Dipl.-Ing. Tobias Scheipel

Prof. Dr. Marcel Baunach

Microcontroller Design, Laboratory "HaDes" SS 2023

Embedded Automotive Systems Group
TU Graz – Institute of Technical Informatics

HADES: *The Greek underworld, in mythology, is an otherworld where souls go after death, and is the original Greek idea of afterlife. At the moment of death the soul is separated from the corpse, taking on the shape of the former person, and is transported to the entrance of the Underworld [by Charon across the river Styx, see Figure 1]. The Underworld itself is described as being either at the outer bounds of the ocean or beneath the depths or ends of the earth. It is considered the dark counterpart to the brightness of Mount Olympus, and is the kingdom of the dead that corresponds to the kingdom of the gods. Hades is a realm invisible to the living, made solely for the dead.*

– Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc. 06 Feb 2017. Web. 09 Feb 2017.

Figure 1: *Crossing the River Styx* by Joachim Patinir (1480 - 1524), Prado Museum, Madrid

In this course, HaDes is simply for "Hardware Design".

Version: January 2, 2023

Contents

1 Preface 1

2 Toolchain Preparation 3

3 Getting ready 5

4 Overview 8

5 Exercises 13

I CPU Construction 14

6 Module: Program Memory 15

7 Module: Instruction Decoder 17

8 Module: Register file 20

9 Module: Control Unit 21

10 Module: Datapath 24

11 Module: Program Counter 29

II Hardware Experiment 40

12 Development Board 41

13 XBus 45

14 XBus Peripherals 48

15 Assembly 54

16 HaDes Object Assembler 56

17 Assembly Exercise 63

III Appendix 65

HaDes Instruction Set 66

Acronyms 69

Bibliography 70

Document Revision History 71

1 Preface

The HaDes Lab is about deeply understanding the internal concepts, structure, and operation of Microcontroller Units (MCUs) . By the end of this course, you will have implemented a specific MCU in a Hardware Description Language (HDL) and software for your MCU in Assembly (ASM). You will have honed your skills in describing digital logic, implementing low level code, debugging software, and hardware using various tools. The exercises will be done in VHDL on a Xilinx® Artix™-7 FPGA, which will be provided on the BASYS3 development board from Digilent, as depicted in Figure 1.1.

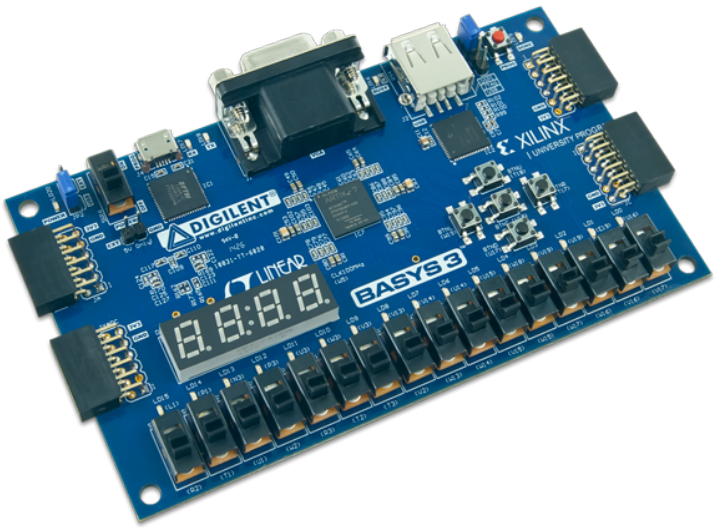


Figure 1.1: The BASYS3 development board featuring an Artix™-7 FPGA.

WE WANT YOU TO BE SUCCESSFUL! The Microcontroller Design Lab (HaDes Lab) takes place as a standalone laboratory. Nevertheless, it requires a sound understanding in a broad spectrum of topics *and* the ability to link this knowledge in a well-structured and creative way. Even though there are no formal requirements on previously passed courses, it is highly advisable to have some knowledge about "CPU architectures" and "Hardware Description Languages".

HOW DOES THE LAB WORK? The lab consists of two parts. In the first part (compulsory), each participant presents his/her implementation to the other participants. This gives you the opportunity to look at other implementations and to discuss the implementation decisions. The second part (voluntary) is a supervised lab, giving the opportunity to ask the supervisors about your implementation troubles.

In any case, the exercises described in Chapter 5 must be completed individually by each participant within the given deadlines. The exercises can be completed at home or in the lab, while the time in the lab offers the opportunity for questions and discussions.

YOU WILL BE EVALUATED! Each participant is evaluated individually. Therefore, we will first review your source code regarding completeness, functionality, structure, and documentation/comments. Furthermore, we will evaluate the presentation of your implementation to other students; therefore the first part of the weekly meeting is mandatory. You should also be able to answer questions (from the supervisor and from students) related to the current exercise and your implementation. The quality of the answers and the participation will influence your final grade.

A FINAL WORD! Passing the HaDes Lab will probably require a significant amount of time. Nevertheless, we believe that the content will give you a good understanding of the functionality of a microcontroller. On the other hand, the lab is new to Graz University of Technology and this is the fourth iteration. Thus, some unexpected problems are likely to occur and the provided documentation or source code will also not be perfect yet. We kindly ask for your understanding and also ask you to help us improve the course for the iterations ahead. Suggestions are always welcome.

Have Fun and Success!

2 Toolchain Preparation

In this chapter, we will guide you through the toolchain installation process and shortly present the used tools. Please do not try to install anything before reading the rest of this chapter! Some details here are important for the correct installation. Before you start with the implementation, please restart your computer.

Our laboratory will be conducted on Linux Virtual Machines. A VirtualBox image with the full installation of the toolchain explained below can be downloaded via the link in the TUGraz TeachCenter course (`https://tc.tugraz.at`). However, Windows® is also supported throughout the lab.

2.1 GHDL

GHDL is an open source simulator for Very High Speed Integrated Circuit Hardware Description Language (VHDL), based on the well-known GCC. The simulator implements the VHDL language standards according to the IEEE 1076-1987, IEEE 1076-1993, and some parts of the IEEE 1076-2000 specification. To simulate the VHDL code, GHDL first translates and compiles your code and generates binaries for your host system. Second, these binaries are executed, and the simulation results are outputted. GHDL is not able to synthesize the VHDL code for a Field Programmable Gate Array (FPGA); other tools are required. You can find the installation file for GHDL in the `_software` folder of our provided repository. Since 2017, you do not need to install this executable since it is already embedded in the provided development environment, distributed at the beginning of the semester.

2.2 GTKWave

The GHDL simulator compiles and executes VHDL code on your host system and outputs the result in textual form. The textual form is useful for information, warnings, errors, and assertion messages. However, it is often helpful to analyze concurrent signals in a so called "wave viewer". A widely used open source wave viewer is GTKWave. You can download the viewer from `http://gtkwave.sourceforge.net/` or use the provided version in the `_software` folder of our provided repository. You have to extract the `ZIP` file to any directory of your choice. To use the program directly from your Windows command line, you also have to manually add the path of the `gtkwave` folder to the Windows® environment variables:

Start → right-click on Computer → Properties → Advanced system settings (on the left) → Environment Variables → Path (under System variables) → Edit → add GTKWave at the end of the path (e.g, `;C:\Program Files\gtkwave\bin`)

2.3 Xilinx®Vivado™

After a successful implementation and simulation of all CPU modules, you probably want to test it on real hardware. To generate a netlist for the FPGA you have to use a synthesis tool, like Vivado™. Vivado™ is offered by Xilinx® in different editions. The Web edition is the free one and sufficient for our purpose. You

can find the WebPack version at http://www.xilinx.com/support/download.html. To get the installer you have to register yourself for free at the Xilinx® homepage. Vivado can also be used for developing and simulating your HDL codes. During installation, chose the WebPACK. Get the *Free SDK, Vivado WebPACK* and press "Connect Now". A browser will open and let you select the free available licenses and activate them.

Note: We are currently using Xilinx® Vivado™ version 2016.2.

2.4 Git

Git is an open source version control system. We will use it to provide the development environment, and you will use it to upload your code. You can use any Git client you prefer (e.g., https://git-scm.com/).

3 Getting ready

After a successful toolchain installation or image import, you can download our prepared development environment from our Git server. We have already created a template to use for your own Git repository. In this chapter, we present how to download the environment step by step, by using the proposed Git client `git-scm` (simple tutorial: `http://rogerdudler.github.io/git-guide/`). When using another tool, please make sure it **does not change the line endings in the files**!

3.1 GIT repository creation

Open the URL

`https://gitlab.tugraz.at/`

and sign in using "'LDAP AD"' and your TUGonline username and password. Create a new GIT repository according to the following naming rules:

`MDLab_<SURNAME>_<REGNUMBER>`

where `<REGNUMBER>` is your 8-digit student registration number. In your repo's settings, navigate to "'Members"' and make sure to add the following users as "'Maintainer"':

- Scheipel, Tobias Peter
- Beikircher, David
- Riedl, Florian

3.2 Environment Checkout

First, create and navigate to the folder to which the repository shall be downloaded. Open a terminal and navigate into your newly created folder. Executing the command

```
git clone --no-checkout https://gitlab.tugraz.at/easpublic/MDLab.git
```

will clone the repository without downloading the files. This is necessary as we first need to set a configuration for this repository. To do so, enter the folder by executing:

```
cd MDLab
```

Now, disable the automatic conversion of the line ending with:

```
git config core.autocrlf false
```

With this, the configuration is finished, and you can proceed downloading the content:

```
git checkout
```

Follow the same procedure to checkout your individual source code repository using the URL

`https://gitlab.tugraz.at/path_to_your_repo`

and place it into our provided environment folder `MDLab`.

3.3 *Project Structure*

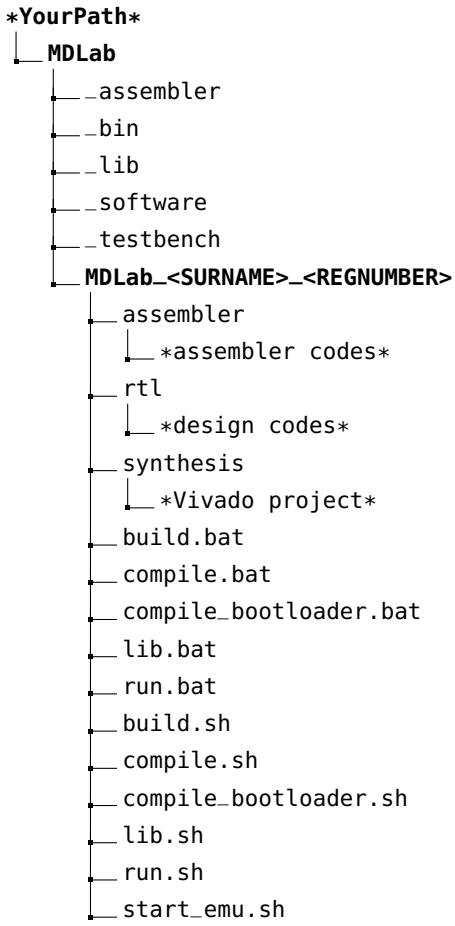


Figure 3.1: Project folder structure.

For a convenient organization, configuration, and maintenance, the project files are located in various directories. Do build your repository with the same structure as shown in Figure 3.1. **Do NOT change this structure!** Figure 3.1 shows further folders from the whole project structure. The folders with the underscore at the beginning of the name (e.g. `_assembler`) are provided by us for a better maintenance of the project.

The folder `_assembler` contains provided assembler software for your MCU. The folder `_bin` contains the required executables for simulating, compiling, linking, and using the emulator. The folder `_lib` contains some useful libraries to simplify the building of a fully working MCU. The `_software` folder contains the installers, which are installed into the `_bin` folder. Finally, you can use the testbenches in the folder `_testbench` to test your implementations.

Your main working directory is your repository `MDLab_<SURNAME>_<REGNUMBER>`. At the beginning, your repository is empty. We have prepared a template folder (`template`), which must be copied into your repository first. Please store your assembly and HDL code into the folders `assembler` and `rtl`, respectively. In folder `synthesis` you will also find a prepared Vivado project, which is required to synthesize your design. For the design, you have to use the provided VHDL templates to guarantee an uniform interface between all modules implemented by you. We also offer four batch/bash files for building and simulating your design. The batch/bash files with the suffixes `.bat` and `.sh` are used under Windows and Linux, respectively. These files are used by our automatic build and test system; therefore, **make sure that they will always work** on a Linux machine!

3.3.1 *How to upload files to git?*

To let us review and evaluate your progress, you have to add, commit, and push your code to the git server. To reduce the storage consumption and potential problems after a compilation or synthesis on a different workstation, you should not add the generated output files. To add files to the version control system you can use the git command:

```
git add <files to add>
```

The simplest way to add all files, except the generated files, is to add all files right at the beginning, i.e., when preparing your folder. To label the current progress of your repository with a meaningful text you should use the `commit` command:

```
git commit -a -m '<text>'
```

All your commits are stored only locally as long as you do not push all your changes to the git server with the command:

```
git push origin
```

3.3.2 *How do the scripts work?*

Your workspace contains a total of **five** scripts for both Linux and Windows operating systems: `build`, `compile`, `compile_bootloader`, `lib` and `run` as well as **one** Linux script to start the HaDes emulator: `start_emu.sh` (see Figure 3.1). In order to work properly, the following workflow is suggested (Linux only) when going through the exercises:

1. build all libraries (must be done at least once) using

   ```
   ./lib.sh all build
   ```

2. build your current module (e.g., `pmemory`) using

   ```
   ./build.sh pmemory
   ```

3. simulate the module and show its waveform in GTKWave using

   ```
   ./run.sh pmemory show
   ```

The rest of the scripts will not be explained in detail. Further information can be found directly in those scripts (mostly self-explanatory).

4 Overview

The processor you will implement in this course is a 32-bit processor. It was initially designed at the University of Würzburg, Germany, before the turn of the millennium. The developer gave it the name *HaDes*, because the processor has been developed in the context of the course **Ha**rdware-**Des**ign.

The processor has a small and non-orthogonal instruction set; therefore, it is classified as a Reduced Instruction Set Computer (RISC) processor. A further characteristic of a RISC processor is the load/store architecture. There, the processor loads data from and to data memory with the help of General Purpose Registers (GPRs) and transfers data between them. HaDes has 8 GPRs in which the zero register $r0$ always has the value zero. Besides, all instruction words have a constant length of 32 bit, which simplifies the instruction decoder.

The memory architecture of the processor follows the Harvard model, in which data and program memory are logically and physically separated from each other, in contrast to the Von-Neumann architecture. HaDes integrates the program memory directly into the processor. However, the data memory is a component (XDMemory) accessible over the external data bus (XBus).

The instruction set, as detailed on Section 4.1, distinguishes among arithmetic & logical, comparison, branch & jump, transportation, and interrupt instructions. The arithmetic & logical and comparison instructions use a three-address code. This means that every instruction has two input source registers and one destination register (`instruction` *dst*, *src1*, *src2*) or, instead of the second source register, a 16-bit immediate operator (`instruction` *dst*, *src1*, *imm*). Every arithmetic instruction is interpreted as a signed operation. The architecture can only generate relative addresses or load the contents of a register into the Program Counter (PC). Since the width of the immediate operator is smaller than the address bus width, there are some restrictions in the branch & jump instructions.

HaDes uses two different instructions for data transfer and transfer to the peripherals. The data transfer between registers and data memory is done with the `LOAD/STORE` instructions. No other instruction has access to directly modify the data memory. However, to address HaDes peripherals (e.g., XTimerXT, XUART, ...) you have to use the `IN/OUT` instructions.

Some special instructions are used to handle the interrupt system, which contains four prioritized interrupt levels. When an interrupt occurs, the interrupt controller jumps to the respective Interrupt Service Routine (ISR) entry address, which was set beforehand with one of the special interrupt instructions.

4.1 Instruction Set

The register set R contains eight registers $R := \{r0, r1, r2, r3, r4, r5, r6, r7\}$ in which the value is defined as $r0 := 0$ and $r1, \ldots, r7 := \{x \in \mathbb{Z} \mid -2.147.483.648 = -2^{31} \leq x \leq 2^{31} - 1 = 2.147.483.647\}$. In the following, lets define $w, a, b \in R$ and the constant value $k := \{x \in \mathbb{Z} \mid -32.768 = -2^{15} \leq x \leq 2^{15} - 1 = 32.767\}$.

Branch & Jump. The branch & jump instructions accept relative addresses, except for the `JREG` instruction with an absolute address in the register. The following instructions are provided by the CPU:

Instruction	Description
`BEQZ` a, #k	Branch with relative offset k, if $a = 0$
`BNEZ` a, #k	Branch with relative offset k, if $a \neq 0$
`BOV` #k	Branch with relative offset k, if the overflow (OV) flag is set
`JMP` #k	Jump with relative offset k
`JAL` w, #k	Call subroutine with relative offset k and store absolute return address into register w
`JREG` a	Jump to absolute address in register a

Arithmetic & Logical operations. All arithmetic and logical instructions (with the exception of `SETOV/GETOV`) have the three-address format (`instr` *dst*, *src1*, *src2*), where *src2* can be replaced by an immediate signed 16-bit constant (`instr` *dst*, *src1*, imm). The instructions `ADD`, `SUB`, `MUL`, `SHL` and `SHR` may influence the Overflow (OV) flag.

Instruction	Description
`ADD` w, a, b	32-bit signed addition
`SUB` w, a, b	32-bit signed subtraction
`MUL` w, a, b	32-bit signed multiplication
`AND` w, a, b	32-bit logical `AND`
`OR` w, a, b	32-bit logical `OR`
`XOR` w, a, b	32-bit logical `EXCLUSIVE-OR`
`XNOR` w, a, b	32-bit logical `NOT-EXCLUSIVE-OR`
`SHL` w, a, b	32-bit non-cyclic left shift
`CSHL` w, a, b	32-bit cyclic left shift
`SHR` w, a, b	32-bit non-cyclic right shift
`CSHR` w, a, b	32-bit cyclic right shift
`SETOV` b	Set Overflow (OV) flag
`GETOV` w	Get Overflow (OV) flag

Comparisons. The comparison instructions compare two registers with each other and, depending on the instruction and the comparison result, the destination register w is set to 1 or 0. There is also an immediate version for each of the instructions below.

Instruction	Description
`SEQ` w, a, b	Set 1 if $a = b$ else 0
`SNE` w, a, b	Set 1 if $a \neq b$ else 0
`SLT` w, a, b	Set 1 if $a < b$ else 0
`SGT` w, a, b	Set 1 if $a > b$ else 0
`SLE` w, a, b	Set 1 if $a \leq b$ else 0
`SGR` w, a, b	Set 1 if $a \geq b$ else 0

Load & Store. To load a value from the data memory or to store a value in the data or program memory you have to use the following instructions. Furthermore, there are two special instructions to enable and disable the write access to the program memory.

Instruction	Description
`LOAD` w, a, #k	Loads a 32-bit value from data memory into register w
`STORE` b, a, #k	Stores register b value into data or program memory
`DPMA`	Disables program memory write access
`EPMA`	Enables program memory write access

Peripheral communication. The software interface to peripherals is handled with the following instructions. If you access an invalid peripheral address, an interrupt will be raised.

Instruction	Description
`IN` w, #k	Loads the value from peripheral address k into register w
`OUT` a, #k	Stores the value in register a into peripheral address k

Interrupt Handling. There are some interrupt handling instructions.

Instruction	Description
`ENI`	Enable interrupts
`DEI`	Disable interrupts
`SISA` #j, #k	Set relative ISR entry address for interrupt #$j \in \{0,1,2,3\}$
`RETI`	Returns from ISR
`SWI` a, #k	Software interrupt with arguments #k and a
`GETSWI` w, #j	Get software interrupt arguments and save in w, for argument #$j \in \{0,1\}$

Syntactic Sugar. To simplify programming, there are some pseudo instructions, which are converted to real instructions by the compiler.

Instruction	Description
`LDI` w, #k	Load signed #k into register w
`LDUI` w, #k	Load unsigned #k into register w
`MOV` a, w	Move register a to register w
`INC` a	Increment register a by 1
`DEC` a	Decrement register a by 1

A more detailed explanation of all instructions can be found in Chapter III.

4.2 Modules

The HaDes CPU contains six top-level modules which are depicted in Figure 4.1. The `CLK` and `RESET` signals were omitted for a better visualization.

