop the program at specific points. If fully atomic assembly debugging of a program is required, there are several options:

- Use ZPU variant with full hardware instruction implementation
- Use ZPUng with emulation masking option (EMUMASK bit set, this is the default operation). This feature is not documented.

On the ZPUng, non-native microcode (outside the 0x0-0x400 area) can not be debugged.

The uniproxy debug agent automatically detects the used CPU type via JTAG, thus no further configuration by the user should be required.

Local variables

Typically, when breaking inside a function, it will display the local variables. On the ZPU/Zealot default platform, these variables may not be correct or not be displayed at all if optimization flags are used or if the `--speed=0` option is given to uniproxy (disables pseudo register stack reads). For critical applications that require repeated dumps of variables, it is recommended to use a global variable in memory. The reason for this behaviour is that the ZPU architecture is a stack machine and full context tracking is limited on the current GCC toolchain.

Interactive register inspection

For interactive register inspection, gensoc creates a GDB auxiliary script `soc_mmr.gdb` for the entire SoC register map. This provides a quick method to dump the bit fields or the value of a register.
For example, a verbose dump of the IRQ SIC_ILAT register is executed by the following command sequence:

```
(gdb) source soc_mmr.gdb
(gdb) dump_irq_sic_ilat 0

Address: 00008304 SIC_ILAT: 00000080
```

The '0' given after the dump call is the device index, in case there are several instances of a device controller. Single device units do not require the '0' on single instance devices.

The script does not explicitely check the bit width of the register being dumped. The MSBs may contain garbage when accessing registers with width < 32 bits.

The `.gdbinit` startup file of the firmware (see `sw/` folder) contains various debug scripts to obtain the current register status. Likewise, manual configuration of hardware registers can take place. For example, the following code sets a GPIO output:

```
source soc_mmr.gdb

define initLED
    set *$Reg_GPIO_GPIO_OUT = 0x8000
    set *$Reg_GPIO_GPIO_DIR = 0x8000
end
```

4

Peripheral modules

The SoC peripherals are mapped automatically from the device description. Several instances of peripheral modules, such as GPIO controller banks, can be mapped into the MMR address space according to the addressing scheme defined in a specific unit map. For register reference, the absolute address of a register is calculated as the sum of the following values:

- A registermap offset per device unit, listed in Table B.1.
- A register offset, listed in the peripheral address map

For external Wishbone peripheral IP that is not auto-generated from the device description, an address relative to the MMR_OFFSET range applies, thus requiring none of the above calculations.
The register map reference is valid only for the following hardware revision of the SoC device: *Device Revision: 0.4[alpha]*

Use the 'sysinfo' GDB script to obtain the current HW revision from the SoC (HWVersion register)

Each component has some software support, or optional regression test scripts to test the interface. See Table 4.1 for an overview on source files corresponding to a peripheral.

Description	C source	Regtest script
General tests and board initialization	test_$(PLATFORM).c	-
System interrupt controller	irqhandler.c	sw/regtest-common/sic.gdb
Timer and PWM	pwm.c	sw/regtest-common/pwm.gdb
Simple UART	uart.c	sw/regtest-common/uart.gdb
16550 UART	uart16550.c	-
Opencores I2C controller	i2c_oc.c	-
LCDIO	lcdio.c	sw/lcdio.gdb

Table 4.1: Peripheral module software support

4.1 System control

The system control section covers the platform specific configuration properties. To synchronize the Firmware with the hardware, the `HWVersion` register should be queried. Typically, this revision code is generated from the system device description XML. One might note that the SysCtrl map follows a different naming scheme than the other agathe peripherals. The reason for this is, that the SysCtrl unit is specific to the system architecture, whereas the basic peripherals are per se system agnostic. For example, if a system supports I/O multiplexing (GPIO and alternate functions) or cache functionality, the SysCtrl unit becomes relevant.

All MaSoCist SoC instances must have at least a SysCtrl unit with the XML id 'sys'.

More complex systems may have a separate I/O multiplexing unit.

Offset [Span]	Name(Id)	Access	Description
0x00 [1]	SysCtrl	RW	System control register
0x04 [2]	CRC16_ValueInit	WO	WriteOnly initialization CRC16 value
0x04 [2]	CRC16_Value	RO	Current CRC16 value
0x08 [1]	CRC16_Data	WO	Writing a 8 bit value to this register updates the CRC16 value
0x34 [4]	Magic	RO	Magic number (typically CPU ID). Used for detection.
0x38 [2]	SocInfo	RO	System type identification field
0x3c [4]	HWVersion	RO	Hardware revision word. Reserved for Simulation: 0xff01

Table 4.2: Address map SysCtrl

In order to facilitate downward compatibility with respect to firmware, it is mandatory for the HWVersion register to remain at the end of the register map. Since the address bits are masked according to the MMR_CFG_SysCtrl definition in the system map VHDL file, it is ensured that queries with higher MSBs set (that are masked out on the older HW revision) still receive the correct HWVersion.

4.1.1 Clock configuration

The agathe and beatrix configurations have a simple clock scheme for the peripherals. The main system clock is simply divided by a divisor value to obtain the clock for the peripheral. The system clock itself depends on the FPGA hardware and oscillator frequencies used. Some platforms do have a PLL, multiplying the external master clock by a fractional value. The configuration of clock frequencies and dividers is done by the menu configuration. The formula to obtain the system clock (see also `sw/driver.h`):

$$SYSCLK = CLOCK\_FREQUENCY * \frac{CONFIG\_PLL\_MUL}{CONFIG\_PLL\_DIV} \tag{4.1}$$

You can output the current system clock in Hz within the GNU debugger by **print g_sysclk** when using the standard test firmware.

For platforms without PLL and missing CONFIG_PLL variables you can assume DIV and MUL = 1. Some FPGAs have internal oscillators whose frequencies may not be arbitrary. These have their own CONFIG variables. Handle with care, do not reconfigure those clocks freely or your system may stop to operate correctly.
The software library support coming with your board supply package typically takes care of the clock configuration.

4.2 SCache - simple software cache

The SCache core is enabled by CONFIG_SCACHE=y. It is only effective on the ZPUng architecture and provides a simple exception mechanism to implement data caches for systems with limited

block ram.
Basically, it detects whether an addressing attempt is made to a I/O space which is not inside the peripheral MMR section. On access (read only), it raises an exception on a LOAD instruction and jumps into a cache handler in uCodespace. This handler can then execute necessary DMA or SPI flash loading functions, copying the requested data page into the cache memory.
The cacheable memory segments are organized in pages of a specific size. On access of cacheable virtual memory, the full page is copied from external memory into the cache area. Note that there may be access restrictions imposed by the CPU core that may, for example, prohibit overlaps of function code to page boundaries. See implementation notes below.

