Modern Computer Architectures
Practical Assignments
2016 - 2017

Joost Hoozemans
j.j.hoozemans@tudelft.nl

Jeroen van Straten
j.vanstraten-1@tudelft.nl

Thomas Marconi
t.marconi@tudelft.nl

Stephan Wong
j.s.s.m.wong@tudelft.nl

Sorin Cotofana
s.d.cotofana@tudelft.nl

Chapter 1

Introduction

This document describes the lab assignments for the Modern/Embedded Computer Architectures course (ET4074/IN4340) given at TU Delft starting with the academic year 2016-2017. The purpose of this lab is to let students exercise the design process of a Very Long Instruction Word (VLIW) processor and its integration into a System-on-Chip (SoC) including caches, busses and other peripherals, optimized for a selected set of applications/benchmarks. Instead of following the "cookbook" lab paradigm where the steps of this process are outlined in detail, we decided to make available to the students a parameterized VLIW core and later on a parameterized SoC platform and ask them to identify the best parameter values in order to optimize the application execution in terms of certain metrics, e.g., performance, power/energy consumption, and/or resource utilization. Subsequently, the students are asked to substantiate their design decisions through measured results from either simulation and/or after FPGA synthesis and describe their proposal and findings in a written report.

The lab comprises two parts. In the first part, the student[1] is asked to perform a design space exploration for a VLIW processor to find an optimal processor instance for two given programs. In the second part, the student will use the gained knowledge to create and evaluate an optimal SoC design for a given workload consisting of multiple programs (> 2). Note that the embedding of the VLIW processor in an SoC also means that certain SoC parameters must be determined to derive an optimal solution. In both assignments, the students must quantify and explain the reasons behind their choices.

Your designs will be implemented using the ρ-VEX platform, which contains a VHDL description of a parametrized VLIW processor, by using the parameter values you identified during the design space exploration process. In this way, your designs can be simulated, synthesized, and evaluated on an FPGA board.

1.1 Assignment 1

Goal The goal of this assignment is to perform and document a Design Space Exploration (DSE) process to determine a VLIW processor configuration opti-

[1] In the remainder of this document, the student will be addressed in a more direct form through the use the word "you".

mized for two given application according to specific metrics. The to be considered metrics are performance and resource utilization.

Tools/platform Each student group is given two applications from the Powerstone benchmark suite and a (software) development platform. The applications from the Powerstone benchmark suite can be run stand-alone without the need for OS support or large software libraries. Within the development platform, the VEX toolchain should be utilized to exercise the DSE process.

Expected Outcome Via the given toolchain, the students must measure the figure of merit of their chosen solution and describe their findings in a report (max 4 pages). The expected content of the report can be found in the assignment description.

A detailed description of Assignment 1 can be found in Chapter 2.

1.2 Assignment 2

Goal The goal of this assignment is to perform and document a DSE process to determine an SoC platform configuration optimized for a given set of applications (called workload) according to specific metrics. The to be considered metrics are performance, energy consumption, and area utilization.

Tools/platform For this assignment, you will be given a new configurable platform that contains multiple VLIW cores. You need to determine its configuration parameters to efficiently execute all given applications, e.g., what type of cores should be used: a single core, multiple homogeneous cores, or several heterogeneous cores. Moreover, other associated aspects of the platform must be considered at the same time, e.g., cache sizes, that impact the aforementioned metrics. Note that the vast majority of the code representing the system is generated for you; the focus is on design-space exploration, not the platform itself.

Expected Outcome After determining the best SoC configuration, you need to demonstrate that your proposed solution provides the best solution for the application set execution in terms of the chosen metrics. As you need to document the DSE process, you need to include the results of your intermediate solutions leading towards the final one.

A detailed description of Assignment 2 can be found in Chapter 3.

1.3 Grading

The two assignments contribute to the final grade as follows:

$$Grade = 0.4 \times A1 + 0.6 \times A2$$

For each assignment grade the following evaluation criteria are in place:

- The performance of your core/platform. Note that while performance improvement is important, as it captures the capability of your solution to faster execute the application, we also take into consideration the cost of this improvement. Therefore, higher appreciation will be given to effective solutions which provide a good *balance* between investment and reward.
- The technical merit of your approach. Aspects as innovation level and implementation quality are considered mostly for Assignment 2.
- The report. Report organization, content, and language are important aspects at this point.

Chapter 2

Assignment 1

2.1 Revision History

Updated 2017-09-19:

- Added section on clustering and how to estimate resource utilization.
- Miscellaneous clarification and typo fixes.

Updated 2016-09-09:

- Updated assignment description for the new virtual machine and workspace organization.

2.2 Overview

VEX ("Very Long Instruction Word Example") is a system conceived by the authors of the book "Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools" (Joseph A. Fisher, Paolo Faraboschi, Clifford Young 2005). It consists of a flexible instruction-set architecture (ISA), a compiler, and a compiled simulation system. The latter two are included in the VEX toolchain that is included in the Assignment 1 archive and Virtual Machine workspace, and can also be downloaded at: `http://www.hpl.hp.com/downloads/vex`. The system is based on the HP/STMicro Lx family of processors (for example the ST200).

