ET4074 Modern Computer Architectures Practical Assignments 2019 - 2020

Nicoleta Cucu Laurenciu
N.CucuLaurenciu@tudelft.nl

Yande Jiang
yande.jiang@tudelft.nl

He Wang
H.Wang-13@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 Computer Architectures course (ET4074). 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 goal is 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 gained knowledge will be used 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 reasons behind parameter value choices must be quantified and explained.

The designs will be implemented using the ρ-VEX platform, which contains a VHDL description of a parametrized VLIW processor, by using the parameter values identified during the design space exploration process. In this way, the 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 optimized for two given application according to specific metrics, i.e., performance and resource utilization.

Tools/platform Each student group gets assigned two applications from the Powerstone benchmark suite and is provided with 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 figure of merit of the chosen solution must be measured and the findings described in a report (maximum 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, i.e., performance, energy consumption, and area utilization.

Tools/platform For this assignment, a new configurable platform that contains multiple VLIW cores will be given. The goal is 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 already generated; the focus is on design-space exploration, not on the platform itself.

Expected Outcome After determining the best SoC configuration, evidence should be provided in order to demonstrate that the proposed solution provides the best support for the application set execution in terms of the chosen metrics. As the DSE process should be properly documented, intermediate instances leading towards the final solutions and the corresponding results have to be also provided.

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:

$$\text{Grade} = 0.4 \times \text{A1} + 0.6 \times \text{A2}$$

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

- Core/platform performance. Note that while performance improvement is important, as it captures the capability of the proposed solution to enable faster execution of 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.

- Approach technical merit. Aspects as innovation level and implementation quality are considered mostly for Assignment 2.
- Report. Report organization, content, and language are important aspects at this point.

Chapter 2

Assignment 1

2.1 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 folder and the Virtual Machine workspace. The VEX toolchain 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 2.2, we present the assignment definition and state its goals. Section 2.3 describes the required tools, Section 2.4 explains how to compile and simulate a program using the VEX toolchain, and Section 2.5 provides inside on how to view and interpret the simulation results. Section 2.6 describes the primary aspects that need to be considered in the area utilization model and briefly explains how clustering works. In Section 2.7, the machine configuration file is described; this file is used to change the design parameters of the processor. Section 2.8 describes how to tweak the compiler settings (optionally). Finally, Section 2.9 entails what need to be turned in to finalize this assignment and the expected content of the report.

2.2 Definition and Goals

In the first assignment, a parametric description of a VLIW processor is provided and each group gets assigned two workload applications taken from the Powerstone benchmark suite. The goal is to perform a Design Space Exploration (DSE) via the VEX toolchain at the simulation level (a multitude of processor parameters can be configured - see Section 2.7) and find a processor configuration which gives the best performance for both applications, while not compromising too much the resource utilization. A maximum 4-page report has to be written, whose expected content is described in Section 2.9.

2.3 Explanation of Tools

2.3.1 Virtual Machine & Initial Setup

For the assignments, we have created a Virtual Machine running OpenSUSE that can be downloaded and run using VMWare or Virtualbox. Username and password are both `user`. The root account also has password `user`. Before proceeding to the assignments, the following are required:

- Running **Set group nr. and receive benchmarks** shortcut on the VM desktop. This script will configure the virtual machine for your group number and select the benchmarks that you will be using.
- **TU Delft VPN connection** For Assignment 2 a VPN connection to TU Delft is required when working outside campus, in order to access the TU Delft licenses necessary for running simulation and synthesis. For information on how to setup a VPN connection please follow `https://www.tudelft.nl/en/student/ict/ict-facilities/virtual-private-network-vpn/`. Note that for Assignment 1 such a VPN connection is not required.

For additional software installation in the Virtual Machine, YaST (GUI) or `zypper` (command line) can be used.

2.3.2 VEX Compiler

The compiler can target a range of different machine organizations and it can be configured by supplying a machine configuration file (`.mm`). 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.

2.3.3 VEX Simulation System

The VEX compiler receives as input a C application and the processor configuration files, and generates as output the assembly code according to the VEX instruction set architecture description. Then, the VEX architectural simulator converts the VEX assembly code to native binary running on the host VM. This is a standard procedure for not yet completely defined processor architectures, which allows one to evaluate the performance of a certain processor instance when executing a given application, without actually implementing the new processor. 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`.

2.4 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 receive benchmarks' command on the desktop to see which programs are assigned to your group. In the remainder of this section we describe the tool operation while making use of the `fir` application as discussion vehicle.