Offset [Span]	Name(Id)	Access	Description
0x00 [1]	CacheControl	WO	Cache control register
0x00 [1]	CacheStatus	RO	Cache status register
0x04 [2]	DCacheOffset	RW	Cache address offset for data cache page
0x08 [2]	ICacheOffset	RW	Cache address offset for instruction cache page
0x10 [3]	DCacheMask	RW	Mask for address evaluation (AND combination)
0x14 [3]	DCachePageMask	RW	Data cache page mask [DEVELOPMENT]. May disappear in future.
0x18 [3]	DCacheAddr	RW	Address to compare with after masking with CacheMask
0x1c [3]	DCacheHitAddr	RO	Stored address of addressing attempt causing a miss
0x20 [3]	ICacheMask	RW	Mask for address evaluation (AND combination)
0x24 [3]	ICachePageMask	RW	Instruction cache page mask [DEVELOPMENT]. May disappear in future.
0x28 [3]	ICacheAddr	RW	Address to compare with after masking with CacheMask
0x2c [3]	ICacheHitAddr	RO	Stored address of addressing attempt causing a miss

Table 4.3: Address map SCACHE

4.2.1 Setup

The SCache uses the exception channel of the SIC. The corresponding SIC_EV0 vector is reserved and typically initialized by the board support package init routines (crt0.o). The code below demonstrates a typical setup sequence:

```
// Trigger a cache exception when Pages 0x10xxx-0x17xxx are hit:
MMR(Reg_DCacheMask)     = 0x038000;
MMR(Reg_DCachePageMask) = 0xfff000;
MMR(Reg_DCacheAddr)     = 0x010000;
MMR(Reg_SIC_EV0)        = (uint32_t) &cache_exc_handler;
// Enable cache address translation
MMR(Reg_CacheControl)   = DCACHE_ENABLE;
```

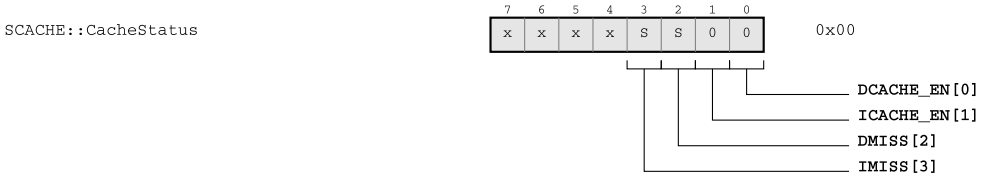

Figure 4.1: CacheStatus

On the ZPUng v0, the `cache_exc_handler()` function must reside in uCodespace to operate correctly

An access to 0x11600, for example, will trigger the exception, once the cache is enabled through the DCACHE_ENABLE bit. The exception address is AND masked with the value from `DCachePageMask` AND the value from `DCacheMask`, the result is placed in `DCacheHitAddr`. Reading this register yields the page base address, in this example, 0x01000. The cache handler will typically use this page address to read from an external memory, like demonstrated in the following example code implementing a SPI flash cache:

```
void cache_handler(void)
{
    uint32_t addr;
    // address of cache memory:
    uint32_t *cacheaddr = (uint32_t *) CACHE_PHYS_ADDR;

    // Set DACK bit to clear exception flag (important!)
    MMR(Reg_CacheControl) = DACK | DCACHE_ENABLE;

    // Do some math to translate into SPI flash offset:
    addr = USER_DATA_OFFSET;
    addr += MMR(Reg_DCacheHitAddr); // Get page address
    addr += _start_xdata;           // See linker script

    // Read data from SPI flash:
    spiflash_read32(addr, cacheaddr, CACHE_SIZE >> 2);
}
```

4.2.2 Instruction caching

When `CONFIG_SCACHE_INSN` is enabled, a second SCACHE instance can be used for instruction caching. When enabled via ICACHE_ENABLE, half of the SCACHE RAM area is used for the instruction cache (ICACHE). In this case, the exception handler must check for the IMISS and DMISS bits of the CacheStatus register to determine which cache triggered an exception. The first half of the cache is reserved for data access, the second for instruction access. If the ICACHE is not enabled, the full cache area can be used for the data cache.
The default board supply code automatically determines the size of the caches, depending on the `CONFIG_SCACHE_INSN` configuration.

Implementation notes

When the internal block ram is filled up with program code, functions can be overlayed to the extended program area where the ICACHE becomes effective. Whenever a jump into the

extended area is made, the virtual address will trigger an exception, the program loader will load the code from SPI flash and the CPU will fetch the instructions from the cache area using a physical address translated by the ICACHE.
For detailed information about program design and function overlays, see Section 3.1.4.
Note that the instruction and data cache areas are separated memory areas by default. If `CONFIG_SCACHE_HARDCODE` is not enabled, the following rules apply:

1. Program code (`.text` segment) can not be allocated in a cacheable data page
2. Data (`.rodata.*` segments) can not be allocated in a cacheable instruction page

This results in data or instruction accesses to the wrong cache page returning false results without warning! This also produces false disassembly output of code in virtual instruction memory.

You can disassemble the overlay function code anyway by examining the raw instruction cache physical memory (see Section A.1)

With `CONFIG_SCACHE_HARDCODE` enabled, CachePageMask and CacheOffset register settings are ignored for both instruction and data cache area. In this case, there can be mixed DATA access from both instruction and data cache areas.

DATA access to an ICACHE page does not trigger an exception in the typical standard design. If this is wanted, the DCacheMask register must be set accordingly

4.2.3 User code

The SCache function is somewhat generic, therefore it can be used to watch accesses to a specific address space. Likewise, it can be configured for a single hardware breakpoint unit, when `CONFIG_SCACHE_HARDCODE` is not effective. This is up to the user to implement.

4.3 Interrupt handling

4.3.1 System interrupt controller

The system interrupt controller (SIC) is a very simple IRQ controller handling up to four channels using the registers listed in Table 4.4
Up to four interrupt sources can be mapped to the channels with a simple prioritized IRQ vector generation.

SICv2 exceptions

The v2 of the SIC has an extra NMI pin plus an EV0 exception vector register. When the NMI pin is asserted, this overrides all pending interrupts and sends an interrupt signal to the CPU. No other interrupts can trigger an event until the condition causing the NMI to be set is cleared. The cause of the exception is not known to the SIC. Instead, it is up to the CPU designer to implement exception cause registers in the SysCtrl map.
Since exceptions are non-reentrant, the programmer must ensure that no other exception occurs inside an exception handler. Likewise, it can be problematic if exceptions occur inside interrupt routines that can interrupt a running exception handler. If the exception handler clears the exception cause too early, interrupts causing another exception will likely trash the stack.