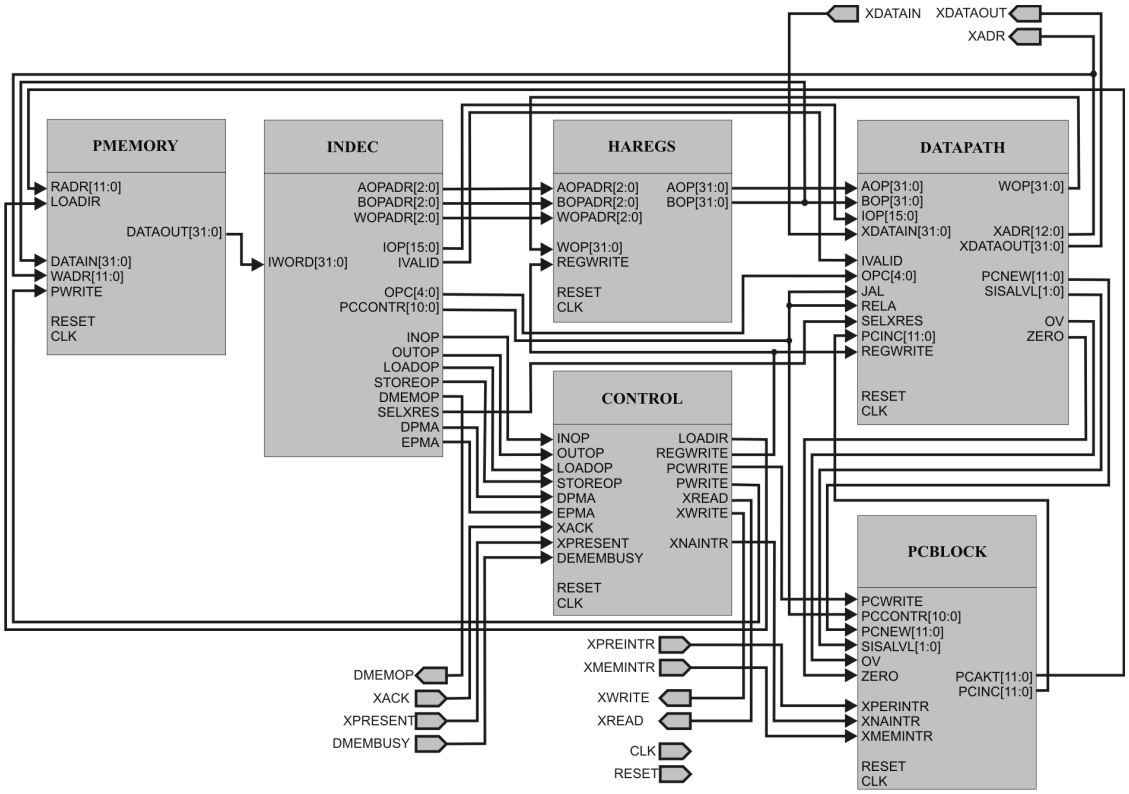


Figure 4.1: HaDes overview.

PMEMORY The PMEMORY module is the program memory. It contains the instructions, which are read by the instruction decoder (INDEC). After a reset or `EPMA` instruction, the memory is writable. Therefore, the boot loader can upload a new firmware to the program memory. When the upload is finished, the `DPMA` instruction, set by the boot loader, restricts the access to the program memory. Further information can be found in Section 6.

INDEC After reading the instruction from the program memory, the instruction decoder decodes the instruction and generates some control signals. See Section 7 for implementation details.

CONTROL Some control signals from the INDEC module are connected to the CONTROL module, which controls the implemented state machine to guarantee a correct instruction execution. More in Section 9.

HAREGS The HAREGS module contains the eight General Purpose Registers (GPRs). They are used to temporarily save the values directly in the CPU. Further information in Section 8.

DATAPATH One of the main components in a CPU is the datapath. It contains the ALU, which is the computation unit of the CPU. More in Section 10.

PCBLOCK The program counter module is responsible for keeping track of the next instruction from the program memory to be executed. Furthermore, it contains the logic for the interrupt management.

Further information in Section 11.

5 Exercises

Implement and hand in all the following exercises according to the Lab schedule:

1. Implement the **program memory** and verify it with the testbench `pmemory_tb.vhd` (cf. Section 6).
2. Implement the **instruction decoder** as combinatorial logic and verify the instruction decoder with the testbench `indec_tb.vhd` (cf. Section 7).

3. Implement the **register file** and verify it with the testbench `haregs_tb.vhd` (cf. Section 8).
4. Implement the **control unit** with a case statement in a *clocked process*. Hint: The next state should take over in the next cycle. Verify the control unit with the testbench `control_tb.vhd` (cf. Section 9).

5. Implement the **ALU**. Hint: You should use the constants from the package `hadescomponents`. Verify the ALU with the testbench `alu_tb.vhd`.
   Implement the **datapath** with the sub-module ALU and verify it with the testbench `datapath_tb.vhd` (cf. Section 10).

6. Implement the **PCLogic** and verify the sub-module with the testbench `pclogic_tb.vhd` (cf. Section 11).
7. Implement the **ISRALogic**. Verify it with the testbench `isralogic_tb.vhd` (cf. Section 11).

8. Implement the **ISRRLogic** and verify it with the testbenche `isrr_tb.vhd` (cf. Section 11).
9. Implement the **CheckIRQ**. Verify the sub-modules CheckIRQ with the testbenche `checkirq_tb.vhd` (cf. Section 11).
10. Build together the **IRQLogic** and verify it with the testbench `irqlogic_tb.vhd` (cf. Section 11).
11. Build together the **PCBlock** and verify it with the testbench `pcblock_tb.vhd` (cf. Section 11).

12. Connect all implemented HaDes modules and verify them with the five testbenches in `cpu_tb.vhd` (cf. Section 15). Synthesize the MCU and test it with the demo software `demo.hix`. Please **push** the Bitfile `.bit` to your repository!

13. Implement a paint program with the usage of the VGA, PS/2, LEDs, buttons, and switches of the BASYS3 board. The program has to follow the specification presented in Section 17 (Hint: You can use the offered emulator to implement this application).

Part I

CPU Construction

6 Module: Program Memory

The memory concept of the HaDes CPU is the Harvard architecture, thus the program memory is separated from the data memory. The program memory contains the instructions that the CPU has to execute. After an initialization phase, the program memory becomes read-only, which appears as a Read Only Memory (ROM).

6.1 Interface

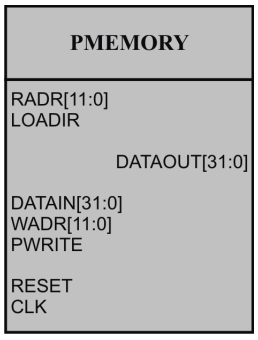


Figure 6.1: Program Memory interface.

Pin	Type	Source/Destination	Description
RADR(11..0)	I	PCBLOCK	Address for next instruction to be read
LOADIR	I	CONTROL	Read signal
WADR(11..0)	I	DATAPATH	Address for next instruction to be written
DATAIN(31..0)	I	HAREGS	Write value to address WADR(11..0)
PWRITE	I	CONTROL	Write signal
DATAOUT(31..0)	O	INDEC	Instruction code on address RADR(11..0)

6.2 Functionality

Real systems use many different types of program memories. Many systems use a ROM, which is not possible to reprogram. Other MCUs use newer technologies, which allows to reprogram the program memory with the help of an external programmer tool. Some MCUs also bring the support to reprogram the program memory without an external tool, only from software, but with some restrictions (e.g. only blocks are programmable and it takes milliseconds to program a block).

Our program memory is a special memory that is initialized with a bootloader. In the initialization phase, the bootloader reprograms the program memory with an executable and initializes some global variables in the data memory. The executable, with instructions and the global data initial values, is received over the serial interface from the working station.

At start-up, the bootloader waits for some data from the serial interface. To store the instructions into the program memory the instruction `STORE` is used. Normally, this instruction stores the data value into the data memory, but not in the initialization phase. To signal the end of the program memory programming, the special instruction `DPMA` is used. It disables the program memory write access, than the next `STORE` instructions are performed in the data memory. Thus, the `PWRITE` signal is only generated before the use of `DPMA`, or after the use of `EPMA`, which re-enables the program memory write access.

The program memory is integrated into the processor, while the data memory (XDMemory) is connected over the external data bus (XBus). The program memory's space is 4096 words in each case with a size of 32 bits and is accessible only aligned. Therefore, to address 4096 words an address width of 12 is necessary.

6.3 Implementation Hint

For the implementation of the PMEMORY module, you have to use the provided `hades_ram32_dp` component in the `work` library. This component creates a memory with a configurable size.

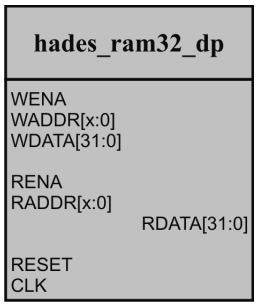


Figure 6.2: HaDes ram32 interface.

In a simultaneous rise of the clock CLK↑ and a high level of the signal WENA the value of WDATA will be written into the address WADDR of the memory. However, a high level at signal RENA and a simultaneous rise of the clock CLK↑, reads out the value at address RADDR and puts it out on output RDATA. Therefore, you can connect directly the ports of the PMEMORY component to the `hades_ram32_dp` component. You have only to initialize the component with the following parameter:

Parameter	Value
`WIDTH_ADDR`	`12`
`INIT_FILE`	Generic INIT parameter from PMEMORY
`INIT_DATA`	`hades_bootloader`

The parameter INIT_FILE allows the programmer to initialize the memory with data. In this specific module, with instructions from an executable file. This generic should be inherited from the generic in the PMEMORY component. It is useful to use this parameter for initializing the program memory with another firmware instead the default bootloader. Thus, you can use different executables on different testbenches. The input file has to be a `HEX`-file in which the data is structured according to the standardized Intel-Hex-Format. If no file is defined (`INIT_FILE = "UNUSED"`), the memory will be initialized with the INIT_DATA. In our case, the `hades_booloader` function returns the default bootloader for the Hardware Experiment. In order to recognize `hades_ram32_dp` and the function `hades_booloader` in the implementing module, you have to include the library work and the package `hadescomponents` as shown in the next Listing:

```
library work;
use work.hadescomponents.all;
use work.all;
```

7 Module: Instruction Decoder

The instruction decoder decodes the current instruction code and generates the necessary information to execute the instruction for other HaDes components.

7.1 Interface

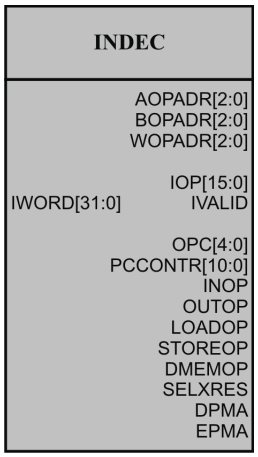


Figure 7.1: Instruction decoder interface.

Pin	Type	Source/Destination	Description
IWORD(31..0)	I	PMEMORY	Current instruction word from program memory
AOPADR(2..0)	O	HAREGS	Register address of the operand A
BOPADR(2..0)	O	HAREGS	Register address of the operand B
WOPADR(2..0)	O	HAREGS	Register address of the destination operand
IOP(15..0)	O	DATAPATH	Immediate value
IVALID	O	DATAPATH	Immediate operand validity
OPC(4..0)	O	DATAPATH	Op-code for ALU
PCCONTR(10..0)	O	PCBLOCK, DATAPATH	Program counter control for PC and interrupt
INOP	O	CONTROL	`IN` instruction
OUTOP	O	CONTROL	`OUT` instruction
LOADOP	O	CONTROL	`LOAD` instruction
STOREOP	O	CONTROL	`STORE` instruction
DMEMOP	O	XBUS	`STORE` or `LOAD` instruction
SELXRES	O	DATAPATH	XBus will be used
DPMA	O	CONTROL	`DPMA` instruction
EPMA	O	CONTROL	`EPMA` instruction

7.2 Functionality

The INDEC component generates control signals, which are needed by the current instruction word, available at IWORD, which is provided from the program memory module. This instruction word remains

stable until the instruction is already executed, therefore the instruction decoder can be implemented as combinatorial logic. That means that you should not use registers (sequential logic). More information about the phases of an instruction execution can be found in module Control Unit (Section 9).

The information on how to set the output signals of this module can be taken directly from the instruction code. The next table shows the instruction codes for all instructions in the HaDes CPU, whereby the bits 27..23, for the Arithmetic Logical Unit (ALU) operations, are shownin a seperate table.

Instruction	31 30 29 28	27 26 25 24 23	22 21 20	19 18 17	16	15 14 13 12 11 10 9 8 7 ... 0
ALU	0 0 0 0	[AluOpCode]	[WREG]	[AREG]	0	[BREG] 0 0 0 0 0 0 ... 0
ALUI	0 0 0 0	[AluOpCode]	[WREG]	[AREG]	1	[IMOP16]
NOP	0 0 0 0	0 0 0 0 0	0 0 0	0 0 0	0	0 0 0 ... 0
SWI	0 0 0 0	0 0 0 1 0	0 0 0	[AREG]	1	[IMOP16]
GETSWI	0 0 0 0	0 0 0 1 1	[WREG]	0 0 0	1	0 0 0 ... 0 i1
IN	0 0 1 0	0 1 1 1 0	[WREG]	0 0 0	1	[IMOP16]
OUT	0 0 1 1	0 1 1 1 0	[BREG]	0 0 0	1	[IMOP16]
ENI	0 0 0 1	0 0 0 0 0	0 0 0	0 0 0	0	0 0 0 ... 0
DEI	0 1 0 0	0 0 0 0 0	0 0 0	0 0 0	0	0 0 0 ... 0
BNEZ	0 1 0 1	0 1 1 0 0	0 0 0	[AREG]	1	x x x x [IMOP12]
BEQZ	0 1 1 0	0 1 1 0 1	0 0 0	[AREG]	1	x x x x [IMOP12]
BOV	0 1 1 1	0 1 1 1 0	0 0 0	0 0 0	1	x x x x [IMOP12]
LOAD	1 0 0 0	1 0 0 0 1	[WREG]	[AREG]	1	[IMOP16]
STORE	1 0 0 1	1 0 0 0 1	[BREG]	[AREG]	1	[IMOP16]
JAL	1 0 1 0	0 1 1 1 0	[WREG]	0 0 0	1	x x x x [IMOP12]
JREG	1 0 1 1	0 0 1 1 0	0 0 0	[AREG]	0	0 0 0 ... 0
RETI	1 1 0 0	0 0 0 0 0	0 0 0	0 0 0	0	0 0 0 ... 0
SISA	1 1 0 1	0 1 1 1 0	0 0 0	0 0 0	1	i1 i2 0 0 [IMOP12]
DPMA	1 1 1 0	0 0 0 0 0	0 0 0	0 0 0	0	0 0 0 ... 0
EPMA	1 1 1 1	0 0 0 0 0	0 0 0	0 0 0	0	0 0 0 ... 0

For a better overview, the next table shows, the op-codes for the ALU located in the DATAPATH for all instructions. Further, it shows the bits 27..23 from the previous table for the ALU operations.

Op-code Name	OPC(4...0)	Instructions
NOP	00000	NOP, ENI, DEI, RETI, DPMA, EPMA
SWI	00010	SWI
GETSWI	00011	GETSWI
SHL	00100	SHL, SHLI
SHR	00101	SHR, SHRI
CSHL	00110	CSHL, CSHLI, JREG
CSHR	00111	CSHR, CSHRI
AND	01000	AND, ANDI
OR	01001	OR, ORI
XOR	01010	XOR, XORI
XNOR	01011	XNOR, XNORI
BNEZ	01100	BNEZ
BEQZ	01101	BEQZ
PassImmed	01110	BOV, JAL, SISA, IN, OUT
SUB	10000	SUB, SUBI
ADD	10001	ADD, ADDI, LOAD, STORE
SETOV	10010	SETOV, SETOVI
GETOV	10011	GETOV
MUL	10100	MUL, MULI
Sxx	11**LGE**	Sxx, SxxI

xx	LGE
NE	000
EQ	001
GT	010
GE	011
LT	100
LE	101

7.3 *Implementation Hint*

The OPC output-port provides the commands for the ALU, which is a sub-module of the data path (DATAPATH). The information can be directly taken from the bits 27..23.

The PCCONTR port provides control signals for the DATAPATH and PCBLOCK modules. This information is used to control jumps, branches and interrupts, thus influencing the program counter. The next table shows, which bit must be set by the presence of each of these instructions. The bit 10 will be set

if a relative address calculation has to be performed. This is the case, when one of the instructions `JAL`, `SISA`, `BOV`, `BEQZ`, `BNEZ` is executed.

PCCONTR:

Bit	10	9	8	7	6	5	4	3	2	1	0
Instruction	*REL_ADD*	SWI	JAL	JREG	RETI	ENI	DEI	SISA	BOV	BEQZ	BNEZ

In the instruction op-code, one bit (bit 16) is used for the identification of an immediate instruction. The output IVALID indicates if the bit is set. In that case the immediate op-code is used, the IOP will be used in the module DATAPATH instead of the second source register. The IOP is used as a 16-bit or 12-bit immediate operator, depending on the instruction. The bits 15..0 can be directly mapped to the IOP output port. Instruction `SISA` uses the IOP in a different way, but the particular bits are then handled in the PCBLOCK.

The AOPADR contains the address for the register in the register file HAREGS, which defines the register used for the operand A for the DATAPATH. This address is defined in the bits 19..17 in the instruction code.

Like the port AOPADR, BOPADR is used to select the register in the register file, for the operand B. The register file address can be taken directly from bits 15..13 with the exception of the instructions `OUT` and `STORE`. There, the register address is in bits 22..20, because the immediate value is also used in these instructions. In the immediate instructions, the operand B is not used. Therefore, the register output value on the register file will be ignored by the data path and instead uses the immediate value.

The WOPADR defines the address in the register file HAREGS where the result from the operation should be stored. If the result should not be stored into a register, the address has to be set to zero, which is the zero register $r0 \in R$. Instructions that are not using the operation result, with exception of `OUT` and `STORE`, are already optimized to use the zero register $r0 \in R$. Thus, you are allowed to use the bits 22..20 for this output port directly, with the execution of instruction `OUT` and `STORE`.

Additionally, there are some further output ports of the instruction decoder INDEC, which are set as described in the following list:

OUTOP	`OUT` instruction
INOP	`IN` instruction
LOADOP	`LOAD` instruction
STOREOP	`STORE` instruction
DMEMOP	`STORE` or `LOAD` instruction
SELXRES	Communication over XBus (`IN`, `OUT`, `LOAD` or `STORE` instruction)
DPMA	`DPMA` instruction
EPMA	`EPMA` instruction

8 Module: Register file

The register file is used to temporary store values inside the CPU. The registers accelerate the execution of the instruction, because it eliminates or at least reduces read and write operations with the slower data memory.

8.1 Interface

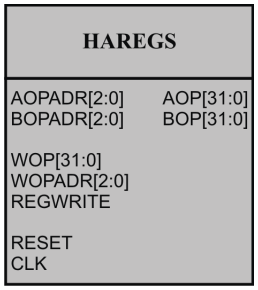


Figure 8.1: Register file interface.

Pin	Type	Source/Destination	Description
AOPADR(2..0)	I	INDEC	Register address to read on output AOP
BOPADR(2..0)	I	INDEC	Register address to read on output BOP
WOPADR(2..0)	I	INDEC	Register address to write
WOP(31..0)	I	DATAPATH	Value to write
REGWRITE	I	CONTROL	Register write signal
AOP(31..0)	O	DATAPATH	Output A
BOP(31..0)	O	DATAPATH, PMEMORY	Output B

8.2 Functionality

The registers of the processor are used to temporarily cache results, for indexing addresses for memory access, for passing parameters, and for the return value of a function call. The assembler uses a three-address-code, thus the instructions can use up to two source registers and one destination register. The register file offers eight General Purpose Registers (GPRs) with a width of 32 bits each. The zero register $r0$ cannot be overwritten and consequently always returns the value zero.

The value on the input WOP is stored in the register defined with the address WOPADR if the write signal REGWRITE is active on a rising edge of the clock (CLK↑). Thus, the update of a register's value is synchronous. However, for the two operands A and B the read commands must be performed asynchronously.

The RESET signal asynchronously clears all registers (set them to zero).

9 Module: Control Unit

The control unit controls other processor modules based on the state of execution of the current instruction. It is implemented as a finite state machine.

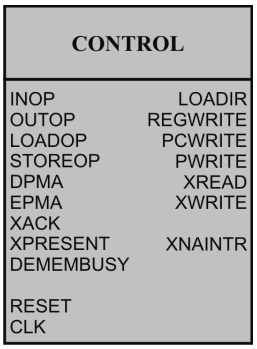


Figure 9.1: Control interface.

