Appendix A

Digital Appendix Documentation

This appendix refers to the software in the companion data archive bundled with this thesis. If this copy of the thesis does not include this archive, it can be downloaded from the following location:

`http://www.jwhitham.org.uk/thesis/`

The layout of this appendix is as follows:

- Section A.1 is an overview of the programs in the archive.
- Additional software required (or recommended) for use with the archive is listed in section A.2.
- General installation advice is given in section A.3.
- Sections A.4 to A.12 describe the programs in the archive: see section A.1 for an overview.
- The third party software included in the distribution is discussed in section A.13.

A.1 Overview

The software archive is a collection of research programs implementing the experiments in chapters 5 to 8. There is no integrated development environment (IDE). Instead, the programs of the Appendix are organised into separate projects, as illustrated in Figure A.1. The projects are:

- `/mcgrep1`: experiments from chapter 5 including the MCGREP-1 CPU generator. These programs are described in sections A.4 to A.6.
- `/testcases`: source code for the MCGREP-2 experiments in section 6.4. The programs are described in section A.7. Benchmark code is written in C, with a test infrastructure written in Python.
- `/mcgrep2-src`: the MCGREP-2 CPU generator, simulator, and tools for compiling and debugging programs, as described in chapter 6. These programs are described in sections A.8 to A.10, and are written in Python. C and VHDL templates are used for code generation.
- `/tracegen`: the trace generator, which is written in C. The trace generator reads trace information from machine code and generates a suitable microprogram. It is specialised at compile time for one MCGREP-2 CPU via the microprogramming API (section 6.2.4). The

