LEON 2 SystemC model on ReSP

José Fernández Alcón

April 11th, 2008

Fernández Alcón, José *jose.fernandez.alcon@esa.int*

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER

ONE

Leon 2

1.1 Introduction

The LEON 2 is 32 bits RISC open-source CPU very popular in space. The actual trends indicate that it will be used in many future European Space Agency missions. The *Reflective Simulation platform* (ReSP) is an open-source virtual platform developed by the European Space Agency and Politecnico di Milano. Having a SystemC-TLM model of the LEON 2 processor will enable ESA to perform design space exploration for LEON 2 based boards.

1.2 Architecture Description

The LEON 2 processor is a fault tolerant version of the SPARC v8, developed by the European Space Agency. Fault-Tolerance is a requirement for space systems so that the processor's state will not be affected by the radiation. Fault-tolerance is completely transparent for the user and so, its behavior will be the same as the one from a standard SPARC v8.

1.2.1 SPARC v8 Registers

Processor State Register

LEON/SPARC Processor State Register (PSR) is a 32 bit register were the status of the processor is stored:

Figure 1.1: LEON/SPARC Processor State Register

icc: Condition Code Flags
EF: Enabled Floating Point Unit (FPU). Set to 1 when there is a FPU available.
PIL: Processor Interrupt Level. Indicates the interrupt level that the processor will be accepting.
S: Supervisor mode flag. Set to 1 while on supervisor mode and to 0 when in user mode.
PS: Previous Supervisor mode flag. used to save the execution mode before the execution of a trap.
ET: Enabled Traps. The processor will accept traps while this flag is high.
CWP: Current Window Pointer. Indicates the window that is being used.

Trap Base Register

The Trap Base Register (TBR) holds the address of the instruction that will be executed after a trap. When a trap is thrown the processor writes in the TBR the address of the corresponding trap handler vector.

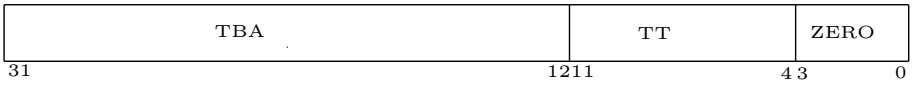

Figure 1.2: LEON/SPARC Trap Base Register

TBA: *Trap Base Address.* The upper part of the address of the trap table. This changes, for instance, when switching execution from ROM (0x000000) to RAM (0x40000000).
TT: *Trap Type.* Indicates which of the 256 possible traps has been thrown.

Window Invalid Mask

It is a 32 bits register which indicates which one of registers windows is invalid. An invalid window means that such a window has old data. At any given time only one bit is set to high. The position of the bit indicates the number of the window that is invalid (WIM=0x8 indicates that the 3rd window is invalid). If

after a window switch the invalid window is reached a trap overflow or underflow will be thrown.

1.3 LEON 2 SystemC model

As mentioned in Section 1.2, the a SystemC model of the LEON 2 and of the SPARC v8 would be identical. There was already available a simple SystemC model of the SPARC v8 from the ArchC group. ArchC's model only supports a basic instruction set and it lacks the important features like:

1. It does not support interruptions. The model can not interact with external hardware as a timer or an UART.
2. Traps are not supported.
3. User modes have not been implemented.
4. It does not handle the Processor State Register as a whole, but as a set of separate variables. This will become an issue if the software tries to read or write the PSR.
5. Several Instructions are missing.
6. Some instructions are wrong or incomplete.
7. Register Windows overflow and underflow traps are hard-coded in the model and so, an operating system will not be able of handling them itself.

Summarizing: ArchC's model supports only a basic instruction set and lack the meachnisms to support a full operating system.

1.3.1 Trap Handler

A trap handler method was written to implement the trap and interrupt handling mechanism of the LEON 2 processor.

Function `trapHandler(int, int)` models the LEON hardware that takes care of the traps and interrupts. Whenever there is a trap (synchronous or asynchronous) the trap handler is called, the handling function address is computed and the processor jumps to that address.