9.1 Interface

Pin	Type	Source/Destination	Description
INOP	I	INDEC	`IN` instruction
OUTOP	I	INDEC	`OUT` instruction
LOADOP	I	INDEC	`LOAD` instruction
STOREOP	I	INDEC	`STORE` instruction
DPMA	I	INDEC	`DPMA` instruction
EPMA	I	INDEC	`EPMA` instruction
XACK	I	XBUS	Acknowledgment from XBus component
XPRESENT	I	XBUS	XBus component presented
DMEMBUSY	I	XBUS	XDMemory busy
LOADIR	O	PMEMORY	Load next instruction code
REGWRITE	O	HAREGS, DATAPATH	Write to register
PCWRITE	O	PCBLOCK	Update program counter
PWRITE	O	PMEMORY	Write to program memory
XREAD	O	XBUS	Read at XBus
XWRITE	O	XBUS	Write at XBus
XNAINTR	O	PCBLOCK	Raise interrupt XBusNotAvailable

9.2 Functionality

Figure 9.2 shows the finite state machine of the module CONTROL. This state machine contains 9 states: IFETCH (instruction fetch), IDECODE (instruction decode), ALU (ALU operation), IOREAD and IOWRITE (XBus operations), XBUSNAINTR (XBus peripheral does not exist), MEMREAD and MEMWRITE (program or data memory access), and WRITEBACK (increment the program counter and write ALU result into register).

To fetch the next instruction, the LOADIR port must be set. Afterwards, in the IDECODE state, the

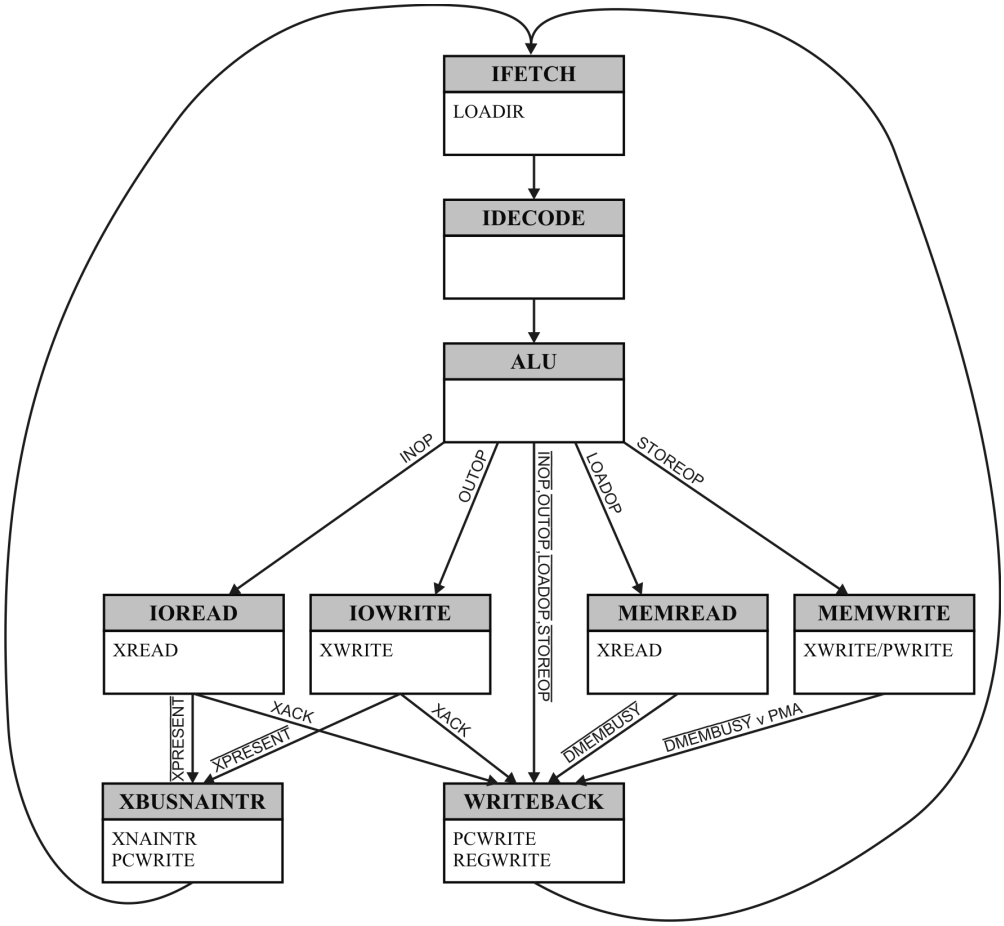


Figure 9.2: Control state automata.

instruction decoder decodes the instruction, sets the control signals for other components, and sets the operand A, B or the immediate operand for the datapath. In the next state, ALU, the datapath which contains the ALU executes the operation; and depending on the four signals INOP, OUTOP, LOADOP or STOREOP from INDEC, branches to the next state:

IOREAD The immediate operand of the `IN` instruction contains the address of the peripheral on the XBus. At IOREAD state, the processor signals the peripheral (with the XREAD signal) that a read operation will be performed. The peripheral recognizes that it is one of its register addresses and immediately sets the XPRESENT signal. If no peripheral is available which uses this address, XPRESENT will not be set. This missing signal is recognized by the control unit raising a XBUSINTR interrupt (by setting the XNAINTR signal). Otherwise, the peripheral offers the requested values (possibly needs some cycles). The operation is finished, when the peripheral sets the XACK signal and then the control unit changes into the WRITEBACK state. Thus, the read value from the peripheral is going to be written into the defined register by using the REGWRITE signal. Also, the PCWRITE is set in this state, which proceeds the Program Counter (PC) to execute the next instruction.

IOWRITE The control unit branches to the state IOWRITE by detecting the `OUT` instruction. There the execution works like the `IN` instruction, but, instead of using the XREAD, the control unit uses the XWRITE signal. This triggers the start of a write command to the peripheral. It is important that the address and the value to be saved are already stable on the peripheral bus. In the WRITEBACK state, the PCWRITE and REGWRITE signal will also be set, but the zero register $r0$ is used as destination, which cannot be overwritten anyways.

MEMREAD The MEMREAD state is reached if the `LOAD` instruction is decoded. The data memory is located on the XBus, therefore also the XREAD signal is set to trigger a read operation. The synchro-

nization process for the data memory is similar to `IOREAD`; here, the state has only to wait for the end of the data access signaled by DMEMBUSY. Finally in the WRITEBACK state, the value will be saved into the register by using the REGWRITE signal. The PCBLOCK will also set the PC to the next instruction with the help of the PCWRITE signal.

MEMWRITE The instruction STORE is very similar to the instruction `LOAD`. However, the result of the ALU will be written to the non-overwritable register zero $r0$. Further, this instruction works not only on the data memory, but also on the program memory. This has to be distinguished in two states, depending off the PMA configuration:

a) Program memory access (PMA set) The writing to the program memory is finished after one cycle, therefore the state machine goes immediately to the next step. To trigger a write, the control unit has to set the PWRITE signal.

b) Data memory access (PMA cleard) The writing access is triggered with the XWRITE signal. The data access could needs some cycles; thus, the end of the operation is signaled by a cleared DMEMBUSY signal from the peripheral. Then the state machine will change to the `WRITEBACK` state.

PMA logic

- If RESET is high, the internal PMA is also high.
- If the input DPMA is set, the PMA flag must be cleared at the next rising clock edge.
- If the input EPMA is set, the PMA flag must be set at the next rising clock edge.
- `PWRITE` must be set, if
  - the state machine is in `MEMWRITE` state (because a `STORE` instruction is used),
  - and the PMA flag is set.
- `XWRITE` must be set, if
  - a `STORE` instruction is used and the PMA flag is cleared,
  - or an `OUT` instruction is used (PMA flag has no influence).
- XWRITE and PWRITE are only set in the states depicted in Figure 9.2, otherwise they are 0.

All other instructions have no communication with the XBus or memory; therefore, all operations are already finished after the ALU operation. The remaining actions are: saving the result into the register with the REGWRITE signal and setting the PC accordingly (by setting the PCWRITE signal). If the result from the ALU is not required, it will be saved into the non-writable register zero $r0$.

10 Module: Datapath

The datapath is one of the most important modules in the HaDes CPU, but it is not so complex to implement. The main part of the datapath is the ALU, which performs the calculations. Further multiplexers in the datapath are used to propagate the corresponding values, depending on the current operation code, to the output ports. Additionally, it uses some registers to buffer values for multi-cycle instructions.

10.1 Interface

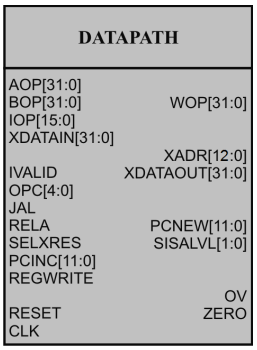


Figure 10.1: Datapath interface.

Pin	Type	Source/Destination	Description
AOP(31..0)	I	HAREGS	Operand A
BOP(31..0)	I	HAREGS	Operand B
IOP(15..0)	I	INDEC	Immediate operand
XDATAIN(31..0)	I	XBUS	XBus data input
IVALID	I	INDEC	Immediate operand validity
OPC(4..0)	I	INDEC	ALU opcode
JAL	I	INDEC	`JAL` instruction (PCCONTR(8)) is set
RELA	I	INDEC	Relative jump instruction, (PCCONTR(10)) is set
SELXRES	I	INDEC	Data is from XBus
PCINC(11..0)	I	PCBLOCK	Current PC + 1
REGWRITE	I	CONTROL	Store into register
WOP(31..0)	O	HAREGS	Result operand
XADR(12..0)	O	PMEMORY, XBUS	XBus address
XDATAOUT(31..0)	O	XBUS	Databus output
PCNEW(11..0)	O	PCBLOCK	New PC
SISALVL(1..0)	O	PCBLOCK	Level for `SISA` instruction
OV	O	PCBLOCK	Overflow (OV) flag
ZERO	O	PCBLOCK	ZERO flag (changed by `BEQZ`,`BNEZ`)

10.2 Functionality

10.2.1 ALU

Figure 10.2 gives an overview of the Arithmetic Logical Unit (ALU). It provides two 32-bit input channels, ACHANNEL and BCHANNEL, and an OPCODE input. The OPCODE defines the operation and outputs the result to the RESULT port. Two additional status flags, ZERO and OVERFLOW, are also generated.

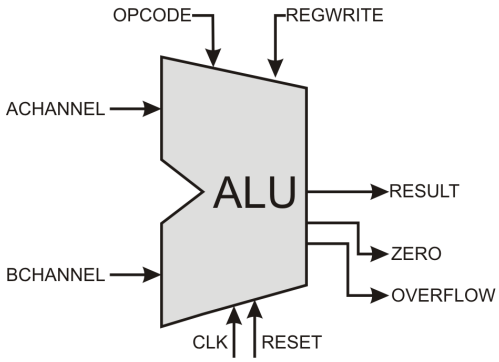


Figure 10.2: ALU interface.

10.2.2 Datapath

Figure 10.3 depicts the overall structure of the datapath module. The main part of this module is the ALU. Additional registers are used to control the input signals for the ALU and the output signals for other modules.

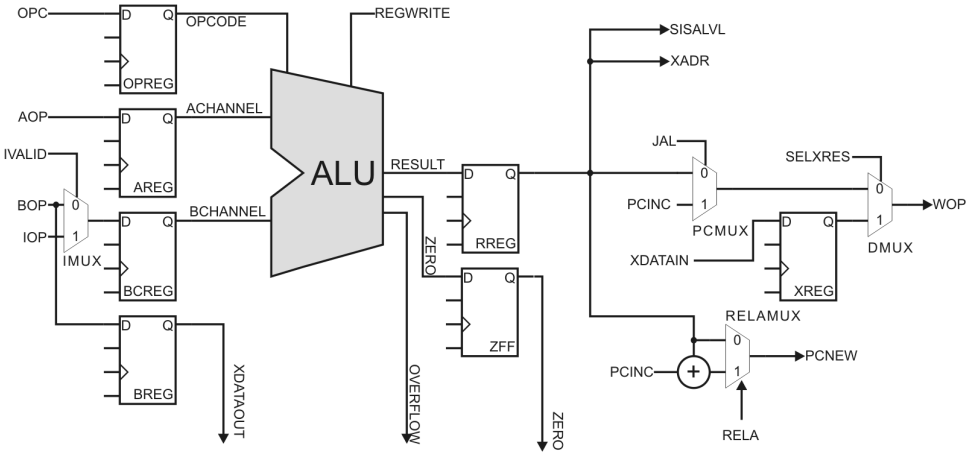


Figure 10.3: Datapath schematics.

10.3 Implementation Hints

10.3.1 ALU

For implementing the ALU you have to use some provided components from the library `work`:

General Each of these provided components uses a generic parameter `N`. It defines the data width.

hades_addsub (`ADD(I)`, `SUB(I)`) The signed input values are added or subtracted:

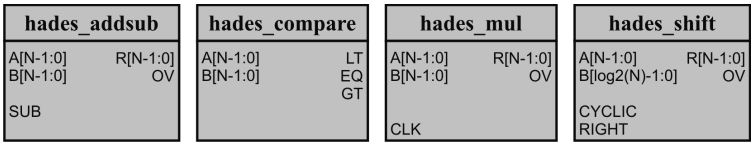


Figure 10.4: Used ALU libraries.

- `SUB = '0'`: Sum of inputs `A` and `B` is applied to output `R`
- `SUB = '1'`: Difference of input `A` and `B` is applied to output `R`
- If the calculated result is too large for output `R`, OV flag will be set.

hades_compare (`SLT, SGT, SLE, SGE, SEQ, SNE`) The signed input values are compared:

- `LT = '1'`: A is less than B
- `EQ = '1'`: A is equal to B
- `GT = '1'`: A greater than B

hades_mul (`MUL(I)`) Multiplication of two signed values:

- Calculates the product of input `A` and `B`.
- If the calculated result is too large for output `R`, OV flag will be set. Since the calculation of this flag is very time-consuming, an additional cycle is required to calculate it. Therefore, the OV flag clear or set is one cycle delayed (multi-cycle instruction).

hades_shift (`SHL(I), SHR(I), CSHL(I), CSHR(I)`) Bit-wise shift of input `A`:

- Output `R` contains the input value `A`, shifted `B` times.
- The `CYCLIC` input signal indicates if a cyclic ('1') or a non-cyclic ('0') shift must be performed.
- To shift the signal to right or left, the `RIGHT` signal has to be set or cleared, respectively.
- If in a non-cyclic operation (`CYCLIC='0'`) a set bit is shifted out, OV flag will be set.

Alternatively, the arithmetic operation can also be implemented by using the Standard Package `ieee.numeric_std`. For logical operations, VHDL offers the Standard Packages `ieee.std_logic_1164`. This allows to easily implement the operators AND, OR, XOR and XNOR by using the VHDL operators `and`, `or`, `xor`, and `xnor`, respectively.

The ALU generates three different signals: `RESULT, ZERO, OVERFLOW`. The next table shows the instructions, which actively use the ALU to produce output values. For all other instructions (e.g., `OPCODE='00000'`), the signals are set to zero.

Instruction	Opcode
SWI	00010
GETSWI	00011
SHL	00100
SHR	00101
CSHL	00110
CSHR	00111
AND	01000
OR	01001
XOR	01010
XNOR	01011
BNEZ	01100
BEQZ	01101
PassImmed	01110

Instruction	Opcode
SUB	10000
ADD	10001
SETOV	10010
GETOV	10011
MUL	10100
SNE	11000
SEQ	11001
SGT	11010
SGE	11011
SLT	11100
SLE	11101

- The ZERO flag is only used for the instructions BNEZ and BEQZ. This flag will be set by the hades_compare component, if the value at ACHANNEL is zero.
- The OV flag is generated by the instructions ADD, SUB, MUL, SHR, SHL, SETOV, and the immediate variants of those instruction. Except for the SETOV instruction, the OV flag is generated by the provided calculation components. If one of the instructions generates an OV flag, it has to be saved into a flip-flop OVFF. Usually, this would happen at an enabled REGWRITE. However, because of the time-consuming OV flag generation in the component hades_mul, the OVFF has to take over the corresponding OV flag one cycle later. Therefore, you have to store the OV flag one cycle after the REGWRITE (Hint: clocked process).
- If a PassImmed or a branch (BNEZ or BEQZ) instruction has to be performed, the immediate operator will be passed directly through the ALU. Thus, the input BCHANNEL is set to the output port RESULT, in which only the 16 Least Significant Bits (LSBs) are used.
- For the hades_shifter component, the 5 LSBs from the BCHANNEL are the shift distance.
- All arithmetic operations (ADD(I), SUB(I), MUL) are interpreted as signed.
- The hades_compare component sets the LSB of the output RESULT depending on the *Set-Condition* instruction. All other bits are cleared.
- For the instruction GETOV, the LSB shows the status of OVFF. Similar to the *Set-Condition* instructions, the other 31 Most Significant Bits (MSBs) are cleared.
- The instruction SWI internally saves the two arguments, if the REGWRITE signal is set:
  - SWIAFF = ACHANNEL
  - SWIBFF = BCHANNEL
  - RESULT is set to zero
- Instruction GETSWI sets, depending on the LSB in BCHANNEL, the value from one of the software interrupt arguments:
  - RESULT = SWIBFF if BCHANNEL(0) = '0'
  - RESULT = SWIAFF if BCHANNEL(0) = '1'

10.3.2 *Datapath*

- To separate the combinationally implemented instruction decoder INDEC (Section 7) and the ALU (Section 10.2.1), the input operands must be stored into a flip-flop. The value is updated at a rising edge of the clock and is asynchronously reset.
- The op-code and the operand A are stored into the registers OPREG and AREG, respectively. These signals are connected directly to the ALU. The operand B cannot be directly connected to the ALU. Thus, the IVALID signal defines the second source for the ALU. If the signal IVALID is set, the immediate operator will be used, otherwise the operand B will be used. For arithmetic operations like ADDI (as well as instructions which use the addition immediate like LOAD and STORE), SUBI, MULI, and *immediate Set-Condition* instructions, the immediate operator has to be expanded to a signed value. All other instructions interpret the immediate operand as unsigned.
- The output OVERFLOW is directly connected to the output port of the datapath module. However, all other output signals from the ALU must be buffered in a register.
- The ALU produces different kinds of results, which are used by other components depending on the instruction. One output signal is XADR, where the 12 LSBs are taken directly from the ALU result register. The remaining 20 bits are OR-ed together and used as a 13th bit, which is used to detect an invalid memory addresses and to raise an interrupt (XMEMINTR) for this invalid state. To easily implement the OR-ing, you can use the comparison with zero (VHDL operator /=).

- For `SISA` instructions, the level for setting the interrupt handler address is located in bits 14 and 15 of ALU's result. These bits are directly handed over to the SISALVL output.
- The `JAL` instruction is used to call a function: it jumps to a label and saves the return address, which is the next address of the current PC (PC+1). Thus, the multiplexer propagates the return address to the output WOP for saving it into a register in the state `WRITEBACK` of the control unit.
- Unless the `JAL` instruction is used, the output value of the datapath depends on the SELXRES input signal. This distinguishes between internal data from the ALU and external data read from the external bus (XBus).
- A further function of the datapath is the calculation of the destination addresses of jumps. It has to distinguish between absolute jumps (`JREG`) and relative jumps from the current PC. Depending on the RELA input signal the PCNEW is the result of the 12 LSBs from ALU's result, or the sum of that value and the PCINC input signal. Listing 10.1 shows two possibilities to implement an unsigned addition with the VHDL Standard Package `ieee.numeric_std`.

```
library ieee;
use ieee.numeric_std.all;
signal a : std_logic_vector(11 downto 0);
signal b : std_logic_vector(11 downto 0);
signal c : std_logic_vector(11 downto 0);
-- ...
c <= std_logic_vector(unsigned(a)+unsigned(b));
-- ...
```

(a) Unsigned addition with `std_logic_vector` variables.

```
library ieee;
use ieee.numeric_std.all;
signal a : unsigned(11 downto 0);
signal b : unsigned(11 downto 0);
signal c : unsigned(11 downto 0);
-- ...
c <= a + b;
-- ...
```

(b) Unsigned addition with `unsigned` variables.

Listing 10.1: Unsigned addition in VHDL.

11 Module: Program Counter

The Program Counter (PCBLOCK) is responsible for the calculation of the next instruction address. Since interrupts also influence the program flow, this module also handles interrupts. In normal operation, the module produces the address of the next instruction, but in case of the `JAL` instruction, it also returns the return address for the Datapath to store it in the Register File.

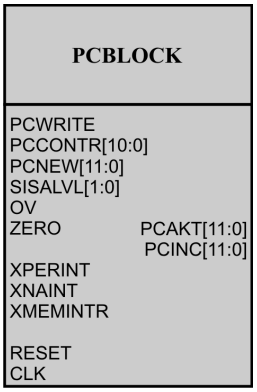


Figure 11.1: Program counter block interface.