In Section 2.3, we present the assignment definition and state its goals. Section 2.4 describes the tools that you will be using in this assignment. Section 2.5 will show you how to compile and simulate a program using the toolchain. Section 2.6 shows how to view and interpret the simulation results. Section 2.7 describes the primary things that you need to consider in your area utilization model and briefly explains how clustering works. In Section 2.8, the machine configuration file is described; you can use it to change the design parameters of your processor. Section 2.9 furthermore describes how you can (optionally) tweak the compiler settings as well. Finally, Section 2.10 will describe what you need to turn in to finalize this assignment and the expected content of your report.

2.3 Definition and Goals

In the first assignment, you are asked to design an "optimal" VLIW processor for two given applications taken from the Powerstone benchmark suite. In the design, you must configure a multitude of parameters (see Section 2.8) and perform measurements to optimize your design in terms of performance and resource utilization. It is important that you understand the interplay between these metrics for your given application. This Design Space Exploration (DSE) process must be documented in a report (max. 4 pages). The expected content is described in Section 2.10.

2.4 Explanation of Tools

2.4.1 Virtual Machine

For the assignments, we have created a Virtual Machine running OpenSUSE that you can download and run using Virtualbox (VMWare should also work). Username and password are both `user`. The root account also has password `user`. The desktop of the VM has some shortcuts on it for maintenance commands:

- **Set group nr. and netID** This script will configure the virtual machine for your group number, select the benchmarks that you will be using, and configure the VPN connection using your netID. When you work from home on assignment 2, you will need this VPN connection to get access to the TU Delft license servers to run simulation and synthesis. Figure 2.1 shows how to enable the VPN connection. If the virtual machine asks you for a wallet password, press cancel, until it presents you with a VPN secrets dialog with two password fields. The first should be your netID password, the second should be set to 'wireless'.
- **Pull toolchain updates Please run this script after downloading the VM to update it to the latest version.** You do not have to run it again unless we instruct you to through brightspace or e-mail (we may need to make changes to the VM scripts in case we encounter problems with them).

If you need additional software, you can use YaST (GUI) or `zypper` (command line).

2.4.2 VEX Compiler

The compiler can target a range of different machine organizations and it can be configured by supplying a machine configuration file. In this way, the programmer can change the number of clusters, execution units, issue width, and functional unit operation latencies without having to recompile the compiler. By compiling and simulating programs using different parameters in the configuration file and analyzing the results, you can perform the DSE process.

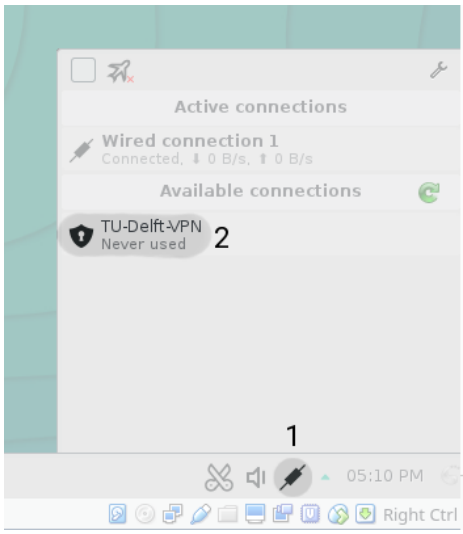

Figure 2.1: Enabling the VPN connection.

2.4.3 VEX Simulation System

While in normal conditions the compiler produces assembly code according to the VEX Instruction Set Architecture (ISA) description, we will use it to perform a cross-compilation for your host machine (the VM). This is a standard procedure for not yet completely defined processor architectures which allows you to evaluate the performance of a certain processor instance when executing a given application, without actually implementing the new processor. The compiler will generate a program that emulates the execution of the compiled benchmark on it. The compiler and simulator include many standard library functions, and the terminal output of the simulation is fed back to the host computer. This way, the programmer can still use functions like `printf`.

2.5 Compiling and Simulating Programs with VEX

Each group is assigned two programs from the Powerstone benchmark suite to analyze. Run the 'Set group nr. and netID' command on the desktop to see which programs are assigned to your group. We will use `blit` as an example to describe the tools.

For your convenience, we have created a command that allows you to compile and run VEX simulations easily. Open the file manager (Dolphin) and select `Assignment 1` in the places menu (this points to `/home/user/workspace/assignment1/`). Notice the terminal at the bottom of the file manager. You will use this to run compilation commands, etc. If it is not there, press F4. Now go into the `configurations/example-2-issue` directory. You need to actually enter the directories, not just unfold them in the file tree, otherwise the terminal will not change its working directory.

In the terminal, type `run blit -O3` and press enter. The `-O3` is a flag passed to the VEX compiler, which selects optimization level 3. You should see `blit: success` appear in the terminal after some compiler warnings, and the directory `output-blit.c` should have appeared. This directory contains all the output files for the simulation that you just ran. Note that the contents of this directory will be completely overwritten when you run the `run blit` command again. This directory contains the following files.