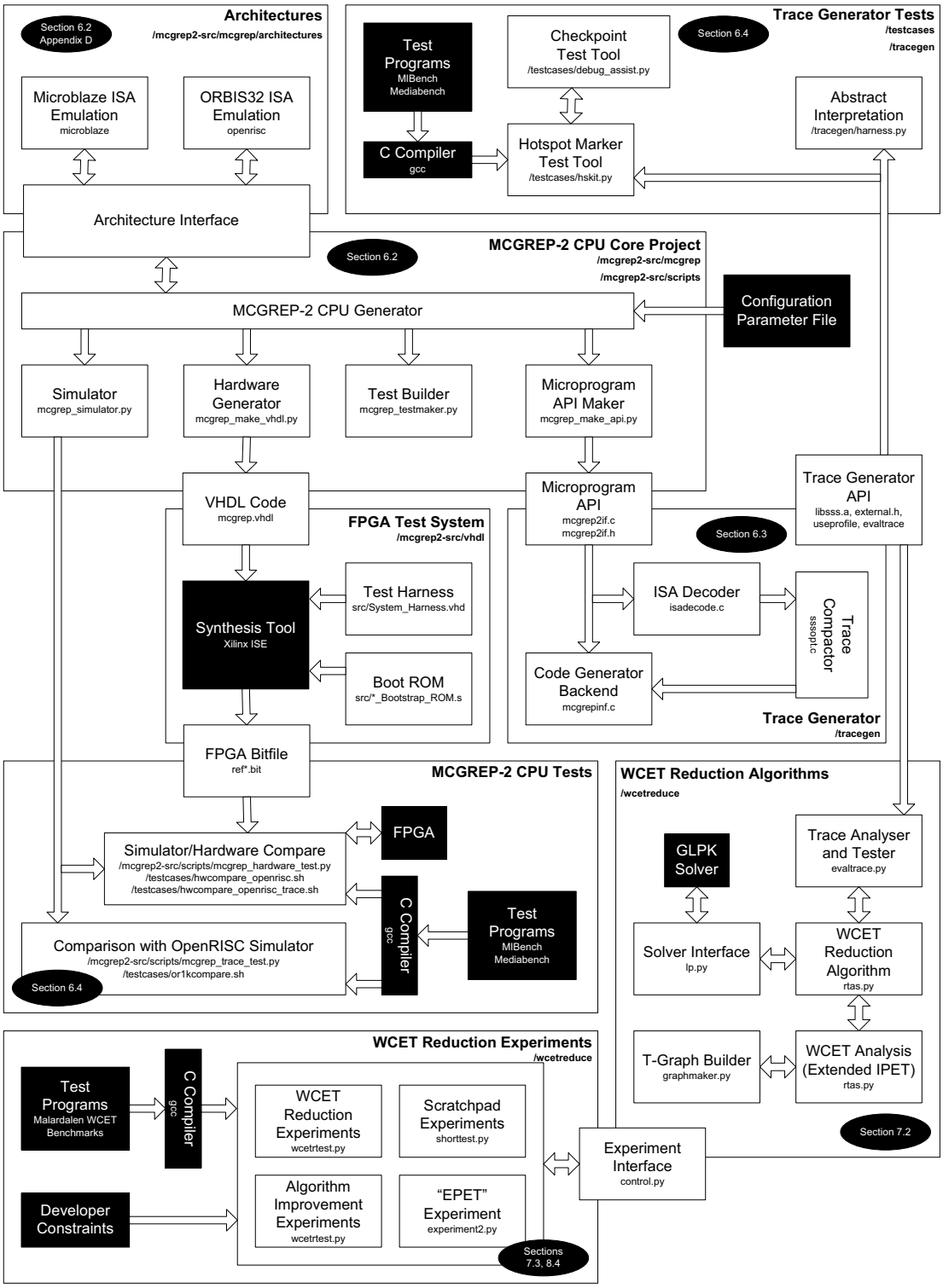

Figure A.1: A map of the software described by chapters 6 through 8, showing file and **path** names within the archive.

`useprofile` and `evaltrace` tools are used by the experiments described in section 6.3.1 and 7.2.7 respectively.

- `/wcetreduce`: experiment source code for chapters 7 and 8.

Your programs can use MCGREP-2's microprogramming features by linking against the `tracegen` library `libsss.a` built for the MCGREP-2 architecture configuration you wish to use. Sample usages of the API for direct microprogramming from C can be found in the `/tracegen/mcgrepinf.c` file. You can also rerun the tests used to check MCGREP-2 as described in section A.12. Some of these tests require an FPGA: information about the required arrangement can be found in section A.11. Please note that the tests are not all "task safe": some tests may fail if run in parallel with each other on the same filesystem, and FPGA access is assumed to be exclusive.

A.2 Required/recommended Software Environment

The archive was prepared on an x86-compatible computer running the Debian Etch Linux distribution. It is also known to be compatible with x86 computers running Slackware Linux version 12.0.0.

In principle, the archive software should be usable on any computer but in practice, some components need a Linux-like environment. Support for x86 Linux binaries is required to support the `gcc` cross compiler for ORBIS32, although this software can be rebuilt using the bundled source code (section A.13). A number of additional programs are required for most operations. These should be provided by the operating system:

- A working `gcc` installation for the host computer (i.e. a version of `gcc` that generates native binaries for the host).

  Recommended versions: `gcc` versions 3.4.6, 4.1.2 and 4.1.3 have been tested.

- A working environment for building C programs (i.e. all header files installed).

  Recommended versions: `libc6` versions 2.3.6 and 2.6.1 have been tested.

- Python, with support for building C extensions.

  Recommended versions: Python 2.4.4 or 2.5.1.

This command could be used on Debian Linux to install the required components:

```
apt-get install python bzip2 python-dev libc6-dev gcc
```

The following additional programs are recommended, because they are required by some operations:

- Xilinx Integrated Software Environment (ISE) for Linux: this is needed if you wish to build any of the hardware designs. This is non-free software, so it is not distributed with Linux.

  Recommended versions: 6.2i is required for building some designs for Spartan-2E FPGAs. For other devices (Virtex-2, Virtex-4, Spartan-3), 8.1i is recommended. Subsequent versions are untested.

- ASL assembler - this is needed if you wish to rebuild any of the T80 control programs (e.g. the program shown in Figure 5.16). This software is not currently distributed with Debian Linux.

  Recommended versions: 1.42 Beta.