Pin	Type	Source/Destination	Description
PCWRITE	I	CONTROL	overwrite PC
PCCONTR(10..0)	I	INDEC	PC control flags
PCNEW(11..0)	I	DATAPATH	new PC
SISALVL(1..0)	I	DATAPATH	interrupt level for SISA instruction
OV	I	DATAPATH(ALU)	OV flag
ZERO	I	DATAPATH(ALU)	ZERO flag
XPERINTR	I	XBUS	raise peripheral interrupt
XNAINTR	I	CONTROL	raise XBusNotAvailable interrupt
XMEMINTR	I	XBUS	raise XMemory interrupt
PCAKT(11..0)	O	PMEMORY	current PC
PCINC(11..0)	O	DATAPATH	next instruction in Memory(PC+1)

11.1 Overview

To program efficient and clear code, the developer has some instructions for branching, jumping and jumping to subroutines. If these concepts were not offered by the CPU, the programmer would have to program only straight sequential code, and the program would not be able to react to external events because no differentiation could be done for each event. Furthermore, sequential code would implement parts of the same code several times, increasing the program memory, reducing the maintainability, and possibly causing other negative effects.

Jumps, branches, and subroutine calls are well known concepts from high level languages like `C` and can be easily mapped to the instruction level. At instruction level, the PC models those behaviors with

defined instructions (e.g. `BEQZ`, `BNEZ`, or `JAL`).

Most of the instructions in the HaDes CPU are PC neutral, which means that the PC is incremented by one word when the current instruction was executed. The relative jump instruction `JMP #k` is one way to manipulate the PC, as it will continue with the instruction at address `PC+1+#k`. Another way to change the PC are branches. The branch instructions `BEQZ` (*branch if equal zero*), `BNEZ` (*branch if not equal zero*), and `BOV` (*branch if overflow*) continue with the relative destination address only if the condition is true, otherwise the PC increments by one word. A further jump instruction is used to call subroutines, the `JAL w, #k` instruction. It saves the incremented PC (PC+1) into register *w*, and the PC is modified to jump to the address `PC+1+#k`, which corresponds to the subroutine. To return from the subroutine, the `JREG w` instruction manipulates the PC to jump to the absolute address in the register *w*.

The PCBLOCK is responsible for managing the PC, which includes the management of interrupts. Interrupts are breaks of the current control flow by an internal or external component to indicate an event. The external interrupts in the HaDes are triggered from peripherals (e.g., press of a button or a byte is received on the serial interface). The CPU also raises internal interrupts to handle internal events like memory interrupts (e.g., invalid memory address), non-valid peripherals (e.g., invalid peripheral address), or software interrupts (e.g., using instruction `SWI`). When the CPU detects an interrupt, the current program flow is interrupted (**after the current instruction**). The next address will be saved, and then an Interrupt Service Routine (ISR) is called. The ISR is part of a code, which handles the interrupt. The instruction `RETI` (*return from interrupt*) leaves the ISR and jumps back to the previously internally saved PC. Section 11.3.1 explains the interrupt handling in more detail.

The PCBLOCK manages all the mentioned functionalities of the PC. To implement all features of this module, we will use a *divide & conquer* approach to simplify the implementation by splitting the functionalities into sub modules. Figure 11.2 shows the PCBLOCK with all its sub modules.

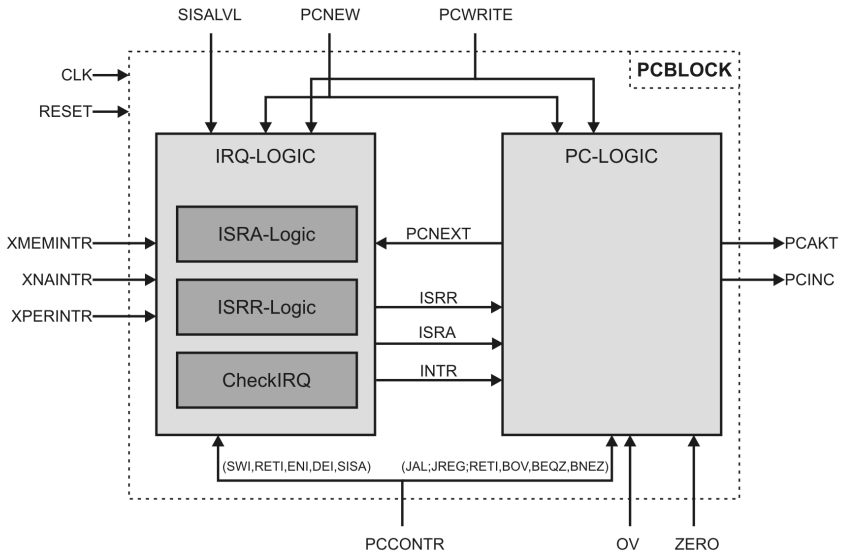


Figure 11.2: Program counter block structure.

PCLOGIC This sub-module implements the PC and calculates the next instruction address. It puts the address of the next instruction on the output port `PCNEXT`. The next instruction address can be calculated in four different ways:

PC=PC+1 The PC is incremented by one.

PC=PCNEW Jump or branch (`JAL`, `JMP`, `JREG`, `BOV`, `BEQZ` or `BNEZ`). PCNEW is taken from datapath.

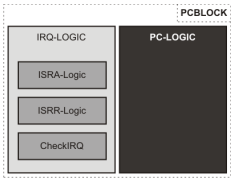
PC=ISRA The `ISRA` (*Interrupt Service Routine Address*) is the address of the routine used to handle the ISR.

PC=ISRR The `ISRR` (*Interrupt Service Routine Return Address*) is the return address from the interrupt handler by using the `RETI` instruction.

Based on the flags INTR, OV, `ZERO` alongside `JAL`, `JREG`, `RETI`, `BOV`, `BEQZ`, and `BNEZ` the sub module decides the value of the outputs `PCNEXT` and `PCAKT`.

IRQ-LOGIC The IRQ-LOGIC handles interrupts. It recognizes interrupts and passes the `ISRA` and `ISRR` addresses to the PCLOGIC. Further, it shows with the signal `INTR` that the program flow has been interrupted and an ISR will be called.

11.2 *Program Counter Logic*



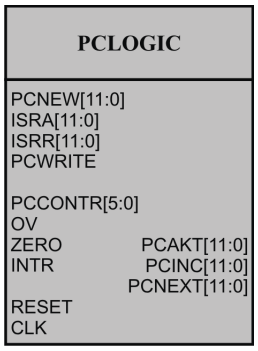


Figure 11.3: Program counter interface.

Pin	Type	Source/Destination	Description
PCNEW(11..0)	I	DATAPATH	jump destination
ISRA(11..0)	I	IRQLOGIC	jump entry address for ISR
ISRR(11..0)	I	IRQLOGIC	return address from ISR
PCWRITE	I	CONTROL	write PC control
PCCONTR(5..0)	I	INDEC	control flags
OV	I	DATAPATH(ALU)	OV flag
ZERO	I	DATAPATH(ALU)	ZERO flag
INTR	I	IRQLOGIC	interrupt triggered
PCAKT(11..0)	O	PMEMORY	current PC
PCINC(11..0)	O	DATAPATH	incremented current PC (PC+1)
PCNEXT(11..0)	O	IRQLOGIC	next PC without interrupt

11.2.1 *Functionality*

The module PCLOGIC contains a 12-bit register PCREG to save the current PC, an adder to calculate PC+1, multiplexers to select the new PC, and a logic element to control the multiplexer MUXSEL. Figure 11.4 shows the structure of this module.

PCMUX This multiplexer is responsible for selecting the correct next address from one of the three addresses `PCINC` (default), `PCNEW` (jump or branch), or `ISRR` (`RETI` instruction, if **no** interrupt occurs). The signal is handed over to the IRQ-LOGIC to save the return address of an Interrupt Request (IRQ). Further, the output is used as an input for the multiplexer IRQMUX. The signals `PCCONTR` (`BOV`, `BEQZ`, `BNEZ`, `JAL`, `JREG` and `RETI`), OV and ZERO flags are used to select the correct address. The input `ISRR` is selected, if the signal `RETI` in the `PCCONTR` is set. The PCNEW address will be selected if a branch or jump is selected, which is the case when the following expression holds:

```
(BOV and OV) or (BEQZ and ZERO) or (BNEZ and (not ZERO)) or JAL or JREG
```

Otherwise PCINC is selected which is the current PC plus one (PC+1).

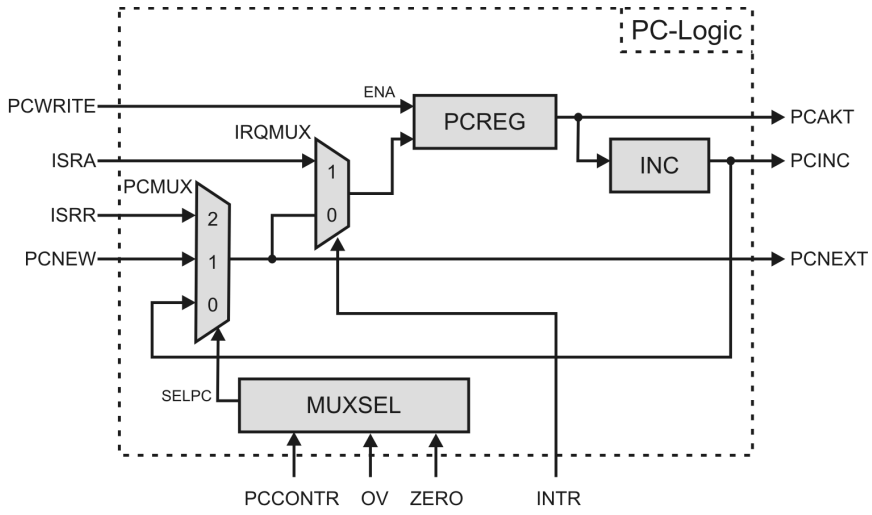


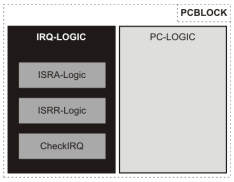
Figure 11.4: PCLOGIC structure.

IRQMUX This multiplexer handles the PC according to the current interrupt state. If no interrupt is active, the address from the `PCMUX` is used. Otherwise the address at the input ISRA, which is generated by the IRQ-LOGIC is used.

The next Table summarizes the calculated output addresses:

Case	**INTR**	**JAL, JREG**	**SELPC**	**RETI**	**PCNEXT**	**PCREG** Input	**PCAKT**	**PCINC**
Default	0	0	0	0	PC+1	PC+1	PC	PC+1
Branch	0	0	1	0	PCNEW	PCNEW	PC	PC+1
JAL, JREG	0	1	1	0	PCNEW	PCNEW	PC	PC+1
RETI	0	0	2	1	ISRR	ISRR	PC	PC+1
INTR+Default	1	0	0	0	PC+1	ISRA	PC	PC+1
INTR+Branch	1	0	1	0	PCNEW	ISRA	PC	PC+1
INTR+JAL	1	1	1	0	PCNEW	ISRA	PC	PC+1
INTR+RETI	1	0	2	1	ISRR	ISRA	PC	PC+1

11.3 Interrupt Logic



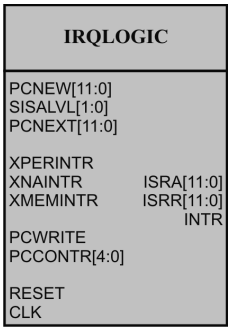


Figure 11.5: Interrupt interface.

Pin	Type	Source/Destination	Description
PCNEW(11..0)	I	DATAPATH	address for `SISA`
SISALVL(1..0)	I	DATAPATH	interrupt level for `SISA`
PCNEXT(11..0)	I	PCLOGIC	return address from ISR
XPERINTR	I	XBUS	peripheral interrupt
XNAINTR	I	CONTROL	XBusNotAvailable interrupt
XMEMINTR	I	XBUS	XMemory interrupt
PCWRITE	I	CONTROL	save data into internal register
PCCONTR(4..0)	I	INDEC	control flags
INTR	O	PCLOGIC	jump into ISR
ISRA(11..0)	O	PCLOGIC	current ISR level jump address
ISRR(11..0)	O	PCLOGIC	current ISR level return address

11.3.1 Overview of interrupt handling

Priority Interrupt System The HaDes CPU supports a priority based interrupt system with five levels:

Level	Description
0	No interrupt: default
1	software interrupt: `SWI` instruction triggers interrupt
2	peripheral interrupt: e.g., button pressed, XUART, XTimerXT
3	XBusNotAvailable interrupt: peripheral address does not exist
4	XMemory interrupt: invalid memory access

We define interrupt levels as $k, l \in \{0, 1, 2, 3, 4\}$. If the processor is running at the interrupt level k, and another interrupt with higher-level $l > k$ is triggered, then the current program flow or ISR will be interrupted and the PC will continue to handle the interrupt l in the corresponding ISR. On a `RETI` instruction, the program is resumed at the last interruption position, and the current level is set back to the level k as long as no higher interrupts as k have been triggered. Thus, an ISR can be interrupted only from higher prioritized interrupts. This interruption can be restricted by disabling interrupts. To disable and enable interrupts, the instruction `DEI` (*disable interrupt*) and `ENI` (*enable interrupt*) are used, respectively. When interrupts are raised while interrupts are disabled, the CPU will save them and as soon as interrupts are re-enabled, they will be handled.

Interrupt Service Routine An Interrupt Service Routine (ISR) makes it possible to immediately react to an event. An interrupt could be triggered by pressing a button, by receiving a value at the communication interface, and many other events. Thus, ISRs are used to handle events which occurrences are unpredictable. Therefore, interrupts are used to react to an *asynchronous* event. The job of the interrupt logic is to detect an interrupt, to store it if interrupts are disabled or with lower interrupt level, and to

jump to the predefined ISR. Before jumping to the ISR, the CPU needs to finish the currently executed instruction.

After calling an ISR, the programmer has the possibility to return immediately from the ISR with the instruction `RETI` or to handle the interrupt before returning with `RETI`. Take notice that the GPRs are not automatically saved nor restored on ISR calls. This must be done manually after calling the ISR and before returning with `RETI`, in case that the GPRs are used in the ISR. The peripheral interrupt (level 2) may be raised by many different peripherals; therefore, the programmer must query the status of each peripheral to detect, which one triggered the interrupt. Some peripherals need a manual clearing of the event; otherwise, the peripheral would trigger the interrupt over and over again.

In the default interrupt level 0 (normal program execution), executing the `RETI` instruction produces a PC reset and the CPU restarts at the first address in the program memory.

The instruction `SISA #j,#k` defines the ISR entry address of the corresponding interrupt. The constant $\#j \in 0,1,2,3$ selects the interrupt and the constant $\#k$ defines the entry address, which is the relative offset of the current program counter plus one (PC+#k+1). E.g., `SISA #1, #300` sets the address of the peripheral interrupt to PC+1+300:

- $\#j = 0$: Software interrupt
- $\#j = 1$: Peripherals interrupt
- $\#j = 2$: XBusNotAvailable interrupt
- $\#j = 3$: XMemory interrupt

11.3.2 *Functionality*

The IRQ-LOGIC is divided into three sub-modules to reduce its complexity and is shown in Figure 11.6. The ISRA-Logic is responsible for calculating the entry address of an interrupt. The ISRR-Logic manages the return address from the ISR, and the CheckIRQ checks if the current program flow must be interrupted. The next Sections go into more details for each sub-module.

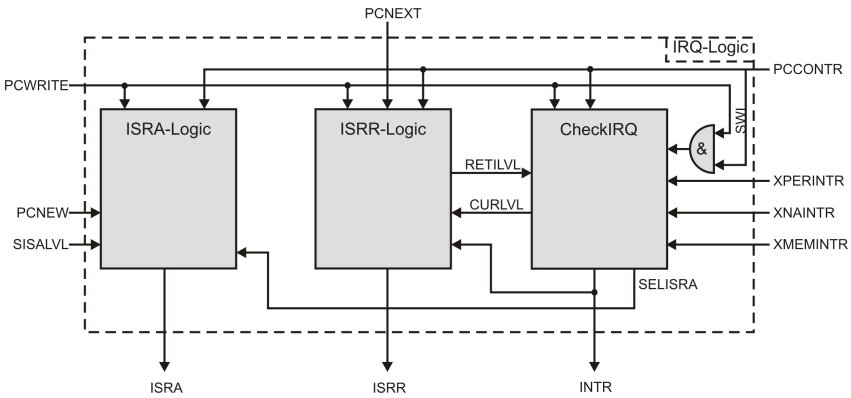
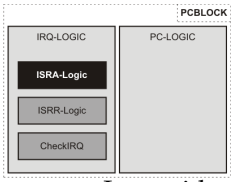


Figure 11.6: Interrupt logic structure.



11.4 ISRA Logic

This sub-module contains four 12-bit registers to store one ISR entry address for each interrupt. It provides the correct entry address at an interrupt for jumping to its corresponding ISR.

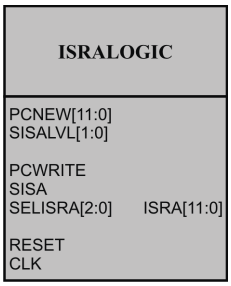


Figure 11.7: ISRA interface.

Pin	Type	Source/Destination	Description
PCNEW(11..0)	I	DATAPATH	address for `SISA`
SISALVL(1..0)	I	DATAPATH	interrupt level for `SISA`
PCWRITE	I	CONTROL	save data into internal registers
SISA (PCCONTR)	I	INDEC	`SISA` instruction
SELISRA(2..0)	I	CHECKIRQ	selected ISRA address
ISRA(11..0)	O	PCLOGIC	current ISR address

11.4.1 Functionality

Figure 11.8 shows the structure of the ISRA-Logic. It contains four 12-bit registers for saving the entry addresses for each interrupt. A multiplexer puts the entry address into the output ISRA. If the default level 0 is active, the output will be set to zero.

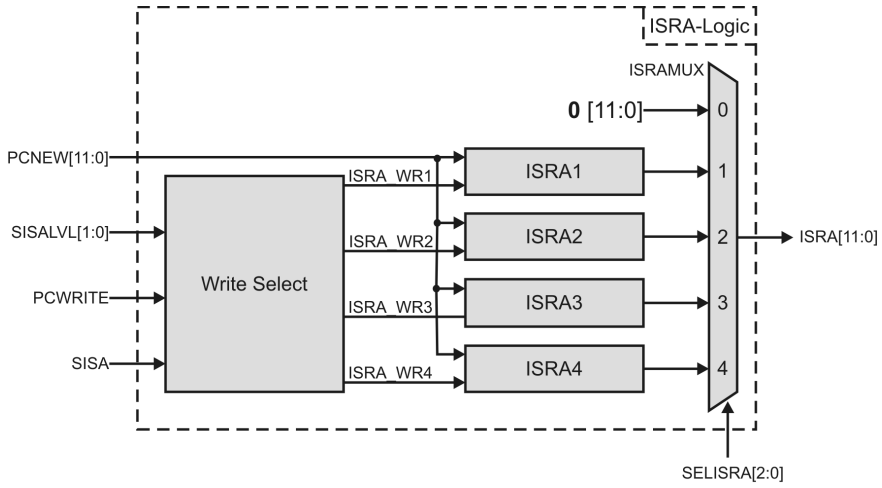
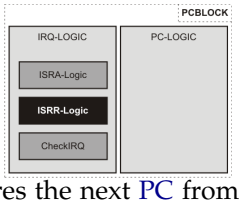


Figure 11.8: ISRA structure.

The `Write Select` box decides, if one of the interrupt entry addresses should be overwritten. After RESET, all addresses are initialized with the address `4095 (0xFFF)`, where also the bootloader is located. On this specific address, the bootloader has a `RETI` instruction that leads to an immediate return from the ISR. The SISALVL input selects, which of the levels should be updated in case the PCWRITE input is set and a `SISA` instruction is active. The following Table summarizes the control write signals depending on the input signals:

PCWRITE	SISA	SISALVL	ISRA_WR1	ISRA_WR2	ISRA_WR3	ISRA_WR4
0	*	**	0	0	0	0
*	0	**	0	0	0	0
1	1	00	1	0	0	0
1	1	01	0	1	0	0
1	1	10	0	0	1	0
1	1	11	0	0	0	1



11.5 ISRR Logic

The ISRR sub-module saves the return address from the ISR. On an interrupt, it stores the next PC from the program flow or from a running lower prioritized ISR. It also manages the interrupt level at a `RETI` instruction.

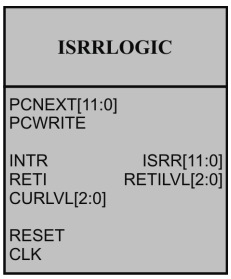


Figure 11.9: ISRR interface.

