Java Environment for Parallel Real-Time Development

Project Number 216682

D2.4– Multiprocessor JOP Documentation

Version 0.3
September 2, 2009
Draft

Public Distribution

Martin Schoeberl, Wolfgang Puffitsch
Technical University of Vienna

Project Partners: **aicas**, **EADS Deutschland**, **FZI**, **RadioLabs**, **SkySoft Portugal**, **SYSGO**, **Technical University Cluj-Napoca**, **The Open Group**, **Technical University of Vienna**, **University of York**

Project Partner Contact Information

aicas Fridtjof Siebert Haid-und-Neu-Strasse 18 76131 Karlsruhe, Germany Tel:+49 721 663 968 23 Fax:+49 721 663 968 93 E-mail:siebert@aicas.com	**EADS Deutschland** Thomas Mahr Woerthstrasse 85 89077 Ulm, Germany Tel: +49 731 392 7469 Fax: +49 731 392 2074 69 E-mail:thomas.mahr@eads.com
FZI Gábor Szeder Haid-und-Neu-Strasse 10-14 76131 Karlsruhe, Germany Tel: +49 721 965 4266 Fax: +49 721 965 4259 E-mail:szeder@fzi.de	**RadioLabs** Filippo Corsini via Cavaglieri 26 00173 Rome, Italy Tel: +39 069 727 8250 Fax: +39 069 727 8268 E-mail:filippo.corsini@radiolabs.it
SkySoft Portugal José Neves Av. D. João II, Torre Fernão Magalhães, 7º 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 Fax: +351 21 386 6493 E-mail:jose.neves@skysoft.pt	**SYSGO** Jacques Brygier 5, Rue Hans List, Batiment F 78290 Croissy-sur-Seine, France Tel: +33 1 300 912 63 Fax: +33 1 301 504 48 E-mail:jacques.brygier@sysgo.fr
Technical University Cluj-Napoca Gheorghe Sebestyen-Pal G. Baritiu 26-28 40027 Cluj-Napoca, Romania Tel:+40 264 401 476 Fax:+40 264 594 491 E-mail:gheorghe.sebestyen@cs.utcluj.ro	**Technical University of Vienna** Martin Schoeberl Treitlstrasse 3 1040 Vienna, Austria Tel: +43 15 880 118 207 Fax: +43 15 869 149 E-mail:mschoebe@mail.tuwien.ac.at
The Open Group Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 675 7721 E-mail:s.hansen@opengroup.org	**University of York** Andrew Wellings Heslington Hall York YO10 5DD, United Kingdom Tel: +44 1904 432 742 Fax: +44 1904 432 767 E-mail:andy@cs.york.ac.uk

Contents

Document Control

Version	Status	Date
0.1	Document start	25 April 2009
0.2	Draft to partners	28 August 2009
0.3	Reworked draft	2 September 2009

Executive Summary

This documents contains the deliverable *D2.4 Multiprocessor JOP Documentation* of work-package 2 of the JEOPARD project due 20 months after project start as stated in the Description of Work. This document presents the user documentation of the JEOPARD JOP multi-core platform as described in deliverable *D2.1*.

1 Introduction

This document gives a brief description of the JOP architecture and the multiprocessor version of JOP. A more detailed description can be found in the *JOP Reference Handbook: Building Embedded Systems with a Java Processor* [13] available online under:

- `http://www.jopdesign.com/doc/handbook.pdf`

or as a printed book from Amazon:

- `http://amzn.com/1438239696`

2 Build Instructions

JOP [10], the Java optimized processor, is an open-source development platform available for different targets (Altera and Xilinx FPGAs and various types of FPGA boards). To support several targets, the resulting design-flow is a little bit complex. There is a `Makefile` available and when everything is set up correctly, a simple

```
make
```

should build everything from the sources and download a *Hello World* example. However, to customize the `Makefile` for a different target it is necessary to understand the complete design flow. It should be noted that an Ant[1] based build process is also available.

2.0.1 Tools

All needed tools are freely available.

- Java SE Development Kit (JDK) Java compiler and runtime
- Cygwin GNU tools for Windows. Packages git, gcc and make are needed
- Quarts II Web Edition VHDL synthesis, place and route for Altera FPGAs

The PATH variable should contain entries to the executables of all packages (java and javac, Cygwin bin, and Quartus executables). Check the PATH at the command prompt with:

```
javac
gcc
make
 git
quartus_map
```

All the executables should be found and usually report their usage.

2.0.2 Getting Started

This subsection shows a quick step-by-step build of JOP for the Cyclone target in the minimal configuration. All directory paths are given relative to the JOP root directory `jop`. The build process is explained in more detail in one of the following subsections.

[1] `http://ant.apache.org/`

Download the Source Create a working directory and download JOP from the GIT server:

```
git clone git://www.soc.tuwien.ac.at/jop.git
```

For a write access clone (for developers) use following URL:

```
git clone ssh://user@www.soc.tuwien.ac.at/home/git/jop.git
```

All sources are downloaded to a directory `jop`. For the following command change to this directory. Create the needed directories with:

```
make directories
```

Tools The tools contain `Jopa`, the microcode assembler, `JopSim`, a Java based simulation of JOP, and `JOPizer`, the application builder. The tools are built with following make command:

```
make tools
```

Assemble the Microcode JVM, Compile the Processor The JVM configured to download the Java application from the serial interface is built with:

```
make jopser
```

This command also invokes Quartus to build the processer. If you want to build it within Quartus follow the following instructions:

1. Start Quartus II and open the project `jop.qpf` from directory `quartus/cycmin` in Quartus with *File – Open Project...*.
2. Start the compiler and fitter with *Processing – Start Compilation*.
3. After successful compilation the FPGA is configured with *Tools – Programmer* and *Start*.

Compiling and Downloading the Java Application A simple *Hello World* application is the default application in the Makefile. It is built and downloaded to JOP with:

```
make japp
```

The “Hello World" message should be printed in the command window.

For a different application change the Makefile targets or override the `make` variables at the command line. The following example builds and runs some benchmarks on JOP:

```
make japp -e P1=bench P2=jbe P3=DoAll
```

The three variables `P1`, `P2`, and `P3` are a shortcut to set the directory, the package name, and the main class of the application.

USB based Boards Several Altera based boards use an FTDI FT2232 USB chip for the FPGA and Java program download. To change the download flow for those boards change the value of the following variable in the Makefile to `true`:

```
USB=true
```

The Java download channel is mapped to a virtual serial port on the PC. Check the port number in the system properties and set the variable `COM_PORT` accordingly.

2.0.3 Xilinx Spartan-3 Starter Kit

The Xilinx tool chain is not well supported by the Makefile or the Ant design flow. Here is a short list on how to build JOP for a Xilinx board:

```
make tools
cd asm
jopser
cd ..
```

Now start the Xilinx IDE wirh the project file `jop.npl`. It will be converted to a new (binary) `jop.ise` project. The `.npl` project file is used as it is simple to edit (ASCII).

- Generate JOP by double clicking 'Generate PROM, ACE, or JTAG File'
- Configure the FPGA according to the board type

The above is a one step build for the processor. The Java application is built and downloaded by:

```
make japp
```

The Java program should now run on JOP/Spartan-3.

2.1 Booting JOP — How Your Application Starts

Basically this is a two step process: (a) configuration of the FPGA and (b) downloading the Java application. There are different possibilities to perform these steps.

2.1.1 FPGA Configuration

FPGAs are usually SRAM based and lose their configuration after power down. Therefore the configuration has to be loaded on power up. For development the FPGA can be configured via a download cable (with JTAG commands). This can be done within the IDEs from Altera and Xilinx or with command line tools such as `quartus_pgm` or `jbi32`.

For the device to boot automatically, the configuration has to be stored in non volatile memory such as Flash. Serial Flash is directly supported by an FPGA to boot on power up. Another method is to use a standard parallel Flash to store the configuration and additional data (e.g. the Java application). A small PLD reads the configuration data from the Flash and shifts it into the FPGA. This method is used on the Cyclone and ACEX boards.

2.1.2 Java Download

When the FPGA is configured the Java application has to be downloaded into the main memory. This download is performed in microcode as part of the JVM startup sequence. The application is a `.jop` file generated by `JOPizer`. At the moment there are three options:

Serial line JOP listens to the serial line and the data is written into the main memory. A simple echo protocol performs the flow control. The baud rate is usually 115 kBaud.

USB Similar to the serial line version, JOP listens to the parallel interface of the FTDI FT2232 USB chip. The FT2232 performs the flow control at the USB level and the echo protocol is omitted.

Flash For stand alone applications the Java program is copied from the Flash (relative Flash address 0, mapped Flash address is 0x80000[2]) to the main memory (usually a 32-bit SRAM).

The mode of downloading is defined in the JVM (`jvm.asm`). To select a new mode, the JVM has to be assembled and the complete processor has to be rebuilt – a full `make` run. The generation is performed by the C preprocessor (`gcc`) on `jvm.asm`. The `make` targets `jopser`, `jopusb` and `jopflash` set the appropriate preprocessor definitions.

VHDL Simulation To speed up the VHDL simulation in ModelSim there is a forth method where the Java application is loaded by the test bench instead of JOP. This version is generated by the `make` target `sim`. The actual Java application is written by `jop2dat` into a plain text file (`mem_main.dat`) and read by the simulation test bench into the simulated main memory.

2.1.3 Combinations

Theoretically all variants to configure the FPGA can be combined with all variations to download the Java application. However, only two combinations are commonly used:

1. For VHDL or Java development configure the FPGA via the download cable and download the Java application via the serial line or USB.
2. For a stand-alone application load the configuration and the Java program from the Flash.

2.1.4 Stand Alone Configuration

The Cycore board can be configured to configure the FPGA and load the Java program from Flash at power up. In order to prepare the Cycore board for this configuration the Flash must be programmed. Depending on the I/O capabilities several options are possible:

SLIP With a SLIP connection the Flash can be programmed via TFTP. For this configuration a second serial line is needed.

Ethernet With an Ethernet connection (e.g., the baseio board) TFTP can be used for Flash programming.

Serial Line With a single serial line the utilities `util.Mem.java` and `amd.exe` can be used to program the Flash.