- uDrawGraph - this is needed for viewing `.udg` graph files which are produced for debugging and visualisation purposes by some tools. uDrawGraph produces graphs like Figure 5.9. This is non-free software, so it is not distributed with Linux. It is only used for viewing results.

  Recommended versions: 3.1.1.

- pygame - this Python module is needed to use the microprogramming GUI for MCGREP-1 (Figure 5.12). It is not needed for MCGREP-2. This software is available in Debian Linux: the package name is `python-pygame`.

  Recommended versions: 1.7.1release.

- matplotlib - this Python module generates charts such as Figure 6.19. It is only used for viewing results. This software is available in Debian Linux: the package name is `python--matplotlib`.

  Recommended versions: 0.90.1.

- Ghostscript - this is used to convert charts into PNG format for inclusion in the test report (section A.12). This software is available in Debian Linux: the package name is `gs-gpl`.

  Recommended versions: 8.56.

- ImageMagick - this set of programs is useful for image manipulation. It can be used by the test software (section A.12) to generate visual image comparisons. This software is available in Debian Linux: the package name is `imagemagick`.

  Recommended versions: 6.2.4.5.

If your computer is not able to execute x86 Linux programs directly, the easiest way to make use of the archive is likely to be through a virtual machine, PC emulator, or PC simulator. Such a program can be used to simulate the architectural environment of a 2008-era PC, allowing you to install an appropriate x86 version of Linux, perhaps from a CDROM `.iso` image file. All of the required programs are part of major Linux distributions such as Debian Etch and Slackware 12, which are available as free software. You will then be able to work with the archive programs in an environment that closely replicates the environment used to write them. If the software proves useful, it should be possible to recompile `gcc` and other programs for more modern hardware as the source code is included (section A.13).

A.3 Installing the Archive Software

The archive should be extracted to a directory with at least 1Gb of free space. The software is not intended to be shared between multiple users, and is not installed using the `root` system administrator account. Your home directory is a good place for it. Use the `tar` command to extract the archive:

`tar xvjf jack-whitham-thesis-sw-dist-2008-`*xx-yy*`.tar.bz2`

Next, `cd` to the newly created directory, and run the `install.sh` program. *Do not run this program as* `root`. The software does not install any programs outside of the directory created by `tar` and therefore does not require `root` privileges. (However, the software does create temporary files in `/tmp`.) The installation begins by displaying an information message. Press Enter, and the process will continue. Installation builds various programs and tests the environment on your computer, checking your version of Python and your C compiler. The correct completion message is as follows:

```
Install process complete.
Be sure to source "setup.sh" before you try to run any of the programs.
(See the manual for your shell to learn how to source a script.)
In the Bash shell, you can type the following to source setup.sh:
    . setup.sh
```

Installation will fail if one of the components listed in section A.2 is missing, or if the version you are using is different in some way to the version that was tested. After a successful installation:

1. Source the `setup.sh` file to load the correct environment for the software. In this context, "source" means that the commands in the file should be executed as if they were directly typed into the shell. This is not the same as running the script, because that will execute the commands within a child process. Files can be sourced using the `.` or `source` command in Bash.

   The script changes the `PATH` variable, creates a new `MCGREP_PATH` variable, and runs a Python program to initialise the `/tmp` directory. `setup.sh`'s changes are not persistent: you must repeat this step every time you log in.

2. *Optional:* If you have the Xilinx ISE tools and you wish to build FPGA hardware designs, you should edit the `xilinx-ise-8.1.sh` script in the `xilinx-ise` subdirectory. This script, which contains an example, should load the Xilinx settings script for the version of Xilinx ISE you are using. If you want to build MCGREP-1 designs for the Spartan-2E FPGA, you must also edit the `xilinx-ise-6.2.sh` script. Version 6.2 of ISE appears to be required to build some Spartan-2E designs.

3. *Optional:* If you have an FPGA and wish to test hardware designs using the MCGREP-2 software, you should also edit the `download-bit.sh` script in the `xilinx-ise`. This script is executed with the absolute path to an FPGA bit file by tools such as `mcgrep_hardware_test.py`. It should program an FPGA with this bit file. The tools also expect the `serial-port` file to contain the Unix device name of a serial port that can be used to communicate with the FPGA, e.g. `/dev/ttyS0`. This may also need to be changed. See section A.11 for information about the required FPGA connections.