The first parameter (`trapID`) indicates the type of interrupt that has been thrown. The second one (`sw_trap_number`) is only used to indicate which kind of software trap is it. If the trap thrown is not of software type, `sw_trap_number` is ignored. Figure 1.3 shows the flow diagram of the `trapHandler` function.

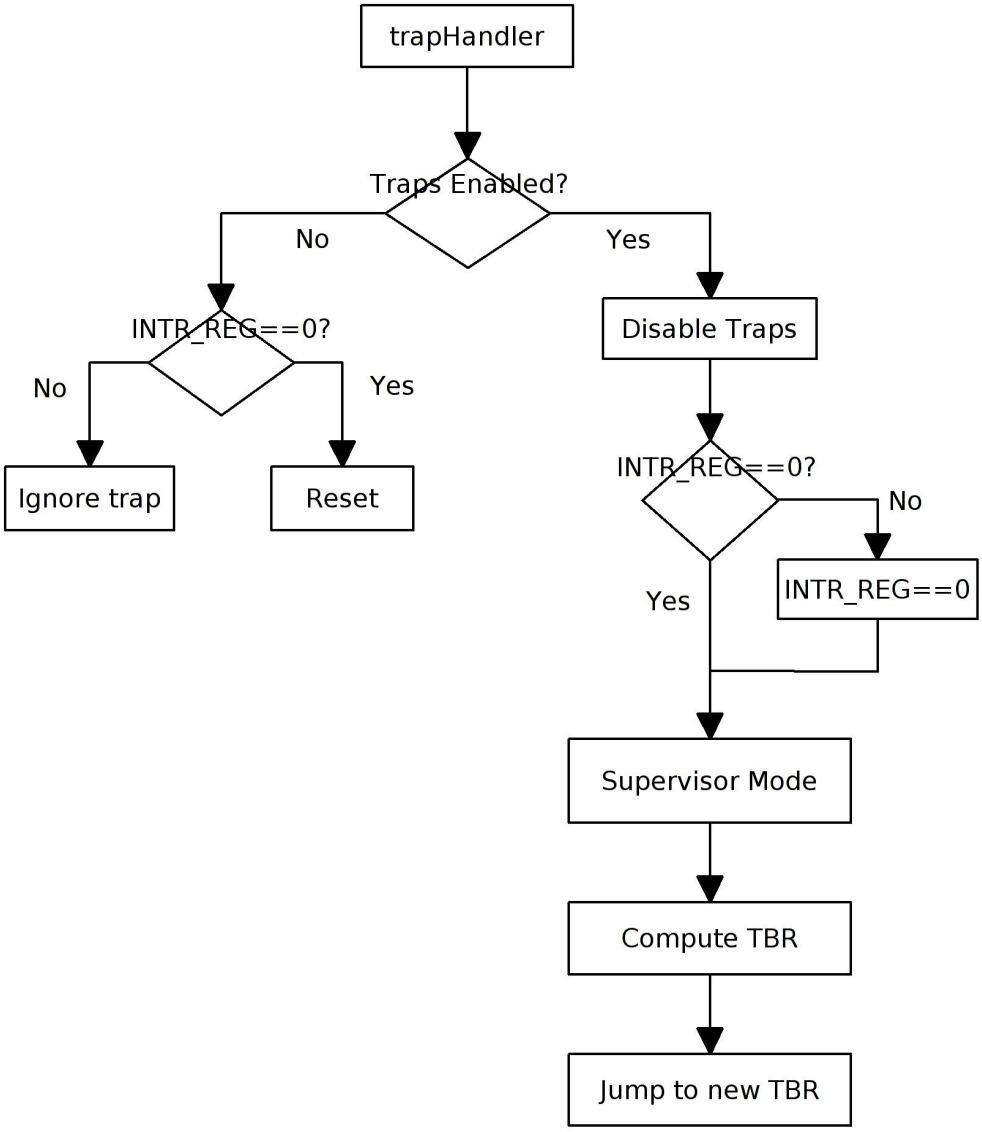

Figure 1.3: Trap Handling Function Flow Diagram

Trap Type	*TT* **value**
Interrupt line 1	0x11
Interrupt line 2	0x12
Interrupt line 3	0x13
Interrupt line 4	0x14
Interrupt line 5	0x15
Interrupt line 6	0x16
Interrupt line 7	0x17
Interrupt line 8	0x18
Interrupt line 9	0x19
Interrupt line 10	0x1A
Interrupt line 11	0x1B
Interrupt line 12	0x1C
Interrupt line 13	0x1D
Interrupt line 14	0x1E
Interrupt line 15	0x1F
Software Trap	0x80 + SW trap number
Division by zero	0x2A
Tag overflow	0x0A
Data Access Exception	0x09
Data Access MMU miss	0x2C
Data Access Error	0x29
Co-processor Exception	0x28
FPU Exception	0x08
Mem. addr. not aligned	0x07
Window Underflow	0x06
Window Overflow	0x05
Watchpoint detected	0x0B
Unimplemented Flush	0x25
Co-processor disabled	0x24
FPU disabled	0x04
Illegal Instruction	0x02
Privileged Instruction	0x03
Instr. access exception	0x01
R register access error	0x20
Instr. access error	0x21
Instr. access MMU miss	0x3C
data store error	0x2B
Reset	

Table 1.1: LEON 2 Trap Table

1.3.2 Interrupt Support

Interrupts are asynchronous events and so, can happen at any moment. At the beginning of every instruction the method `ac_behavior{instruction}` of `leon2_isa.cpp` which is the generic instruction method is called.

We have created a fake register (`INTR_REG`) that can have values from 0 to 15. 0 if no interrupt has been received, and 1 to 15 depending on the interruption line that went high.

In `ac_behavior{ instruction }` we check if that register is active. If so, we jump to corresponding trap handler. If not, we continue with the normal execution.