- `a.out`: this is the x86 executable that simulates the benchmark. The `run` script calls this automatically.
- `blit.c`: this is the C source code of the benchmark, as passed to the VEX compiler. Note that this file is a *symlink*, which is a bit like a shortcut on Windows but more powerful: if you change this file, the original file is also modified, and vice versa.
- `blit.cs.c`: this is the C source code for `a.out`, generated by the VEX compiler to simulate `blit.c` with the given architecture. It is simply compiled with `gcc`.
- `blit.s`: this is the VEX assembly file corresponding to the simulation. You can (and should) analyze the assembly code with a text editor to get an idea of where the performance bottlenecks are.
- `gmon*.out`: these files contain `gprof` performance logs of the simulated benchmark taking various caches into consideration. Evaluating cache performance is beyond the scope of assignment 1, so we only need `gmon-nocache.out`. Note that these are binary files. To interpret them, run `gprof a.out gmon-nocache.out > gmon-nocache.txt` in the console and open `gmon-nocache.txt` in a text editor.
- `ta.log.001`: this is the simulation log file. Unlike the gmon files, this is already a text file. **The most important number is the number of 'execution cycles'.** The 'total cycles' also takes into account the cache performance, which, again, is beyond the scope of this assignment.
- `vex.cfg`: this is a symlink to the cache configuration file. There is no need to do anything with this file.

2.6 Analyzing Simulation Results

As mentioned in the file description, the most important metric (together with your estimation of the area utilization of a given design) is the 'execution cycles' number in the generated `ta.log.001` files. While optimizing your design though, you may want more information about why a certain design gives certain results.

The VEX toolchain has a number of tools available to this end. These tools can analyze call graphs, control flow graphs, instruction schedules, and visualize them using the VCG (Visualization of Compiler Graph) tool that is also included. Of course, you can also try to analyze the assembly files yourselves.

You are advised to read the `vex.pdf` documentation (go to Documentation in the file manager) and use it to your advantage!

As an example, we will demonstrate the use of the `pcntl` program. It can be invoked as
`pcntl blit.s` from the simulation output directory. It should produce the following output (ignoring the `perl` deprecation warning at the top).

```
Procedure: blitf::
 Trace      IPC  Cycles    Oper    Copy     Nop
-----------------------------------------------
     7    1.53      17      26       0       0
     1    1.52      23      35       0       2
    17    1.80       5       9       0       0
    16    1.80       5       9       0       0
    14    1.80       5       9       0       0
     8    1.12       8       9       0       0
     9    2.00       9      18       0       0
     4    1.52      23      35       0       2
    22    1.80       5       9       0       0
    21    2.00       5      10       0       0
    19    1.80       5       9       0       0
    11    1.25       8      10       0       0
    10    1.38       8      11       0       0
     3    1.53      15      23       0       2
    23    1.83       6      11       0       0
    20    1.83       6      11       0       0
    18    1.83       6      11       0       0
    12    1.62       8      13       0       0
    13    1.89       9      17       0       0
Operations    = 285
Instructions  = 176
Reg. moves    = 0
Nops          = 6
Avg ILP = 1.61932
Procedure: main::
 Trace      IPC  Cycles    Oper    Copy     Nop
-----------------------------------------------
     1    1.09      22      24       0       1
     2    1.00       5       5       0       1
Operations    = 29
Instructions  = 27
Reg. moves    = 0
Nops          = 2
Avg ILP = 1.07407
```

As we can see, this utility analyzes information for every function in the program, and every compilation trace per function with statistics. Let us look into some of the metrics stated above:

- IPC: Instructions per Cycle: The number of instructions per second for a processor can be derived by multiplying the instructions per cycle and the clock speed of the processor, indicating the performance of the processor.

- Cycles: Number of scheduled cycles
- Operations: Number of scheduled operations
- Nops: Number of "No Operations" in each trace; The final value is a sum of the total number of operations in each trace
- Average ILP: ILP is a measure of how many of the operations in a computer program can be performed simultaneously

2.7 Resource utilization

As stated, we expect you to balance performance vs. area utilization, but as you have probably noticed, the tools do not give you any area numbers. This makes sense, because at this point in the design-space exploration process there is no processor implementation yet! Instead, you will need to come up with a model that converts the machine configuration to a number that you think will be approximately proportional to the area of the implemented processor.

In a VLIW processor, the register file is a major contributor to the area utilization. This is because a VLIW with many functional resources also need many registers to keep those functional resources fed with data. At the same time, more operations running in parallel requires more read/write access ports to the register file. The size of a memory is roughly proportional to its depth (i.e. the number of registers), times the number of read ports, times the number of write ports (there are two read ports and one write port for each issue slot in VEX). Thus, there is an approximately cubic relationship between the amount of functional resources and the area utilization for a VLIW with a sufficiently large register file. Note that you select the number of registers independently from the issue width, so the register file area is proportional to the number of registers times the square of the issue width.

It is possible to reduce the area of a VLIW without reducing its functional resources by means of a technique called clustering. A clustered VLIW consists of multiple sets of functional resources, which each have their own register file (the clusters). The area utilization of each *individual cluster* is thus roughly proportional to the cube of its functional resources, whereas the total area utilization is merely the sum of the clusters. VLIW clusters operate in lockstep; i.e., they share the same program counter, which points to a bundle of instructions mapping to each cluster. In other words, all clusters work together to execute a single, common thread. Because all clusters are always working on exactly the same part of a program, inter-cluster communication can be done without any kind of additional synchronization primitives. The compiler automatically handles this communication for the programmer; the programmer can write a normal single-threaded program exactly like they otherwise would.

There is, of course, a performance penalty for a clustered VLIW compared to a non-clustered VLIW with the same amount of functional resources, because inter-cluster register moves require instructions that would not otherwise exist. It is up to you to investigate whether clustering is worthwhile for your benchmarks, and to what degree. Note that VEX supports only processors with 1, 2, or 4 clusters, and that each cluster needs at least an issue width of 2 instructions per cycle.