Once all files are installed, refer to sections A.4 through A.13 for information about the programs, libraries and hardware designs included within the appendix.

A.4 Building the MCGREP-1 Test Cases

To build the MCGREP-1 test cases, move to the test case directory (shown above). The `build` program offers the following options:

Post-installation Configuration Files

▷ `/xilinx-ise`

Xilinx ISE/FPGA configuration directory.

MCGREP-1 Test Cases

▷ `/mcgrep1/testcases`

Test case directory.

▷ `/mcgrep1/testcases/build`

Test case build program.

▷ `/mcgrep1/testcases/bin`

Test case output directory for binaries.

- `./build or`

  Entering this command will build all the test cases for the MCGREP-1 platform. It automatically includes microcode, and patches the program binaries. The resulting programs are ready to run inside the MCGREP-1 simulator or on the hardware. These programs can be used to obtain the MCGREP-1 performance figures (section 5.3.6).

- `./build ror`

  This command builds all the test cases for OpenRISC or MCGREP-1. Microcode is not included, and the programs only make use of operations supported by both OpenRISC and MCGREP-1. These programs can be used to obtain the OpenRISC performance figures (section 5.3.6).

- `./build aror`

  This command builds the interference experiment (section 5.3.5). The binary to be used is:

  ```
  /mcgrep1/testcases/bin/aes.bin
  ```

  This should be executed on both the MCGREP-1 hardware and the OpenRISC CPU to obtain a full set of results.

Please note that it is not easy to extend the set of test cases because custom microprograms must be generated for each one using a manual process (Figure 5.12). To use the microprogramming GUI, you should use the `make_ucode.py` script in a test case subdirectory, but bear in mind that the inconvenience of this process was one of the main motivators for the automatic microprogram generator in MCGREP-2 (chapter 6).

MCGREP-1 Hardware

▷ `/mcgrep1/hw`
Bitfile output directory.
▷ `/mcgrep1/hw/build`
Hardware build program.
▷ `/mcgrep1/hw/debug-monitor/mc_spartan2e.vhd`
MCGREP-1 VHDL file.
▷ `/mcgrep1/hw/debug-monitor/mc_virtex2.vhd`
MCGREP-1 VHDL file.

MCGREP-1 Simulator

▷ `/mcgrep1/testcases/*/run_orig.py`
Simulator program.
▷ `/mcgrep1/testcases/*/run_accel.py`
Simulator program.

A.5 Using the MCGREP-1 Hardware Generator

To build the MCGREP-1 hardware, run the `build` program listed above. The program requires a working installation of Xilinx ISE. It builds four bitfiles:

1. `mcgrep-eth-burched.bit` - MCGREP-1 plus test harness for "BurchEd B5" Spartan-2E board.
2. `mcgrep-eth-virtex.bit` - MCGREP-1 plus test harness for "Amadeus" Virtex-2 board.
3. `openrisc-burched.bit` - OpenRISC OR1200 plus test harness for "BurchEd B5" Spartan-2E board.
4. `openrisc-virtex.bit` - OpenRISC OR1200 plus test harness for "Amadeus" Virtex-2 board.

Once the process has completed, you can find the MCGREP-1 VHDL source code in the locations shown above. This only interacts with external components via the test harness.

The bit files can be downloaded to an appropriate FPGA board for testing. However, the test harness assumes that the external interface will be compatible with the York RTS Group Virtual Lab (section A.11). If this interface is not available, you will need to change the top level VHDL files to implement your own external interface.

A.6 Using the MCGREP-1 Simulator

To use the MCGREP-1 simulator to run a test case, you should use either of the two simulator programs listed above, which can be found in the subdirectory of each test case. `run_orig.py` runs

MCGREP-2 Test Cases

▷ `/testcases`

Test case directory.