The basic model provided by the ArchC team lacks interrupts lines. Then, it is necessary to add the TLM ports that will model the interrupt lines. This is done modifying the file `leon2.ac` by adding:

```
ac_tlm_intr_port INTERRUPT_PORT_1
...
ac_tlm_intr_port INTERRUPT_PORT_15
```

Executing `acsim` will create the TLM ports for the 15 interrupts lines in a template file `leon2_intr_handlers.cpp.tmpl`. As that file has been automatically created, there is no behavior defined for the interrupts ports. In order to implement the behavior we will copy `leon2_intr_handlers.cpp.tmpl` to `leon2_intr_handlers.cpp`[1] and then modify the method of each interrupt to write on `INTR_REG` the number of the interrupt line that went active.

```
// Interrupt handler behavior for interrupt port INTERRUPT_PORT_4.
void ac_behavior(INTERRUPT_PORT_4, value) {
INTR_REG.write(4);
}
```

Figure 1.4 shows the flow diagram of the interrupt handling mechanism developed for the Leon2 Model.

1.3.3 Re-modeling the Processor Status Register

In Section 1.2.1 SPARC's PSR was shown. In the original model the PSR was modeled as a set of independent variables (CWP, ,PSR_icc_n, PSR_icc_z, PSR_icc_v, PSR_icc_c). The rest of the fields were missing as the original model did not need them because of the lack of support for other processor features (like traps, user modes or interrupts). The LEON 2 model has to be the most accurate possible so, it was necessary to model it as a single variable of 32 bits with all its fields just as it is in the real hardware. In order to do so, we eliminated the above mentioned variables from `leon2.ac` and substitute them for a single called PSR.

A set of auxiliary functions was created to read or write each of the fields of the PSR independently, in order to increase the readability of the code and the ease of programming. This functions are described on Table 1.2

[1] The model uses `leon2_intr_handlers.cpp` the `.tmpl` file is only an automatically created template to ease the development. Every time `acsim` is executed `leon2_intr_handlers.cpp.tmpl` is overwritten and all the previous changes are lost.