Note also that the register file size is not the only contributing factor to the area utilization. Try to expand your area utilization model to the other parameters of the machine configuration file as well.

2.8 Changing the Machine Configuration

Your goal for this assignment is to optimize the simulated processor for the given workload. The specification of the processor is given to the VEX toolchain by means of a machine configuration file. Such a file must always be named `configuration.mm` for the `run` command to recognize it, and the `run` command must be run from the directory that contains the file. Two of these machine configuration files are given to you as examples; a basic 2-way VLIW and a basic 4-way 2-cluster VLIW. You may want to make a copy of the configuration directories before you change their configuration.

Note that the VEX compiler seems to silently ignore the configuration file if there are problems with it. The overall issue width and the issue width per cluster must be at least 2, and there must be at least one resource of each type in the processor. Please read the `vex.pdf` file in the Documentation directory for other constraints and information about the machine config file.

You will see that at some point, increasing the issue width and/or number of functional units will not substantially increase the performance. This is the law of diminishing returns at work. The question you need to ask yourself is: what do I get in return (in terms of increasing performance) by investing in a larger chip (area, power consumption)? You should include one or multiple plots about this in your report.

You should not change the delay (`DEL`) values; they match the ρ-VEX implementation you will be using in assignment 2.

2.9 Changing the Compiler Flags

In addition to the processor itself, the compiler and its configuration also affect performance. Any flags (the parts of a command line after a `-`, such as `-O3`) passed to the `run` script *after* the benchmark name are passed to the VEX compiler. Again, read `vex.pdf` for information about the available flags. You are also allowed to add `#pragma` statements to the benchmark C files to give more fine-grained information to the compiler if you want, as long as you do not change the actual code. Doing these things is optional.

2.10 What to turn in

In order to successfully complete this assignment you need to turn in the following:

- The machine configuration file;
- The source files for your benchmarks (if you changed them);
- A report.

The report should be maximum 4 pages and should contain at least the following:

- Description of the given benchmarks from a high-level perspective (C-Code);
- Assumptions about the environment that you think that the programs will be used. For example; a desktop computer, a mobile phone, copying machine etc. These assumptions should be used in your design decisions (a desktop machine needs to be fast, an embedded devices needs to be small and low-power);
- Discussion about the processor resources and how they impact the performance of the benchmarks and your area model;
- Your final solutions and results;
- Reflection about what you learned in this assignment.

The report must reflect the choices you made during the design space exploration process. Gather results using different configuration parameters and elaborate on them in your report. Analyze the results in your report and create a plot of the most interesting designs. Choose which design you think is the most well-balanced and explain why. The `pcntl` and related tools and the `ta.log` file present you with a lot of information. Part of the assignment is to show that you are able to identify what information is important, and how to base your design decisions on them. So do not produce graphs of all the numbers you can find, but only present the data that you think is most relevant! You will find that your report will become way longer than 4 pages otherwise.

Chapter 3

Assignment 2

3.1 Revision History

Updated 2017-09-19:

- Updated the boardserver section for the new websocket-based protocol.
- Miscellaneous clarification and typo fixes.

Updated 2016-10-08:

- Updated assignment description for the new virtual machine and workspace organization.

3.2 Overview

This chapter describes the second part of the lab:

- Section 3.3 presents the goals for this assignment.
- Section 3.4 introduces the platform that you will be using.
- Section 3.5 describes how to configure the hardware and what the generated directory structure looks like.
- Section 3.6 describes how to configure and modify the C source code for your platform.
- Section 3.7 describes the commands that are at your disposal for running your design using modelsim simulation and the boardserver.
- Section 3.8 describes how to interpret the results that are generated by synthesis and the boardserver.
- Section 3.9 describes what you need to turn in at the end of this assignment.

3.3 Goals and Definitions

In the first part of the lab, you learned how to analyze a program and select well-balanced VEX design parameters for it. In the second part, you will use an actual SoC design with one or more ρ-VEX cores to run a small workload consisting of four Powerstone benchmarks, two of which are the benchmarks you used in assignment 1. Your task is to create a well-balanced SoC design that will run your workload as efficient as possible based on performance, area utilization, and energy consumption.

You should start by analyzing the two new benchmarks in the same way you did for assignment 1, in order to determine what kind of processor would execute them efficiently. You can use the workspace for assignment 1 for this. The goal is now to map the programs to a real ρ-VEX core however, which does not have quite as much configuration options as the VEX compiler can handle. In fact, the only things you can vary in the machine configuration file are the issue width (2, 4, or 8) and the number of multipliers. The `rvex.mm` file in the assignment 2 root directory contains the values that you need to use for the other parameters for the configuration to be valid. You can copy this file into a new configuration directory within the assignment 1 folder and rename it to `configuration.mm` to simulate with it.

Based on the simulation results, you should form an idea of how to run your workload on a SoC design with one or more ρ-VEX processors, and discuss this in your report. Subsequently, you need to implement your design using the given framework, simulate it, and run it on an actual FPGA board. You will get results about your design in terms of performance (delay), area (resource usage on the FPGA), and energy (measured on the FPGA). Based on the output from the board server, you can also modify the cache sizes and stop bit configuration of the processor(s) and iterate.

Consider 3 scenario's; a design targeting an embedded low-power environment, a design targeting a high-performance environment, and a well-balanced design. Plot your results of these three designs in your report and discuss what you think is the best design. Using the results, reflect back on your expectations based on the design exploration process with the VEX simulator.