▷ `/testcases/run_through.sh`

Builds test cases with and without custom RFU configurations. Uses checkpoints to compare the execution of each type of build.

▷ `/testcases/or1kcompare.sh`

Compares MCGREP-2 execution against the OpenRISC simulator.

MCGREP-2 Hardware Generator

▷ `/mcgrep2-src/scripts/mcgrep_make_vhdl.py`

Standalone VHDL generator.

▷ `/mcgrep2-src/vhdl`

Test system build directory.

▷ `/mcgrep2-src/vhdl/build_ref_hw.sh`

Builds the Avnet/Memec MM1 test system used for evaluation in chapter 6.

a test case in ORBIS32 mode only, without using any custom microprograms, while `run_accel.py` uses custom microprograms.

You will find that the MCGREP-2 simulator (sections 6.2.5 and A.9) is much faster than the MCGREP-1 simulator. However, the MCGREP-2 simulator only supports the OpenRISC ORBIS32 and Microblaze ISAs and the MCGREP-2 microprogramming interface. It cannot be used to run general MCGREP-1 programs because the microcode is not compatible.

A.7 Using the MCGREP-2 Test Cases

The MCGREP-2 test cases are built as part of an integrated experiment environment. They can also be built using the `mcgrep_testmaker.py` program: the `hskit.py` program includes examples of the usage of this program. The experiments are designed to execute unattended and produce results that are formatted into the tables and charts found in chapter 6.

A.8 Using the MCGREP-2 Hardware Generator

The MCGREP-2 hardware generator can be used in two ways:

- As part of the hardware building system for a supported FPGA, which generates the MCGREP-2 VHDL and then synthesises it. Currently, the supported FPGA for MCGREP-2 is the Spartan-3 `xc3s400-ft256-4` FPGA on the Avnet/Memec MM1 "mini module" FPGA board [22] (Figure A.2). The `build_ref_hw` program builds a bit file for this FPGA us-

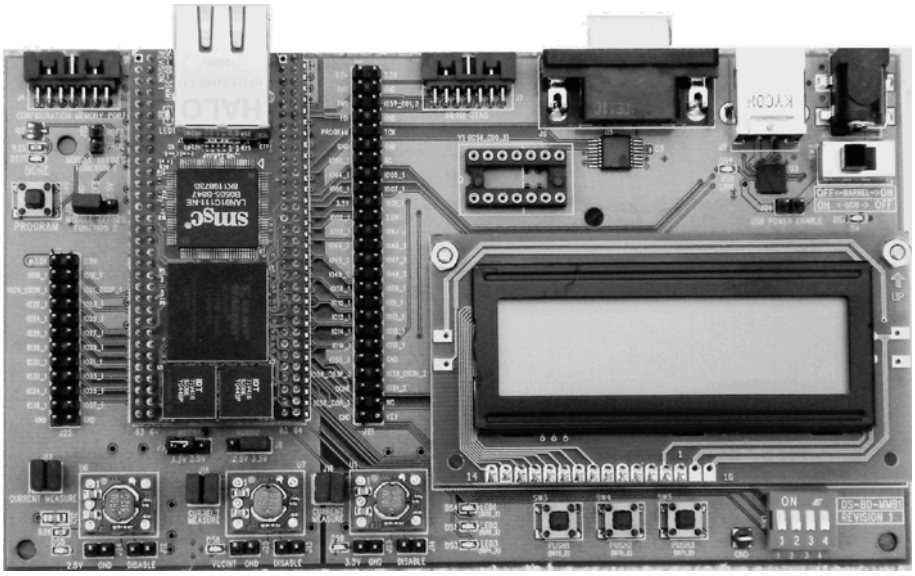

Figure A.2: Photograph of the MM1 "mini module" FPGA prototyping board, from Avnet/Memec documentation [22]. The mini module is plugged into the left-hand side of a development board that provides a serial port and JTAG interface in addition to a variety of other components (unused in this application) such as the display on the right-hand side.

ing Xilinx ISE. The bit file includes the test harness described in section 6.2.2.6. The `build_ref_core_hw` program builds the same system minus the test harness: in this configuration, the system is just an MCGREP-2 CPU plus memory and serial port drivers. Bit files are placed in `/mcgrep2-hw` along with temporary files produced by Xilinx ISE.