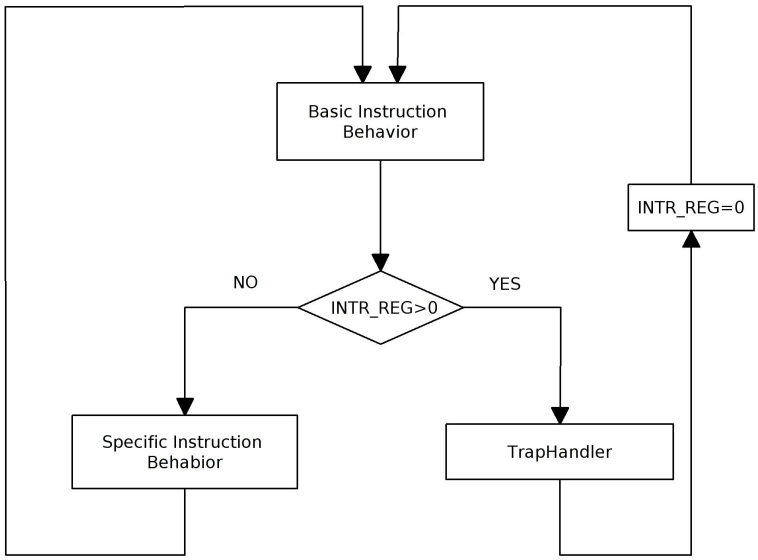

Figure 1.4: Flow Diagram of the Interrupt Handling mechanism

1.3.4 Enabling the user modes

The LEON2 processor has to user modes: *user* and *supervisor*. While the processor is in *supervisor* mode it can execute all the instructions. In *user* mode only the unpriviledged instructions are allowed to be executed. If the processor attempts to execute a priviledged instruction while in *user* mode, a trap (privileged_instruction, $tt = 0x03$) will be thrown. The set of privileged instructions for LEON 2 are shown on Table 1.3.

Bit 7 of the PSR is supervisor flag. If one the processor is in *supervisor* mode, and if zero in *user* mode.

To implement the *supervisor* and *user* modes we added a conditional statement at the beginning of every privileged instruction. If the supervisor flag was equal to one, then execute instruction if not, throw a trap:

```
void ac_behavior( instruction )
{
  if(read_s()){
    execute_instruction_code;
  else{
    trapHandler(PRIVILEGED_INSTRUCTION, 0)
};
```

Where `read_s()` `write_s()` are two functions added to read and write the supervisor mode bit. `PRIVILEGED_INSTRUCTION` is a macro to tell the trap handler that the trap was caused by the execution of a privileged instruction.

Function	Description
write_icc_n(bool)	Writes the N bit of PSR
write_icc_z(bool)	Writes the Z bit of PSR
write_icc_v(bool)	Writes the V bit of PSR
write_icc_c(bool)	Writes the C bit of PSR
read_icc_n()	Reads the N bit of PSR
read_icc_z()	Reads the Z bit of PSR
read_icc_v()	Reads the V bit of PSR
read_icc_c()	Reads the C bit of PSR
write_cwp(int)	Writes the Current Window Pointer
read_cwp()	Reads the Current Window Pointer
write_pil(int)	Writes the Processor Interrupt Level
read_pil()	Reads the Processor Interrupt Level
write_et(bool)	Writes *Enabled Traps* flag of the PSR
read_et()	Reads the *Enabled Traps* flag of the PSR
write_ef(bool)	Writes *Enabled Floating Point Unit* flag of the PSR
read_ef()	Reads the *Enabled Floating Point Unit* flag of the PSR
write_s(bool)	Writes *Supervisor mode* flag of the PSR
read_s()	Reads the *Supervisor mode* flag of the PSR
write_ps(bool)	Writes *Previous Supervisor mode* flag of the PSR
read_ps()	Reads the *Previous Supervisor mode* flag of the PSR

Table 1.2: Set of Processor State Register auxiliary functions

Assembly	Name
RETT	Retrun from Trap
RDASR	Read Ancillary State Register
RDPSR	Read Processor State Register
RDWIM	Read Window Invalid Mask Register
RDTBR	Read Trap Base Register
WRASR	Write Ancillary Register
WRPSR	Write Processor State Register
WRWIM	Write Window Invalid Mask Register
WRTBR	Write Trap Base Register

Table 1.3: LEON 2 Privileged Instructions

1.3.5 Register Windows

Register window switching is a special method of the LEON and SPARC architectures to save the context when there is a system call. At any given time the processor has access to 32 general purpose registers:

- 8 32bit *Global* registers.
- 8 32bit *In* registers.
- 8 32bit *Local* registers.
- 8 32bit *Out* registers.