3.4 ρ-VEX softcore

The Computer Engineering lab at TU Delft has implemented the VEX ISA in a softcore using VHDL. This design, called ρ-VEX ("reconfigurable VEX"), can be synthesized using different configurations, much like the configuration parameters you experienced in the first lab. For programs to run correctly on the hardware, the machine configuration used by the compiler and the flags passed to the assembler must match the hardware parameters. These include the issue width, and the number and location of different functional units and their respective latencies.

The ρ-VEX can also be configured to be runtime-reconfigurable. This allows the processor to either function as one wide-issue VLIW, or as a number of more narrow-issue VLIWs. From a programmer's perspective, one reconfigurable ρ-VEX behaves as multiple virtual ρ-VEX cores, which can each have varying issue widths or be disabled altogether. It is somewhat similar to hyperthreading,

except it is the responsibility of the programmer[1] to divide the resources between the functional units. Note that reconfiguration is almost instantaneous; the cost is in the order of only 5-10 clock cycles. This is primarily due to the required pipeline flush.

In order to allow a single program to run correctly for multiple issue widths, we have developed so-called generic binaries for the ρ-VEX. A generic binary is compiled as if it is to run on an 8-issue ρ-VEX. The assembly is then post-processed by a tool called VEXparse (as we do not have access to the HP VEX compiler source code) and the assembler to ensure that no data dependencies are violated when the 8-way bundles are run as if they are four 2-way bundles or as two 4-way bundles.

The ρ-VEX includes a single-level data and instruction cache. The cache organization is non-assiciative buffered write-through, using bus snooping to maintain coherency. The size is configurable, and is part of the parameters that you can (and should) vary for assignment 2. Performance counters that measure hit rate are included in the processor, to allow you to analyze the behavior of the cache on the FPGA platform.

When the ρ-VEX reconfigures at runtime, so must the cache. This is implemented by instantiating what we call a cache block for each part of the processor that can be individually assigned to differing virtual cores. When different cache blocks are mapped to the same virtual processor, they work together to behave like a single, larger cache. For example, if a 4-issue reconfigurable ρ-VEX that has 16 kiB worth of instruction cache splits into two 2-issue virtual cores, each virtual core has access to 8 kiB worth of icache.

Finally, the ρ-VEX utilizes a bit in the syllable (operation) encoding called the stop bit to determine whether the next syllable is part of the current instruction bundle or the next. A set stop bit is allowed only every Nth syllable; if a bundle is not an integer multiple of N in size, no-operation syllables must be inserted as padding. N is configurable at design time, and is part of the parameters that you can (and should) vary for the lab. Decreasing the number increases area somewhat, but decreases the size of the binary, and thus instruction cache pressure. It also significantlty improves the performance of a generic binary running on a less-than-8-issue core.

Refer to `rvex.pdf` if you would like to learn more about the ρ-VEX processor.

3.5 Hardware configuration

The platform for assignment 2 is configured in two steps. The first step configures the hardware; that is, the number of cores, and the configuration for each core. The second step configures the software.

The hardware is configured using a single configuration file, similar to the machine model file from assignment 1. You can find a generously commented example configuration file for a platform with a single, basic 4-way ρ-VEX processor in the `configurations/example` folder within the `Assignment 2` workspace. Like in assignment 1, you can modify the file in-place, or you can copy the `example` folder for as many configurations as you like.

[1] It can also be managed by an operating system or hardware scheduler, driven by the compiler and/or runtime measurements. This is a subject of ongoing research at the CE lab.

When you have updated the configuration file, you can use it to generate the platform. Open the file manager or a terminal, change to the directory that contains the file, make sure that the file is named `configuration.rvex`, and then run `configure` in the terminal. Similar to the `run` command from assignment 1, this will generate some files and folders, although this time it is not specific to a benchmark. Note that if the files that it would generate already exist, it will ask you if you really want to replace them. This precaution is in place because it may override sources that you have created and/or delete synthesis results otherwise, which may take a long time to recreate.

The `configure` command should generate three folders and one file, listed below.

- `src`: this folder contains the sources for the software that will be run on the platform. More information is presented in Section 3.6.
- `results`: this folder will contain the results as they are generated. More information about the results can be found in Section 3.8.
- `data`: this folder contains the compilation and FPGA synthesis output directories. In general, you should not need to concern yourself with the contents of this directory. The only things that may prove useful are the compilation output directories (`data/compile/core*`), in case you want to analyze the actual assembly files for your program (`*.s`), the assembly files after VEXparse post-processing (`*.sv`, only for reconfigurable cores), or the disassembly file for the complete program (`out.disas`).
- `Makefile`: this is a script file for a standard Linux tool called `make`, which is typically used to handle incremental builds of software. In this case, it is also used to start modelsim and synthesis, and to run the design on the FPGA boardserver. You should not edit this file.

3.6 Software configuration

When you first generate hardware from the `configuration.rvex` file, example sources for the software to run on it are also generated, and placed in the `src` directory. The example source files are already customized for your group's benchmark selection, but they simply run all the benchmarks sequentially on the first (virtual) core. The other cores, if they exist, simply terminate without doing anything.

The list of generated files is as follows.

- `config.compile`: this file controls which C files are mapped to which virtual core, and what compiler flags are used to compile them. It is described in more detail below.
- `<benchmark>.c` (4x): *copies* of the *original* Powerstone benchmarks selected for your group. Thus, in contrast to assignment 1, the sources are not symlinked, allowing you to easily have different benchmark sources for different configurations. Also, if you changed the benchmark sources in assignment 1, you will need to change them again here.