The following text describes the Flash programming and PLD reconfiguration for a stand alone configuration. Fist we have to build a JOP version that will load a Java program from the Flash:

```
make jopflash
```

[2] All addresses in JOP are counted in 32-bit quantities. However, the Flash is connected only to the lower 8 bits of the data bus. Therefore a store of one word in the main memory needs four loads from the Flash.

As usual a `jop.sof` file will be generated. For easier reading of the configuration it will be converted to `jop.ttf`. This file will be programmed into the Flash starting at address 0x40000. Therefore, we need to save that file and rebuild a JOP version that loads a Java program (the Flash programmer) from the serial line:

```
copy quartus\cycmin\jop.ttf  ttf \cycmin.ttf
make jopser
```

As a next step we will build the Java program that will be programmed into the Flash and save a copy of the `.jop` file. `Hello.java` is the embedded version of a *Hello World* program that blinks the watchdog LED at 1 Hz.

```
make java_app -e P1=test P2=test P3=Hello
copy java\target\dist\bin\Hello.jop .
```

To program the Flash the programmer tool `util.Mem` will run on JOP and `amd.exe` is used at the PC side:

```
make japp -e P1=common P2=util P3=Mem COM_FLAG=
amd Hello.jop COM1
amd ttf\cycmin.ttf  COM1
```

As a last step the PLD will be programmed to enable FPGA configuration form the Flash:

```
make pld_conf
```

The board shall now boot after a power cycle and the LED will blink. To read the output from the serial line the small utility `e.exe` can be used.

In the case the PLD configuration shall be changed back to JTAG FPGA configuration following make command will reset the PLD:

```
make pld_init
```

Note, that in a stand alone configuration the watchdog (WD) pin has to be toggled every second (e.g., by invoking `util.Timer.wd()`. When the WD is not toggled the FPGA will be reconfigured after 1.6 seconds.

2.2 The Design Flow

This subsection describes the design flow to build JOP in greater detail.

2.2.1 Tools

There are a few tools necessary to build and download JOP to the FPGA boards. Most of them are written in Java. Only the tools that access the serial line are written in C.[3]

[3]The Java JDK still comes without the javax.comm package and getting this optional package correctly installed is not that easy.

Downloading These little programs are already compiled and the binaries are checked in into the repository. The sources can be found in directory `c_src`.

down.exe The workhorse to download Java programs. The mandatory argument is the COM-port. Optional switch `-e` keeps the program running after the download and echoes the characters from the serial line (`System.out` in JOP) to stdout. Switch `-usb` disables the echo protocol to speed up the download over USB.

JavaDown A replacement for down.exe, written in Java. It allows downloading Java programs in a platform-independent way.

e.exe Echoes the characters from the serial line to stdout. Parameter is the COM-port.

amd.exe A utility to send data over the serial line to program the on-board Flash. The complementary Java program `util.Mem` must be running on JOP.

USBRunner.exe Download the FPGA configuration via USB with the FTDI2232C chip (dspio board). Also present for Linux.

Generation of Files These tools are written in Java and are delivered in source form. The source can be found under `java/tools/src` and the class files are in `jop-tools.jar` in directory `java/tools/dist/lib`.

Jopa The JOP assembler. Assembles the microcoded JVM and produces on-chip memory initialization files and VHDL files.

BlockGen converts Altera memory initialization files to VHDL files for a Xilinx FPGA.

JOPizer links a Java application and converts the class information to the format that JOP expects (a `.jop` file). `JOPizer` uses the bytecode engineering library[4] (BCEL).

Simulation

JopSim reads a `.jop` file and executes it in a debug JVM written in Java. Command line option `-Dlog="true"` prints a log entry for each executed JVM bytecode.

pcsim simulates the BaseIO expansion board for Java debugging on a PC (using the JVM on the PC).

2.2.2 Targets

JOP has been successfully ported to several different FPGAs and boards. The main distribution contains the ports for the FPGAs:

- Altera Cyclone EP1C6 or EP1C12
- Xilinx Spartan-3
- Altera Cyclone-II (Altera DE2 board)
- Xilinx Virtex-4 (ML40x board)
- Xilinx Spartan-3E (Digilent Nexys 2 board)

[4] `http://jakarta.apache.org/bcel/`

I/O board	Quartus	I/O top level
simpexp, baseio	`cycmin`	`scio_min.vhd`
dspio	`usbmin`	`scio_dspiomin.vhd`
baseio	`cycbaseio`	`scio_baseio.vhd`
bg263	`cybg`	`scio_bg.vhd`
lego	`cyclego`	`scio_lego.vhd`
dspio	`dspio`	`scio_dspio.vhd`

Table 1: Quartus project directories and VHDL files for the different I/O boards

For the current list of the supported FPGA boards see the list at the web site.[5] Besides the ports to different FPGAs there are ports to different boards.

Cyclone EP1C6/12 This board is the workhorse for the JOP development and comes in two versions: with an Cyclone EP1C6 or EP1C12. The board contains:

- Altera Cyclone EP1C6Q240 or EP1C12Q240 FPGA
- 1 MB fast SRAM
- 512 KB Flash (for FPGA configuration and program code)
- 32 MB NAND Flash
- ByteBlasterMV port
- Watchdog with LED
- EPM7064 PLD to configure the FPGA from the Flash (on watchdog reset)
- Voltage regulator (1V5)
- Crystal clock (20 MHz) at the PLL input (up to 640 MHz internal)
- Serial interface (MAX3232)
- 56 general purpose I/O pins

The Cyclone specific files are `jopcyc.vhd` or `jopcyc12` and `mem32.vhd`. This FPGA board is designed as a module to be integrated with an application specific I/O-board. There exist following I/O-boards:

simpexp A simple bread board with a voltage regulator and a SUBD connector for the serial line

baseio A board with Ethernet connection and EMC protected digital I/O and analog input

bg263 Interface to a GPS receiver, a GPRS modem, keyboard and a display for a railway application

lego Interface to the sensors and motors of the LEGO Mindstorms. This board is a substitute for the LEGO RCX.

dspio Developed at the University of Technology Vienna, Austria for digital signal processing related work. All design files for this board are open-source.

Table 1 lists the related VHDL files and Quartus project directories for each I/O board.

[5] `http://www.jopwiki.com/FPGA_boards`

Project	Content
`jop`	The target sources
`joptools`	Tools such as `Jopa`, `JopSim`, and `JOPizer`
`pc`	Some PC utilities (e.g. Flash programming via UDP/IP)
`pcsim`	Simulation of the basio hardware on the PC

Table 2: Eclipse projects

Xilinx Spartan-3 The Spartan-3 specific files are `jop_xs3.vhd` and `mem_xs3.vhd` for the Xilinx Spartan-3 Starter Kit and `jop_trenz.vhd` and `mem_trenz.vhd` for the Trenz Retro-computing board.

2.3 Eclipse

In folder `eclipse` there are four Eclipse projects that you can import into your Eclipse workspace. However, do not use *that* directory as your workspace directory. Choose a directory outside of the JOP source tree for the workspace (e.g., your usual Eclipse workspace) and copy the for project folders `joptarget`, `joptools`, `pc`, and `pcsim`.

All projects use the Eclipse path variable[6] `JOP_HOME` that has to point to the root directory (`.../jop`) of the JOP sources. Under *Window – Preferences...* select *General – Workspace – Linked Resources* and create the path variable `JOP_HOME` with *New...*.

Import the projects with *File – Import..* and *Existing Projects into Workspace*. It is suggested to an Eclipse workspace that is not part of the jop source tree. Select as the root directory (e.g., your Eclipse workspace), select the projects you want to import, select *Copy projects into workspace*, and press *Finish*. Table 2 shows all available projects.

Add the libraries from `.../jop/java/lib` (as external archives) to the build path (right click on the `joptools` project) of the project `joptools`.[7]

2.4 Simulation

This subsection contains the information you need to get a simulation of JOP running. There are two ways to simulate JOP:

- High-level JVM simulation with `JopSim`
- VHDL simulation (e.g. with ModelSim)

2.4.1 JopSim Simulation

The high level simulation with `JopSim` is a simple JVM written in Java that can execute the JOP specific application (the `.jop` file). It is started with:

```
make jsim
```

To output each executing bytecode during the simulation run change in the Makefile the logging parameter to `-Dlog="true"`.

[6] Eclipse (path) variables are workspace specific.
[7] Eclipse can't use path variables for external .jar files.

VHDL file	Function	Initialization file	Generator
sim_jop_types_100.vhd	JOP constant definitions	-	-
sim_rom.vhd	JVM microcode ROM	mem_rom.dat	Jopa
sim_ram.vhd	Stack RAM	mem_ram.dat	Jopa
sim_jbc.vhd	Bytecode memory (cache)	-	-
sim_memory.vhd	Main memory	mem_main.dat	jop2dat
sim_pll.vhd	A dummy entity for the PLL	-	-
sim_uart.vhd	Print characters to stdio	-	-

Table 3: Simulation specific VHDL files

2.4.2 VHDL Simulation

This subsection is about running a VHDL simulation with ModelSim. All simulation files are vendor independent and should run on any versions of ModelSim or a different VHDL simulator. You can simulate JOP even with the free ModelSim XE II Starter Xilinx version, the ModelSim Altera version or the ModelSim Actel version.

To simulate JOP, or any other processor design, in a vendor neutral way, models of the internal memories (block RAM) and the external main memory are necessary. Beside this, only a simple clock driver is necessary. To speed-up the simulation a little bit, a simulation of the UART output, which is used for `System.out.print()`, is also part of the package.

Table 3 lists the simulation files for JOP and the programs that generates the initialization data. The non-generated VHDL files can be found in directory `vhdl/simulation`. The needed VHDL files and the compile order can be found in `sim.bat` under `modelsim`.

The actual version of JOP contains all necessary files to run a simulation with ModelSim. In directory `vhdl/simulation` you will find:

- A test bench: `tb_jop.vhd` with a serial receiver to print out the messages from JOP during the simulation
- Simulation versions of all memory components (vendor neutral)
- Simulation of the main memory

`Jopa` generates various `mem_xxx.dat` files that are read by the simulation. The JVM that is generated with `jopsim.bat` assumes that the Java application is preloaded in the main memory. `jop2dat` generates a memory initialization file from the Java application file (`MainClass.jop`) that is read by the simulation of the main memory (`sim_memory.vhd`).