The 8 *Global* registers are always accessible by the processor. The other 24 are what is called a register window. While the SPARC v8 can have between 2 and 32 windows[2] the LEON 2 has only 8.

[2]Depending on the implementation.

The *Local* registers can only be accessed while on their specific window. The *In* registers of one window are the *Out* registers of the following one.

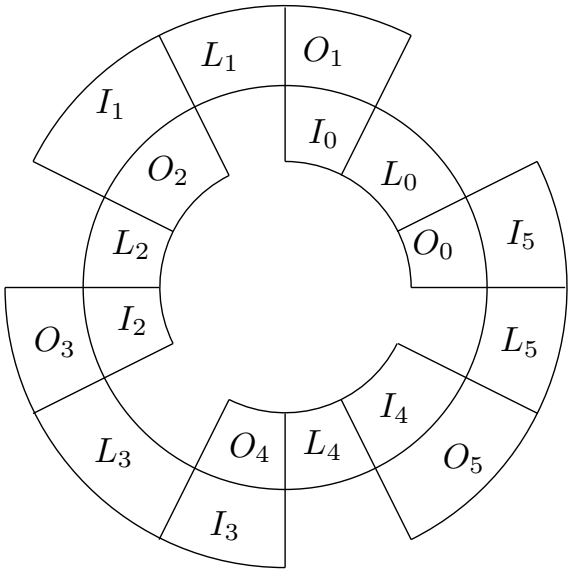

Figure 1.5: LEON SPARC register windows.

To implement the register window behavior we created a buffer of 128 registers[3]

When we want to switch window, we save the contents of that window into the buffer. Figures 1.6 and 1.7

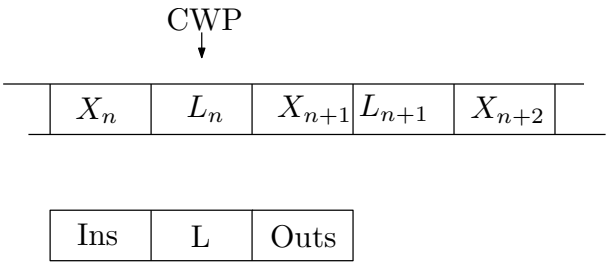

Figure 1.6: Registers Buffer and register window

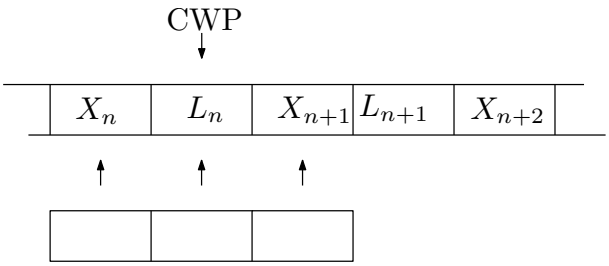

Figure 1.7: Copying the registers to the buffer

Then the current window pointer is changed to point to the new window. Figure 1.8

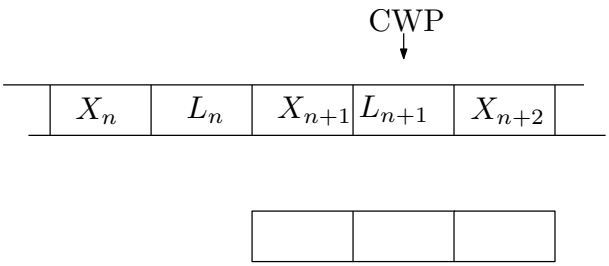

Figure 1.8: Moving register window

And finally the data is copied from the buffer onto the window. Figure 1.9. Notice that the *Out* registers of the n window where on X_{n+1}, and for the next window $(n+1)$ that that was placed at the *In* registers of the $n+1$ window.