Pin	Type	Source/Destination	Description
PCNEXT(11..0)	I	PCLOGIC	next PC (for jumping back)
PCWRITE	I	CONTROL	save data into internal register
INTR	I	CHECKIRQ	interrupt raised
RETI (PCCONTR)	I	CONTROL	`RETI` instruction
CURLVL(2..0)	I	CHECKIRQ	current interrupt level
RETILVL(2..0)	O	CHECKIRQ	current return interrupt level
ISRR(11..0)	O	PCLOGIC	current ISR return address

11.5.1 Functionality

The ISRR module has four 15 bit registers to save the return address (lower 12 bits) and the return interrupt level (bits 14, 13 and 12) while an IRQ is active. The four registers are used as a stack. Every item in this stack will be asynchronously set to 4094 (0xFFE) on a reset, as this address contains the instruction `JMP #-1` corresponding to an endless loop. If an interrupt is triggered (INTR input signal is set), the return address and the current interrupt level are pushed onto the stack. Later, the `RETI` instruction pulls these values from the stack. The return address and the return interrupt level at the top of the stack are accessible from the output ports ISRR and RETILVL, respectively. Figure 11.10 gives an overview of the structure of this sub-module.

RETI, INTR and PCWRITE control the DOSHIFT control signal and the four multiplexers. The multiplexers are controlled according to one of the following four cases:

No interrupt, no `RETI`*:* All registers keep their current values.

Interrupt, no `RETI`*:* If PCWRITE is set, the return address and the current interrupt level are stored into ISRR1 and LVL1, respectively, and all other register values are shifted to the next registers: ISRR1 to ISRR2, ISRR2 to ISRR3, ISRR3 to ISRR4, LVL1 to LVL2, LVL2 to LVL3 and LVL3 to LVL4.

`RETI`***, no Interrupt****:* The values in ISRR1 and LVL1 are used by PCLOGIC and CheckIRQ. If PCWRITE is set, the stack executes a pop command and all items in the stack are shifted (left in the Figure). The return address ISRR4 will be set to the default value 4094 (0xFFE).

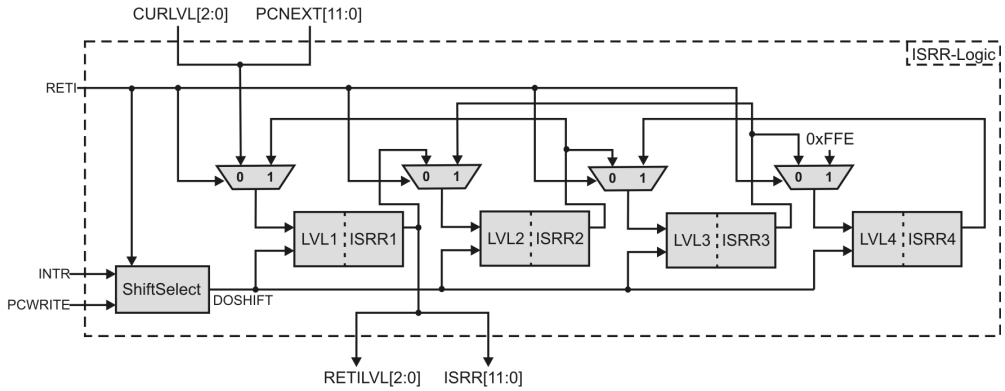
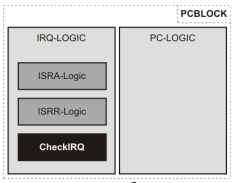


Figure 11.10: ISRR structure.

`RETI` ***and simultaneous interrupt:*** All registers keep their values, as the return address of the current ISR is equal to the return address of the current triggered interrupt.



11.6 CheckIRQ

CheckIRQ saves the current interrupt level and interrupt enable status. It detects interrupts and triggers an interrupt depending on the current interrupt level. Its output signals INTR, CURLVL, and SELISRA control the other sub-modules ISRA, ISRR and PCLOGIC in the PCBLOCK.

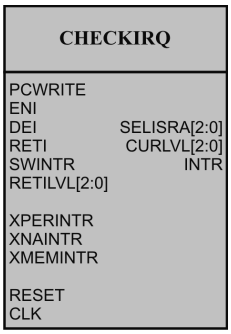


Figure 11.11: CheckIRQ interface.

Pin	Type	Source/Destination	Description
PCWRITE	I	CONTROL	Save data into internal register
ENI	I	INDEC	enable interrupt instruction (PCCONTR(5))
DEI	I	INDEC	disable interrupt instruction (PCCONTR(4))
RETI	I	INDEC	return from ISR instruction (PCCONTR(6))
SWINTR	I	INDEC	software interrupt instruction (PCCONTR(9))
XPERINTR	I	XBUS	peripheral interrupt
XNAINTR	I	CONTROL	XBusNotAvailable interrupt
XMEMINTR	I	XBUS	XMemory interrupt
RETILVL(2..0)	I	ISRRLOGIC	current return interrupt level
SELISRA(2..0)	O	ISRALOGIC	selected ISR address
CURLVL(2..0)	O	ISRRLOGIC	current interrupt level
INTR	O	ISRRLOGIC, PCLOGIC	interrupt raised

11.6.1 Interrupt categories

To detect interrupts, two different interrupt categories exist: *Pulse interrupts* and *level interrupts*. In the HaDes Central Processing Unit (CPU), the interrupts SWINT, XNAINTR and XMEMINTR are pulse interrupts. This means that the trigger for the corresponding interrupt is available only for a short time (few cy-

cles). Therefore, the module has to buffer these triggers. The sub-module (`hadescomponents.irqreceiver`) depicted in Figure 11.12 can be used to buffer the interrupt.

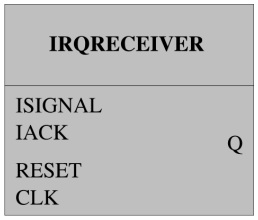


Figure 11.12: IRQRECEIVE interface.

The `ISIGNAL` is sampled with each rising CLK↑ signal. If the `ISIGNAL` is set at least one cycle, the output `Q` will be set to 1 as long as `IACK` does not clear `Q`. Therefore, the IRQRECEIVER buffers the interrupt as long as the interrupt is not acknowledged. The interrupt will be acknowledged if the ISR, from the correct priority, is called. The interrupt signal at the input should only be saved if it is required. This means that it should be buffered if and only if the corresponding ISR is not executed immediately in order to prevent calling the ISR twice. You can also implement the functionality of the IRQRECEIVER directly into the sub-module CHECKIRQ.

The other kind of interrupts are level interrupts, like the XPERINTR. These interrupt signals from the peripheral remain active as long as the peripheral is not acknowledged on its event. For this kind of interrupt, no buffering is required and the peripheral interrupt signal can be used directly.

11.6.2 *Functionality*

The CheckIRQ sub-module is responsible for saving the interrupt enable status and entering an ISR. The ISRss are only called if interrupts are enabled. To enable and disable interrupts, instructions `ENI` and `DEI` can be used, respectively. By default, interrupts are enabled at start-up.

The current interrupt level changes its state whenever a `RETI` instruction is executed or a new ISR is called. This only happens in case interrupts are enabled and the new interrupt is higher prioritized than the current interrupt level. The output signal `CURLVL` is only updated in case the `PCWRITE` signal is set.

Every interrupt has a different priority. Therefore, the highest interrupt has to be handled first. Pay attention that CheckIRQ checks for recently triggered interrupts and buffered interrupts. The next Table summarizes the request levels, depending on the active interrupt:

XMEMBuf or **XMEMINTR**	**XNABuf** or **XNAINTR**	**XPERINTR**	**SWBuf** or **SWINTR**	**REQLVL[2..0]**
0	0	0	0	000 (0)
0	0	0	1	001 (1)
0	0	1	*	010 (2)
0	1	*	*	011 (3)
1	*	*	*	100 (4)

The requested interrupt level `REQLVL` can be compared with the current interrupt level to decide which level will be executed next. In case of a `RETI`, this could lead to an incorrect interrupt priority level handling. An example: The processor is running at interrupt level 0 and an interrupt level 4 is called, concurrently the interrupt level 3 is triggered. The ISR handling interrupt level 4 will be executed and the current interrupt level `CURLVL` is raised to 4. By executing the instruction `RETI`, the current interrupt level CURLVL would be set to 0, because the RETILVL from the ISRRLOGIC is 0. Therefore, it would execute an instruction from the interrupt level 0, instead of the expected interrupt level 3.

Therefore, if the input signal RETI is set, the requested level REQLVL has to be compared with the return level `RETILVL` from the ISRRLOGIC. If the comparison results in a new interrupt trigger(REQLVL

$>$ RETILVL) and interrupts are enabled, the SELISRA and the INTR signal will be set. Otherwise, if RETILVL is greater or equal than REQLVL, we will proceed executing the interrupt at RETILVL without setting the INTR signal.

To guarantee that the signal `INTR` is set during a write phase, the signal has to be AND-ed with the `PCWRITE` signal. The signal INTR is also used to clear the buffered interrupt signals in the sub-module IRQRECEIVER. In the following table some examples of CURLVL, REQLVL and RETILVL combinations are shown:

CURLVL[2..0]	**REQLVL[2..0]**	**RETILVL[2..0]**	**RETI**	**CURLVL^{+1}[2..0]**	**INTR^{+1}**
000 (0)	000 (0)	000 (0)	0	000 (0)	0
000 (0)	011 (3)	000 (0)	0	011 (3)	1
011 (3)	100 (4)	000 (0)	0	100 (4)	1
100 (4)	100 (4)	000 (0)	0	100 (4)	0
100 (4)	100 (4)	001 (1)	1	100 (4)	1
100 (4)	011 (3)	000 (0)	0	100 (4)	0
100 (4)	011 (3)	000 (0)	1	011 (3)	1
100 (4)	011 (3)	011 (3)	1	011 (3)	0

Part II

Hardware Experiment

12 Development Board

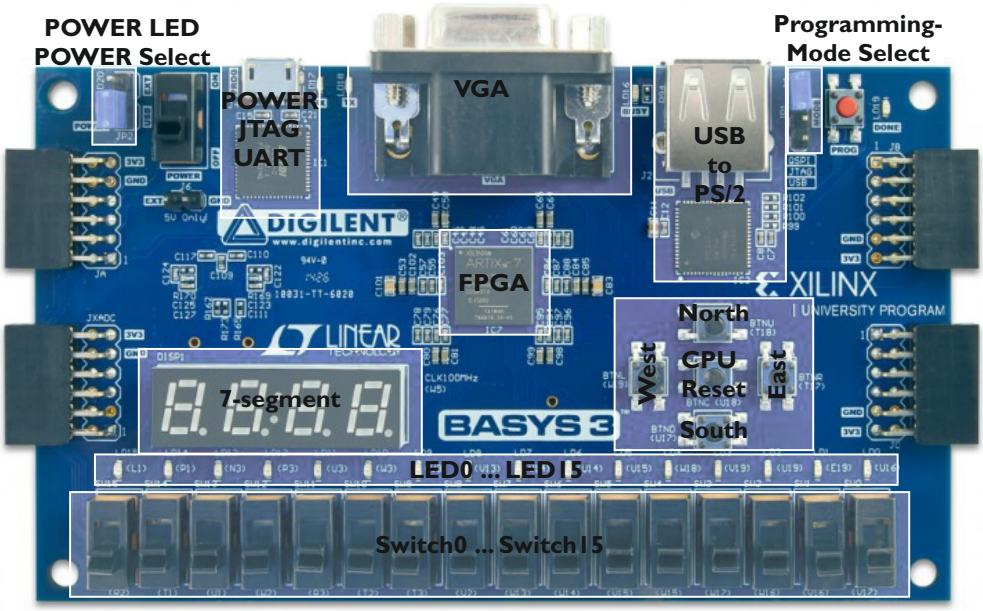


Figure 12.1: The BASYS3 board.

The Basys3 is a complete, ready-to-use digital circuit development platform based on the latest Artix®-7 FPGA from Xilinx®. BASYS3 offers the following ports and peripherals:

- 16 user switches
- 16 user LEDs
- 5 user push buttons
- 4-digit 7-segment display
- 3 Pmod connectors
- Pmod for XAnalog-to-Digital-Converter (ADC) signals
- 12-bit VGA output
- USB-UART bridge
- serial flash
- Digilent USB-JTAG port
- USB HID (USB to PS/2)

Power Supplies The board is powered by the Digilent USB-JTAG port (J4) or an external 5 V power supply. Jumper JP3 determines, which source is used. A power LED (LD20), driven by the "power good" output of the LTC3633 supply, indicates that the supplies are turned on and operating normally.

The USB port delivers enough power for the vast majority of designs. A few demanding applications, including any that drive multiple peripheral boards, might require more power than the USB port can provide. An external power supply can be used by using the external power header (J6) and setting jumper JP2 to "EXT". Power supplies must offer voltage ranging from 4.5 V_{DC} to 5.5 V_{DC} and at least 1 A of current (i.e., at least 5 W of power).

FPGA Configuration After power-on, the Artix®-7 FPGA must be configured before it can perform any functions. The on-board jumper (JP1) selects between three different configurations modes:

1. a computer can use the Digilent USB-JTAG port (J4, labeled "PROG") to configure the FPGA any time the power is on
2. a file stored in the non-volatile serial flash device can be transferred to the FPGA by using the SPI port
3. a configuration file can be transferred from a USB memory stick attached to the USB HID port

Oscillators / Clocks BASYS3 has a single 100 MHz oscillator connected to pin W5 (W5 is a MRCC input on bank 34). The input clock can drive MMCMs or PLLs to generate clocks of various frequencies and with known phase relationships that may be needed throughout a design.

Basic I/O (LEDs, Switches, Buttons) Figure 12.2 shows the basic I/O devices of the BASYS3 board. It includes sixteen individual LEDs, which are anode-connected to the FPGA via 330 Ω resistors and can be switched on with high signals.
The sixteen switches are located at the bottom of the board. They (and the five buttons above) are connected to the FPGA via serial resistors to prevent damage if accidentally defined as output.

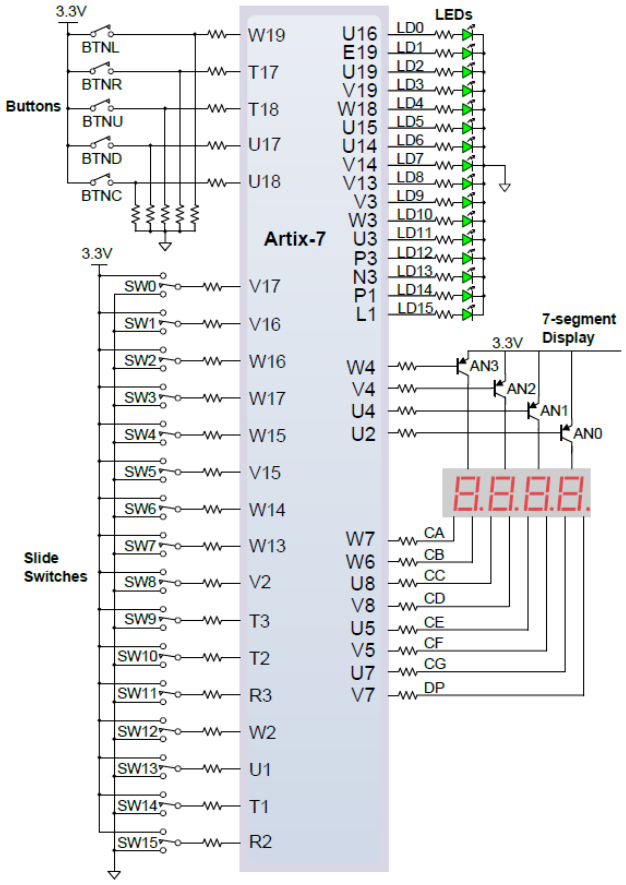


Figure 12.2: Basic I/O devices.

Seven-Segment Display The BASYS3 board provides a four-digit common anode seven-segment LED display. Every digit uses one common anode for every digit. In total, the circuit has four anodes and 7 cathodes for the whole display. The cathodes are shared between all four digits, which results in a multiplexed display design. The update rate between the single digits must be higher than 45 Hz to prevent a flickering display. All four digits should be driven once every 1 to 16 ms (1 kHz to 60 Hz)

USB HID Host The auxiliary function microcontroller (Microchip PIC24FJ128) provides the BASYS3 with USB HID host capability. After power-up, the microcontroller is in configuration mode, either downloading a bitstream to the FPGA or waiting for it to be configured from other sources. Once the FPGA is configured, the microcontroller switches to application mode, which in this case is the USB HID host mode. The firmware in the microcontroller can drive a mouse or a keyboard attached to the type A USB connector at J2 labeled "USB".

VGA Beside the LEDs and the seven-segment display, another visual output is available. The VGA port with 4 bits per color supports up to 4096 different colors. The provided VGA peripheral for the XBus uses only one bit per color and one intensity bit (necessary reduction to save valuable memory). Additional to the color information are two more signals required. One signal for vertical synchronization and one for the horizontal synchronization. The BASYS3 board uses 14 FPGA pins to create the VGA port as shown in Figure 12.3.

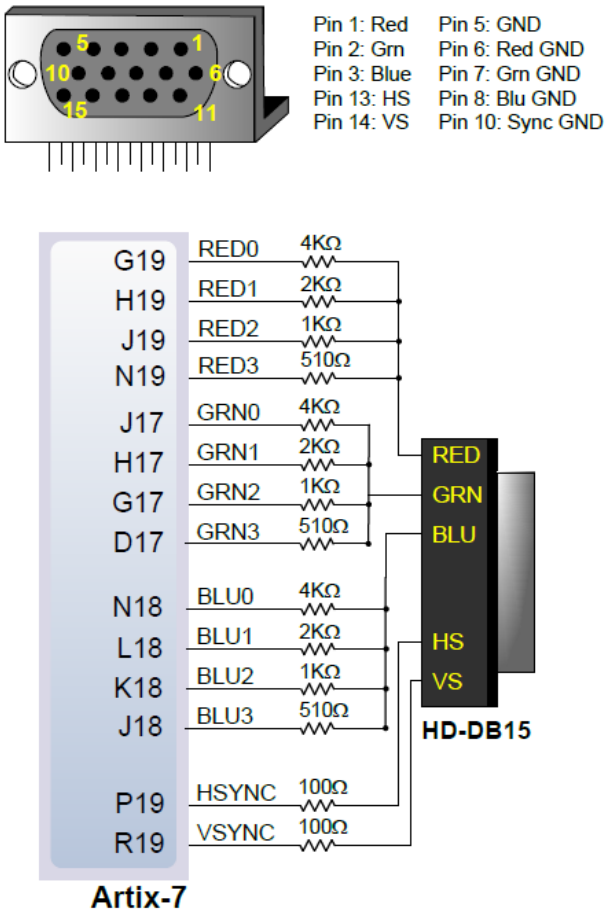


Figure 12.3: Pin configuration for the VGA port

The next Table shows the connected FPGA pins with the VHDL top level ports:

Name	FPGA pin	Type	Connected with	Description
clk	W5	in	xbus_syn.clk	onboard clock (100 MHz)
reset_in	U18	in	xbus_syn.reset_in	button reset
btn_in<0>	T18	in	xbus_syn.btn_in(0)	button north
btn_in<1>	W19	in	xbus_syn.btn_in(1)	button west
btn_in<2>	T17	in	xbus_syn.btn_in(2)	button east
btn_in<3>	U17	in	xbus_syn.btn_in(3)	button south
led_out<0>	U16	out	xbus_syn.led_out(0)	LED output 0
led_out<1>	E19	out	xbus_syn.led_out(1)	LED output 1
led_out<2>	U19	out	xbus_syn.led_out(2)	LED output 2
led_out<3>	V19	out	xbus_syn.led_out(3)	LED output 3
led_out<4>	W18	out	xbus_syn.led_out(4)	LED output 4
led_out<5>	U15	out	xbus_syn.led_out(5)	LED output 5
led_out<6>	U14	out	xbus_syn.led_out(6)	LED output 6
led_out<7>	V14	out	xbus_syn.led_out(7)	LED output 7
led_out<8>	V13	out	xbus_syn.led_out(8)	LED output 8
led_out<9>	V3	out	xbus_syn.led_out(9)	LED output 9
led_out<10>	W3	out	xbus_syn.led_out(10)	LED output 10
led_out<11>	U3	out	xbus_syn.led_out(11)	LED output 11
led_out<12>	P3	out	xbus_syn.led_out(12)	LED output 12
led_out<13>	N3	out	xbus_syn.led_out(13)	LED output 13
led_out<14>	P1	out	xbus_syn.led_out(14)	LED output 14
led_out<15>	L1	out	xbus_syn.led_out(15)	LED output 15
ps2_clk	C17	inout	xbus_syn.ps2_clk	PS/2 Device
ps2_data	B17	inout	xbus_syn.ps2_data	PS/2 Device
swt_in<0>	V17	in	xbus_syn.swt_in(0)	switch 0
swt_in<1>	V16	in	xbus_syn.swt_in(1)	switch 1
swt_in<2>	W16	in	xbus_syn.swt_in(2)	switch 2
swt_in<3>	W17	in	xbus_syn.swt_in(3)	switch 3
swt_in<4>	W15	in	xbus_syn.swt_in(4)	switch 4
swt_in<5>	V15	in	xbus_syn.swt_in(5)	switch 5
swt_in<6>	W14	in	xbus_syn.swt_in(6)	switch 6
swt_in<7>	W13	in	xbus_syn.swt_in(7)	switch 7
swt_in<8>	V2	in	xbus_syn.swt_in(8)	switch 8
swt_in<9>	T3	in	xbus_syn.swt_in(9)	switch 9
swt_in<10>	T2	in	xbus_syn.swt_in(10)	switch 10
swt_in<11>	R3	in	xbus_syn.swt_in(11)	switch 11
swt_in<12>	W2	in	xbus_syn.swt_in(12)	switch 12
swt_in<13>	U1	in	xbus_syn.swt_in(13)	switch 13
swt_in<14>	T1	in	xbus_syn.swt_in(14)	switch 14
swt_in<15>	R2	in	xbus_syn.swt_in(15)	switch 15
uart_rx	B18	in	xbus_syn.uart_rx	RS232 RXD
uart_tx	A18	out	xbus_syn.uart_tx	RS232 TXD
vga_b<0>	N18	out	xbus_syn.vga_b(0)	VGA blue bit 0
vga_b<1>	L18	out	xbus_syn.vga_b(0)	VGA blue bit 1
vga_b<2>	K18	out	xbus_syn.vga_b(0)	VGA blue bit 2
vga_b<3>	J18	out	xbus_syn.vga_b(0)	VGA blue bit 3
vga_g<0>	J17	out	xbus_syn.vga_g(0)	VGA green bit 0
vga_g<1>	H17	out	xbus_syn.vga_g(1)	VGA green bit 1
vga_g<2>	G17	out	xbus_syn.vga_g(2)	VGA green bit 2
vga_g<3>	D17	out	xbus_syn.vga_g(3)	VGA green bit 3
vga_hsync	P19	out	xbus_syn.vga_hsync	VGA horizontal sync
vga_r<0>	G19	out	xbus_syn.vga_r(0)	VGA red bit 0
vga_r<1>	H19	out	xbus_syn.vga_r(1)	VGA red bit 1
vga_r<2>	J19	out	xbus_syn.vga_r(2)	VGA red bit 2
vga_r<3>	N19	out	xbus_syn.vga_r(3)	VGA red bit 3
vga_vsync	R19	out	xbus_syn.vga_vsync	VGA vertical sync
seg<0>	W7	out	xbus_syn.seg(0)	7 Segment bit a
seg<1>	W6	out	xbus_syn.seg(1)	7 Segment bit b
seg<2>	U8	out	xbus_syn.seg(2)	7 Segment bit c
seg<3>	V8	out	xbus_syn.seg(3)	7 Segment bit d
seg<4>	U5	out	xbus_syn.seg(4)	7 Segment bit e
seg<5>	V5	out	xbus_syn.seg(5)	7 Segment bit f
seg<6>	U7	out	xbus_syn.seg(6)	7 Segment bit g
dp	V7	out	xbus_syn.dp	7 Segment decimal point
an<0>	W4	out	xbus_syn.an(0)	7 Segment anode 1
an<1>	V4	out	xbus_syn.an(1)	7 Segment anode 2
an<2>	U4	out	xbus_syn.an(2)	7 Segment anode 3
an<3>	U2	out	xbus_syn.an(3)	7 Segment anode 4

13 XBus

The XBus is an external databus. It is used to communicate with the data memory (XDMemory) and the peripherals (XTimerXT, XUART, XConsole, etc.). For communication with the data memory, the `LOAD` and `STORE` instructions must be used, and for peripheral access, the instructions `IN` and `OUT` are to utilize. There are two versions of this module: a simulation version (suitable to use in the simulator), and a synthesizable version (with more components to be used on real hardware).

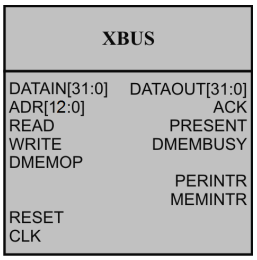


Figure 13.1: XBus interface.

Pin	Type	Source/Destination	Description
DATAIN(31..0)	I	DATAPATH	data input
ADR(12..0)	I	DATAPATH	XBus address
READ	I	CONTROL	read control from XBus
WRITE	I	CONTROL	write control to XBus
DMEMOP	I	INDEC	uses data memory
DATAOUT(31..0)	O	DATAPATH	data output
DMEMBUSY	O	CONTROL	data memory busy
ACK	O	CONTROL	acknowledgment from peripheral
PRESENT	O	CONTROL	addressed peripheral exists
PERINTR	O	PCBLOCK	peripheral interrupt
MEMINTR	O	PCBLOCK	data memory interrupt

13.1 Functionality

Figure 13.2 depicts the structure of the XBus. Every peripheral some defined registers and every register uses its unique address to be accessed using the signals address (ADR), read (READ) or write (WRITE). In a real Integrated Circuit (IC) the output signals of the components would be connected with a tristate signal to prevent signal collisions. In FPGAs this is only possible to be used for IO pins.

On the HaDes processor, the data memory is only available by accessing the XBus. The peripherals could have a register at the same address as a global variable in the data memory. Therefore, the HaDes instruction set distinguishes between data memory and peripheral access by using different instructions. Thus, the XBus has to handle both kinds in a different way:

Memory access The memory access is signaled with the DMEMOP port if a `LOAD` or `STORE` instruction is used (in the case that the PMA is disabled). At memory access, all bits from ADR, with the exception of the MSB, are used to select the position in the memory. The MSB has to be zero; otherwise, the calculated address in the datapath is incorrect and a memory interrupt (XMemory) is raised. The

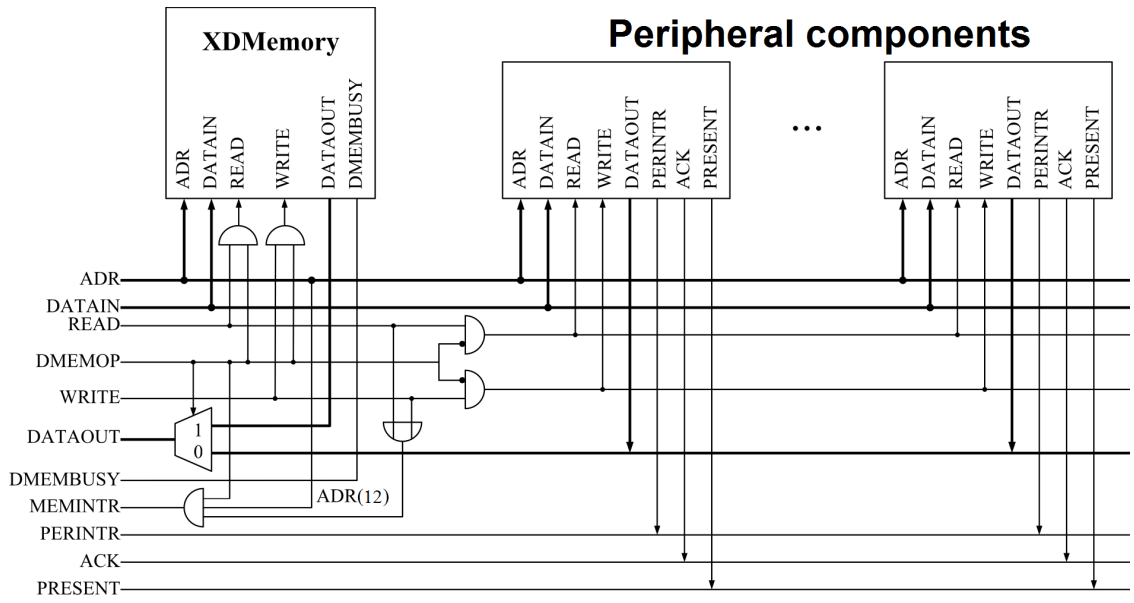


Figure 13.2: Structure of the XBus.

output signal of the memory is forwarded to the XBus if the DMEMOP port is set, otherwise the peripheral output value is hatched. The end of the memory access is shown by the DMEMBUSY flag if it is set to zero. Only after that, the read values are valid.

Peripheral access By the use of an `IN` or an `OUT` instruction, the DMEMOP is not set, thus a peripheral communication has to be performed. Every peripheral listens to the address bus. From the whole address bus width only the 8 Least Significant Bits are used, the other bits are ignored.

When the peripheral recognizes one of its addresses, it immediately sets the PRSENT signal and get the appropriated value to the DATAOUT output. Unless the address is valid, no peripheral sets the PRSENT signal. The control unit (CONTROL) realize the invalid address and raises an XBusNotAvailable interrupt.

Many peripherals also use interrupts to notify the CPU about an important state in it. The peripheral interrupt is sent to the CPU and saved into a status register. The interrupt will jump into the ISR and there the software queries the peripheral, to check from whom the interrupt comes. The interrupt remains active as long as the interrupts are enabled or the peripheral interrupt is acknowledged. This approach makes sure that no interrupt will be missed, especially if more than one peripheral raises an interrupt.

Table 13.1 and Table 13.2 will show the synchronization steps between the CPU and a peripheral *P* in case of a read and write command, respectively.

Step	`IN` **instruction** (CPU reads data from XBus)
1	CPU sets address to ADR
2	*P* recognizes its address and sets PRESENT
3	CPU control unit sets READ
4	*P* sets ACK signal and sets the data to DATAOUT
5	CPU control unit clears READ signal
6	*P* clears ACK and PRESENT signal

Table 13.1: Synchronization steps for an `IN` instruction

Step	`OUT` **instruction** (CPU write data to XBus)
1	CPU sets address to ADR
2	*P* recognizes its address and sets PRESENT
3	CPU control unit sets WRITE
4	*P* sets ACK signal
5	*P* latches the data on a risign clock edge
6	CPU control unit clears READ signal
7	*P* clears ACK and PRESENT signal

Table 13.2: Synchronization steps for an `OUT` instruction

14 XBus Peripherals

The XBus contains the data memory and many peripherals. This section introduces all offered HaDes peripherals.

14.1 XConsole

The XConsole peripheral has the job to manage LEDs, switches, buttons of the BASYS3 development board. It contains 5 different registers to manage the status of the external components. Furthermore, it has the possibility to enable interrupts for switches and buttons.

Address	Bit	Descpription
64 LED-control	15...0 31...16	LED15...LED0 activate LED15...LED0 blinking
65 switch-status	15...0	switch15...switch0 enabled
66 button-status	0 1 2 3 3...0	IN button NORTH has been set IN button EAST has been set IN button SOUTH has been set IN button WEST has been set OUT reset bit
67 interrupt-control	15...0 16 17 18 19	switch15...switch0 interrupt enable button NORTH interrupt enable button EAST interrupt enable button SOUTH interrupt enable button WEST interrupt enable
68 interrupt-status	15...0 16 17 18 19 19...0	IN switch15...switch0 has triggered an interrupt IN button NORTH triggered an interrupt IN button EAST triggered an interrupt IN button SOUTH triggered an interrupt IN button WEST triggered an interrupt OUT reset bit

The BASYS3 board contains 16 different LEDs. The LED-control register controls the behavior of the LED according to the next table:

active bit	blinking bit	result
0	*	LED turned off
1	0	LED turned on
1	1	LED blinking with 2 Hz

The address 65 (switch-status) shows the status of all 16 switch on the BASYS3 board. If the switch is enabled, the status returns a '1' at its position. For every switch, an optional interrupt can be enabled. You have to enable the interrupt in the interrupt-control register of this peripheral. If the interrupt is triggered, it must be acknowledged by writing a '1' to the interrupt-status register.

For the four buttons an own register is available. The button-status register shows with a '1' that a button has been pressed. If a '1' is written to the bit, the bit will be cleared by the hardware to be ready to recognize the next button press. As for the switches, there is the possibility to enable an interrupt on a pressed button. The control and status register is the same as for the buttons.

14.2 XTimerXT

The XTimerXT is a time programmable signal generator. The peripheral generates a signal on a defined time. This module is also supported to use in the simulator.

Address	Bit	Description
16 time	15...0	countdown time (ms)
17 status	0 1 2 2	timer activated / restart interrupt enable `IN` timeout expired `OUT` clear bit

The time register is used to define in which time (resolution 1 ms) a interrupt signal will be generated. In the simulator, the resolution is 1 µs. The Bit 0 in the status register activates the timer and the time register remains on the set value. To restart the timer, you have to set one more time the timer activate bit in the status register. It is also possible to restart the timer although the timer is not expired.

The interrupt bit enables an interrupt when the timer is expired. The expiration is also shown in the timeout expire bit. The interrupt stays active as long as this bit is set. A write to this bit clears it. Another way to clear it is to restart the timer.

14.3 XUART

The XUART peripheral is a serial communication interface (RS232) for communicating with a working station. The baud rate is fixed set to 115200 kbits.

Address	Bit	Description
96 buffer	31...0	`IN` receive buffer `OUT` transmit buffer
97 status	0 1	receive buffer full transmit buffer empty
98 control	0 1 2	receive buffer full interrupt enable transmit buffer empty interrupt enable UART mode: 0=byte, 1=word

To reduce the arise of receive buffer overflows under a high working load in the CPU, the XUART peripheral offers a 32-bit send-and-receive buffer, which is called Word-mode. The register at address 96 is used to transmit the 4 Bytes and to receive them. All 4 Bytes are sent in big-endian format: the most significant byte will be sent first. Additionally, the 8-bit transmission mode (Byte-mode) can be selected in the control register to send and receive only 1 Byte. If the receive buffer contains information, it will show this in the status register. In the Word-Modus, the receive buffer full bit is set if all four bytes are received; however, in the Byte-Modus already if one byte is received.

Further, the control register contains bits to enable an interrupt when one or both status bits are set. To clear the bit in the status register, which triggers the interrupt, you have to read out the receive buffer or to fill up the transmit buffer.

14.4 XPS2

The XPS2 is a peripheral to communicate with a PS/2 device. The BASYS3 board offers to connect Human Machine Interfaces (HMIs) (mouse or keyboard) only via an USB. The assembled microcontroller converts the USB protocol into a PS/2 signal. The peripheral abstracts the low level, and the user has only to handle the external device via three offered peripheral registers.

The communication interface is byte wise. It starts with a start bit ('1'). Afterward, the payload is sent with LSB first. The transfer ends with a parity and a stop bit ('1'). To synchronize the communication, a

Address	Bit	Description
128 data	7...0	OUT send data to the device IN read last received data
129 status	0 1 2 3 4 5	IN receive buffer full IN transmit buffer empty IN device initialized IN device connected IN device type 1 keyboard, 0 mouse OUT do a device reset
130 control	0 1	Receive buffer full interrupt enable Transmit buffer empty interrupt enable

further clock line is used. The clock is always driven by the PS/2 device.

To use the PS/2 after a start or reset, the programmer has to check if the device has been initialized. The reason for this approach is that after a CPU reset also the PS/2 device executes a reset (i.e., the keyboard flashes all three control LEDs). After the reset, the peripheral needs some time to initialize the external device and to detect the type. The bit 2 in register 129 indicates the end of the initialization process, and then the device is ready to send and receive commands.

The peripheral provides some status bits to show if a device is connected and the type if it is connected. The following listing shows an initialization sequence for a mouse.

```
@ldef stat "r3"
@ldef temp "r4"
INIT_LOOP:
  IN   @stat, #129
  ANDI @temp, @stat, #0b100       ; Bit 2 selected
  BEQZ @temp, #INIT_LOOP          ; wait until init is done
  // init is done
  // check if a mouse is conected
  ANDI @temp, @stat, #0b11000     ; Bit 3 and 4 selected
  SEQI @temp, @temp, #0b01000     ; device detected + mouse type
  BEQZ @temp, #ERROR              ; if false: ERROR
  ; ...                           ; else:     resume
```

The high level communication with the PS/2 device must be done in software. The connected devices are HMIs, therefore the programmer has, in most cases, only to handle the received bytes and to interpret them. The keyboard immediately sends the information to the PS/2 interface if a key is pressed or released. However, the mouse must receive a special command (*Enable Data Reporting*), to send the movement information from the mouse to the host. For further information visit: `http://www.computer-engineering.org`.

The peripheral provides to trigger an interrupt, if the receive buffer is full or the send buffer is empty. To enable and disable the interrupt the register at address 130 must be used. Both interrupts remain active as long as the event is not handled. To clear a receive buffer full interrupt and a transmit buffer empty interrupt you have to read the data register and to write to the data register, respectively. If no interrupts are active, the programmer has to poll the data register 128 periodically to check if a new data is received, which is shown by bit 0 in the status register 129. If new data should be sent to the PS/2 device, the programmer has to check if the transmit buffer is already empty, otherwise the value written to the data register is ignored.

The next two tables show the scan codes for the mouse and for the keyboard.

Mouseevents

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1:	Overflow Y	Overflow X	Sign Y	Sign X	1	Middle Button	Right Button	Left Button
Byte 2:	Movement X							
Byte 3:	Movement Y							

Keyboard Scan Codes (make = *key pressed*, break = *key released*)

key	make	break	key	make	break	key	make	break
A	1C	F0,1C	9	46	F0,46	[	54	FO,54
B	32	F0,32	‘	0E	F0,0E	INSERT	E0,70	E0,F0,70
C	21	F0,21	-	4E	F0,4E	HOME	E0,6C	E0,F0,6C
D	23	F0,23	=	55	FO,55	PG UP	E0,7D	E0,F0,7D
E	24	F0,24	\	5D	F0,5D	DELETE	E0,71	E0,F0,71
F	2B	F0,2B	BKSP	66	F0,66	END	E0,69	E0,F0,69
G	34	F0,34	SPACE	29	F0,29	PG DN	E0,7A	E0,F0,7A
H	33	F0,33	TAB	0D	F0,0D	U ARROW	E0,75	E0,F0,75
I	43	F0,43	CAPS	58	F0,58	L ARROW	E0,6B	E0,F0,6B
J	3B	F0,3B	L SHFT	12	FO,12	D ARROW	E0,72	E0,F0,72
K	42	F0,42	L CTRL	14	FO,14	R ARROW	E0,74	E0,F0,74
L	4B	F0,4B	L GUI	E0,1F	E0,F0,1F	NUM	77	F0,77
M	3A	F0,3A	L ALT	11	F0,11	KP /	E0,4A	E0,F0,4A
N	31	F0,31	R SHFT	59	F0,59	KP *	7C	F0,7C
O	44	F0,44	R CTRL	E0,14	E0,F0,14	KP -	7B	F0,7B
P	4D	F0,4D	R GUI	E0,27	E0,F0,27	KP +	79	F0,79
Q	15	F0,15	R ALT	E0,11	E0,F0,11	KP EN	E0,5A	E0,F0,5A
R	2D	F0,2D	APPS	E0,2F	E0,F0,2F	KP .	71	F0,71
S	1B	F0,1B	ENTER	5A	F0,5A	KP 0	70	F0,70
T	2C	F0,2C	ESC	76	F0,76	KP 1	69	F0,69
U	3C	F0,3C	F1	05	F0,05	KP 2	72	F0,72
V	2A	F0,2A	F2	06	F0,06	KP 3	7A	F0,7A
W	1D	F0,1D	F3	04	F0,04	KP 4	6B	F0,6B
X	22	F0,22	F4	0C	F0,0C	KP 5	73	F0,73
Y	35	F0,35	F5	03	F0,03	KP 6	74	F0,74
Z	1A	F0,1A	F6	0B	F0,0B	KP 7	6C	F0,6C
0	45	F0,45	F7	83	F0,83	KP 8	75	F0,75
1	16	F0,16	F8	0A	F0,0A	KP 9	7D	F0,7D
2	1E	F0,1E	F9	01	F0,01	]	5B	F0,5B
3	26	F0,26	F10	09	F0,09	;	4C	F0,4C
4	25	F0,25	F11	78	F0,78	’	52	F0,52
5	2E	F0,2E	F12	07	F0,07	,	41	F0,41
6	36	F0,36	PRNT SCRN	E0,12, E0,7C	E0,F0, 7C, E0, F0,12	.	49	F0,49
7	3D	F0,3D	SCROLL	7E	F0,7E	/	4A	F0,4A
8	3E	F0,3E	PAUSE	E1,14,77,E1 F0,14, F0,77	-NONE-			

14.5 XSevenSeg

XSevenSeg is the peripheral responsible for setting the four seven segment displays with their dots. The colon between the first two and the second two displays is not connected in hardware therefore cannot be controlled by software.

This module offers an easy way to use the interface as shown in the following table.