In directory `modelsim` you will find a small batch file (`sim.bat`) that compiles JOP and the test bench in the correct order and starts ModelSim. The whole simulation process (including generation of the correct microcode) is started with:

```
make sim
```

After a few seconds you should see the startup message from JOP printed in ModelSim's command window. The simulation can be continued with `run -all` and after around 6 ms *simulation time* the actual Java `main()` method is executed. During those 6 ms, which will probably be minutes of simulation, the memory is initialized for the garbage collector.

2.5 Files Types You Might Encounter

As there are various tools involved in the complete build process, you will find files with various extensions. The following list explains the file types you might encounter when changing and building JOP.

The following files are the *source* files:

`.vhd` VHDL files describe the hardware part and are compiled with either Quartus or Xilinx ISE. Simulation in ModelSim is also based on VHDL files.

`.v` Verilog HDL. Another hardware description language.

`.java` Java — the language that runs native on JOP.

`.c` There are still some tools written in C.

`.asm`, `.inc` JOP microcode. The JVM is written in this stack oriented assembler. Files are assembled with `Jopa`. The result are VHDL files, `.mif` files, and `.dat` files for ModelSim.

`.bat` These DOS batch files are a legacy of the original build process and are mostly deprecated.

`.xml` Project files for Ant. Ant is an attractive substitution to `make`.

Quartus II and Xilinx ISE need configuration files that describe your project. All files are usually ASCII text files.

`.qpf` Quartus II Project File. Contains almost no information.

`.qsf` Quartus II Settings File defines the project. VHDL files that make up your project are listed. Constraints such as pin assignments and timing constraints are set here.

`.cdf` Chain Description File. This file stores device name, device order, and programming file name information for the programmer.

`.tcl` Tool Command Language. Can be used in Quartus to automate parts of the design flow (e.g. pin assignment).

`.npl` Xilinx ISE project. VHDL files that make up your project are listed. The actual version of Xilinx ISE converts this project file to a new format that is not in ASCII anymore.

`.ucf` Xilinx Foundation User Constraint File. Constraints such as pin assignments and timing constraints are set here.

The Java tools `javac` and `jar` produce following file types from the Java sources:

`.class` A class file contains the bytecodes, a symbol table and other ancillary information and is executed by the JVM.

`.jar` The Java Archive file format enables you to bundle multiple files into a single archive file. Typically a `.jar` file contains the class files and auxiliary resources. A `.jar` file is essentially a zip file that contains an optional `META-INF` directory.

The following files are generated by the various tools from the source files:

`.jop` This file makes up the linked Java application that runns on JOP. It is generated by `JOPizer` and can be either downloaded (serial line or USB) or stored in the Flash (or used by the simulation with `JopSim` or ModelSim)

.mif Memory Initialization File. Defines the initial content of on-chip block memories for Altera devices.

.dat memory initialization files for the simulation with ModelSim.

.sof SRAM Output File. Configuration file for Altera devices. Used by the Quartus programmer or by `quartus_pgm`. Can be converted to various (or too many) different formats. Some are listed below.

.pof Programmer Object File. Configuration for Altera devices. Used for the Flash loader PLDs.

.jbc JamTM STAPL Byte Code 2.0. Configuration for Altera devices. Input file for `jbi32`.

.ttf Tabular Text File. Configuration for Altera devices. Used by flash programming utilities (`amd` and `udp.Flash` to store the FPGA configuration in the boards Flash.

.rbf Raw Binary File. Configuration for Altera devices. Used by the USB download utility (`USBRunner`) to configure the dspio board via the USB connection.

.bit Bitstream File. Configuration file for Xilinx devices.

2.6 Information on the Web

Further information on JOP and the build process can be found on the Internet at the following places:

- `http://www.jopdesign.com/` is the main web site for JOP
- `http://www.jopwiki.com/` is a Wiki that can be freely edited by JOP users.
- `http://tech.groups.yahoo.com/group/java-processor/` hosts a mailing list for discussions on Java processors in general and mostly on JOP related topics

2.7 Porting JOP

Porting JOP to a different FPGA platform or board usually consists of adapting pin definitions and selection of the correct memory interface. Memory interfaces for the SimpCon interconnect can be found in directory `vhdl/memory`.

2.7.1 Test Utilities

To verify that the port of JOP is successful there are some small test programs in `asm/src`. To run the JVM on JOP the microcode `jvm.asm` is assembled and will be stored in an on-chip ROM. The Java application will then be loaded by the first microcode instructions in `jvm.asm` into an external memory. However, to verify that JOP and the serial line are working correctly, it is possible to run small test programs directly in microcode.

One test program (`blink.asm`) does not need the main memory and is a first test step before testing the possibly changed memory interface. `testmon.asm` can be used to debug the main memory interface. Both test programs can be built with the `make` targets `jop_blink_test` and `jop_testmon`.

Blinking LED and UART output The test is built with:

```
make jop_blink_test
```

After download, the watchdog LED should blink and the FPGA will print out 0 and 1 on the serial line. Use a terminal program or the utility `e.exe` to check the output from the serial line.

Test Monitor Start a terminal program (e.g. HyperTerm) to communicate with the monitor program and build the test monitor with:

```
make jop_testmon
```

After download the program prints the content of the memory at address 0. The program understands following *commands*:

- A single CR reads the memory at the current addres and prints out the address and memory content
- `addr=val;` writes val into the memory location at address $addr$

One tip: Take care that your terminal program does not send an LF after the CR.

2.8 Extending JOP

JOP is a soft-core processor and customizing it for an application is an interesting opportunity.

2.8.1 Native Methods

The *native* language of JOP is microcode. A native method is implemented in JOP microcode. The interface to this native method is through a *special* bytecode. The mapping between native methods and the special bytecode is performed by `JOPizer`. When adding a new (*special*) bytecode to JOP, the following files have to be changed:

1. `jvm.asm` implementation
2. `Native.java` method signature
3. `JopInstr.java` mapping of the signature to the name
4. `JopSim.java` simulation of the bytecode
5. `JVM.java` (just rename the method name)
6. `Startup.java` (only when needed in a class initializer)
7. `WCETInstruction.java` timing information

First implement the native code in `JopSim.java` for easy debugging. The *real* microcode is added in `jvm.asm` with a label for the special byctecode. The naming convention is `jopsys_name`. In `Native.java` provide a method signature for the native method and enter the mapping between this signature and the name in `jvm.asm` and in `JopInstr.java`. Provide the execution time in `WCETInstruction.java` for WCET analysis.

The native method is accessed by the method provided in `Native.java`. There is no calling overhead involved in the mechanism. The *native* method gets substituted by `JOPizer` with a *special* bytecode.

2.8.2 A new Peripheral Device

Creation of a new peripheral devices involves some VHDL coding. However, there are several examples in `jop/vhdl/scio` available.

All peripheral components in JOP are connected with the SimpCon [12] interface. For a device that implements the Wishbone [5] bus, a SimpCon-Wishbone bridge (`sc2wb.vhd`) is available (e.g., it is used to connect the AC97 interface in the `dspio` project).

For an easy start use an existing example and change it to your needs. Take a look into `sc_test_slave.vhd`. All peripheral components (SimpCon slaves) are connected in one module usually named `scio_xxx.vhd`. Browse the examples and copy one that best fits your needs. In this module the address of your peripheral device is defined (e.g. 0x10 for the primary UART). This I/O address is mapped to a negative memory address for JOP. That means 0xffffff80 is added as a base to the I/O address.

By convention this address mapping is defined in `com.jopdesign.sys.Const`. Here is the UART example:

```
// use negative base address for fast constant load
// with bipush
public static final int IO_BASE = 0xffffff80;
...
public static final int IO_STATUS = IO_BASE+0x10;
public static final int IO_UART = IO_BASE+0x10+1;
```

The I/O devices are accessed from Java by *native*[8] functions: `Native.rdMem()` and `Native.wrMem()` in package `com.jopdesign.sys`. Again an example with the UART:

```
// busy wait on free tx buffer
// no wait on an open serial line, just wait
// on the baud rate
while ((Native.rdMem(Const.IO_STATUS)&1)==0) {
    ;
}
Native.wrMem(c, Const.IO_UART);
```

Best practise is to create a new I/O configuration `scio_xxx.vhdl` and a new Quartus project for this configuration. This avoids the mixup of the changes with a new version of JOP. For the new Quartus project only the three files `jop.cdf`, `jop.qpf`, and `jop.qsf` have to be copied in a new directory under `quartus`. This new directory is the project name that has to be set in the Makefile:

```
QPROJ=yourproject
```

The new VHDL module and the `scio_xxx.vhdl` are added in `jop.qsf`. This file is a plain ASCII file and can be edited with a standard editor or within Quartus.

2.8.3 A Customized Instruction

A customized instruction can be simply added by implementing it in microcode and mapping it to a native function as described before. If you want to include a hardware module that implements this instruction a new microinstruction has to be introduced. Besides mapping this instruction to a native method the instruction has also be added to the microcode assembler `Jopa`.

[8] These are not real functions and are substituted by special bytecodes on application building with JOPizer.

2.8.4 Dependencies and Configurations

As JOP and the JVM are a mix of VHDL and Java files, of changes some configurations or changes in central data structures needs an update in several files.

Speed Configuration By default, JOP is configured for 80 Mhz. To build the 100 MHz configuration, edit `quartus/cycmin/jop.qsf` and change `jop_config_80` to `jop_config_100`.

Method Cache Configuration The default configuration (for the Altera Cyclone) is a 4 KB method cache configured with 16 blocks (i.e., a variable block cache). For the Xilinx targets, the cache size is 2KB because Xilinx does not (or did not) support easily configurable block RAMs.