Offset [Span]	Name(Id)	Access	Description
0x00 [1]	SIC_IMASK	RW	System interrupt mask. When '1', interrupt is enabled
0x04 [1]	SIC_ILAT	RO	IRQ latch register. When IRQ was just received, the corresponding bit is set
0x04 [1]	SIC_ILAT_W1C	WO	When IRQ is latched, clear the corresponding ILAT flag by writing a '1' to this register
0x08 [1]	SIC_IPEND	RO	IRQ pending register. When enabled in SIC_IMASK, a latched IRQ is marked '1' in this register when pending.
0x08 [1]	SIC_IPEND_W1C	WO	When IRQ is pending, clear the corresponding IPEND flag by writing a '1' to this register
0x0c [2]	SIC_IAR	RW	Interrupt assignment register
0x10 [2]	SIC_IV0	RW	Interrupt vector 0
0x14 [2]	SIC_IV1	RW	Interrupt vector 1
0x18 [2]	SIC_IV2	RW	Interrupt vector 2
0x1c [2]	SIC_IV3	RW	Interrupt vector 3
0x20 [2]	SIC_EV0	RW	Exception Vector 0
0x24 [2]	SIC_MISS	RO	Interrupt MISS register
0x24 [2]	SIC_MISS_W1C	WO	Interrupt MISS register writeonly write-one-to-clear

Table 4.4: Address map IRQ

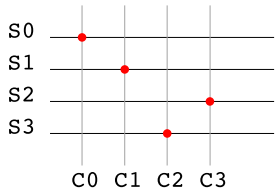

Figure 4.3: Channel assignment scheme

The ZPUng v1.1 prohibits stack trashing within an exception handler and does not allow reentrancy until a IUNLK opcode is executed.

Interrupt handling

When a '1' level occurs on an interrupt source, the corresponding bit in the SIC_ILAT register is set. The corresponding bit in the SIC_IMASK register determines, whether the interrupt triggers an IRQ request to the core. If the SIC_IMASK bit is set, the corresponding SIC_ILAT bit will be latched into SIC_IPEND and the CPU will receive a pulse on the IRQ line. The interrupt vector is generated by the SIC according to the SIC_IAR register.
By default, each interrupt source [0..n-1] is assigned to the corresponding channel [0..n-1] where n is the maximum number of channels (four in this fixed implementation). Channels with lower numbers have higher priority. The channel assignment can be changed by setting the corresponding bit field in the SIC_IAR register.
The user must initialize the interrupt vectors in SIC_IVx to the IRQ handlers before enabling the interrupts using the SIC_IMASK register.

Channel assignment

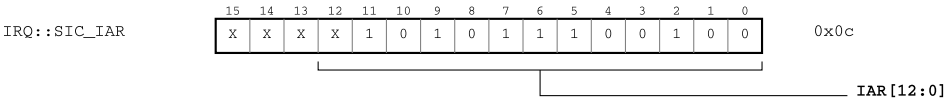

Figure 4.2: SIC_IAR register

Fig. 4.3 shows the connection scheme used by the SIC_IAR register. S[0..3] denote the IRQ sources, C[0..3] the IRQ channels. If S0 is supposed to trigger an IRQ on lowest priority C3, a value of 3 needs to be written to the SIC_IRQCH0_SEL bit field. This mapping is effective after all latching and masking logic.
If several interrupt sources are assigned to the same channel, they are technically ORed. The developer needs to determine the event source according to the SIC_ILAT register in the irq handler corresponding to the assigned channel.

When using a Zealot CPU core, the SIC_IVx settings will be ignored, due to only one global interrupt routine support (`zpu_interrupt()`). The zpu_interrupt routine will have to determine the IRQ cause by checking the SIC_IPEND register.

How the interrupt sources are mapped to the peripherals, is listed in the SoC specific section Appendix A.

Keep in mind that the IMASK/IPEND/ILAT register represents the channel assignment, not the priority assignment (when you have changed the order using SIC_IAR).

Interrupt priorities

Interrupt priorities are sorted out by the System Interrupt Controller (SIC). By default, all SIC implementations handle lower channel numbers with higher priority such that:

- Simultaneous events on two channels will trigger the higher priority event and queue the lower priority request
- Higher priority events during a low prio service will cause an override action (more below)
- Lower priority events while another interrupt is in service will queue and execute after the current interrupt handler exits

The SIC will assert the override signal if it detects an event with higher priority than the one currently latched. The override signal clears the internal IRQ flag in the ZPUng core and therefore allows reentrancy. It may happen, that low priority interrupt requests during the high prio service are lost. If an interrupt of a specific channel had occured within a service routine can be determined by the SIC_IPEND register. It is therefore recommended to clear the corresponding SIC_IPEND_W1C flag at the **beginning** of the routine, if repeated IRQs during a service are expected.

Interrupt requests **can** get lost if several IRQ pulses occur during the service handler (with SIC_IPEND flag still set). If your software relies on a correct number of pulses, make sure to use peripheral controllers with hardware counters

When using the default SIC_IAR settings, IRQ priorities match the channel order [0..n-1].

4.3.2 Interrupt source mapping

Currently, only four interrupt channels are used. The mapping is platform specific, see Appendix A.

4.3.3 Core specific interrupt handling

This section applies to the 'ZPUng' CPU architecture only

Typically, an IRQ routine is a just a normal C subroutine with no specific interrupt attributes. However, this only is valid when no memreg instanciations are used inside the IRQ handler. If this is the case, restore() and save() macros need to be called inside the interrupt routine. Whenever an interrupt is serviced from the handler, it must signal to the SIC by clearing the IPEND bit to the corresponding IRQ channel that the interrupt logic can rearm for the next event to happen on this channel.
The ZPU legacy architecture does not have a protection against reentrancy of an interrupt handler on recurring events, so too frequent interrupt events could trash the stack. The ZPUng logic prevents this by internal usage of an IRQ flag. While this flag is set, no other IRQ source can interrupt a current handler. The flag is always cleared at the end of the interrupt handler, more details below.

Interrupt priorities

Interrupt priorities are sorted out by the System Interrupt Controller (SIC). By default, all SIC implementations handle lower channel numbers with higher priority such that:

- Simultaneous events on two channels will trigger the higher priority event only
- Higher priority events will cause an override action (more below)
- Lower priority events while another interrupt is in service will queue and execute after the current interrupt handler exits

The SIC will assert the override signal if it detects an event with higher priority than the one currently latched. The override signal clears the internal IRQ flag in the ZPUng core and therefore allows reentrancy.

Exceptions

Exceptions are also handled like an interrupt in the ZPUng, however, they only occur on special occasions like memory accesses. The ZPUng v1 only covers memory access exceptions via the SCache system peripheral. An exception can be assigned to any property, however a higher priority IRQ can interrupt the exception handler **only after** the cause for the exception has been cleared. If the exception cause clearing is forgotten inside the exception handler, the system will hang on the next memory access.