[3] 8 windows × 8 *Local* registers per window = 64. *Ins* and *Outs* are shared by two consecutive windows, so we need another 64 registers. In total 128 registers.

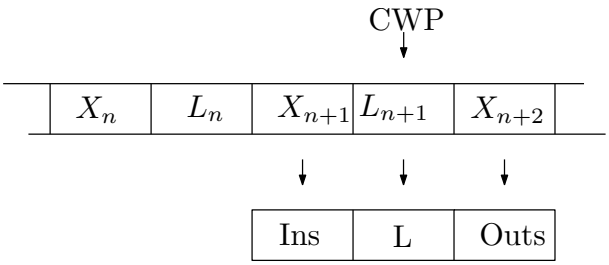

Figure 1.9: Updating the registers window

Below is shown the function that performs the window switching. `RB` is the buffer where the register windows are stored and retrieved.

```
void leon2_isa::change_window(int old_cwp, int new_cwp){
    //Saving current window to buffer RB
    for (int i=0; i<24; i++) {
      RB.write(((unsigned int)(old_cwp*16 + i -8)) % RB.size() , REGS[i+8]);
    }
    //Retieving window from buffer RB
    for (int i=0; i<24; i++) {
      REGS[i+8] = RB.read(((unsigned int)(new_cwp*16 -8 + i))  % RB.size());
    }
}
```

CHAPTER

TWO

RTEMS on the LEON 2 model

2.1 Real-Time Executive for Multiprocessor Systems

RTEMS stands for Real-Time Executive for Multiprocessor Systems. RTEMS is a very popular Real-Time Operating System (RTOS) for aerospace application mainly due to is maturity and stability[1]. Another important characteristic is support for wide variety of platform. There is already a Board Support Package (BSP) for the LEON2 platform developed by Gaisler Research.

[1] It was originaly developed as a military application. RTEMS used to stand for Real-Time Executive for Missile Systems

2.2 Installation

All the necessary tools to install and compile RTEMS are avialble on the web. RTEMS source code and sample applications can be downloaded from the RTEMS website (www.rtems.org). The tools to build RTEMS are available at the GNU foundation website (www.gnu.org). At the time of writing of this document the most recent stable version was 4.8.

2.2.1 RTEMS Cross Compilation System from Gaisler Research

This is the recommended installation method. Gaisler Research has released a Cross Compilation System for both the LEON2 and the old ERC32. The Cross Compilation System is only available for RTEMS-4.6.5 and RTEMS-4.7.99.2. Using this tool-chain with any other version of RTEMS will translate into compilation errors. In order to use a version of RTEMS different than 4.6.5 ot 4.7.99.2 it will be necessary to build your own tools.

```
wget ftp://gaisler.com/gaisler.com/rcc/bin/linux/sparc-rtems-4.6.5-gcc-3.2.3-1.0.19-linux.tar.bz2
```

or

```
wget ftp://gaisler.com/gaisler.com/rcc/bin/linux/sparc-rtems-4.7.99.2-gcc-4.2.1-1.0.4-linux.tar.bz2
```

2.2.2 Installing the Pre-built tools from the RTEMS website

If you have a Red Hat or Debian based linux, there are a set of pre-built utils at the RTEMS website that can be installed using their respective package managers.

2.2.3 Building your own tool-chain

To build your own tool-chain autoconf 2.61 and automake 1.10 must be used.
First, it is necessary to get the source code for the build utils.

```
mkdir rtems-utils-sources
cd rtems-utils-sources

wget ftp://ftp.rtems.com/pub/rtems/SOURCES/binutils-2.18.tar.bz2
wget ftp://ftp.rtems.com/pub/rtems/SOURCES/gcc-4.2.1.tar.bz2
wget ftp://ftp.rtems.com/pub/rtems/SOURCES/newlib-1.15.0.tar.gz
```