- `main-core*.c`: these files contain the entry points for each of the virtual cores in your platform. **You will need to modify these manually when you move benchmarks from one core to another in `config.compile`.** `config.compile` only determines how the benchmark sources are compiled, not when and where they are actually run.
- `benchmarks.h`: this file declares the entry point functions for each of your benchmarks. Each of these should be called by the toplevel `main`s exactly once in total.
- `intercore.h`: this file declares a `struct` that is placed in a shared memory region, allowing the cores to communicate with each other. Note that you should not need this file for most designs; it is there in case you want to do something fancy. You can modify the `struct` as you fit, as long as its size is no larger than 16 MiB. Note that the `struct` is not initialized, and thus contains garbage at the start of your program.

The `config.compile` file specifies which source file should be compiled for which virtual processor and how. It contains a section for each virtual core in your platform, marked by the `[coreX.Y]` tags, where `X` is the index of the physical core within the platform, and `Y` is the index of the virtual core within the physical one. These sections should correspond to your hardware configuration, and to how you call the benchmarks in the `main-core*.c` files.

Within the sections, each line represents a (set of) sources that should be compiled for the virtual core. The first word specifies the source(s), using the following format:

- `name`: compiles the C source `name.c` from the source directory.
- `name-sub`: compiles the C source `name.c` from the source directory, in such a way that all global function and variable names are prefixed with `name_`. Taking `blit.c` as an example, this would cause the `main()` function defined therein to be known as `blit_main()` outside the scope of that source file. This allows multiple programs each having its own `main()` to be run as functions within a larger program.
- `OTHERS`: this line refers to the startup and library source files that are always compiled alongside your program (even if you remove the line, actually). It is only used to specify the compilation flags to be used for these sources.

Everything after the first whitespace character on each of these lines is interpreted as a set of flags to be passed to the compiler for those sources. Thus, if you would like to compile one of your benchmarks with aggressive loop unrolling, you would add `-H4` to the line referring to that benchmark source.

The source files mapped to each virtual core must contain exactly one `main()` function (not counting the `main`s that are prefixed with the benchmark name), which is used as the entry point for that core. That is, each core runs a program that is not dependent on and in general does not (need to) communicate with the programs running on the other cores. If, for some reason, you do need communication (for elaborate reconfiguration schemes for instance), you can use the `struct` defined in `intercore.h`.

3.6.1 Reconfiguration

The ρ-VEX can be configured to be a runtime-reconfigurable core. This allows the software running on the processor to change some of the parameters of the processor at runtime. If you want an extra challenge, you can make use of these reconfiguration capabilities in your design, but you do not have to do this to get a passing grade for the lab.

As an example use case, let's say that you have a very short and a very long program that can be run in parallel, and you want them both to complete as soon as possible (as is typically the case in the lab). You could then make a design with one very wide core and one small core, each running respectively the long and the short benchmark. Unless the difference in size between the two processor is large enough to account for the difference in program length however, the small core will be idle most of the time, effectively wasting area and energy.

With a reconfigurable core, you could map both benchmarks to the same processor. They can start by executing in parallel, each getting access to half the resources of the processor. However, when the first benchmark finishes, it can transfer its computational resources to the other program, to let the other benchmark finish as soon as possible. Now all computational resources are used all the time if both benchmarks are sufficiently parallel.

If you want to use reconfiguration, use the `ab:ab:ab:ab`, `abcd:abcd`, or `ab:ab` format for the `CONFIG` tag in `configuration.rvex`. The sections divided by the colons are called lane groups. Each lane group can be individually assigned to a different virtual core (also called a context), in such a way that a virtual core can be mapped to zero, one, or multiple lane groups. A lane group can also be turned off to save power. Changing the reconfiguration is done by writing a configuration word to `CR_CRR`, defined in `rvex.h`. It behaves like an `int` variable. Chapter 6 of Appendix C of `rvex.pdf` (starts on page C-83) explains the format of the configuration word.

Please note that we will prioritize answering questions related to the required parts of the lab over supporting reconfiguration. Think of it as a challenge to try to figure something out on your own.

3.7 Running your design

When you have finished configuring your design and want to run it, use the file manager or terminal to go to the directory that contains your `configuration.rvex` file. The following commands are now available to you:

- `make compile`: compiles your software.
- `make sim`: simulates the FPGA platform with the current software in modelsim. This command requires you to be connected to the campus network or use the VPN connection (Figure 2.1). Modelsim simulation is described in more detail in Section 3.7.1.
- `make synth`: synthesizes the design to allow you to run it on the FPGA using the boardserver. This command requires you to be connected to the campus network or use the VPN connection (Figure 2.1). You do not need to resynthesize if you only change the software. Synthesis usually takes at

least half an hour, but it depends greatly on your design complexity. It is also possible for synthesis to fail entirely if your design is too complex, of course.

- `make run`: sends the design to the FPGA boardserver. This command requires you to be connected to the campus network or use the VPN connection (Figure 2.1). This will run `make compile` first if you have changed the source files. If you have not synthesized yet, the command will ask you if you want to do that (and cancel if not). The functionality of the boardserver is described in more detail in Section 3.7.2.
- `make clean`: deletes all intermediate files. If you get weird errors from the other commands, running this and then trying again is typically not a bad idea. This command does not touch your source files, the synthesized design, or the `results` directory.
- `make pack`: compresses your configuration file, `src` directory, `results` directory, and FPGA bitstream to `design.tgz`. Please send the generated archive(s) for your design(s) along with your report when handing in your work.
- `make`: prints an overview of the commands shown above.