Writing handlers

Interrupt routines on systems that allow interrupt handler reentrancy must clear the internal interrupt flag before the IRQ routine is left. On the ZPUng v1, the IRQ_REARM() macro must be called at the end of the irq handler. Below is a standard example with the v0.1 toolchain release.

```
void irq_generic_handler(void)
{
    MMR(Reg_SIC_IPEND_W1C) =  PINMAP_IRQ_GPIO_A; // clear IRQ and rearm
    g_counter++;
    IRQ_REARM();
}
```

With the v0.2 toolchain and ZPUng v1.1, the IRQ_REARM() macro is no longer required, instead, the `__attribute__((interrupt))` decorator must be used as preamble to the function declaration.

4.4 GPIO module

Simple general purpose I/O controller. The available registers are shown in Table 4.5.
The GPIO bank has a very simple functionality. Configure this bank in your firmware as follows:

1. Choose direction (input or output) using the GPIO_DIR register
2. Read a value using the GPIO_IN register, write via GPIO_OUT register
3. Atomically set GPIO outputs high using the GPIO_SET register (write only)
4. Atomically clear GPIO outputs using the GPIO_CLR register (write only)

Offset [Span]	Name(Id)	Access	Description
0x00 [2]	GPIO_DIR	RW	IO direction register. '1': Output, '0': input
0x04 [2]	GPIO_IN	RO	GPIO input (read) value
0x08 [2]	GPIO_OUT	RW	GPIO output value
0x0c [2]	GPIO_SET	WO	Write 1 to set
0x10 [2]	GPIO_CLR	WO	Write 1 to clear

Table 4.5: Address map GPIO

4.5 TIMER and PWMPlus module

The PWMPlus is an enhanced PWM core with programmable waveforms and the possibility to seamlessly switch between the waveforms using a data register. It can be used to stream specific waveforms using a data bit sequence.

4.5.1 Timer configuration

The timer module contains the configuration that apply to all PWMs at the same time. It derives the PWM clock from the incoming system clock which is divided by the the PWMCLKDIV register. This contains a divisor value minus one. The typical setup procedure follows this scheme:

1. Configure PWMCLKDIV
2. Configure PWM units needed
3. Start PWMs using the TIMER_START register

A write to the TIMER_START register starts the PWM unit whose corresponding bit is set. Likewise, a write to TIMER_STOP stops the corresponding PWMs.

Offset [Span]	Name(Id)	Access	Description
0x00 [2]	TIMER_START	WO	Write a '1' for each timer unit to be started.
0x04 [2]	TIMER_STOP	WO	Write a '1' for each timer unit to be stopped.
0x08 [2]	TIMER_STATUS	RO	Timer status register (reserved)
0x0c [2]	TIMER_CONFIG	RW	Timer config register

Table 4.6: Address map TIMER

4.5.2 PWMPlus configuration

Each PWM unit has a corresponding register map Table 4.7. The PWM_WIDTHx register defines the width of the full PWM interval. The PWM_PERIODx value determines when the timer event occurs and the PWM output level is toggled.
To transmit a seamless PWM sequence, data must be written to the PWM_DATA register fast enough such that the FIFO is never underrun during the transmission phase. This can be achieved by DMA on some platforms, when using the CPU, the WREADY bit has to be checked whether the FIFO can receive data. Overruns are marked by the ORUN bit, when an underrun is detected, the URUN bit is set.

Offset [Span]	Name(Id)	Access	Description
0x00 [2]	PWM_CONFIG	RW	PWM configuration register
0x04 [1]	PWM_STATUS	RO	PWM status
0x08 [2]	PWM_WIDTH0	RW	The PWM interval length minus one. When the counter reaches this value, it resets to zero.
0x0c [2]	PWM_PERIOD0	RW	The PWM period value (<= PWM_WIDTH) minus one
0x10 [2]	PWM_WIDTH1	RW	The PWM interval length minus one. When the counter reaches this value, it resets to zero.
0x14 [2]	PWM_PERIOD1	RW	The PWM period value (<= PWM_WIDTH) minus one
0x18 [2]	PWM_COUNTER	RO	Current PWM counter value. Volatile.
0x1c [4]	PWM_DATA	WO	Data word

Table 4.7: Address map PWM

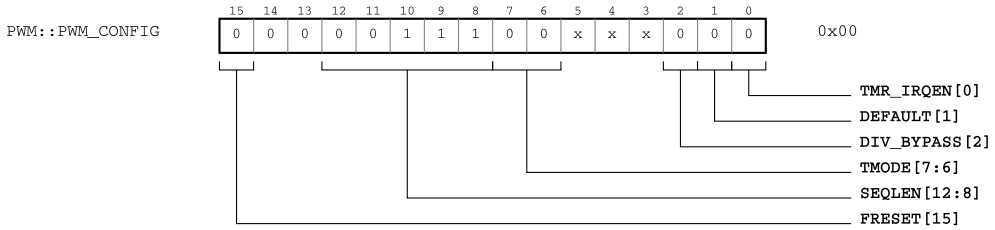

Figure 4.4: PWM_CONFIG

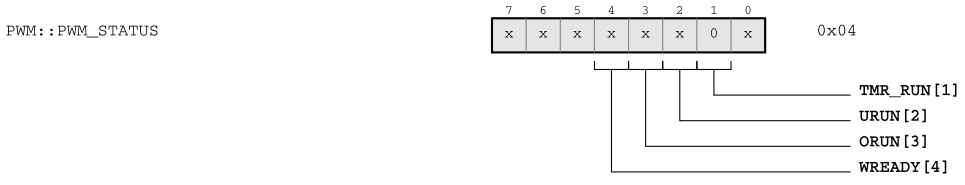

Figure 4.5: PWM_STATUS register

Value	Mode name	Description
0	CFG0	Standard mode, PWM configuration 0
1	CFG1	Standard mode, PWM configuration 1
2	RESERVED	Reserved
3	DATA	Data mode

Table 4.8: Mode TMODE – possible values

Prior to the transmission, the number of data bits must be configured using the SEQLEN bit field of the PWM_CONFIG register. Also, TMODE must be set to DATA, see Table 4.8.

The data written to PWM_DATA must be MSB (32 bit) aligned!

For example, if a 24 bit value is to be transmitted, SEQLEN is written with 23 and the original data has to be shifted left by 8 before writing to PWM_DATA.

4.6 Simple UART

The UARTSimple module implements a primitive UART with limited functionality:

- 8 bit frame size only, one start, one stop bit
- No parity support
- No break detection
- No handshaking

The UART is configured by setting the UART_CONTROL::UART_CLKDIV bitfield to a value of $sysclk/16/baudrate - 1$. Whenever the clock is changed, the (UART_CONTROL::UART_RESET) bit needs to be set and released as well for proper clock operation.
When the RX_IRQ_ENABLE bit is set, UART receive events trigger an IRQ on the UART irq line. See Appendix A how this IRQ pin corresponds to the SIC interrupt channels.