- As a standalone program. The `mcgrep_make_vhdl.py` program generates the self-contained VHDL source of an MCGREP-2 CPU. The program expects to be supplied with the name of a configuration file which specifies the parameters of the CPU to be generated. Sample configuration files can be found in `/mcgrep2-hw`: each has the extension `.cfg`. Some of the parameters that are supported are listed in Table A.1. Typically, this program is executed as follows:

```
mcgrep_make_vhdl.py -r mcgrep2-hw -n ref.cfg output.vhdl
```

This creates the file `output.vhdl` after reading the configuration file `ref.cfg` from the directory `mcgrep2-hw`.

The default bus used by MCGREP-2 CPUs is Wishbone [190]. However, the generator can also produce a version of the CPU with an On-chip Peripheral Bus (OPB) connector. If this is selected, then the output of the generator is a component for the Xilinx Embedded Development Kit (EDK) with an OPB interface. This component includes several files and is placed in a new subdirectory for inclusion in EDK, where it can act as a drop-in replacement for Microblaze if suitably configured (section D).

Parameter Name	Supported Values	Effect
num_units	Integer ≥ 1	Sets the total $l + m + n$ for the CPU (Figure 4.5).
hw_full_debug_chain	Boolean	Synthesise a long debugging chain that includes all CPU components rather than a subset.
memory_latency	≥ 1	Expect the specified memory latency. (Note: does *not* affect the latency of RAM accesses in the simulator - a command-line parameter is used for that purpose.)
multiply_on_any_unit	Boolean	Add a multiplier to every functional unit rather than just the first.
target	”spartan2e”, ”spartan3”	Selects the block RAM driver to be used. The ”spartan3” selection is also suitable for Virtex-II and subsequent devices.
arch_name	”openrisc”, ”microblaze”	Selects the ISA of the CPU.

Table A.1: Configuration parameters for the MCGREP-2 hardware generator, tester, and simulator.

MCGREP-2 Simulator

▷ /mcgrep2-src/scripts/mcgrep_simulator.py

Standalone simulator.

▷ /mcgrep2-src/scripts/mcgrep_mcuc_test.py

Debugging aid for microprogramming problems.

▷ /mcgrep2-src/scripts/mcgrep_hardware_test.py

Compares hardware and simulator execution using debugging mechanism.

A.9 Using the MCGREP-2 Simulator

The MCGREP-2 simulator is widely used by test cases and experiments. For example, it is used to check the correct operation of every test case in `/testcases`, and it is also used to carry out the experiments in chapters 6 to 8. Many programs extend the simulator with hooks (section 6.2.5), so the simulator is executed via Python. However, the simulator can also be executed manually by running `mcgrep_simulator.py`.

The simulator program uses a configuration file like the one accepted by the hardware generator. (The same file can be used by both.) The supported parameters are listed in Table A.1. Microprograms are not portable between different configurations of MCGREP-2, so it is important to ensure that the simulator, the hardware and the code generator all share the same configuration.

The simulator's debugging switch (`-d`) causes traces to be emitted at the microprogram level. This is useful for debugging microprograms. More sophisticated debugging is possible using `mcgrep_mcuc_test.py`, an enhanced version of the simulator which supports the following additional features:

- `--debug-entry`: activates debugging after a specific microprogram state has been reached. This avoids the need to trace all the microinstructions executed before that point.
- `--dump_to`: dumps memory and registers to two files beginning with the specified name when the simulator exits.
- `--rebuild-debugging`: keep each successive version of the automatically generated C components of the simulator.

Adding features to MCGREP-2 CPUs may involve writing both VHDL and C implementations for each new type of microoperation. This type of extension can be debugged using `mcgrep_hardware_test.py`, which compares the functionality of the hardware and software models (section 6.4.1). This is done through the debugging harness on the embedded system (implemented by `System_Harness.vhd`), which communicates by a RS232 serial link with the `hardware_test.py` program on your workstation. Section A.11 has information about setting up an FPGA prototyping board to host the embedded system.