To change from a variable block cache to a dual-block cache, you will need to edit the top-level VHDL. Here is an example from `vhdl/top/jopcyc.vhd`:

```
entity jop is

generic (
    ram_cnt : integer := 2;   -- clock cycles for external ram
    rom_cnt : integer := 15; -- clock cycles for external rom for 100 MHz
    jpc_width : integer := 12; -- address bits of java bytecode pc = cache size
    block_bits : integer := 4  -- 2*block_bits is number of cache blocks
);
```

The power of 2 of the `jpc_width` is the cache size, and the power of 2 of the `block_bits` is the number of blocks. To simulate a dual block cache, `block_bits` has to be set to 1. (To use a single block cache, `cache.vhd` has to be modified to force a miss at the cache lookup.)

Stack Size The on-chip stack size can be configured by changing following constants:

- `ram_width` in `jop_config_xx.vhd`
- `STACK_SIZE` in `com.jopdesign.sys.Const`
- `RAM_LEN` in `com.jopdesign.sys.Jopa`

3 WCET Analysis

Worst-case execution time (WCET) estimates of tasks are essential for designing and verifying real-time systems. WCET estimates can be obtained either by measurement or static analysis. The problem with using measurements is that the execution times of tasks tend to be sensitive to their inputs. Measurement does not guarantee safe WCET estimates. Instead, static analysis is necessary for hard real-time systems.

The WCET analysis is split into several phases. First, the control flow graph of the program is generated, and loop bounds are annotated (manually or automatically). Then, the execution time of each basic block is computed. In a third step, global effects of hardware features such as caches are determined. Finally, the WCET is computed by transforming the control flow graph with the timing information and program annotations into an integer linear programming (ILP) problem and solving that problem.

The first WCET analysis tool that targets JOP has been developed by Rasmus Pedersen [15]. Benedikt Huber implemented a new version of the analyzer with support of bytecodes implemented in Java and a better method cache approximation has [3]. The new tool also contains a module to extract the low-level bytecode timing from the microcode assembler program (`jvm.asm`) automatically. Both programs use the ILP solve `lp_solve`.[9] Furthermore, a framework for automated loop bound detection [9] has been developed and integrated into the WCET analysis tools.

JOP users who want to perform WCET analysis must take into account two fundamental limitations of static WCET analysis:

- Programs must not contain recursion.
- The upper bound of each loop has to be known.

These requirements may seem straight-forward to fulfill. However, one has to take care that library functions such as concatenating two strings often lead to loops without obvious bounds.

3.1 WCET Analysis by Example

Listing 1 shows the example we will use throughout this section. The example does not contain any recursion and all loops are bounded, so the fundamental requirements for WCET analysis are fulfilled.

Now assume that the method `test.WcetExample.run()` is called from a program `test.Main`. After the Java program has been built, the following command is used to compute the WCET for `test.WcetExample.loop()`:

```
make wcet P1=test P2=test P3=Main WCET_METHOD=test.WcetExample.loop WCET_DFA=yes
```

The parameters `P1`, `P2`, and `P3` select the main class. The variable `WCET_METHOD` selects the method for which the WCET should be computed. `WCET_DFA=yes` enables the data-flow analysis for automatic loop bound detection. Please note that the `lp_solve` library must be made visible to the JVM through the system property `java.library.path`. Further options for the WCET analysis tool can be displayed through the make target `wcet_help`. In the current default configuration, the above command should report a WCET of 1573 cycles. Of course, different memory or bytecode timings would change this result.

3.1.1 Manual loop annotation

Data-flow analysis cannot find all loop bounds. Also, it may be too costly for large programs. In such cases, the loop bounds have to be annotated manually. If, for example, the loop bound for the outer loop in Listing 1 cannot be determined, the following annotation would be appropriate:

[9] `http://lpsolve.sourceforge.net/5.5/`

```
package test;

public class WcetExample {

    public static void run(boolean b, int val) {
        Worker w;
        w = new SlowWorker();
        w = new Worker();
        WcetExample.loop(w, b, val);
    }

    public static int loop(Worker w, boolean b, int val) {
        int i;
        for (i=0; i<10; ++i) {
            val = w.run(b, val);
        }
        return val;
    }
}

class Worker {
    int run(boolean b, int val) {
        return val;
    }
}

class SlowWorker extends Worker {
    int run(boolean b, int val) {
        int j;
        if (b) {
            for (j=0; j<3; ++j) {
                val *= val;
            }
        } else {
            for (j=0; j<4; ++j) {
                val += val;
            }
        }
        return val;
    }
}
```

Listing 1: The example used for WCET analysis

```
for (i=0; i<10; ++i) { // @WCA loop=10
```

The annotation `//@WCA loop=`*N* tells the WCET analysis that the loop is executed *N* times. An alternative form is the annotation `//@WCA loop<=`*N*, which tells the WCET analysis that the loop is executed *up to N* times.

3.1.2 Virtual Method Dispatch

One feature of the data-flow analysis is the computation of receiver types, i.e., which method is actually invoked for virtual method calls. In Listing 1, a naive approach would have to assume that either `Worker.run()` or

```
27          | class SlowWorker extends Worker {
28          |         int run(boolean b, int val) {
29          |                 int j;
30  [5]     |                 if (b) {
31  [116]   |                         for (j=0; j<3; ++j) {
32          |                                 val *= val;
33          |                         }
34          |                 } else {
35          |                         for (j=0; j<4; ++j) {
36          |                                 val += val;
37          |                         }
38          |                 }
39  [24]    |                 return val;
40          |         }
41          | }
```

Figure 1: WCET path highlighting in the source code.

`SlowWorker.run()` is invoked. In this case, the computed WCET would be 2783 cycles. The data-flow analysis can determine that only `Worker.run()` is actually invoked, which leads to a WCET of 1573 cycles. However, the receiver types cannot always be determined precicely, which can be a source of pessimism for the computed WCET.

3.1.3 Results

After completing WCET analysis, WCA creates detailed reports to provide feedback to the user and annotate the source code as far as possible. All reports are formatted as HTML pages, and located under the directory `java/target/wcet`. Each individual method is listed with basic blocks and execution time of bytecodes, basic blocks, and cache miss times.

The overall WCET is also shown under the assumption that all invokes and returns hit the methods cache and that they are all caches misses. This provides feedback to whether a larger method cache could considerably enhance performance. In the case of our example, a minimum WCET of 1463 cycles is reported, i.e., no more than 110 cycles could be gained if the method cache is always hit.

Furthermore, the source is annotated with execution time and the WCET path is marked, as shown in Figure 1 for the class `SlowWorker` from Listing 1. This provides feedback that is easily understandable, and helps developers to spot opportunities to lower the WCET.

4 Chip-Multiprocessor

The following section gives a brief introduction how to use the CMP version of JOP, JopCMP. A more technical description of the background of the design is available in [8] and [7].

To build the CMP version, several synthesis projects are available. The project `cyccmp` allows up to 3 cores on the dspio board. The project `altde2cmp` targets the Altera DE2 board, which fits up to 8 cores. The number of cores is configured through setting `cpu_cnt` in the appropriate design file, e.g., `vhdl/top/jopmul.vhd` for the `cyccmp` project.

Figure 2 shows the principal architecture of JopCMP. A central arbiter regulates the synchronization on memory read and write operations. Keeping the access to the shared memory time-predictable is key to achieving a predictable overall system. The CMP also includes a synchronization unit, which acts as a central lock to implement mutual exclusion between cores. Furthermore, the CMP contains an I/O unit; this I/O unit is not necessarily tied to a single processor, as discussed in Section 4.1.1.

4.1 Memory Arbitration

Three different arbiters are available for the access policy to the main memory: priority based, fairness based, and a time division multiple access (TDMA) arbiter. The main memory is shared between all cores. The source files for the different arbiters are `vhdl/simpcon/sc_arbiter_fixedpr`, `vhdl/simpcon/sc_arbiter_fair`, and `vhdl/simpcon/sc_arbiter_tdma`, respectively. The arbiter can be selected by adding the appropriate file to the project file of the synthesis tool.

The TDMA based memory arbiter provides a static schedule for the memory access. Therefore, access time to the memory is independent of tasks running on other cores. The worst-case execution time (WCET) of a memory loads or stores can be calculated by considering the worst-case phasing of the memory access pattern relative to the TDMA schedule [6].

In the default configuration each processor cores has an equally sized slot for the memory access. The TDMA schedule can also be optimized for different utilizations of processing cores. The TDMA schedule can be optimized to distribute slack time of tasks to other tasks with a tighter deadline [16].

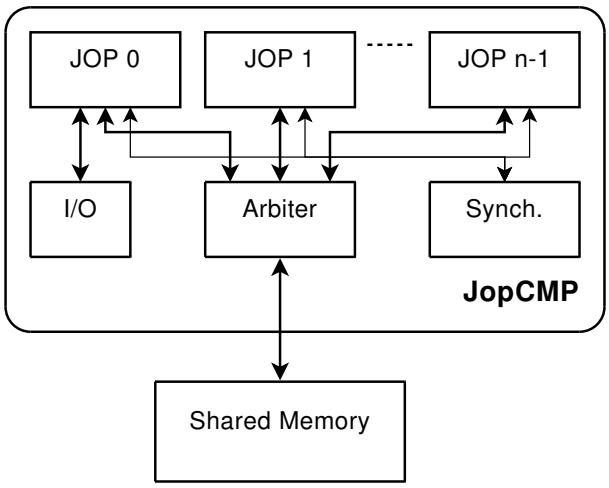

Figure 2: Block diagram of JopCMP

4.1.1 I/O Devices

Each core contains a set of local I/O devices, needed for the runtime system (e.g., timer interrupt, lock support). The serial interface for program download and a *stdio* device is connected to the first core.

For additional I/O devices two options exist: either they are connected to one core, or shared by all/some cores. The first option is useful when the bandwidth requirement of the I/O device is high. As I/O devices are memory mapped they can be connected to the main memory arbiter in the same way as the memory controller. In that case the I/O devices are shared between the cores and standard synchronization for the access is needed. For high bandwidth demands a dedicated arbiter for I/O devices or even for a single device can be used.

An interrupt line of an I/O device can be connected to a single core or to several cores. As interrupts can be individually disabled in software, a connection of all interrupt lines to all cores provides the most flexible solution.