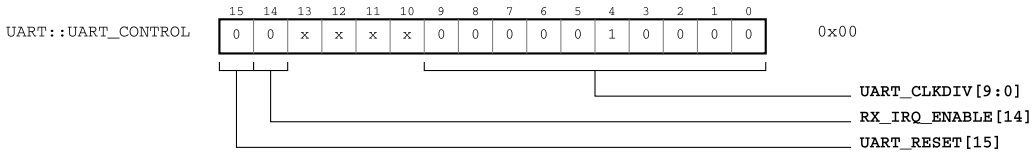

Figure 4.6: UART control register

For simple operation, the RXREADY/TXREADY bits of the UART_STATUS are polled before writing to UART_RXR and UART_TXR, respectively. Error bits are reset upon issue of a UART_CONTROL::UART_RESET toggle. The meaning of the status/error bits is as follows:

TXBUSY
When set, characters are left to transmit in the FIFO queue. However, you can keep writing data as long as TXREADY is asserted high

FRERR
When set, a framing error occured (less than 8 bits transmitted, or glitch)

TXOVR
The transmit FIFO was overrun, i.e. a write occured although TXREADY is low

RXOVR
The receive FIFO was overrun, because the UART_RXR register was not read in time

The non-pipelined Zealot CPU may have difficulties keeping up with high baud rates. There is no guarantee of a maximum baud rate and depends on the deployed system configuration and software

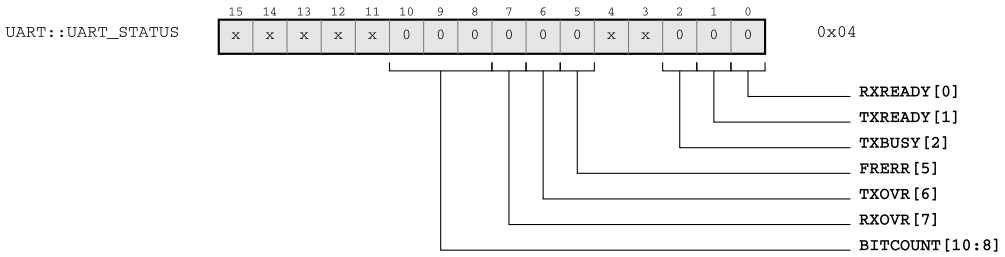

Figure 4.7: UART Status register

Offset [Span]	Name(Id)	Access	Description
0x00 [2]	UART_CONTROL	RW	UART Control register
0x04 [2]	UART_STATUS	RO	UART status register
0x08 [2]	UART_RXR	RO	UART receiver register
0x08 [1]	UART_TXR	WO	UART transmitter register

Table 4.9: Address map UART

4.7 SPI Master

This module implements a simple SPI master controller with the register map shown in Table 4.10.
The SPI master can be configured to use a RX/TX register width up to 32 bits. This upper limit is defined by CONFIG_SPI_BITS_POWER. See below on bit width configuration.

Offset [Span]	Name(Id)	Access	Description
0x00 [2]	SPI_CONTROL	RW	SPI control register
0x04 [1]	SPI_STATUS	RO	SPI status register
0x08 [4]	SPI_TX	WO	SPI transmit register
0x08 [4]	SPI_RX	RO	SPI receive register
0x0c [2]	SPI_CLKDIV	RW	SPI clock divider register

Table 4.10: Address map SPI

First, the SPI_CLKDIV register should be set with the desired clock divider. Again, the resulting SPI clock is the system clock divided by 4 divided by (SPI_CLKDIV-1). When the PUMP bit is set, a write to SPI_TX or read from SPI_RX will trigger a SPI send/receive cycle.

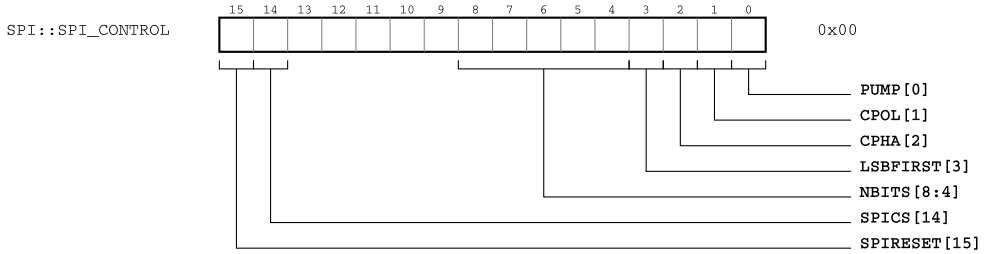

Figure 4.8: SPI_CONTROL register

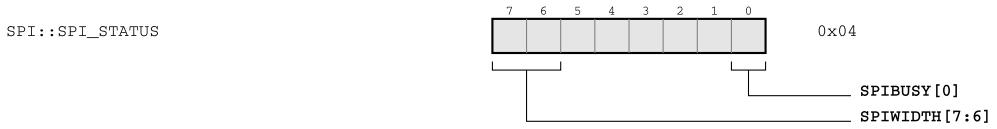

Figure 4.9: SPI_STATUS register

During a RX/TX cycle, the SPIBUSY bit is set. A driver routine has to poll this bit before issuing the next command.
When doing simultaneous transfers, the user will have to turn off the PUMP bit for either of the SPI_TX or SPI_RX access. More advanced peripherals as the **UniSI** core offer a few more configuration options and DMA support for improved throughput.
Other configuration bits:

CPOL
: Invert clock polarity

CPHA
: When set, assert first bit on first clock cycle

LSBFIRST
: When set, transmit LSB first

NBITS
: Number of bits to transmit minus one

SPIRESET
: Toggle to reset the SPI controller

4.7.1 Bit width configurations

The SPI peripheral can be hard-configured for a maximum transfer width of 8, 16 and 32 bits using the CONFIG_SPI_BITS_POWER variable. When the PUMP bit is set and a value to the SPI_TX register is written, the controller will start transferring the number of bits specified by the NBITS field, starting with the MSB by default (LSBFIRST = 0). Even if the number of transferred bits is lower than the maximum register width, the data has to be MSB-aligned. For example, assuming you have configured the maximum width to 32 bits (CONFIG_SPI_BITS_POWER = 5) and you need to write a byte value of 0xaa, you have to set NBITS to 7 and SPI_TX to 0xaa000000. It is recommended to drive the SPI controller in full 32 bit mode on the ZPU small architecture for greater speed.
If you have LSBFIRST set, the data is LSB-aligned, i.e. if you are writing a burst sequence 0xaa, 0xbb, 0xcc of 8 bits each, you have to write 0x00ccbbaa to the SPI_TX register and set NBITS to 23.

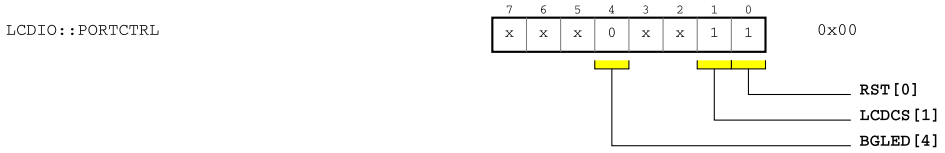

Figure 4.10: PORTCTRL register

Typically, the software should be configured to match the hardware configuration. If unsure or if you need to have a generic driver routine that determines the SPI bit width at runtime, you need to evaluate the SPI_STATUS::SPIWIDTH bit field.