A.10 Extending MCGREP-2

The bulk of the MCGREP-2 software is stored within the `/mcgrep2-src/mcgrep` directory. This directory is a Python package named `mcgrep`: if changes are made, it may be necessary to run `install.sh` again to update Python.

The parts of MCGREP-2 that are explicitly extensible are found in `/mcgrep2-src/mcgrep/-architectures`. The `microblaze` architecture is derived from the base `openrisc` architecture. Many new features can be added by deriving a new architecture from one of these. Each directory must include a standard set of files which are loaded by the `arch` module of MCGREP-2. The architecture to be used is selected by a configuration parameter (Table A.1).

The trace generating software (section 6.3) is not part of MCGREP-2: it communicates with the MCGREP-2 tools via an interface as shown in Figure 6.7. This software can be found in `/tracegen`. The trace generator itself is found in `sssopt.c`, with the MCGREP-2 interface in `mcgrepinf.c` and the machine code interface in `isadecode.c`.

The WCET reduction software (section 7.2) is located in `/wcetreduce/rtas.py`. This code exposes a variety of interfaces to allow extensions and experiments: usage examples can be found in `/wcetreduce/shorttest.py` and `/wcetreduce/control.py`.

A.11 Connecting an FPGA

Some of the tests and experiments require an FPGA. At present, FPGAs come in a wide variety of packages on a wide variety of prototyping boards. Although VHDL is standard [17], there is no universal standard for FPGA hardware or prototyping boards, and each device has different pin connections and peripherals. Therefore, the components of the work that require an FPGA will probably need to be changed to meet your requirements.

MCGREP-1 experiments make use of the York RTS Group Virtual Lab. This provides a terminal interface to FPGA hardware, allowing users to send commands to FPGA hardware and see the results in an applet (Figure 5.16). From the perspective of the T80 CPU that acts as a microcontroller for the debugging hardware (section 5.3.1), the 40x25 text display and the serial input are memory mapped devices. If necessary, the Virtual Lab features can be recreated by replacing these two devices with the implementations in `mcgrep2-src/vhdl/src/generic/vga_module.vhd` and `mcgrep2-src/vhdl/src/generic/uart_module.vhd`. This can be done by changing the `mcgrep1/hw/common/monitor_bus_bridge.vhd` module so that bus transactions are sent to local devices rather than being encoded for the external Virtual Lab interface.

Because of the potential difficulty of reproducing Virtual Lab features in future systems, the MCGREP-2 experiments and tests use a simple RS232 serial connection to communicate with an FPGA. The FPGA prototyping board used for the MCGREP-2 tests is the Avnet/Memec MM1 "mini module" (section A.8), but this can be replaced by any prototyping board that provides RS232 line drivers and at least 1Mb of SRAM in addition to a suitable FPGA. To use a different FPGA or prototyping board with the existing tools, refer to the files in `mcgrep2-src/vhdl/boards/ref`. These specify the prototyping board parameters. `ref.vhd` is the top level VHDL file and should provide pins for a RS232 serial connection, a clock and SRAM. `ref.ucf` specifies the pin names. The other files are used by various parts of the Xilinx synthesis process.

The `ref` subdirectory can be copied to create an entirely new board target. The `build_ref_hw` script in `mcgrep2-src/vhdl` should be modified to specify the board directory (`NAME`) and the FPGA type (`PART`). Various other board targets exist in the archive, but not all of these have been tested.

The MCGREP-2 software does not set the baud rate of the serial connection on the workstation, so a terminal emulator program must be used to do this. The baud rate expected by the MM1 system is 57600 bits per second (8 bits per character, no parity, 1 stop bit, no flow control). This can be changed by modifying the `uart_divisor` parameter of the `System` component in `ref.vhd`. The equation is:

$$d = \frac{f}{64b} \tag{A.1}$$

where d is the divisor to be rounded to nearest integer, f is the FPGA input clock frequency in Hz, and b is the baud rate in bits per second. For the MM1 system, $f = 100$MHz and the divisor $d = 27$.