Address	Bit	Description
224	31	OUT activate the displays
	30-20	OUT unused
	16-19	OUT set the four dots
	11-15	OUT BCD of 1000
	8-11	OUT BCD of 100
	4-7	OUT BCD of 10
	0-3	OUT BCD of 1

Each display can be set by a 4-bit Binary Coded Decimal (BCD) code and its dot in the corresponding dot flag. The MSB indicates, whether the display is switched on or off. The four bits 19 to 16 set the dots from left to right. Therefore, bit 19 sets the dot of the outer left, 18 the inner left, 17 the inner right and 16 the right display.

To set the *decimal values* of each display, one has to set the corresponding 4-bit BCD code into the register. Because of the BCD code, only values from 0 to 9 can be displayed.

If a single switched-off display is wanted, a value above 9 must be set into the corresponding four bits.

The following code snipped shows the usage:

```
LDUI r1, #0x8002  ; enable and set inner right dot
SHLI r1, r1, #16  ; shift register left by 16 bit
ORI r1, #0xF147   ; disable outer left and set 145 to others
OUT r1, #224      ; write to module
```

One can see that our CPU only supports 16-bit operations, therefore we need to shift the register to the left to set its bits higher 16. When applied, the snippet should show 14.7 with a disabled outer left display.

14.6 XVGA

The XVGA peripheral controls the graphical output on the VGA output. The output resolution is predefined to 640x480 pixel with 16 colors. The used color system is the Color Graphics Adapter or iRGB system (https://en.wikipedia.org/wiki/Color_Graphics_Adapter). It is a 4-bit color system. Beside the 3 primary colors, red, green and blue, an intensity bit exists describing the intensity of the RGB color. The next table shows the assignment of the individual bits:

Bit	Descpription
0	blue
1	green
2	red
3	intensity

With 4 bits it is possible to generate 16 different colors. In the next table, all possible colors are shown. The table shows that the intensity bit distinguishes the intensity of the RGB color. The only exceptions are the brown and yellow color. The standard defines to generate a brown color instead of a dark yellow, for a better distinction of the two colors.

Color	Bits	Color	Bits	Color	Bits	Color	Bits
black	0000	red	0100	gray	1000	light red	1100
blue	0001	magenta	0101	light blue	1001	light magenta	1101
green	0010	brown	0110	light green	1010	yellow	1110
cyan	0011	light gray	0111	light cyan	1011	white	1111

The component uses internally five different block rams with a size of $16kx4$. Therefore, the completely graphic memory has a size of 1310720 bits which corresponds to 327680 pixel. For the representation for the whole screen, the full memory space is not required. For the screen only 640 pixel $\cdot$ 480 pixel $=$ 307200 pixel are used. Therefore, the remaining space in the memory can be used to temporary save some figures. The remaining space in the memory is 327680 pixel $-$ 307200 pixel $=$ 20480 pixel. It can be used like the visible space; you have to use only the graphic memory address outside the visible region.

Address	Bit	Description
160 address	18...0	Address of the graphic memory Increments automatically after a buffer or fastbuffer read or write.
161 buffer	3...0	`IN` read a pixel `OUT` write a pixel
162 fastbuffer	15...0	`IN` read four pixel `OUT` write four pixel

The address register (160) is the current graphic address in the graphic memory. It can be read out and set to a new value. The address 0 is the pixel on the top left $(x = 0,\ y = 0)$ of the screen. The rising address goes from the left to the right til the end of the row. Then it jumps to the next row and starts on the left. Therefore, the next pixel of address $n \cdot (640 - 1), n \in \{0, \ldots, 478\}$ $(x = 639,\ y = n)$ is on the next row on the left $(x = 0,\ y = n + 1)$. After finishing the last line $(x = 639,\ y = 479)$, the next pixel starts at the beginning $(x = 0,\ y = 0)$.

The buffer and fastbuffer are used to write or read the color into the component's graphic buffer. The register 161 (buffer) is used to exchange one pixel. It contains the three colors plus the intensity bit. All other bits are ignored. With the fastbuffer, you have the possibility to transfer four pixels in one write or read command. Therefore, you can reduce the transportation overhead of a full image. Take care that in this mode, the graphic memory address must be aligned to 4 Pixel. After every read or write command to one of the two buffers, the address is incremented automatically. In the 1 pixel transfer by 1 and in the fastbuffer mode by 4.

The next Listing sets a pixel at the address `@pixaddr` with the value `@pixdata`. Before doing it, the old graphic memory pointer is saved into `@temp` for future use.

```
IN  @temp, #160     ; save old graphic address
OUT @pixaddr, #160  ; Write pixel...
OUT @pixdata, #161  ; ...into the buffer
OUT @temp, #160     ; restore old graphic address
```

The next Listing fill the whole display with a white blue striped mask:

```
@data mask {
  #0000FF11h    ; white blue striped mask
@}

@code __init {
  @ldef counter "r3"
  @ldef mask    "r4"
  LDUI @counter, #0b1001011
  SHLI @counter, @counter, #10   ; init counter 307200/4
  LOAD @mask, r0, *d*mask        ; load mask
loop:
  OUT  @mask, #162               ; save mask, FASTBUFFER
  DEC  @counter                  ; decrement counter
  BNEZ @counter, #loop           ; check if finish

forever:
  JMP #forever                   ;
@}
```

15 Assembly

The implemented modules are used to create the whole HaDes processor. Figure 15.1 shows one more time, the whole CPU with its modules and the interconnections to each other. The connection to the XBus is still missing. Figure 15.2 shows the connection to the XBus to create a fully functional processor.

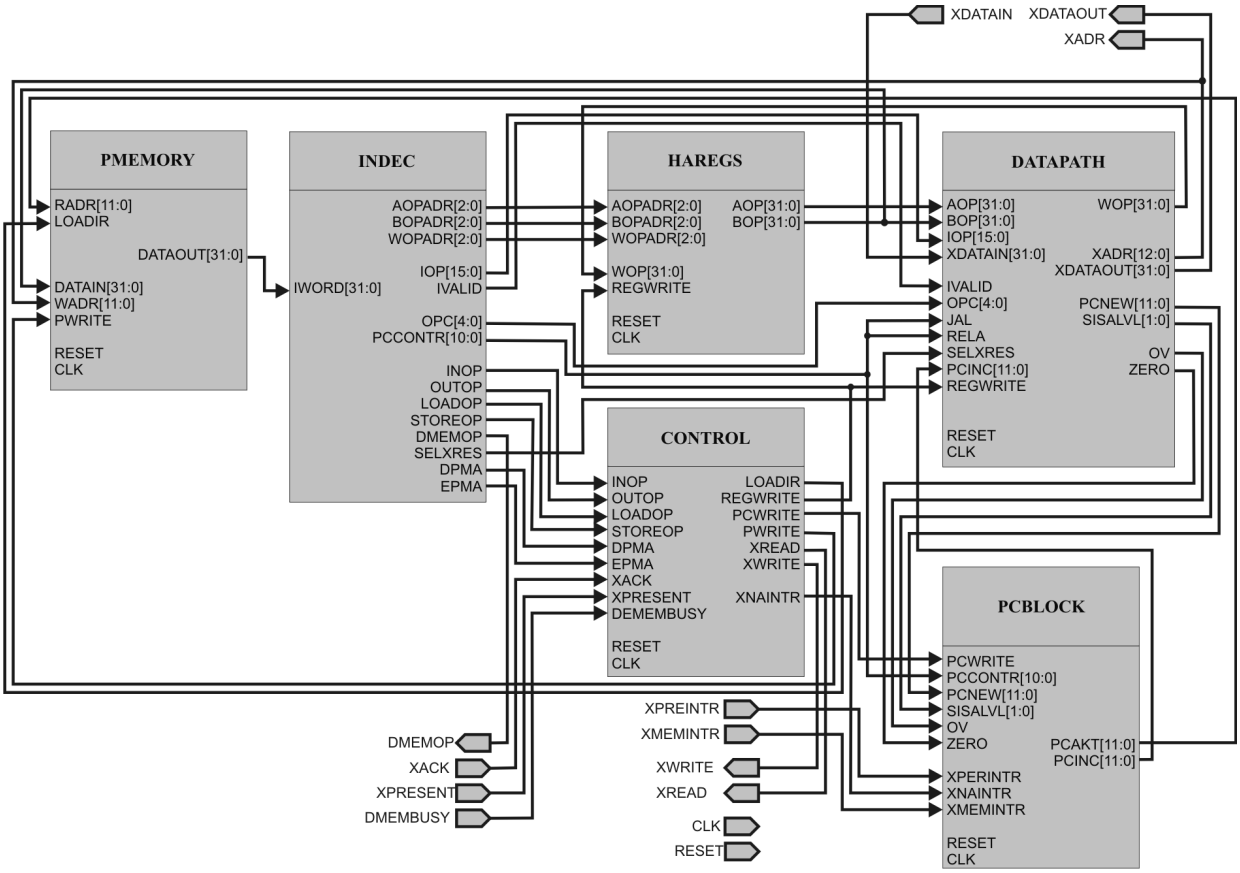


Figure 15.1: HaDes top level.

15.1 Bootloader

The bootloader has the job to upload an application into the program memory. After configuring the FPGA with the bitstream the CPU executes the first word in the program memory. The first address is read and jumps to the bootloader, by default. This address is used as a trampoline because the bootloader sits at the end of the program memory. Afterwards, it waits for an incoming word on the serial interface (XUART). The bootloader expects a `hix`-file (HaDes Internal Exchange Format) which is formatted according the next table:

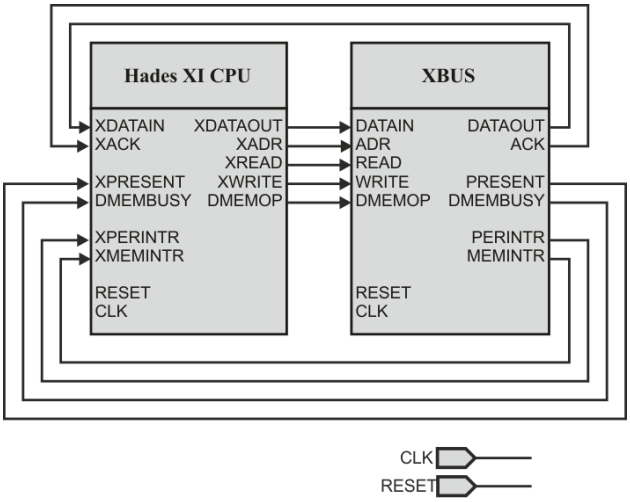


Figure 15.2: HaDes test environment.

Index	Meaning
0	Number of instructions l
1	1st instruction
...	...
l	lth instruction
$l+1$	Checksum of instructions (32 bits signed)
$l+2$	Number of datas d
$l+3$	1st data
...	...
$l+2+d$	dth data
$l+2+d+1$	Checksum of data (32 bits signed)

The progress of the bootloader is shown on the LEDs. If the bootloader process is finish, all LEDs are cleared otherwise the bootloader hangs in one of the bootloader steps, or stops in one of the four bootloader error conditions, which is shown with a blinking LED. The next Table shows all possible LED states in the bootloader:

LED LDx on the board																State
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	◉	Bootloader started; wait for program length
●	●	●	●	●	●	●	●	●	●	●	●	●	●	◉	◉	Program length received
●	●	●	●	●	●	●	●	●	●	●	●	●	◉	◉	◉	Program read; wait for check sum
●	●	●	●	●	●	●	●	●	●	●	●	◉	◉	◉	◉	Check sum read
●	●	●	●	●	●	●	●	●	●	●	◉	◉	◉	◉	◉	Data length received
●	●	●	●	●	●	●	●	●	●	◉	◉	◉	◉	◉	◉	Data read; wait for check sum
●	●	●	●	●	●	●	●	●	◉	◉	◉	◉	◉	◉	◉	Check sum read
●	●	●	●	●	●	●	◖	●	●	●	●	●	●	●	●	Program too long
●	●	●	●	●	●	◖	●	●	●	●	●	●	●	●	●	Program check-sum error
●	●	●	●	●	◖	●	●	●	●	●	●	●	●	●	●	Data too long
●	●	●	●	◖	●	●	●	●	●	●	●	●	●	●	●	Data check-sum error

After the bootloader has finish, the program memory access is disabled (`DPMA` instruction), all registers are cleared to zero, all peripherals are cleared, the interrupts are enabled (`ENI` instruction) and jumps to the first address in the program memory.

The bootloader will never be overwritten, only the program memory before the bootloader including the first word, which was the trampoline address for the bootloader. This will be overwritten by the new program but the application code has the possibility to jump into the bootloader because it is not changed.

16 HaDes Object Assembler

The *HaDes Object Assembler* is an assembler for the HaDes CPU. It allows the programmer to place the software into different files, to declare variables, to create names for registers and constants, and many other features to simplify the programming.

16.1 Syntax

The HaDes assembler files are text files with code, data symbols, and further declarations. Every line defines an own instruction; spaces and tabs are only used to separate and will be ignored by the assembler.

The assembler is case-insensitive, therefore upper-case and lower-case characters are interpreted as the same. Thus, `ShiftRight:`, `Shiftright:` and `SHIFTRIGHT:` are the same label. This is also the same for declarations (e.g., `@InC`) and instruction names (e.g., `shRi`). **To increase the readability, you have to format the code well, use a uniform style, and to use comments.**

16.1.1 Comment

The assembler supports three different styles for commenting code:

- Assembler row comment:

```
; Comment until the end of the row.
```

- `C++` row comment:

```
// Comment until the end of the row.
```

- Multi row `C` comment:

```
/* This command uses
 * several lines... */
```

16.1.2 Numbers

Numbers can be presented in different ways, depending on its base or representation. A constant value must be started always with a hash character #. Take notice that the range of the constant cannot exceed the range of its usage. Therefore, the range for immediate variable is limited to signed 16 bit = $[-32767, 32767)$.

Decimal It is declared without a special character: `#123` or `#-45`.

Binary For declaring a binary value, you can use the prefix `0b` or the postfix `b`: `#0b00101011` or `#10010010b`.

Hexadecimal Hexadecimal constants can be used like in `C` with the prefix `0x` or with the postfix `h`: `#0x1234` or `#ac73h`. (Note: Where `0x` can only be used to represent positive numbers between `0x00000000` and `0x7FFFFFFF`)

Character HaDes also supports characters that are declared with apostrophe `'`: 'Z'. These characters are ASCII values and therefore they are mapped to a value in the range of 0 to 255.

16.1.3 Global Declaration

`@inc` The declaration `@inc "file"` includes the assembler file in this position. The file is included verbatim like the header file in `C`:

```
@inc ''init.has''
```

`@def` The declaration `@def NAME "CONST"` can be used to declare a register or constant value with a name. At assembling, the assembler substitute all NAME definitions with the constant CONST. The scope of the definition is from the declaration until the end of the file. If it is declared inside an include file, then the NAME is also accessible in the file which includes it.

```
@def sp ''r1''          ; Definition of SP
@def offset ''#1''
  ;  ...
STORE r3, @sp, @offset  ; Usage of SP
```

`@mac` Macros are used to declare a piece of code, which will be inlined into the caller code. The scope of the declaration is till the end of the file.

```
@mac MACNAME {
  ; Macro ...
@}
```

The macro can be called with `@call` with up to nine parameters. In the macro they are handled with the macro parameter @1 to @9. Pay attention, the macro is concluded with the character @ before the closing bracket.

```
@mac enableLED {     ; Define macro enableLED
  LDUI @1, @2        ; (Call with @call enableLED TempReg BitMask),
  OUT  @1, @LED
@}
```

`@call` After declaring a macro, it can be used with the `@call MACNAME PARAM1 ...PARAMn` instruction. The assembler replaces the macro call with the content of the macro; therefore, everything is inlined. Pay attention, the macro parameters are not separated with a comma.

```
@call enableLED r5 #0x0001
```

16.1.4 Code Symbol

Code symbols are blocks that contain sequenced instructions. These instructions will be programmed into the HaDes to execute it. The HaDes assembler allows you to implement code only in code symbols and macros. The layout of the code symbols is managed by the linker. It also includes a garbage collector, which removes unused code symbols to save memory. Thus, it is impossible to suppose the relative location of code symbols. Therefore, the assembler does not allow to use reference labels from other code symbols.

Every function is declared as a code symbol. Therefore, a function call should not be done with a jump to a label (`JAL @ra, #mark`), but with a code symbol references (`JAL @ra, *reference`). The assembler ignores these references, and the linker replaces the call with the offset, represented as an immediate value, to the entry address of the called function.

`@code` The declaration of a code symbol is shown in the next Listing:

```
@code NAME {
  ; instructions, local definitions, labels, ...
@}
```

The code symbol has to define a name NAME. The name is further used to call the function by using the symbol referencing `*NAME`. Like the macro, the code symbol must be closed with the character @ before the closing bracket.

`@ldef` The local definition works like the global definition, but the scope of this definition is limited. The scope of the definition is only valid inside the code symbol where it is declared. Therefore, they are useful to assign local variables to registers, which are used only inside a code symbol.

```
@ldef temp ''r5''
```

`Instructions` The important content of a code symbol are the instructions. Each instruction is declared in a new line, like all other declarations. The parameters are separated with a comma. The order of operands in a arithmetic operation is *destination*, *source1*, *source2* and produces the following mathematical equation: `destination := source1 OP source2`.

The parameters of the instruction are in most cases the registers (*r*0 to *r*7), but some instructions use an immediate and maybe an additional constant value. The immediate instructions, which are also offered as register version, are distinguished by adding a `I` to the instruction: e.g., `ADDI`. The immediate values are constants values(#Values), labels (#Label) or symbols (*Function). To load or store a variable to a global variable the data symbol reference can be used: e.g., `LOAD regDest, r0, *d*varName`.

`Pseudo instructions` The HaDes assembler offers some pseudo instructions, which are replaced in the assembling stage with a real instruction. For more details, see Section III.

`Two address format` Further to pseudo instructions, the assembler offers a two address format for arithmetic and set-condition instructions. For example, the x86 architecture also uses this style. There, the source operand is implicitly the destination operand:

- `ADD r1, r2` equivalent to `ADD r1, r1, r2`
- `ADDI r1, #87` equivalent to `ADDI r1, r1, #87`
- `SGT r1, r4` equivalent to `SGT r1, r1, r4`

`Labels` The HaDes CPU allows the programmer to jump and branch to relative addresses. The relative addresses must not be calculated by the programmer, instead you can use the label names (e.g., `LABEL:`). The branch and jump instruction (`BxxZ, JAL, ...`) can use these labels. The assembler resolves the relative addresses inside a code symbol. Thus, it is not allowed to use a label of another code symbol.

`Code symbol reference` For jump instructions the programmer can also use code symbols instead of labels. This is often used to call a function `JAL @ra, *functionname`. For using the code symbol reference, the assembler does not need a declaration like in other languages. The linker then resolves the references. If the called symbol reference cannot be found, the linker aborts the linking with an error. Pay attention that symbol names are case-sensitive, which is an exception.

16.1.5 Data Symbol

Data symbols are global variables from high level programming languages, which are already initialized at startup by the bootloader. The initialization value is stored into the `HIX`-file. To access the variables you can use the `LOAD` or `STORE` instruction.

`Data symbol reference` The reference to the data symbol is similar to the code symbol, only a prefix `*d` declares it as a data symbol. E.g., The address of the global variable `icon` can be read as `LDUI regDest, *d*icon`.

`Internal Data` The global variables are declared with the `@data` declaration. For every variable, a name must be used. The size of the variable depends on the number of defined initialized values. Every added value increases the variable size by the address bus size, which is in the HaDes architecture 4 Bytes. The assembler does not allow declaring variables without an initialization value.

```
@data VARIABLENNAME {
  ; List of values, separated with commas
@}
```

The initialization value can be initialized with three different types:

Integer To initialize it with an integer value, you can use an integer constant.

Symbol reference The variable can be initialized with a pointer to a data or code symbol.

String The variable can be initialized with a string. The string has to be in quotes, e.g. ”Hello, world!”. Every ASCII character is stored into a separate aligned address. Pay attention, the string is not terminated with a termination character, like zero.