4.8 LCDIO: asynchronous 8 bit parallel interface

The LCDIO module implements a simple, dedicated 8 bit asynchronous interface for the agathe/beatrix architecture of the SoC. It is used by a number of embedded TFT/STN displays from Sitronix, Ilitek, etc. These use a specific command / data mode, represented by an address line a0. This implementation of the interface handles the address bit through the I/O data register's 9th bit, see Fig. 4.11. Table 4.11 shows the register map of the LCDIO driver.

4.8.1 Register configuration

Offset [Span]	Name(Id)	Access	Description
0x00 [1]	PORTCTRL	WO	I/O port control
0x00 [1]	PORTSTAT	RO	I/O port state
0x04 [2]	IODATA	WO	I/O data, 8 bit write
0x04 [1]	IODATA_R	RO	I/O data, 8 bit read
0x08 [1]	TIMINGS	RW	LCD timings

Table 4.11: Address map LCDIO

The PORTCTRL register contains bits mapped to specific LCD pins. Prior to access of the LCD, a reset should be triggered by toggling the RST bit for the time specified by the LCD controller (see corresponding data sheet). RST is low active. To enable the LCD display, LCDCS must be set to 0.

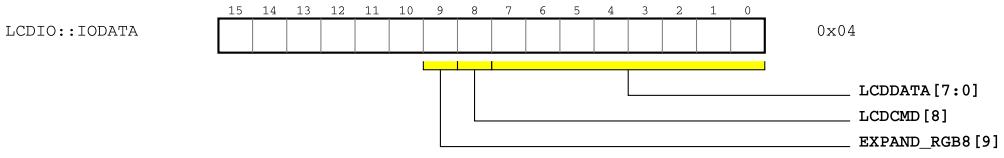

Figure 4.11: IODATA register

Since the asynchronous bus to the LCD needs to follow the timings specified by the on-LCD controller hardware, the TIMINGS register must be configured with the SETUP and HOLD timings.

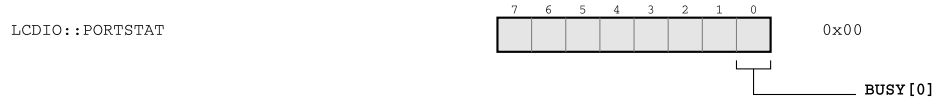

Figure 4.13: PORTSTAT register

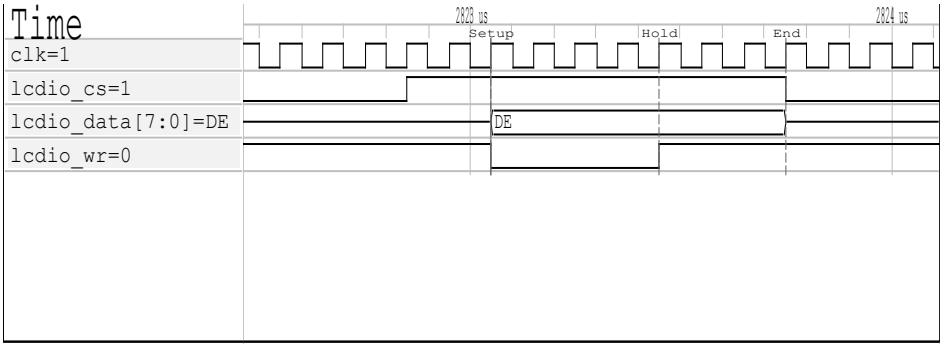

Figure 4.14: SETUP and HOLD timings

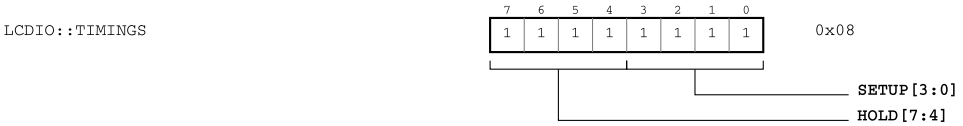

Figure 4.12: TIMINGS register

When writing to the IODATA register, the lcdio driver will follow the timing as outlined in figure Fig. 4.14 below. The beatrix variant of the SoC does not stall the CPU when **writing** the asynchronous interface, i.e. no wait cycles are introduced to the CPU per se upon I/O access. Therefore, the CPU needs to check the BUSY bit of PORTSTAT before issueing another transaction to the lcdio bus. For code optimization, there are calibrated routines available for a specific SoC setup that respect the bus timings without having to check the BUSY bit.

On the ZPU small architecture (Zealot), checking the BUSY bit takes more cycles than actually inserting the needed nummer of NOPs manually

The setup phase begins with a slight delay upon addressing of the IODATA register space. The data byte is asserted and the `lcdio_wr` signal pulled low. During the setup phase of SETUP+2 cycles, the relevant signals will not change. The LCD controller latches the data byte on the rising edge of the `lcdio_wr` signal. Likewise, a read phase is issued, however, the CPU will then stall also on the beatrix configuration until the End time is reached, introducing some performance penalty. Typically, no reads from a LCD controller are necessary during the display phase.
After the rising edge of `lcdio_wr`, the data word needs to be hold until the end of the transaction. Likewise, the signals don't change for (HOLD+1) cycles. In the above timing example, a value of 0x22 is written to the TIMINGS register.
The lcdio_cs pin is displayed inverted. FIXME.
Configurations from variant **cordula** on allow DMA transfers to the IODATA register. See SoC specific implementation section in Appendix A for details.

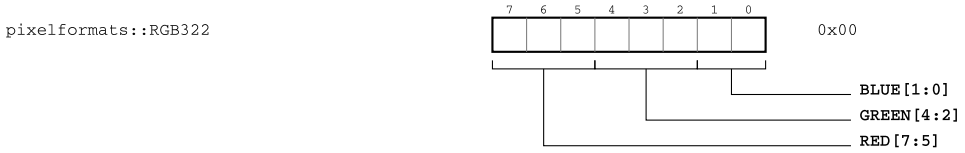

Figure 4.15: RGB 332 layout

Special 8 bit RGB332 decompression

When the `EXPAND_RGB8` bit is set while writing to IODATA, the written 8 bit value will be expanded to a two byte RGB565 value that is understood by the display controller and two writes will in fact be executed by the lcdio core. Fig. 4.15 shows the RGB component layout of the data byte to be written with the EXPAND_RGB8 bit set

4.8.2 Software support

TFT driver library

Depending on the board supply package, your configuration may come with driver routines for a specific TFT screen. Supported TFT screens are listed in Table 4.12. The LCD library is documented separately and not part of the default board supply package.