Then we get the patches for binutils and newlib.

```
wget ftp://ftp.rtems.com/pub/rtems/SOURCES/binutils-2.18-rtems4.8-20071104.diff
wget ftp://ftp.rtems.com/pub/rtems/SOURCES/newlib-1.15.0-rtems4.8-20071221.diff
```

Unarchive the sources:

```
tar xjvf binutils-2.18.tar.bz2
tar xjvf gcc-4.2.1.tar.bz2
tar xzvf newlib-1.15.0.tar.gz
```

Apply the patches

```
cd binutils-2.18
cat ../$BINUTILS_PATCH | patch -p1
cd ..
cd newlib-1.15.0
cat ../$NEWLIB_PATCH | patch -p1
cd ..
```

```
mkdir b-binutils
cd b-binutils
../binutils-2.18/configure --target=sparc-rtems --prefix=/opt/rtems-4.8.0
make -j4 all info install
cd ..
```

```
export PATH=/opt/rtems-4.8.0/bin:${PATH}

cd $GCC/
ln -s ../newlib-1.15.0/newlib .
cd ..
mkdir b-gcc
cd b-gcc
../$GCC/configure --target=sparc-rtems --with-gnu-as --with-gnu-ld --with-newlib \
     --verbose --enable-threads --enable-languages="c,c++" --prefix=/opt/rtems-4.8.0
make -j4 all info install
cd ..
```

2.2.4 Creating an RTEMS image

RTEMS must have already been compiled before trying to embed an application.

Lets suppose that RTEMS has already been built on `/tools/c-leon2`
Then we need to add the following variable to the environment[2].

```
export RTEMS_MAKEFILE_PATH=/tools/c-leon2/sparc-rtems/c/leon2/make/
```

add the location of the `sparc-rtems-*` executables in case they haven't been added to the path yet:

```
export PATH=$PATH:/opt/rtems-4.8/bin
```

In order to make you application you need a Makefile that should look like this:

```
EXEC=name_of_the_output_binary
PGM=${ARCH}/$(EXEC)

# optional managers required
MANAGERS=all

# C source names
CSRCS = name_of_source_file
COBJS_ = $(CSRCS:.c=.o)
COBJS = $(COBJS_:%=${ARCH}/%)

# C++ source names
CXXSRCS =
CXXOBJS_ = $(CXXSRCS:.cc=.o)
CXXOBJS = $(CXXOBJS_:%=${ARCH}/%)

# AS source names
ASSRCS =
ASOBJS_ = $(ASSRCS:.s=.o)
ASOBJS = $(ASOBJS_:%=${ARCH}/%)

# Libraries
LIBS = -lrtemsall -lc

include $(RTEMS_MAKEFILE_PATH)/Makefile.inc

include $(RTEMS_CUSTOM)
include $(PROJECT_ROOT)/make/leaf.cfg

OBJS= $(COBJS) $(CXXOBJS) $(ASOBJS)

all:    ${ARCH} $(PGM)

$(PGM): $(OBJS)
$(make-exe)
```

[2]Or manually to the Makefile

Component	Init Addr.	End addr.
ROM	0x00000000	0x3fffffff
RAM	0x40000000	0x7fffffff
Timers 7 0x80000040	0x8000006f	
UART 1	0x80000070	0x8000007C
UART 2	0x80000080	0x8000008C
Parallel Port	0x800000A0	0x800000AC

Table 2.1: Components Address Mapping

After executing make in the directory where the source code of the application is located a binary file with the name given before will be created. That file can be found under the `o-optimize` directory.

2.3 Running RTEMS executables on ReSP

In order to avoid having a different Board Support Package (BSP) the SPARC port[3] of RTEMS generates binaries that do not perform any board initialization, and that are linked to the beginning of RAM.

A normal simulator will detect that the start address of the binary is not 0x0, and will perform the board initialization itself. As ReSP is not a simulator for an specific board but a virtual platform, it behaves in exactly the same way the real hardware would do.

For some specific boards there exist an application called *mkprom*, which is used to create a self extracting prom image. That image will initialize all the boards registers. It will decompress and load the application into the ram and will start it.