4.2 Booting a CMP System

One interesting aspect of a CMP system is how the startup or boot-up is performed. On power-up, the FPGA starts the configuration state machine to read the FPGA configuration data either from a Flash memory or via a download cable from the PC during the development process. When the configuration has finished, an internal reset is generated. After this reset, microcode instructions are executed, starting from address 0. At this stage, we have not yet loaded any application program (Java bytecode). The first sequence in microcode performs this task. The Java application can be loaded from an external Flash memory, via a PC serial line, or an USB-port.

In the next step, a minimal stack frame is generated and the special method `Startup.boot()` is invoked, even though some parts of the JVM are not yet setup. From now on JOP runs in Java mode. The method `boot()` performs the following steps:

- Send a greeting message to *stdout*
- Detect the size of the main memory
- Initialize the data structures for the garbage collector
- Initialize `java.lang.System`
- Print out JOP's version number, detected clock speed, and memory size
- Invoke the static class initializers in a predefined order
- Invoke the `main` method of the application class

The boot-up process is the same for all processors until the generation of the internal reset and the execution of the first microcode instruction. From that point on, we have to take care that *only one* processor performs the initialization steps.

All processors in the CMP are functionally identical. Only one processor is designated to boot-up and initialize the whole system. Therefore, it is necessary to distinguish between the different CPUs. We assign a unique CPU identity number (CPU ID) to each processor. Only processor CPU0 is designated to do all the boot-up and initialization work. The other CPUs have to wait until CPU0 completes the boot-up and initialization sequence. At the beginning of the booting sequence, CPU0 loads the Java application. Meanwhile, all other processors are waiting for an *initialization finished* signal of CPU0. This busy wait is performed in microcode. When the other CPUs are enabled, they will run the same sequence as CPU0. Therefore, the initialization steps are guarded by a condition on the CPU ID.

4.3 CMP Scheduling

There are two possibilities to run multiple threads on the CMP system:

1. A single thread per processor
2. Several threads on each processor

For the configuration of one thread per processor the scheduler does not need to be started. Running several threads on each core is managed via the JOP real-time threads `RtThread`.

The scheduler on each core is a preemptive, priority based real-time scheduler. As each thread gets a unique priority, no FIFO queues within priorities are needed. The best analyzable real-time CMP scheduler does not allow threads to migrate between cores. Each thread is pinned to a single core at creation. Therefore, standard scheduling analysis can be performed on a per core base. Threads cannot migrate from one core to another one.

Similar to the uniprocessor version of JOP, the application is divided into an initialization phase and a mission phase. During the initialization phase, a predetermined core executes only one thread that has to create all data structures and the threads for the mission phase. During transition to the mission phase all created threads are started.

The uniprocessor real-time scheduler for JOP has been enhanced to facilitate the scheduling of threads in the CMP configuration. Each core executes its own instance of the scheduler. The scheduler is implemented as `Runnable`, which is registered as an interrupt handler for the core local timer interrupt. The scheduling is not tick-based. Instead, the timer interrupt is reprogrammed after each scheduling decision. During the mission start, the other cores and timer interrupts are enabled.

Another interesting option to use a CMP system is to execute exactly one thread per core. In this configuration scheduling overheads can be avoided and each core can reach an utilization of 100% without missing a deadline. To explore the CMP system without a scheduler, a mechanism is provided to register objects, which implement the `Runnable` interface, for each core. When the other cores are enabled, they execute the `run` method of the `Runnable` as their *main* method.

4.3.1 One Thread per Core

The first processor executes, as usual, `main()`. To execute code on the other cores a `Runnable` has to be registered for each core. After registering those Runnables the other cores need to be started. The code in Listing 2 shows an example that can be found in `test/cmp/HelloCMP.java`.

```
public class HelloCMP implements Runnable {

    int id;
    static Vector msg;

    public HelloCMP(int i) {
        id = i;
    }

    public static void main(String[] args) {

        msg = new Vector();
        System.out.println("Hello World from CPU 0");

        SysDevice sys = IOFactory.getFactory().getSysDevice();
        for (int i=0; i<sys.nrCpu-1; ++i) {
            Runnable r = new HelloCMP(i+1);
            Startup.setRunnable(r, i);
```

```
        }

        // start the other CPUs
        sys.signal = 1;
        // print their messages
        for (;;) {
            int size = msg.size();
            if (size!=0) {
                StringBuffer sb = (StringBuffer) msg.remove(0);
                System.out.println(sb);
            }
        }
    }

    public void run() {
        StringBuffer sb = new StringBuffer();
        sb.append("Hello World from CPU ");
        sb.append(id);
        msg.addElement(sb);
    }
}
```

Listing 2: A CMP version of Hello World

4.3.2 Scheduling on the CMP System

Running several threads on each core is possible with `RtThread` and setting the core for each thread with `RtThread.setProcessor(nr)`. The example in Listing 3 (`test/cmp/RtHelloCMP.java`) shows registering of 50 threads on all available cores. On `missionStart()` the threads are distributed to the cores, a scheduler for each core registered as timer interrupt handler, and the other cores started.

```
public class RtHelloCMP extends RtThread {

    public RtHelloCMP(int prio, int us) { super(prio, us); }

    int id;
    public static Vector msg;
    final static int NR_THREADS = 50;

    public static void main(String[] args) {
        msg = new Vector();
        System.out.println("Hello World from CPU 0");
        SysDevice sys = IOFactory.getFactory().getSysDevice();
        for (int i=0; i<NR_THREADS; ++i) {
            RtHelloCMP th = new RtHelloCMP(1, 1000*1000);
            th.id = i;
            th.setProcessor(i%sys.nrCpu);
        }
        RtThread.startMission();      // start mission and other CPUs
        for (;;) {                    // print their messages
            RtThread.sleepMs(5);
            int size = msg.size();
            if (size!=0) {
                StringBuffer sb = (StringBuffer) msg.remove(0);
```

```
                for (int i=0; i<sb.length(); ++i) {
                    System.out.print(sb.charAt(i));
    }}}}

    public void run() {
        StringBuffer sb = new StringBuffer();
        StringBuffer ping = new StringBuffer();
        sb.append("Thread "); sb.append((char) ('A'+id)); sb.append(" start on CPU ");
        sb.append(IOFactory.getFactory().getSysDevice().cpuId); sb.append("\r\n");
        msg.addElement(sb);
        waitForNextPeriod();
        for (;;) {
            ping.setLength(0);
            ping.append((char) ('A'+id));
            msg.addElement(ping);
            waitForNextPeriod();
    }}
}
```

Listing 3: A CMP version of Hello World with the scheduler

4.4 Parallel Executor

To simplify the parallelization of algorithms we have built a small *executor* framework. The client of the framework has to create an executor with the problem size n (independent units of work) and provide a class that implements an interface with a single method `execute(int nr)`. This method implements the unit of work and is automatically invoked by the framework n times. The framework automatically distributes the workload to all available processor cores. Therefore, several iterations of `execute(int nr)` execute in parallel. Any access to shared data needs to be properly synchronized.

Conceptually there is a single thread per core to execute the workload. The executor framework is more lightweight than starting a thread per unit of work. Our approach is similar, but simpler, than the *Executor* framework introduced in Java 1.5. The following listing shows the usage of the executor framework.

```
public static void main(String[] args) {
        Execute e = new Test();
        ParallelExecutor pe = new ParallelExecutor();
        pe.executeParallel(e, Test.N);
        Test.result(); }

private static class Test implements Execute {
        final static int N = 100;
        static int a[] = new int[N];

    // the work method for one iteration
        public void execute(int nr) {
                a[nr] = nr;
        }
        public static void result() {
                for (int i=0; i<N; ++i) {
                        System.out.println(a[i]);
                }
        } }
```

5 Caching

Caches are a mandatory feature of current processors to deliver instructions and data to a fast processor pipeline. However, standard cache organizations are designed to increase the average case performance. They are hard to model for worst-case execution time (WCET) analysis. Unknown abstract cache states during the analysis result in conservative WCET bounds. Therefore, we organized caches to simplify the analysis.

Different memory areas are cached in different caches:

- A stack cache
- An instruction cache for complete methods
- A cache for static data
- A small, fully associative buffer for heap access
- A cache for constants

The stack is the most heavily accesses data area in a JVM. Caching is therefore mandatory to achieve acceptable performance. Furthermore, the stack contains only thread-local data, i.e., no cache coherence mechanisms are required. Figure 3 shows a simplified block diagram of JOP, where cache memories are highlighted. As can be seen in this figure, the stack cache is an integral part of JOP's core pipeline.

The instruction cache holds complete methods and is also referred to as *method cache*. Cache misses can only occur upon `invoke` and `return` instructions. This reduces the points at which the WCET analysis has to compute the appropriate information and therefore reduces the complexity of the analysis. As depicted by Figure 3, the method cache is located in the memory controller unit, which allows the core pipeline to resume executing microcode instructions while the cache is loaded. Therefore, some of the latency for method cache loads can be hidden.

The other three caches are located in a separate data cache unit. They take into account the varying properties of JVM data memory areas w. r. t. caching. Memory accesses can be classified as follows:

- The address is *always* known statically. This is the case for static variables, which are resolved at link time, and for the constant pool, which only depends on the currently executed method.
- The address depends on the dynamic type of the operand, but not on its value. Therefore, the set of possible addresses is restricted by the receiver types determined for the call site. The class info table, the interface table and the method table are in this category.
- The address depends on the value of the reference. The exact address is unknown, as some value on the managed heap is accessed, but in addition to the symbolic address a relative offset is known. Instance fields and array fields, both showing some degree of spatial locality, belong to this category.