Controller description	Format
ILI9163	128x128 RGB
Sitronix ST77xx models and compatible	128x128 RGB
ILI9320, ILI9325 and compatible	320x240 landscape RGB
OTM3225	320x240 landscape RGB

Table 4.12: Supported TFT controllers

A

Custom SoC documentation

A.1 'agathe'/'beatrix' specific maps

A.1.1 ZPUng address map

The ZPUng has a no shared Program/Data/Heap and stack memory, instead, it uses a separate stack as shown in Table A.1.

Address range	Description
0x0000-0x3fff	Program memory and Heap
0x4000-0x41ff	Stack memory
0x8000-0xffff	Memory mapped registers (MMR)

Table A.1: ZPUng v1 RAM default address map

The stack memory start address always immediately follows the program memory (which is configured by CONFIG_BRAM_ADDR_WIDTH)

B

Register map reference

B.1 Device peripheral map

Table B.1 shows the address map offsets for the peripheral units in this SoC.

Device id	Address base	Description	Details
SysCtrl	0x0F8000	System controller	Section 4.1
GPIO0	0x0F8200	GPIO bank controller	Section 4.4
GPIO1	0x0F8220	GPIO bank controller	Section 4.4
IRQ	0x0F8100	System interrupt controller	Section 4.3.1
SPI	0x0F8300	Simple SPI controller	Section 4.7
PWM[0..CONFIG_NUM_TMR]	0x0F8400, 0x0F8420, ..	PWM controller (Advanced)	Section 4.5
TIMER	0x0F8500	Global timer unit	Section 4.5.1
UART	0x0F8600	Simple UART controller	Section 4.6
LCDIO	0x0F8700	Simple async I/O LCD controller	Section 4.8
SCACHE	0x0F8F00	Software cache	Section 4.2

Table B.1: Device map MaSoCist_bertram

B.2 Register details

B.2.1 MaSoCist_bertram registers

'SysCtrl' core registers

Bit(s)	Name	Description
7	UART_SEL	When set, UART will use the alternate pinout tx1/rx1
2	SYS_RESET	System reset. Resets entire CPU (1 active)
0	PERIO_RESET	1: Reset peripheral I/O section

Table B.2: SysCtrl register (RW) Offset: 0x00

Bit(s)	Name	Description
15:10	CPUFLAGS	Core specific CPU flags (undocumented)
9	HAVE_ICACHE	1: SoC has ICACHE support
8	HAVE_DCACHE	1: SoC has DCACHE support
7:4	CPUARCH	CPU architecture field (undocumented)
3:0	SOCREV	SoC Revision

Table B.3: SocInfo register (RO) Offset: 0x38

Bit(s)	Name	Description
31:16	CONFIG_ID	Configuration ID
15:8	REV_MAJOR	Major hardware revision number
7:0	REV_MINOR	Minor hardware revision number

Table B.4: HWVersion register (RO) Offset: 0x3c

'SCACHE' core registers

Bit(s)	Name	Description
3	IACK	Cache interrupt acknowledge. Writing this bit as '1' clears the cache interrupt signal.
2	DACK	Cache interrupt acknowledge. Writing this bit as '1' clears the cache interrupt signal.
1	ICACHE_ENABLE	Set to enable cache
0	DCACHE_ENABLE	Set to enable cache

Table B.5: CacheControl register (WO) Offset: 0x00

Bit(s)	Name	Description
3	IMISS	Read access MISS to ICACHE page
2	DMISS	Read access MISS to DCACHE page
1	ICACHE_EN	ICACHE active
0	DCACHE_EN	DCACHE active

Table B.6: CacheStatus register (RO) Offset: 0x00

'IRQ' core registers

Bit(s)	Name	Description
12:0	IAR	IRQ channel mapping

Table B.7: SIC_IAR register (RW) Offset: 0x0c

'SPI' core registers

Bit(s)	Name	Description
15	SPIRESET	Set 1 to reset SPI, 0 to resume
14	SPICS	Set '0' to select SPI
8:4	NBITS	Number of bits to transfer (up to 32)
3	LSBFIRST	1; Send LSB first, 0: MSB first
2	CPHA	Capture phase bit
1	CPOL	Invert clock polarity
0	PUMP	When set, access (R or W) to the SPIData register pumps a new transfer

Table B.8: SPI_CONTROL register (RW) Offset: 0x00

Bit(s)	Name	Description
7:6	SPIWIDTH	SPI data width: 0: 8 bit, 1: 16 bit: 2: 32 bit
0	SPIBUSY	When 1, SPI is busy transferring data

Table B.9: SPI_STATUS register (RO) Offset: 0x04

Bit(s)	Name	Description
15	CLKDIV_BYPASS	When set, bypass clock divider
7:0	CLKDIV	Clock divider value

Table B.10: SPI_CLKDIV register (RW) Offset: 0x0c

'TIMER' core registers

Bit(s)	Name	Description
15	CRESET	When set, a write to TIMER_START resets clock div counter
14:0	PWMCLKDIV	Clock divider for PWM

Table B.11: TIMER_CONFIG register (RW) Offset: 0x0c

'PWM' core registers

Bit(s)	Name	Description
15	FRESET	FIFO reset
12:8	SEQLEN	Sequence length minus one
7:6	TMODE	Timer mode.
2	DIV_BYPASS	Set to bypass the PWM pre-divider
1	DEFAULT	Default pin value at PWM output
0	TMR_IRQEN	Enable system IRQ for timer unit if '1'

Table B.12: PWM_CONFIG register (RW) Offset: 0x00

Bit(s)	Name	Description
4	WREADY	Underrun
3	ORUN	Overrun, wrote to FIFO when full
2	URUN	Underrun, FIFO drained before end of transfer
1	TMR_RUN	'1' when timer running

Table B.13: PWM_STATUS register (RO) Offset: 0x04

'UART' core registers

Bit(s)	Name	Description
15	UART_RESET	'1': Reset UART. Clear to run.
14	RX_IRQ_ENABLE	Enable receive IRQ
9:0	UART_CLKDIV	(16x) Clock divider

Table B.14: UART_CONTROL register (RW) Offset: 0x00

Bit(s)	Name	Description
10:8	BITCOUNT	RX bit counter
7	RXOVR	Receiver FIFO overrun. Cleared by UART_RESET.
6	TXOVR	Transmitter FIFO overrun. Cleared by UART_RESET.
5	FRERR	Sticky framing error. Set when stop bit not null. Reset by UART_RESET.
2	TXBUSY	'1' when TX in progress
1	TXREADY	Set when TX FIFO ready for data
0	RXREADY	Set when data ready in RX FIFO

Table B.15: UART_STATUS register (RO) Offset: 0x04

Bit(s)	Name	Description
8	DVALID	1 when data valid (mirror of RXREADY bit)
7:0	RXDATA	UART receive data