Thus, in total you have three options to analyze your design:

- The VEX simulator from assignment 1. This can only do one core at a time and does not simulate cache behavior, but it provides results almost instantaneously.
- Modelsim (Section 3.7.1). This is a relatively accurate simulation that simulates your entire platform, albeit with a simplified model for the DDR memory, so the numbers you get will not match the real world exactly. Simulation is *very* slow, but it has the advantage that you do not have to synthesize first, which also takes a lot of time. Hint; with some benchmarks you can relatively easily decrease the input size, which allows you to simulate them faster.
- The boardserver (Section 3.7.2). This will run your complete design on actual hardware, to provide you with the results that you need to report. Running a design takes about a minute if there is no queue, but you need to resynthesize every time you change the hardware configuration, and synthesis can take quite a bit of time.

3.7.1 Modelsim simulation

To simulate your design using modelsim, run `make sim` in the directory that contains the `configuration.rvex` file. If you are not on the TU Delft campus, first enable the VPN connection (Figure 2.1). This command will first compile your workload if it is out of date, then compile the VHDL sources of the platform if this is the first run (this takes several minutes), launch the modelsim GUI, and start the simulation. The simulation will stop automatically when all cores have finished running their program, with a message that includes the total amount of cycles that were needed to complete the program.

Note that execution does not start immediately at the start of your `main()`. Before a C function can be executed, the stack needs to be set up, and the so-called BSS section needs to be filled with zeros. This section contains all the global variables of your program, excluding those with an explicit, nonzero initialization value. This is a space-saving optimization, because these zero-initialized values now do not need to be part of the loaded binary. Unfortunately, initializing this piece of memory can take quite some time for some of the benchmarks.

While the simulation fully models the ρ-VEX cores, it does not model the memory completely. On the physical platform, the memory is an off-chip DDR3 DIMM, which, due to the memory controller, has a varying and seemingly non-deterministic latency. The simulation model assumes a fixed latency per 32-bit access. Therefore, the amount of cycles that the program needs to run does not completely match the hardware results.

Aside from providing the debug output of your program and the total cycles consumed, modelsim also provides you with an instruction trace. You can use this to gain a deeper insight of what the processor is doing. To see it, click the waveform view (the black area on the right) to select it, then press [F] to zoom out. Now click on the topmost "signal", i.e. the one marked "Core 0" on the left. You can now use [Tab] and [Shift]+[Tab] to move through the processor cycles. The signal values shown in the gray area to the left of the waveform view show you what the processor is doing for each cycle.

"Ctxt" is short for context, and means virtual core. The information shown to the right of it indicates the program counter and the current state of the core. "Ln" is short for lane, which can be regarded as a functional unit that can handle one syllable/operation per cycle. The information shown on the first line for each lane shows the program memory address that it is executing or, depending on the situation, ignoring, as well as disassembly for that operation. The second line shows the runtime effects that the operation has on the register files and memory.

3.7.2 Using the boardserver

You should get your final performance results by running your design on the boardserver. If you are not on the TU Delft campus, first enable the VPN connection (Figure 2.1). You can then send your design to the server for evaluation by running the `make run` command in the same directory as your `configuration.rvex` file. This command will first compile your workload if it is out of date. Next, if you have not synthesized the design yet, it asks if you want to do that now. Finally, when all dependencies are met, it sends the design to the server.

Running a design on the server takes about a minute on average if there is no queue. The boardserver can only handle one request at a time though, so if multiple groups are running things at the same time, a queue may form. The server uses round-robin scheduling between groups, so if you queue up multiple runs simultaneously, the server may bump some of your runs further down the queue if other groups are also queueing runs.

If the boardserver cannot be reached for some reason, the run script will continuously retry. It will only fail (with a nonzero exit code) if the boardserver actively reports that there is a problem with your design.

3.7.3 Debugging

To allow you to debug your software, each virtual core has its own output stream, akin to `stdout`. In modelsim, these streams all map to the modelsim log window, without any synchronization whatsoever. This unfortunately means that if two cores are writing for instance "hello" at the same time, you might end up with something like "hheelllolo" in the log. Therefore, when debugging things with modelsim, you may want to use just a single characters followed by a newline to indicate certain events. When you run your software on the boardserver, you will get a separate log file for each core, so then this is not an issue.

The functions that you can use to write to these streams are defined in `assignment2/utils/record.h`. Note that while writing to the streams is relatively cheap in terms of performance, it is not free: you may want to disable all the debug prints when you do your final measurements. Note also that a `printf`-like function is missing; there is in fact no C standard library at all within the scope of the lab. You will need to make due with the (simpler) functions provided to you, or write your own.

3.8 Results

The `make synth` and `make run` commands provide you with quite a lot of numbers and other results that give you information about how well your design performs, and if it performs correctly at all. All these results are stored in the `results` directory. Roughly, the results can be divided into four categories: correctness, performance, area, and energy.

3.8.1 Correctness

The first thing that you need to verify after running your design is that it worked correctly. This goes for hardware synthesis as well as the software.