The test programs call the `download-bit.sh` script in `xilinx-ise` to send bit files to the FPGA. This can call any other program to do the work: the Xilinx Impact program is a possibility.

Appendix Software Tests

▷ `/utils/test`

Test case directory.

▷ `/results`

Test results directory.

▷ `/regtest`

Regression test data.

By default, the script uses the Virtual Lab to download bit files.

A.12 Appendix Software Tests

The tests produce most of the results printed in this thesis. Some tests require the Xilinx tools, others require an FPGA. The `b` programs in the test case directory have the following functions:

- `b1.py`: installation sanity checks. This test is very short.
- `b2.py`: synthesis tests. All of the tests carried out by this program require Xilinx ISE. This test takes around 3 hours on a 2008-era PC.
- `b3.py`: software tests, part 1. MCGREP-1 software is tested, along with some of MCGREP-2. This test takes around 24 hours.
- `b3a.py`: software tests, part 2. MCGREP-2 tests are completed, and the WCET reduction experiments described in chapters 7 and 8 are performed. This test takes around 24 hours.
- `b4.py`: hardware test. The MCGREP-2 hardware is compared against the simulator in both machine code and custom RFU execution mode for each benchmark. In this test case, the trace generator software is executed on the FPGA itself before each benchmark is executed. Before starting this test, see section A.11. This test takes around 4 hours.
- `b6.py`: result production. The test results are finalised and a report is produced in `/results/reportcharts.html`. This report can be viewed using a Web browser. If the ImageMagick program `compare` is installed, the generated script `/results/reportcharts.sh` can be executed to generate a visual comparison between the regression test data and the latest results. This test is very short.

A.13 Third Party Software and Hardware

The archive includes a number of software programs and hardware designs written by others. These are redistributed under the terms of the GNU General Public License version 2. This licence can be found in the file named `/LICENSE` in the root of the archive.

Complete source code for all of the following programs can be found in `/3rdparty-sw/src`:

- `/3rdparty-sw/glpk`: the GNU Linear Programming Kit, version 4.22 [100]. This component is sourced from the Free Software Foundation website. No modifications have been made.
- `/3rdparty-sw/openrisctools`: `newlib`, `gcc` and `binutils` for the OpenRISC ORBIS32 ISA [151]. These components are sourced from the Opencores.org website. Some changes were necessary in order to compile this software: these are present within three patch files in the `src` directory.
- `/3rdparty-sw/microblazetools`: `newlib`, `gcc`, `libgloss` and `binutils` for the Microblaze ISA [286]. These components are sourced from Petalogix. Some changes were necessary in order to compile this software: these are present within one patch files in the `src` directory.
- `/3rdparty-sw/openrisctools`: `libgloss` for the OpenRISC ORBIS32 ISA. This component is based on `libgloss` for Microblaze. It includes modifications to use the simulator system call interface (section 6.4.1) for access to files on the host workstation.
- `/3rdparty-sw/or1ksim`: the simulator for ORBIS32. This component is sourced from the Opencores.org website. No modifications have been made.
- `hex2rom`: this converts binary files into read-only memory implemented in block RAM. It has been modified to generate ROMs with a Wishbone bus.

Complete VHDL/Verilog source code for the following third-party hardware designs can be found in `/3rdparty-cores`:

- `/3rdparty-cores/t80`: the T80 CPU [269]. This component is sourced from the Opencores.org website.
- `/3rdparty-cores/or1200`: the OpenRISC CPU [151]. This component is sourced from the Opencores.org website.
- `/3rdparty-cores/mem_ctrl`: a memory controller from Opencores.org.

Some of the benchmark programs are *not* licenced under the GNU General Public Licence version 2. In these cases, a file named `LICENSE` is present in the benchmark source directory detailing the terms for that specific program. These terms only apply to files in that directory, and have previously permitted redistribution of the code within MIBench [108] and Mediabench [154].

Finally, the following additional third-party code is used:

- `/utils/virtual-python.py`: clones an installation of Python. This component is public domain software.