Table B.16: UART_RXR register (RO) Offset: 0x08

'LCDIO' core registers

Bit(s)	Name	Description
4	BGLED	Set to turn background LED on
1	LCDCS	LCD chip select, low active
0	RST	RESET pin to the TFT screen, low active

Table B.17: PORTCTRL register (WO) Offset: 0x00

Bit(s)	Name	Description
0	BUSY	1 when I/O port busy

Table B.18: PORTSTAT register (RO) Offset: 0x00

Bit(s)	Name	Description
9	EXPAND_RGB8	1: Expand 8 bit RGB into RGB565 (resulting in two LCD write accesses)
8	LCDCMD	Set when LCD command
7:0	LCDDATA	Data written to LCD interface

Table B.19: IODATA register (WO) Offset: 0x04

Bit(s)	Name	Description
7:4	HOLD	Hold time after WR/RD cycle
3:0	SETUP	Setup cycles before WR/RD cycle

Table B.20: TIMINGS register (RW) Offset: 0x08

C

Known issues

This appendix lists known bugs for cores and peripheral hardware versions.

C.1 CPU cores

C.1.1 Zealot small

For these issues, no workaround is present.

- Too frequent interrupts can trash the stack
- Emulation steps inside microcode

C.1.2 ZPUng overview

Known issues about the ZPUng architecture are listed in Table C.1. For a detailed workaround description, see below.

Version	Description	Workaround	Fixed
0alpha	Exception/IRQ handlers require at least four cycles from clearing the IPEND register until return	n	v1
0alpha	ICE single stepping would under certain conditions freeze the system (IRQ during immediate opcode)	n	v1
0alpha	Interrupts may no longer be triggered when using the reentrancy software trick	y	v1
v1.0	Stack trashing (underrun) not guarded. Program will just crash.	y	-
1.0	Code in extended program memory can not be debugged	y	-
1.0	Data in extended ROM area can not be dumped directly	y	-
1.0	Low priority interrupt handlers can become accidentally reentrant	n	v1.1

Table C.1: ZPUng issues

C.1.3 ZPUng v0

All issues below are fixed in v1

Issues without workarounds

- Exception/IRQ handlers require at least four cycles from clearing the IPEND register until return
- ICE single stepping would under certain conditions freeze the system (IRQ during immediate opcode)

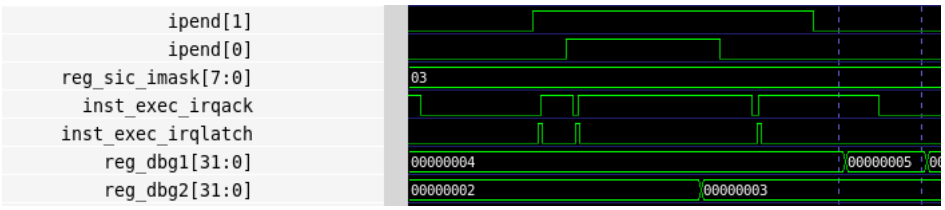

Figure C.1: IRQ failure scenario

Issues with workarounds

Interrupts may no longer be triggered when using the reentrancy software trick
Make sure to check for other interrupts inside a handler being serviced and clear ALL IPEND bits at the end of the routine. In very rare cases, if current IRQ reentrancy is wanted, you may have to use semaphores with guaranteed atomic behaviour.

C.1.4 ZPUng v1 rev0

Issues without workarounds

Stack trashing (underrun) not guarded. Program will just crash.
If stack underrun occurs due to recursive calling of a function, undefined behaviour may occur. This can be fixed using the WPU from all 'c' architectures (cranach, ...)

Low priority interrupt handlers can become accidentally reentrant
If a low priority IRQ routine is interrupted by a high priority interrupt, the interrupt rearming logic may cause a situation where, after returning from the high prio handler, the low prio IRQ handler interrupts itself again. This is causing unwanted reentrancy, as shown in Fig. C.1. This could cause a counter to be incremented twice. Fixed in v1rev1.

Issues with workarounds

Code in extended program memory can not be debugged
Place the routines in the .text default area first. Once you have found them to be stable, you can move them to the extended program memory. For examining the data, see below. Experimental software based workarounds do exist, however they don't work with the standard debugger. .

Data in extended ROM area can not be dumped directly
Set a break point in the cache handler after the data has been read from the ROM and examine the data from the cache area. Use gdb helper scripts to calculate the proper address.

C.1.5 ZPUng v1 rev1

Change in behaviour:

- During an exception in service, IRQs can no longer interrupt the exception handler, even if the exception cause is cleared. The pending interrupt request is serviced after a IUNLK (opcode 0xf) instruction is issued.

C.2 Peripheral hardware 'agathe' and 'beatrix'

C.2.1 Revision 0.1

This section applies to the hardware definition v0.1. All listed issues are fixed in the v0.3 revision, if not noted otherwise.

SIC

- No hardware support for reentrant interrupt services

Timer and PWM

- Changing the PWMCLKDIV register would not reset the timer. This could cause a somewhat nondeterministic behaviour of the PWM

SPI

- Pre-Division of system clock for SPI is too high, therefore SPI max. clock is limited
  - v0.2 uses predivider 4 instead of 16
  - v0.3 introduces BYPASS bit for highest possible speed

C.2.2 Revision 0.2

SIC

- When exception handling interrupt had lower priority and a simultaneous high priority event occured, the higher priority interrupt service handler would be executed instead
  - 0.3 SICv2 uses specific exception (NMI) input with highest priority, but allows lower priority IRQs to occur once the exception cause has been cleared.

Exceptions are BY DEFAULT non-reentrant!

Timer and PWM

- PWM_WIDTH and PWM_PERIOD were swapped in behaviour. This did not have any effect with the used pwm.c driver. Fixed in 0.3, make sure to use the functions in pwm.c

C.3 Peripheral hardware 'bertram'

C.3.1 Revision 0.3

PWMPlus

- Experimental: Has swapped register behaviour with PWM. Fixed in 0.4

C.4 Peripheral hardware 'cordula', 'cranach'

Cordula/Cranach use different CPU cores, but same peripherals and mandatory MMR 32 bit access width

C.4.1 Revision 0.0

All listed issues are fixed in v0.1.

SIC

- Interrupt handling could show false behaviour in simulation model. Does not apply to hardware!

C.4.2 Revision 0.1

This section applies to the hardware definition v0.1. All listed issues are fixed in the v0.3 revision, if not noted otherwise.

DMA

- DMA engine revised for one clock cycle delay, but higher speed and compatibility to vendor specific FIFOs

ScratchPad memory 'cranach'

- On cranach, only the ScratchPad memory is DMA capable. On cordula, all data memory can be accessed by DMA.

C.5 Gensoc issues

Wishbone wait states on custom peripherals not supported
This is a current limitation of the perio_mmr module generated by the gensoc v0.2 release. You may have to use manual fixes for the time being.