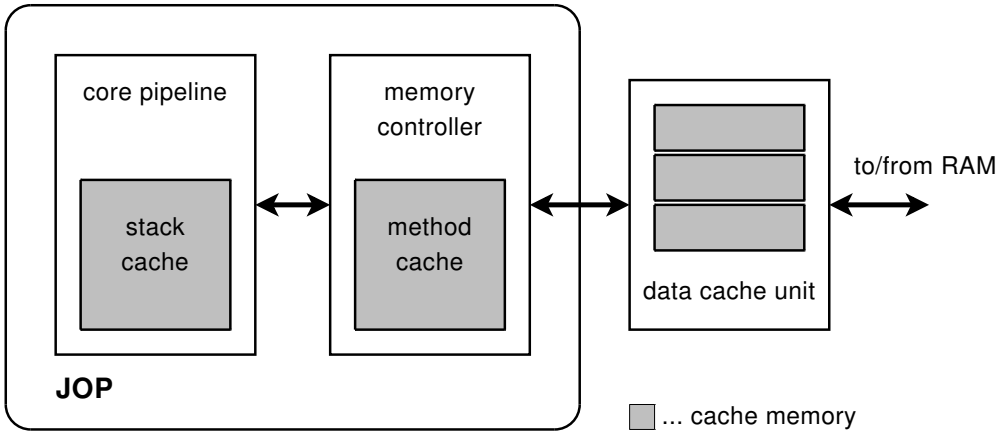

Figure 3: JOP Caches

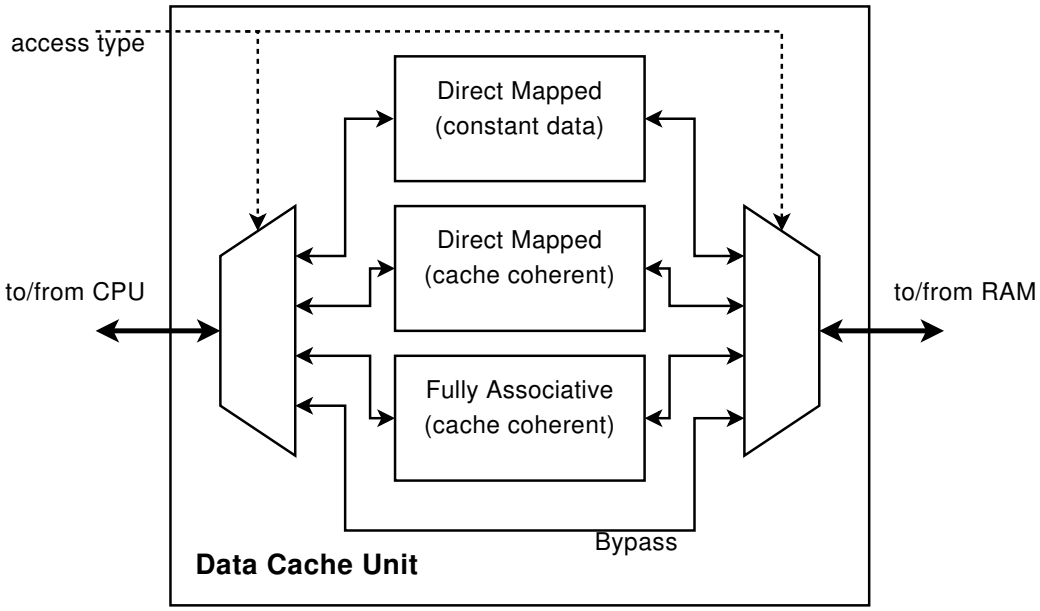

Figure 4: Data cache design

- The last category contains handles, reference to the method dispatch table, and array lengths. They reside on the heap as well, but we only know the symbolic address.

We use a direct-mapped cache for the first two classes. If addresses are statically known, WCET analysis can classify accesses for such a simple cache. If the address depends on the dynamic type, some pessimism is introduced, but receiver type analysis can considerably lower this pessimism.

The new two classes are handled by a small, fully associative cache. If an address is unknown, WCET analysis cannot predict which cache set is affected and therefore has to assume that all sets are affected. The cache is therefore effectively reduced to a single set, rendering parts of the cache useless. In reverse, a fully associative cache can be modeled properly by the WCET analysis. However, such a cache is expensive in terms of hardware resources and therefore has to be kept small.

A second criterion to classify memory accesses is whether the accessed data requires to be held coherent. Accesses to static variables and object fields must follow the Java memory model, which requires some coherence mechanism. Accesses to constant data such as the constant pool or the method table are implicitly cache coherent. Coherence mechanisms can therefore be avoided.

5.1 Data Cache Design

The structure of the resulting data cache design is shown in Figure 4. The memory unit outputs a signal for the access type, such that accesses can be directed to the appropriate cache. Accesses that should not be cached, e.g., loading bytecodes into the method cache, can bypass the data caches.

5.2 Cache Coherence

Cache coherence is achieved through a simple mechanism: the caches are invalidated upon `monitorenter` and reads from volatile variables. This scheme is compliant with the Java memory model, but provides only few guarantees for code that is not properly synchronized. Although the coherence implementation is probably

less efficient than other coherence protocols, cache invalidation is always a local action. The cache state does not depend on the behavior of other cores. WCET analysis can therefore take these operations into account, which helps to minimize the pessimism for WCET bounds.

5.3 Cache Configuration

Not all projects use the data cache – for a uniprocessor with fast memory, caching provides only minor performance benefits. The projects that do use the cache include "`cache`" in their names. The sizes of the individual data caches can be configured in the file `vhdl/cache/datacache.vhd`. The mapping between accesses and caches can be changed in two places: the file `vhdl/memory/mem_sc.vhd` implements the memory unit, where signals to select a cache are generated. If accesses are generated from microcode (as it is the case for invoke instructions), the appropriate file for changes is `asm/src/jvm.asm`.

6 Garbage Collection

Garbage Collection (GC) is an essential part of the Java runtime system. GC enables automatic dynamic memory management which is essential to build large applications. Automatic memory management frees the programmer from complex and error prone explicit memory management (`malloc` and `free`).

JOP's collector can operate in two modes: (1) as stop-the-world collector triggered on allocation when the heap is full, or (2) as concurrent real-time collector running in its own thread.

6.1 Algorithm

The garbage collector is a copying collector, which uses a snapshot-at-beginning write barrier to ensure consistency during concurrent operation. To simplify the relocation of objects, objects are accessed through a handle, which acts as a light-weight read barrier.

The real-time collector is scheduled periodically and within each period it performs the following steps:

Flip An atomic flip exchanges the roles of tospace and fromspace.

Mark roots All references static and local variables are pushed onto the mark stack.

Mark and copy An object is popped from the mark stack, all referenced objects, which are still white, are pushed on the mark stack, the object is copied to tospace and the handle pointer is updated.

Sweep handles All handles in the use list are checked if they still point into tospace (black objects) or can be added to the handle free list.

Clear fromspace At the end of the collector work the fromspace that contains only white objects is initialized with zero. Objects allocated in that space (after the next flip) are already initialized and allocation can be performed in constant time.

To reduce blocking time on a uniprocessor, a hardware unit performs copies of objects and arrays in an interruptible fashion, and records the copy position on an interrupt. On an object or array access the hardware knows whether the access should go to the already copied part in the tospace or in the not yet copied part in the fromspace. It has to be noted that splitting larger arrays into smaller chunks, as done in Metronome [1] and in the GC for the JamaicaVM [17], is a software option to reduce the blocking time.

The collector has two modes of operation: one for the initialization phase and one for the mission phase. At the initialization phase it operates in a stop-the-world fashion and gets invoked when a memory request cannot be satisfied. In this mode the collector scans the stack of the single thread conservatively. It has to be noted that each reference points into the handle area and not to an arbitrary position in the heap. This information is considered by the GC to distinguish pointers from primitives. Therefore the chance to keep an object artificially alive is low.

As part of the mission start one stop-the-world cycle is performed to clean up the heap from garbage generated at initialization. From that point on the GC runs in concurrent mode in its own thread and omits scanning of the thread stacks.

6.2 Concurrent Mode

To concurrently perform garbage collection, the garbage collector must be executed in a periodic thread. Listing 4 shows an example for such a thread. The period of the garbage collector in this listing is `GC_PERIOD`. The period must be carefully chosen to meet two requirement: the garbage collector must be able to meet its deadline, and the period must be short enough so the garbage collector can keep up with the application threads.

```
static class GCThread extends RtThread {

    public GCThread() {
        super(1, PERIOD_GC);
        GC.setConcurrent();
    }

    public void run() {
        for (;;) {
            GC.gc();
            waitForNextPeriod();
        }
    }
}
```

Listing 4: Concurrent GC thread

6.2.1 GC Period

To determine the maximum GC period, information about the allocation behavior of the application is required. We denote the maximum amount a thread allocates within its period T_i with a_i. The maximum amount of live memory is denoited by L_{max}. Furthermore, the overall heap size, H_{CC} must be known. The maximum GC period T_{GC} can then be computed as

$$T_{GC} \leq \frac{H_{CC} - 2L_{max} - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}}$$

Background on the GC period equation can be found in [11].

7 Low-level I/O Access

The following subsection describes the low-level mechanism for I/O access and interrupts on JOP. As JOP is a Java processor no native functions (usually written in C) are available to access I/O directly or use C written interrupt handler. We need access to those low-level functionality from Java for an embedded system. In the following a hardware abstraction layer (HAL) in Java is described, where I/O devices are mapped to Java objects and interrupt handlers can be implemented in Java as `Runnable`. This subsection is based on [14] and [4].

7.1 Hardware Objects

Hardware objects map object fields to device registers. Therefore, field access with bytecodes `putfield` and `getfield` accesses device registers. With a correct class that represents a device, access to it is safe – it is not possible to read or write to an arbitrary memory address. A memory area (e.g., a video frame buffer) represented by an array is protected by Java's array bounds check. Representing I/O devices as first class objects has following benefits:

Object-oriented An object representing a hardware device is the most natural integration into an OO language

Safe The safety of Java is not compromised. We can access only those device registers that are represented by the class definition

Efficient Device register access is performed by single bytecodes `getfield` and `putfield`. We avoid expensive native calls.