In the case of synthesis, so-called timing errors may occur, which indicate that your design is too complex for the FPGA at the targetted clock frequency. This should not be a problem for the lab as the target clock frequency is set almost twice as low as what we know the core is capable of, but it is reported nonetheless in the `timing.txt` file. If this file contains errors, your design will probably behave erratically when run on the FPGA.

The correctness of the benchmarks is recorded in the debug output, which is logged in the `run*-core*.log` files. To determine whether your task distribution functions are working as you expect them to, you may need to add your own debug prints to your code.

Unfortunately, the HP VEX compiler is not bug-free. Some benchmark-configuration combinations will cause the benchmarks to report failure. In such cases you can ignore the failure (it will not detract from your grade). If you are unsure if you broke something or if the compiler is the problem, ask the lab assistents.

3.8.2 Performance

The overall performance (cycle count) of your solution is recorded in the `performance.txt` file. Recall that the timing of the memory controller can be regarded as nondeterministic, causing there to always be somewhat of a difference in performance between individual runs. The boardserver runs your program three times after it configures the FPGA, to allow you to see how big these differences are, and to also give an average value to use for your figure of merit.

In addition to this, the auto-generated sources also dump performance information to the debug output, using the `log_perfcount(...)` function. You may use these numbers as a guide for further optimizations. The function reports the following statistics for the core itself:

- `CYC`: the total number of cycles that this virtual core has been active in so far. Note that in reconfigurable ρ-VEX processors this may not match the total cycle count, because the virtual cores will be halted when they are not assigned any resources.
- `STALL`: the amount of cycles during which the core was stalled, i.e. waiting for a memory access to complete. Stalled cycles also count towards `CYC`.
- `BUN`: the amount of instruction bundles that have thus far been fetched and executed.
- `SYL`: the amount of operations/syllables that have thus far been fetched and executed. For non-reconfigurable cores without stop-bit support, this value is just `BUN` times the issue width.
- `NOP`: the amount of no-operation operations/syllables that have thus far been fetched and executed. A lower value means that your binary is more efficiently packed.

It also provides statistics from the cache. **However, these statistics are wrong for reconfigurable cores due to the rather crude way in which they are currently measured.**

- `IACC`: instruction cache accesses.
- `IMISS`: instruction cache misses.
- `DRACC`: data cache read accesses.
- `DRMISS`: data cache read misses.
- `DWACC`: data cache write accesses.
- `DWMISS`: data cache write misses.

You may notice that in some cases the numbers do not add up completely. For instance, `SYL / BUN` may appear to be very slightly higher than the issue width if stop bits are disabled. This is caused by the fact that the application itself is reading out the performance counters one by one, causing them to change while they are being read out. This effect is small enough to ignore.

3.8.3 Area

The synthesis command provides you with an extensive report of the FPGA resources used by your design. This report is recorded in `area.txt`. Unfortunately, there is no single area number that summarizes all the others, as you would get for an ASIC. For your report, we would like you to consider:

- the number of slice registers,
- the number of slice LUTs (the generic computational resources of an FPGA),
- the number of RAM blocks (please count each RAMB36E1 as two RAM blocks and each RAMB18E1 as one RAM block, as the RAMB36E1 elements are twice as large as the RAMB18E1 elements),
- and the number of DSP48E1s (used by the multiplication units of the ρ-VEX).

Try to make your comparison based on all of these. The best way to do this is to create a model that combines these four numbers into a single value, based on the relative sizes of the resources. You will need to do some research to figure out what those are (approximately). The boardserver uses a Virtex 6 FPGA (XC6VLX240T-1FFG1156).

3.8.4 Energy

We have a device connected to the FPGA that records power usage and energy consumption while your design is loaded and your software is running. The data from this device is recorded in the `energy.txt` and `run*-power.csv` files.

If your program is long enough, the `run*-power.csv` files include an idle power measurement, the extra energy consumed by your program compared to doing nothing for the period of the program, and the data to plot the extra power over time while your program is running. You will notice that the idle power is several orders of magnitude greater than the extra power consumed by running a program. This is simply a characteristic of FPGAs, and is precisely the reason why we subtract the idle power to get a somewhat meaningful energy number.

The `energy.txt` file simply contains this energy number for each run, as well as an average for the three runs. Please use this average to compare your designs in your report.

Unfortunately, the power/energy consumption of the FPGA is *highly* temperature-dependent, to the point where the die temperature can no longer even be considered to be uniform. Therefore, make note of the variance in the energy measurements and take it into consideration when drawing conclusions. When you have narrowed down your design-space to only a few designs and software configurations, you may want to run these setups several times to get better averages.

3.9 What to turn in

In order to successfully complete this assignment you need to turn in the following:

- Your report.
- The archive generated by means of the `make pack` command (Section 3.7) for your best design. Discuss why you think it is the best in the report - is it the fastest or most efficient?

The report should be maximum 5 pages and should contain at least the following:

- Design-Space Exploration results of the benchmarks.
- Proposals for a high-performance and a low-power design based on DSE.
- Measurement results and discussion of these designs.
- Proposal for well-balanced design and related measurement results.
- Discussion about what you think is the best design of the three you have measured and in what aspects.
- Reflection about what you learned in this assignment.

It is important that there are (at least) three solutions presented in the report, i.e., one optimized for performance, one optimized for area/energy efficiency, and one balanced design. Naturally, you are allowed to explore more designs if you want to invest the required time to perform synthesis and board runs.