`External Data` To load many data into the data memory (e.g. wallpapers), the assembler offers to load an external binary file in it.

```
@data VARIABLENNAME "FILE"
```

The file will be read and loaded into the object file. It is important that the file size has to be a multiple of four, because the data width is four. The binary input file has to be formatted into big-endian format with four bytes per word:

Byte	Content
0	Bits 31..24 of word 0
1	Bits 23..16 of word 0
2	Bits 15..8 of word 0
3	Bits 7..0 of word 0
⋮	⋮
$4 \cdot k + 0$	Bits 31..24 of word k
$4 \cdot k + 1$	Bits 23..16 of word k
$4 \cdot k + 2$	Bits 15..8 of word k
$4 \cdot k + 3$	Bits 7..0 of word k

16.1.6 Short Reference

```
; Comment
/* Multi-line
   comment*/
// C++ comment.

@inc "incl.has"         ; Includes the incl.has file

@def sp "r3"            ; Uses @sp instead of r3
@def LED "#64"          ; Uses @LED instead of #64

@mac blinkLED1 {        ; macro definition
  LDUI @1, #256         ; (function call @call blinkLED1 TempReg),
  OUT  @1, @LED         ; @1 ... @9 can be used for parameter
@}


@data my_var_name {     ; Create a data symbol (size in words)
  #1,                   ; Decimal value.
  #12h,                 ; #12h and #0x12 are the value 12(hex)
  #0x12,                ; therefore 18(decimal).
  #1000101010b,         ; Binary value and
  #-12,                 ; negative values are allowed.
  "Hello, world!",      ; Strings use one word per character
  #0                    ; and are not automatically terminated.
@}

/* Big data can also be loaded from an external binary. The data have to
   be in big-endian format. */
@data my_img_var_name "img.raw"

@code my_func_name {     ; Create code symbol (function)
  STORE r2, @sp, #3      ; Save return address
  @ldef temp "r5"        ; Local definition
  LOAD  @temp, @sp, #2   ; Instruction with the use of local definition
MkGreen:                 ; Scope of label is inside symbol
  @call blinkLED1 @temp  ; Macro call will be inlined.
                         ; Parameters are not separated with comma.
  JAL   r2, *subfunction ; Function call.
  LOAD  @temp, r0, *d*my_var_name
                         ; Load first item (#1) from variable
  BEQZ  @temp, #MkGreen  ; Jump to label
  LOAD  r2, @sp, #3      ; Restore return address.
  JREG  r2
@}
```

16.2 Program Structure

To make the software more readable and maintainable, it is needed to use macros or function calls otherwise it looks like spaghetti code. Macros are inlined into the code; therefore, it can increase the code size extremely if they are used many times. Thus, function calls are used to prevent an extremely increasing code size. To handle functions, it is needed to use the program stack. This is used to overhand parameters, to get return values and to buffer results. Buffering results is important, because every function uses registers and the code could overwrite their content.

16.2.1 Register and Memory management

A stack is needed to call a function. The stack is normally managed with a single stack pointer. This pointer points to the first free item in the stack where you can add parameters, return values, and buffered values in any order. The items are accessible with a position relative to the stack pointer (i.e., stack pointer address and a displacement).

The stack pointer is used in every function call, therefore an own register is allocated to it. A further register is allocated for the return address from a function call. Next Listing defines the two global declarations:

```
@def ra "r1"  ; return address
@def sp "r2"  ; stack pointer
```

It is important that this two register will not be used as a variable. For buffering results, the register *r*3 to *r*7 can be used. Register *ra* is used to save the return value from the function call `JAL`. Unless the function calls a further function, the register *ra* can be used to return to the caller address with the instruction `JREG`. The next Listing shows these steps.

```
JAL @ra, *function
; ...
@code function {
  ; ...
  JREG @ra
@}
```

However, if another function were called then the register `ra` would be overwritten. Thus, the register must be saved on the stack before calling the function. A more detailed explanation can be found in Section 16.2.2

The stack should be initialized at start-up to a free memory. The bootloader initializes the data by increasing addresses. Thus, the memory after the global variables (or heap pool) is free and the stack pointer can be initialized with this address (e.g. 0x7FF). The best practice is to initialize the stack pointer in the `__init` function. This should be included into your main file, which implements a main code symbol. The initialization code could look like the next Listing:

```
@code __init {
  LDUI @sp, #0x7FF ; Set SP to 0x7FF
  ; some other initialization code
  JAL  @ra, *main  ; call main function
endless:           ; endless loop
  JMP  #endless    ; if main returns
@}
```

16.2.2 *Function call*

Some provided functions (PlotPixel and DrawLine) use a standardized function call convention. Therefore, we recommended to use the suggested Application Binary Interface (ABI) convention:

- Stack pointer points always to a free item.
- Uses *Callee-Save* approach: the called function saves all registers before using them and restore them before returning.
- The parameters are pushed on the stack.
- The return values are pushed on the stack.
- All additional free items on the stack can be used by the function, e.g., context save, local variables, and further function calls.

The ABI is also depicted in Figure 16.1.

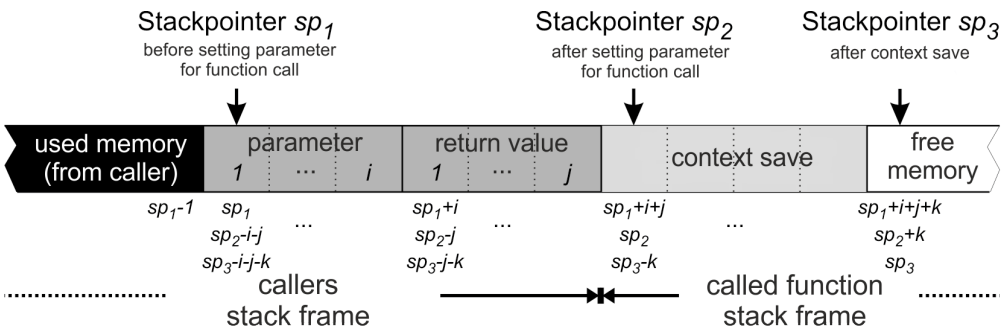


Figure 16.1: ABI.

Next Listing demonstrates a typical function call. The stack pointer points to the first free item on the stack. The function uses one parameter and one return value:

```
; ...
ADDI  @sp, #2            ; Increment stack pointer by i+j (here i+j=2)
STORE @param, @sp, #-2   ; Set parameter

JAL   @ret, *function    ; Call function

LOAD  @retval, @sp, #-1  ; Get return value from stack
SUBI  @sp, #2            ; Restore stack pointer
; ...
```

The function call itself could look like the next Listing. It saves the used registers before using them and restores them before returning:

```
@code function {
  @ldef mypar  "rx"
  @ldef myrval "ry"
  ADDI  @sp, @sp, #3     ; Increment stack pointer (here k=3)
  STORE @ra, @sp, #-1    ; Save return address
  STORE rx, @sp,  #-2    ; Save other...
  STORE ry, @sp,  #-3    ; ...used registers
  ; ...
  LOAD  @mypar, @sp, #-5 ; Load parameter into register(hier i+j+k=5)
  ; ...
  STORE @myval, @sp, #-4 ; Set return value(hier j+k=4)
  ; ...
  LOAD  ry, @sp,  #-3    ; Restore ...
  LOAD  rx, @sp,  #-2    ; ... saved ...
  LOAD  @ra, @sp, #-1    ; ... registers
  SUBI  @sp, @sp, #3     ; Restore stack pointer
  JREG  @ra              ; Return to address in @ra
@}
```

17 Assembly Exercise

Initialization The program should clean the drawing area with the background color, which is always set to black. Afterward, the mouse cross is initialized into the middle of the screen and the screen frame is set to the current drawing color.

Drawing area The whole VGA resolution of the component is 640 x 480 pixel, which represents the physical coordinates. The drawing area is reduced to 636 x 476 pixel, which are the logical coordinates. The logical coordinates are mapped to the middle of the physical coordinates, which results that a 2 pixel frame arises. This frame is colored to the current drawing color.

Key Lock The switch SW4 is the key lock. If it is enabled, the program does not react to anything.

Color selection The background color of the drawing area is always set to black. However, the drawing color can be changed at runtime. The VGA peripheral offers 16 different colors. The switches SW3, SW2, SW1, and SW0, are used to set the current drawing color bits intensity, red, green, and blue, respectively. The LEDs show the current selected drawing color, like the frame of the drawing area. The next table shows the selected colors to draw and which LEDs are active in respect to the switch state:

SW4	SW3	SW2	SW1	SW0	Color	LED active
0	0	0	0	0	black	LED0 blinking
0	0	0	0	1	blue	LED0 & LED1
0	0	0	1	0	green	LED0 & LED2
0	0	0	1	1	cyan	LED0 & LED3
0	0	1	0	0	red	LED0 & LED4
0	0	1	0	1	magenta	LED0 & LED5
0	0	1	1	0	brown	LED0 & LED6
0	0	1	1	1	light gray	LED0 & LED7
0	1	0	0	0	gray	LED0 & LED8
0	1	0	0	1	light blue	LED0 & LED9
0	1	0	1	0	light green	LED0 & LED10
0	1	0	1	1	light cyan	LED0 & LED11
0	1	1	0	0	light red	LED0 & LED12
0	1	1	0	1	light magenta	LED0 & LED13
0	1	1	1	0	yellow	LED0 & LED14
0	1	1	1	1	white	LED0 & LED15
1	*	*	*	*	-	-

Mouse cross The mouse position is shown with the mouse cross. The mouse cross contains four "wings", each one has four pixels and are drawn in white. One pixel around and the pixel at the mouse position is not overdrawn, the background is shown there. Figure 17.1 shows how you should store the mouse cross.

Mouse movement The mouse cross is adapted to the movement of the external connected mouse. Before moving the mouse cross, the buffered background (16 pixel) of the old mouse position is restored, then the background of the new position is saved, at the end, the graphic buffer is overdrawn with the mouse cross on the new position. The buffered background can be saved on a fixed buffer in the

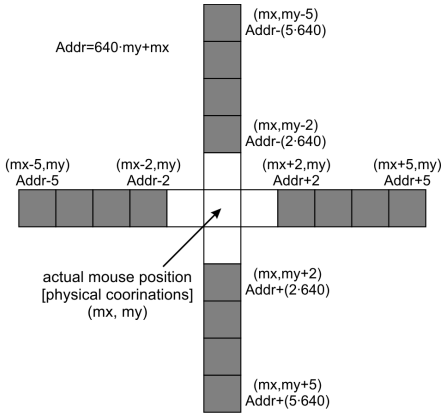


Figure 17.1: Mouse cross.

data memory. Take notice, the position of the mouse cannot underflow, or overflow the drawing area, however the "wings" of the cross can overlap the graphic frame.

7Segment The current position of the logical mouse coordinates should be shown on the 7Segment. The x-axis and y-axis should switch in a 2 s period.

Mouse buttons

- The left mouse click draws a line from the old mouse position to the new position with the current drawing color. You can use the provided `DrawLine` function.
- If the middle button on the mouse is clicked, then the screen will be cleaned with the background color black. This can take some time.
- The right click is used to rubber. Thus, it draws a point with the background color. You can use the provided `DrawPixel` function.

Buttons The four buttons simulate the movement of the mouse. If a button is pressed, the mouse will move for 4 pixel in the corresponding direction. The pressing of the four buttons does not draw a line it moves only the cross. But if the left button of the mouse is pressed, the line has to be drawn by the program.

Note: Do not forget to push the sourcefile `.has` and also the compiled file `.hix` into the `_assembler` folder into your repository!

Part III

Appendix

HaDes Instruction Set

The register set R contains eight registers $R := \{r0, r1, r2, r3, r4, r5, r6, r7\}$ in which the value is defined as $r0 := 0$ and $r1, \ldots, r7 := \{x \in \mathbb{Z} \mid -2.147.483.648 = -2^{31} \leq x \leq 2^{31} - 1 = 2.147.483.647\}$. In the following, lets define $w, a, b \in R$ and the constant value $k := \{x \in \mathbb{Z} \mid -32.768 = -2^{15} \leq x \leq 2^{15} - 1 = 32.767\}$ and the annotations $_s$ stands for signed, $_u$ for unsigned, $_{32}$ for a 32-bit extended constant value, and $_{(x...0)}$ for the respective bits.

The ALU instructions use one destination and two source values, with a few exceptions. The destination is always a register and the sources are either two registers or only one register with an 16-bit immediate value. The 16-bit immediate value is expanded to a 32-bit value.

Instruction	**Semantic**	**Effects**	
		OV	ZERO
`NOP`	PC++		
`ADD w,a,b`	$w := a + b$	X	
`SUB w,a,b`	$w := a - b$	X	
`MUL w,a,b`	$w := a \times b$	X	
`AND w,a,b`	$w := a$ and b		
`OR w,a,b`	$w := a$ or b		
`XOR w,a,b`	$w := a$ xor b		
`XNOR w,a,b`	$w := a$ xnor b		
`SHL w,a,b`	$w := a$ shl $b_{(4...0)}$	X	
`CSHL w,a,b`	$w := a$ cshl $b_{(4...0)}$		
`SHR w,a,b`	$w := a$ shr $b_{(4...0)}$	X	
`CSHR w,a,b`	$w := a$ cshr $b_{(4...0)}$		
`ADDI w,a,#k`	$w := a + \#k_{s32}$	X	
`SUBI w,a,#k`	$w := a - \#k_{s32}$	X	
`MULI w,a,#k`	$w := a \times \#k_{s32}$	X	
`ANDI w,a,#k`	$w := a$ and $\#k_{u32}$		
`ORI w,a,#k`	$w := a$ or $\#k_{u32}$		
`XORI w,a,#k`	$w := a$ xor $\#k_{u32}$		
`XNORI w,a,#k`	$w := a$ xnor $\#k_{u32}$	X	
`SHLI w,a,#k`	$w := a$ shl $\#k_{u32(4...0)}$		
`CSHLI w,a,#k`	$w := a$ cshl $\#k_{u32(4...0)}$	X	
`SHRI w,a,#k`	$w := a$ shr $\#k_{u32(4...0)}$		
`CSHRI w,a,#k`	$w := a$ cshr $\#k_{u32(4...0)}$		
`GETOV w`	$w :=$ OV		
`SETOV b`	OV $:= b_{(0)}$	X	
`SETOVI #k`	OV $:= \#k_{(0)}$	X	

The next instructions are the *Set-Condition* instructions. Here, the register a is compared with the register$b$ or an immediate value. The register w will be set with 1 when the condition is true. Otherwise, it will be set with 0. Like in the ALU immediate instructions, the immediate value will be expanded to a 32-bit signed value.

Instruction	Semantic	Effects	
		OV	ZERO
`SEQ w,a,b`	$w := (a = b)\ ?\ 1 : 0$		
`SNE w,a,b`	$w := (a \neq b)\ ?\ 1 : 0$		
`SLT w,a,b`	$w := (a < b)\ ?\ 1 : 0$		
`SGT w,a,b`	$w := (a > b)\ ?\ 1 : 0$		
`SLE w,a,b`	$w := (a \leq b)\ ?\ 1 : 0$		
`SGE w,a,b`	$w := (a \geq b)\ ?\ 1 : 0$		
`SEQI w,a,#k`	$w := (a = \#k_{s32})\ ?\ 1 : 0$		
`SNEI w,a,#k`	$w := (a \neq \#k_{s32})\ ?\ 1 : 0$		
`SLTI w,a,#k`	$w := (a < \#k_{s32})\ ?\ 1 : 0$		
`SGTI w,a,#k`	$w := (a > \#k_{s32})\ ?\ 1 : 0$		
`SLEI w,a,#k`	$w := (a \leq \#k_{s32})\ ?\ 1 : 0$		
`SGEI w,a,#k`	$w := (a \geq \#k_{s32})\ ?\ 1 : 0$		

HaDes offers some instructions for jumps and branches. The destination address is a signed 12-bit offset (immediate value) of PC+1, or the absolute address in a register. The branch of the `BOV` depends on the last instruction that effects the OV flag.

Instruction	Semantic	Effects	
		OV	ZERO
`BEQZ a,#k`	$PC := (!a)\ ?\ ((PC + 1 + \#k_s) \bmod 4096) : (\text{PC} + 1)$		X
`BNEZ a,#k`	$PC := (a)\ ?\ ((PC + 1 + \#k_s) \bmod 4096) : (\text{PC} + 1)$		X
`BOV #k`	$PC := (\text{OV})\ ?\ ((PC + 1 + \#k_s) \bmod 4096) : (\text{PC} + 1)$		
`JMP #k`	`BEQZ 0,#k`		X
`JAL w,#k`	$w := PC + 1; PC := ((\text{PC} + 1 + \#k_s) \bmod 4096)$		
`JREG a`	$\text{PC} := a$		

The next instructions are used to transfer data between registers and memory. The 16-bit immediate signed value #*k* will be extended to a 32-bit signed value. If the data memory address is negative or greater than its size, an XMEMINTR interrupt will be raised. The `STORE` instruction writes to the data memory or to the program memory, depending on the program memory access status.

Instruction	Semantic	Effects	
		OV	ZERO
`LOAD w,a,#k`	$w :=$ XDMEMORY$[a + \#k_{s32}]$	X	
`STORE b,a,#k`	XDMemory$[a + \#k_{s32}] := b$ (after `DPMA`)	X	
	PMemory$[(a + \#k_{s32}) \bmod 4096] := b$ (before `DPMA`)	X	
`DPMA`	Disable program memory access		
`EPMA`	Enable program memory access		

The next instructions are used to write and read information to and from the peripheral, respectively. The 16-bit constant value is interpreted as unsigned, and is used to choose the peripheral register.

Instruction	Semantic	Effects	
		OV	ZERO
`IN w,#k`	$w :=$ IOReg$[\#k_u]$		
`OUT a,#k`	IOReg$[\#k_u] := a$		

The HaDes CPU offers four different interrupts. Each one has to set the entry address with the `SISA` instruction. The entry point address is the signed 12-bit offset (immediate value) of PC+1. The `SWI` instruction is used to raise a software interrupt. It is executed with two parameters, which can be read out in the interrupt handler.

Instruction	Semantic	Effects	
		OV	ZERO
`ENI`	Enable Interrupt		
`DEI`	Disable Interrupt		
`SISA #0,#k`	Set Software interrupts entry address := $(PC + 1 + \#k_s) \bmod 4096$		
`SISA #1,#k`	Set XPeripheral interrupts entry address := $(PC + 1 + \#k_s) \bmod 4096$		
`SISA #2,#k`	Set XBusNotAvailable interrupts entry address := $(PC + 1 + \#k_s) \bmod 4096$		
`SISA #3,#k`	Set XMemory interrupts entry address := $(PC + 1 + \#k_s) \bmod 4096$		
`RETI`	Return to last interrupt level		
`SWI a, #k`	Raise software interrupt with the arguments $\#k$ and a		
`GETSWI w,#0`	$w := \#k$ of the last software interrupt		
`GETSWI w,#1`	$w := a$ of the last software interrupt		

The assembler offers some instructions which have not an own op-code; they are also called syntactic sugar. They will use other instruction to describe its behavior.

Instruction	Semantic	Effects	
		OV	ZERO
`LDI w,#k`	`ADDI w,r0,#k` (Load Signed)		
`LDUI w,#k`	`ORI w,r0,#k` (Load Unsigned)		
`MOV a,w`	`ADDI w,a,#0` (Move Register)		
`INC a`	`ADDI a,a,#1` (Increment)	X	
`DEC a`	`SUBI a,a,#1` (Decrement)	X	

Acronyms

ABI	Application Binary Interface
ADC	Analog-to-Digital-Converter
ASCII	American Standard Code for Information Interchange
ALU	Arithmetic Logical Unit
ASM	Assembly
BCD	Binary Coded Decimal
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GPR	General Purpose Register
HDL	Hardware Description Language
HID	Human Interface Device
HMI	Human Machine Interface
IEEE	Institute of Electrical and Electronics Engineers
IC	Integrated Circuit
IRQ	Interrupt Request
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LED	Light Emitting Diode
LSB	Least Significant Bit
MCU	Microcontroller Unit
MMCM	Mixed-Mode Clock Manager
MRCC	Multi-region Clock-Capable
MSB	Most Significant Bit
OV	Overflow
PC	Program Counter
PLL	Phase Locked Loop
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RS232	Recommended Standard 232
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VGA	Video Graphics Array

Bibliography

Digilent. Basys3 Schematic. https://reference.digilentinc.com/_media/basys3:basys3_sch.pdf, 2014.

Digilent. Basys3 FPGA Board Reference Manual. https://reference.digilentinc.com/_media/reference/programmable-logic/basys-3/basys3_rm.pdf, March 2017.

Xilinx. XA Artix-7 FPGAs Overview. http://www.xilinx.com/support/documentation/data_sheets/ds197-xa-artix7-overview.pdf, October 2014.

Document Revision History

This document's revision history and changelog.

- 24.02.2016 – Initial release
- 04.01.2017 – Adaptation to summer term 2017
- 09.02.2017 – Various fixes and clarifications
- 19.12.2017 – Adaptation to summer term 2018
- 20.02.2019 – Adaptation to summer term 2019 (lecturer change: Mauroner - Scheipel)
- 23.04.2019 – Language improvements implemented (still ongoing)
- 06.09.2019 – Adaptation to summer term 2020
- 11.02.2021 – Restructured and adapted to summer term 2021. Language updated by Gabriel Stoll.
- 24.01.2022 – Adaptation to summer term 2022
- 02.01.2022 – Adaptation to summer term 2023