7.1.1 Definition

All hardware classes have to extend the abstract class `HardwareObject` (see Lising 5). This empty class serves as type marker. Implementations may use it to distinguish between plain objects and hardware objects for the field access. The package visible only constructor disallows creation of hardware objects by the application code that resides in a different package.

```
public abstract class HardwareObject {
    HardwareObject() {};
}
```

Listing 5: The marker class for hardware objects

Listing 6 shows a class representing a serial port with a `status` register and a `data` register. The status register contains flags for receive register full and transmit register empty; the data register is the receive and transmit buffer. Additionally, we define device specific constants (bit masks for the `status` register) in the class for the serial port. All fields represent device registers that can change due to activity of the hardware device. Therefore, they must be declared `volatile`.

```
public final class SerialPort extends HardwareObject {

    public static final int MASK_TDRE = 1;
    public static final int MASK_RDRF = 2;

    public volatile int status;
```

```
    public volatile int data;

    public void init (int baudRate) {...}
    public boolean rxFull()  {...}
    public boolean txEmpty() {...}
}
```

Listing 6: A serial port class with device methods

In this example we have included some convenience methods to access the hardware object. However, for a clear separation of concerns, the hardware object represents only the device state (the registers). We do not add instance fields to represent additional state, i.e., mixing device registers with heap elements. We cannot implement a complete device driver within a hardware object; instead a complete device driver owns a number of private hardware objects along with data structures for buffering, and it defines interrupt handlers and methods for accessing its state from application processes. For device specific operations, such as initialization of the device, methods in hardware objects are useful.

7.1.2 Access Control

Usually each device is represented by exactly one hardware object (see Section 7.1.5). However, there might be use cases where this restriction should be relaxed. Consider a device where some registers should be accessed by system code only and some other by application code. In JOP there is such a device: a system device that contains a 1 MHz counter, a corresponding timer interrupt, and a watchdog port. The timer interrupt is programmed relative to the counter and used by the real-time scheduler – a JVM internal service. However, access to the counter can be useful for the application code. Access to the watchdog register is required from the application level. The watchdog is used for a sign-of-life from the application. If not triggered every second the complete system is restarted. For this example it is useful to represent one hardware device by two *different* classes – one for system code and one for application code. We can protect system registers by private fields in the hardware object for the application. Listing 7 shows the two class definitions that represent the same hardware device for system and application code respectively. Note how we changed the access to the timer interrupt register to `private` for the application hardware object.

```
public final class SysCounter extends HardwareObject {

    public volatile int counter;
    public volatile int timer;
    public volatile int wd;
}

public final class AppCounter extends HardwareObject {

    public volatile int counter;
    private volatile int timer;
    public volatile int wd;
}
```

Listing 7: System and application classes with visibility protection for a single hardware device

Another option, shown in Listing 8, is to declare all fields private for the application object and use setter and getter methods. They add an abstraction on top of hardware objects and use the hardware object to implement their functionality. Thus we still do not need to invoke native functions.

```
public final class AppGetterSetter extends HardwareObject {

    private volatile int counter;
    private volatile int timer;
    private volatile int wd;

    public int getCounter() {
        return counter;
    }

    public void setWd(boolean val) {
        wd = val ? 1 : 0;
    }
}
```

Listing 8: System and application classes with setter and getter methods

7.1.3 Using Hardware Objects

Use of hardware objects is straightforward. After obtaining a reference to the object all that has to be done (or can be done) is to read from and write to the object fields. Listing 9 shows an example of client code. The example is a *Hello World* program using low-level access to a serial port via a hardware object.

```
import com.jopdesign.io.*;

public class Example {

    public static void main(String[] args) {

        BaseBoard fact = BaseBoard.getBaseFactory();
        SerialPort sp = fact.getSerialPort();

        String hello = "Hello World!";

        for (int i=0; i<hello.length(); ++i) {
            // busy wait on transmit buffer empty
            while ((sp.status & SerialPort.MASK_TDRE) == 0)
                ;
            // write a character
            sp.data = hello.charAt(i);
        }
    }
}
```

Listing 9: A 'Hello World' example with low-level device access via a hardware object

7.1.4 Hardware Arrays

For devices that use DMA (e.g., video frame buffer, disk, and network I/O buffers) we map that memory area to Java arrays. Arrays in Java provide access to raw memory in an elegant way: the access is simple and safe due to the array bounds checking done by the JVM. Hardware arrays can be used by the JVM to implement higher-level abstractions from the RTSJ such as `RawMemory` or scoped memory [18].

7.1.5 Board Configurations

An object that represents a device is a typical Singleton [2]. Only a single object should map to one instance of a device. Therefore, hardware objects cannot be instantiated by a simple `new`: (1) they have to be mapped by some JVM mechanism to the device registers and (2) each device instance is represented by a single object.

The board specific factory object is created at class initialization and is retrieved by a static method. Listing 10 shows an example of a base factory and a derived factory. Note how `getBaseFactory()` is used to get a single instance of the factory. We have applied the idea of a factory two times: the first factory generates an object that represents the board configuration. That object is itself a factory that generates the objects that interface to the hardware device.

```
public class BaseBoard {

    private final static int SERIAL_ADDRESS = ...;
    private SerialPort serial;
    BaseBoard() {
        serial = (SerialPort) jvmHWOCreate(SERIAL_ADDRESS);
    };
    static BaseBoard single = new BaseBoard();
    public static BaseBoard getBaseFactory() {
        return single;
    }
    public SerialPort getSerialPort() { return serial; }

    // here comes the JVM internal mechanism
    Object jvmHWOCreate(int address) {...}
}
```

Listing 10: A base class of a hardware object factory

The shown example base factory represents the minimum configuration with a single serial port for communication (mapped to `System.in` and `System.out`) represented by a `SerialPort`. The derived configuration `ExtendedBoard` (listing 11) contains an additional serial port for a GPS receiver and a parallel port for external control.

```
public class ExtendedBoard extends BaseBoard {

    private final static int GPS_ADDRESS = ...;
    private final static int PARALLEL_ADDRESS = ...;
    private SerialPort gps;
    private ParallelPort parallel;
    ExtendedBoard() {
        gps = (SerialPort) jvmHWOCreate(GPS_ADDRESS);
        parallel = (ParallelPort) jvmHWOCreate(PARALLEL_ADDRESS);
    };
    static ExtendedBoard single = new ExtendedBoard();
    public static ExtendedBoard getExtendedFactory() {
        return single;
    }
    public SerialPort getGpsPort() { return gps; }
    public ParallelPort getParallelPort() { return parallel; }
}
```

Listing 11: An extended class of a hardware object factory for a board variation

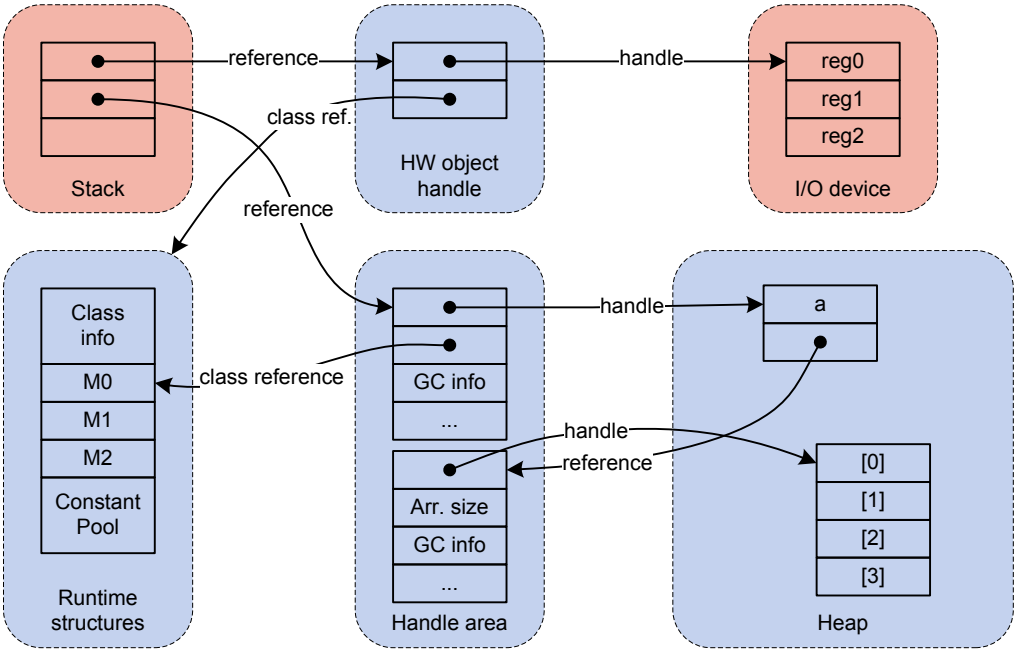

Figure 5: Memory layout of the JOP JVM

Furthermore, we show in those examples a different way to incorporate the JVM mechanism in the factory: we define well known constants (the memory addresses of the devices) in the factory and let the native function `jvmHWOCreate()` return the correct device type.

7.1.6 Implementation

In this subsubsection the internals of the hardware object creation are described. Just to use hardware objects this subsection can be skipped. To create new types of hardware objects and the companion factory this subsection contains the needed details.

In JOP, objects and arrays are referenced through an indirection called *handle*. This indirection is a lightweight read barrier for the compacting real-time GC (see Section 6). All handles for objects in the heap are located in a distinct memory region, the handle area. Besides the indirection to the *real* object the handle contains auxiliary data, such as a reference to the class information, the array length, and GC related data. Figure 5 shows an example with a small object that contains two fields and an integer array of length 4. The object and the array on the heap just contain the data and no additional hidden fields. This object layout greatly simplifies our object to device mapping. We just need a handle where the indirection points to the memory mapped device registers instead of into the heap. This configuration is shown in the upper part of Figure 5. Note that we do not need the GC information for the hardware object handles. The factory, which creates the hardware objects, implements this indirection.