It is necessary to download the tool-chain from Gaisler, because it is the only one that has `sparc-rtems-mkprom` which is necessary for the to boot the Leon 2 and 3 boards. There is also another mkprom application for the ERC32 that can be found at

`http://hacks-galore.org/aleix/mkprom-erc32/`

To create the boot image we only need to call the `mkprom` parsing the RTEMS application as a parameter.

```
/opt/rtems-4.8/bin/sparc-rtems-mkprom -v application.exe
```

After that, a ROM executable file (`prom.out`) will be created.

2.3.1 ReSP LEON Board

Figure 2.1 shows the minimum set of components necessary to run RTEMS.

2.4 Things that can go wrong

C compiler can not create executables.

The configure script test a simple program to check that the tool-chain is properly built (confdefs.h). If this error is shown the most likely your tool-chain is faulty.

[3] As SPARC and LEON uses the same instruction set, we are using the SPARC port of the RTEMS buildutils to create the binaries.

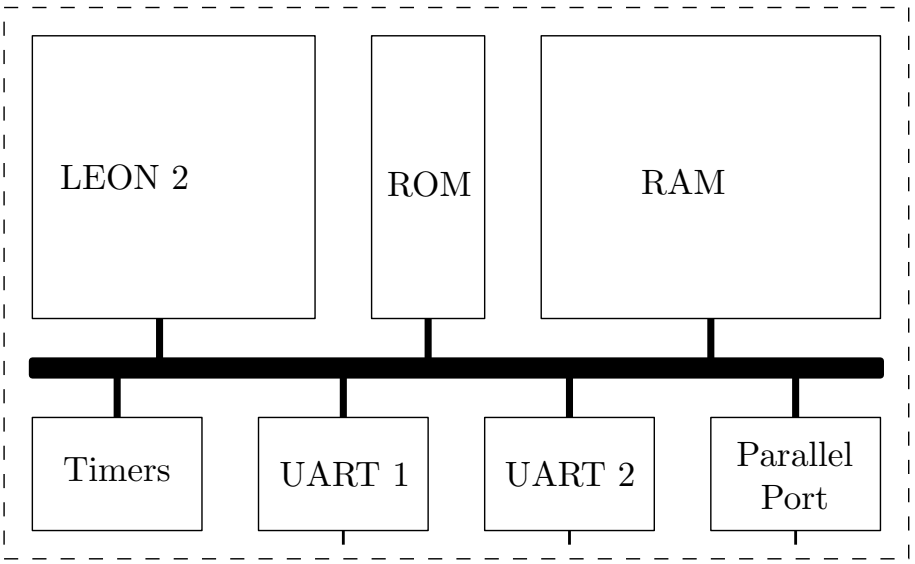

Figure 2.1: Board used for RTEMS testing

2.5 Modification of RTEMS to enable interrupts trap handler

In order to enable the console driver so that RTEMS will handle the interrupts produced by the UARTs, it is necessary to add the following flag: `CONSOLE_USE_INTERRUPTS=1` to the makefile when building an RTEMS executable.

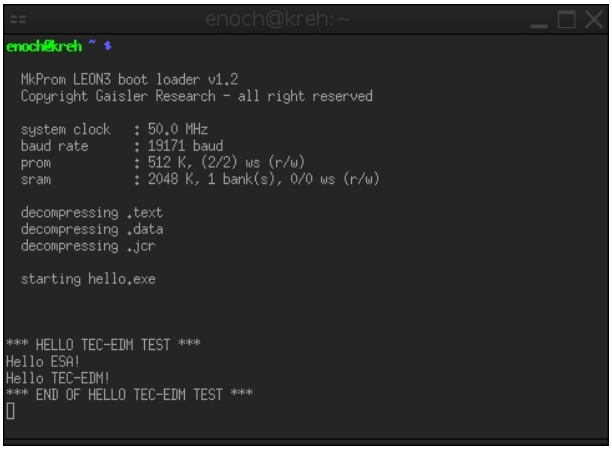

Figure 2.2: Screen-shot of a console connected to UART of a LEON 2 showing a Boot-prom output and the execution of a Hello World! RTEMS application.