As described in Section 7.1.5 we do not allow applications to create hardware objects; the constructor is private (or package visible).[10] Listing 12 shows part of the base hardware object factory that creates the hardware object `SerialPort` and `SysDevice`. Two static fields (`SP_PTR` and `SP_MTAB`) are used to store the handle to the serial port object. The first field is initialized with the base address of the I/O device; the second

[10]For creation of hardware objects with `new` we would need to change the implementation of bytecode `new` to distinguish between normal heap allocated objects and hardware objects. In the implementation on JOP the hardware object constructor is package visible to allow the factory to create a plain object of that type.

field contains a pointer to the class information.[11] The address of the static field `SP_PTR` is returned as the reference to the serial port object.

```
public class IOFactory {
    private SerialPort sp;
    private SysDevice sys;

    // Handles should be the first static fields !
    private static int SP_PTR;
    private static int SP_MTAB;
    private static int SYS_PTR;
    private static int SYS_MTAB;

    IOFactory() {
        sp = (SerialPort) makeHWObject(new SerialPort(), Const.IO_UART1_BASE, 0);
        sys = (SysDevice) makeHWObject(new SysDevice(), Const.IO_SYS_DEVICE, 1);
    };
    // that has to be overridden by each sub class to get the correct cp
    private static Object makeHWObject(Object o, int address, int idx) {
        int cp = Native.rdIntMem(Const.RAM_CP);
        return JVMHelp.makeHWObject(o, address, idx, cp);
    }
    private static IOFactory single = new IOFactory();
    public static IOFactory getFactory() {
        return single;
    }

    public SerialPort getSerialPort() { return sp; }
    public SysDevice getSysDevice() { return sys; }
}
```

Listing 12: Simplified version of the JOP base factory

The class reference for the hardware object is obtained by creating a *normal* instance of `SerialPort` with `new` on the heap and copying the pointer to the class information. To avoid using *native*[12] methods in the factory class we delegate JVM internal work to a helper class in the JVM system package. That helper method returns the address of the static field `SP_PTR` as reference to the hardware object.

7.1.7 Legacy Code

Before the implementation of hardware objects, the access to I/O devices was performed with memory read and write methods. Those methods are `Native.rdMem()` and `Native.wrMem()`. Due to historical reasons[13] the same Methods also exist as `Native.rd()` and `Native.wr()`.

Those native methods are mapped to a system bytecode and perform direct memory access – not a save abstraction at all. However, there exists still some Java code that uses those public visible methods. Those methods are deprecated and new device drivers shall use hardware objects.

[11] In JOP's JVM the class reference is a pointer to the method table to speed-up the invoke instruction. Therefore, the name is `XX_MTAB`.

[12] There are no *real* native functions in JOP – bytecode is the native instruction set. The very few native methods in class `Native` are replaced by special bytecodes during class linking.

[13] In an older version of JOP I/O and memory had different busses.

7.2 Interrupt Handlers

Interrupts are notifications from hardware components to the processor. As a response to the interrupt signal some method needs to be invoked and executed. To allow implementation of first-level interrupt handler (IH) in Java we map interrupt requests to invocations of the `run()` method of a `Runnable`. Listing 13 shows an example of such an interrupt handler and how it is registered.

```
public class InterruptHandler implements Runnable {

    public static void main(String[] args) {

        // get factory
        InterruptHandler ih = new InterruptHandler();
        fact.registerInterruptHandler(1, ih);

        // enable interrupt 1
        fact.enableInterrupt(1);

        // start normal work
    }

    public void run() {
        // do the first level interrupt handler work
    }

}
```

Listing 13: A simple first-level interrupt handler in Java

7.2.1 Synchronization

When an interrupt handler is invoked, it starts with global interrupt disabled. The global interrupt mask is enabled again when the interrupt handler returns. On the uniprocessor version of JOP the monitor is also implemented by simply masking all interrupts. Therefore, those critical subsections cannot be interrupted and synchronization between the interrupt handler and a normal thread is fulfilled. For a CMP version of JOP synchronization is performed via a global hardware lock. As the CMP system offers true concurrency, the IH has to protect the access to shared data when the handler and the thread are located on different cores.

A better approach for data sharing between a first-level IH and a device driver task (or second-level IH) is the usage of non-blocking queues. An object oriented version of a non-blocking single reader/writer queue is available in class `SRSWQueue` in package `rtlib`. The redesigned TCP/IP stack `ejip` uses such non-blocking queues for communication of network packets between different protocol layers.

7.2.2 Interrupt Handler Registration

Interrupt handlers can be registered for a interrupt number n. On the CMP system the interrupt is registered on the core where the method is invoked. Following methods are available in the factory class for interrupt handler registration/deregistration and enable/disable of a specific interrupt.

```
public void registerInterruptHandler(int nr, Runnable logic) { }
public void deregisterInterruptHandler(int nr) { }
public void enableInterrupt(int nr) { }
public void disableInterrupt(int nr) { }
```

Interrupt 0 has a special meaning as it is the reprogrammable timer interrupt for the scheduler. On the transition to the mission phase (`startMission()`) a scheduler, which is a simple `Runable`, is registered for each core for the timer interrupt.

7.2.3 Implementation

The implemented interrupt controller (IC) is priority based. The number of interrupt sources can be configured. Each interrupt can be triggered in software by a IC register write as well. There is one global interrupt enable and each interrupt line can be enabled or disabled locally. The interrupt is forwarded to the bytecode/microcode translation stage with the interrupt number. When accepted by this stage, the interrupt is acknowledged and the global enable flag cleared. This feature avoids immediate handling of an arriving higher priority interrupt during the first part of the handler. The interrupts have to be enabled again by the handler at a *convenient* time. All interrupts are mapped to the same special bytecode. Therefore, we perform the dispatch of the correct handler in Java. On an interrupt the static method `interrupt()` from a system internal class gets invoked. The method reads the interrupt number and performs the dispatch to the registered `Runnable` as illustrated below. Note, how a hardware object of type `SysDevice` is used to read the interrupt number.

```
static Runnable ih[] = new Runnable[Const.NUM_INTERRUPTS];
static SysDevice sys = IOFactory.getFactory().getSysDevice();

static void interrupt() {
    ih[sys.intNr].run();
}
```

The timer interrupt, used for the real-time scheduler, is located at index 0. The scheduler is just a plain interrupt handler that gets registered at mission start at index 0. At system startup, the table of Runnables is initialized with dummy handlers. The application code provides the handler via a class that implements `Runnable` and registers that class for an interrupt number.

For interrupts that should be handled by an event handler under the control of the scheduler, the following steps need to be performed on JOP:

1. Create a `SwEvent` with the correct priority that performs the second level interrupt handler work
2. Create a short first level interrupt handler as `Runnable` that invokes `fire()` of the corresponding software event handler
3. Register the first level interrupt handler and start the real-time scheduler

7.2.4 An Example

The system device (`sc_sys.vhd`) contains input lines for external interrupts (in port `io_int`). The number of interrupt lines is configurable with `num_io_int`. Each hardware interrupt can also be triggered by a write into the system device. Listing 14 shows registering and using a first level interrupt handler. The interrupt is triggered in software by the main thread.

```
public class InterruptHandler implements Runnable {

    public static void main(String[] args) {

        IOFactory fact = IOFactory.getFactory();
        SysDevice sys = fact.getSysDevice();
```

```
        InterruptHandler ih = new InterruptHandler();
        fact.registerInterruptHandler(1, ih);

        // enable software interrupt 1
        fact.enableInterrupt(1);

        for (int i=0; i<20; ++i) {
            Timer.wd();
            int t = Timer.getTimeoutMs(200);
            while (!Timer.timeout(t)) ;
            // trigger a SW interrupt via the system HW object
            System.out.println("Trigger");
            sys.intNr = 1;
            if (i==10) {
                fact.disableInterrupt(1);
            }
        }
    }

    public void run() {
        System.out.println("Interrupt fired!");
    }
}
```

Listing 14: Interrupt register/deregister methods in the factory class

7.3 Standard Devices

A minimum version of JOP consists of two standard devices: the system device and a UART device for program download and as a representation of `System.out`.

7.3.1 The System Device

The system device contains all the logic for interrupts, CMP interaction, timers, and the watchdog control. The registers definition is shown in Table 4.

7.3.2 The UART

The UART contains a contral/status register and a data read/write register is shown in Table 5.

Address	Read	Write
0	Clock counter	Interrupt enable
1	Counter in μs	Timer interrupt in μs
2	Interrupt number	SW interrupt
3	—	Watchdog
4	Exception reason	Generate exception
5	Lock request status	Lock request
6	CPU ID	—
7	Polled in JVM startup	Start CMP
8	—	Interrupt mask
9	—	Clear pending interrupts
11	Nr. CPUs	—

Table 4: Registers in the system device.

Address	Read	Write
0	status	control
1	receive data	transmit buffer

Table 5: Registers in the UART device.

A Source Access

The design files for this deliverable are open-source under the GNU General Public License, version 3. The sources can be downloaded with *git* anonymously as follows:

```
git clone git://www.soc.tuwien.ac.at/jop.git
```

For partners with write access to the git repository following command applies:

```
git clone ssh://user@www.soc.tuwien.ac.at/home/git/jop.git
```

Build instructions for JOP can be found in Chapter 2 of the *JOP Reference Handbook* [13], available from: `http://www.jopdesign.com/doc/handbook.pdf`.

References

[1] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994.

[3] Benedikt Huber and Martin Schoeberl. Comparison of implicit path enumeration and model checking based WCET analysis. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dublin, Ireland, July 2009. OCG.

[4] Stephan Korsholm, Martin Schoeberl, and Anders P. Ravn. Interrupt handlers in Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[5] Wade D. Peterson. WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores, revision: B.3. Available at http://www.opencores.org, September 2002.

[6] Christof Pitter. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, 2008.

[7] Christof Pitter. *Time-Predictable Java Chip-Multiprocessor*. PhD thesis, Vienna University of Technology, Austria, 2009.

[8] Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *Trans. on Embedded Computing Sys. accepted for publication.*, 2009.

[9] Wolfgang Puffitsch. Supporting WCET analysis with data-flow analysis of Java bytecode. Research Report 16/2009, Institute of Computer Engineering, Vienna University of Technology, Austria, February 2009.

[10] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[11] Martin Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006. IEEE.

[12] Martin Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*, Graz, Austria, October 2007.

[13] Martin Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. Number ISBN 978-1438239699. CreateSpace, August 2009. Available at `http://www.jopdesign.com/doc/handbook.pdf`.

[14] Martin Schoeberl, Stephan Korsholm, Christian Thalinger, and Anders P. Ravn. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[15] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[16] Martin Schoeberl and Peter Puschner. Is chip-multiprocessing the end of real-time scheduling? In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dublin, Ireland, July 2009. OCG.

[17] Fridtjof Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. Number ISBN: 3-8311-3893-1. aicas Books, 2002.

[18] Andy Wellings and Martin Schoeberl. Thread-local scope caching for real-time Java. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.