For your convenience, a command that allows you to easily compile and run VEX simulations was created. 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-4-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 fir -O3` and press enter. The `-O3` is a flag passed to the VEX compiler, which selects optimization level 3. After a successful execution of the benchmark, `fir: success` is displayed in the terminal after some compiler warnings, and the directory `output-fir.c` is created. 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 fir` 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.
- `fir.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.
- `fir.cs.c`: this is the C source code for `a.out`, generated by the VEX compiler to simulate `fir.c` with the given architecture. It is simply compiled with `gcc`.
- `fir.s`: this is the VEX assembly file. 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 Assignment 1 scope, 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, is beyond the scope of Assignment 1.
- `static_prof_file.txt` and `dyn_prof_file.txt`: count of instructions composing the benchmark.
- `vex.cfg`: this is a symlink to the cache configuration file. There is no need to do anything with this file.

2.5 Analyzing Simulation Results

As mentioned in the file description, the most important metric (together with your estimation of the area utilization corresponding to 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.

For example, the files `static_prof_file.txt` and `dyn_prof_file.txt` relate to static and dynamic profiling in terms of instructions count. The VEX toolchain has also a number of tools available for profiling. These tools can analyze call graphs, control flow graphs, instruction schedules, and visualize them. You are advised to read the `vex.pdf` documentation (go to Documentation in the file manager) and use it to your advantage!

One such profiling utility is `gprof` which displays the execution timing profile for the benchmark call graph. To interpret the `gprof` performance log files (`gmon-nocache.out`), run `gprof a.out gmon-nocache.out > gmon-nocache.txt`. To visualize the profiling information, the VEX distribution includes the `rgg`

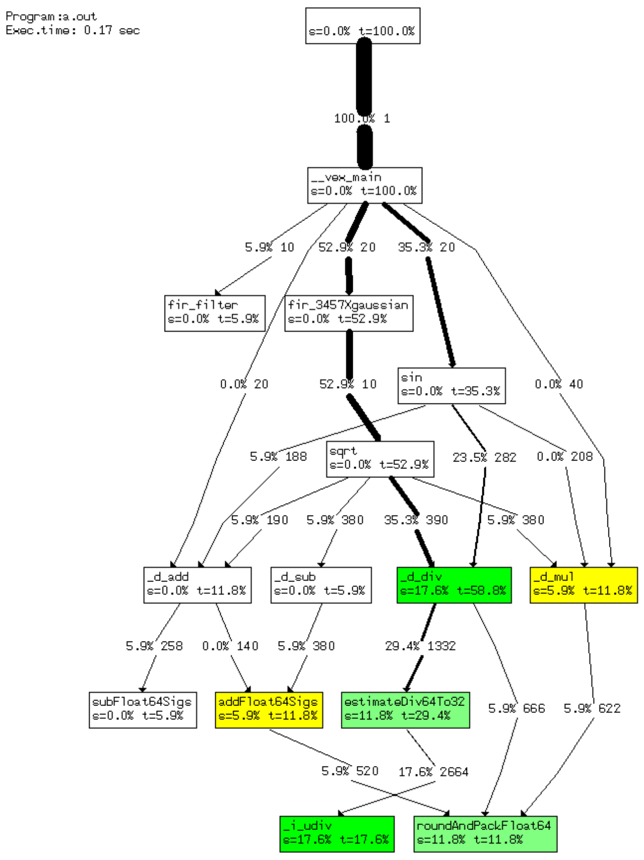


Figure 2.1: Profiling call graph.

utility that converts the standard gprof output file (`gmon-nocache.out`) into a graph representation. Run `rgg a.out -g gmon-noncache.out` in the simulation output directory to get the visualization result as presented in Figure 2.1. Nodes are color-coded for importance and labeled with the total execution times for the corresponding function ($t[\%]$) which consist of the execution times for themselves ($s[\%]$) and for the descendent subgraph ($(t-s)[\%]$). Edges are labeled with execution counts, with % execution time, and have a thickness based on their importance. For example, the `_estimateDiv64To32` function is called 1332 times by `_d_div` and amounts to 29.4% of the benchmark total execution time (11.8% in the function itself and $29.4\% - 11.8\% = 17.6\%$ for the subgraph `_i_udiv`).

Graph specific compilation flags (e.g., `-fdraw-dag`) can enable the generation of VEX operation dependence graphs, which can be later graphically vizualized using the `vcg` tool. Figure 2.2 shows an example of such a graph.

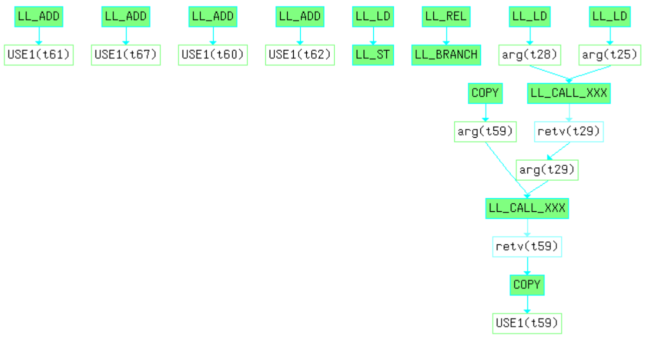
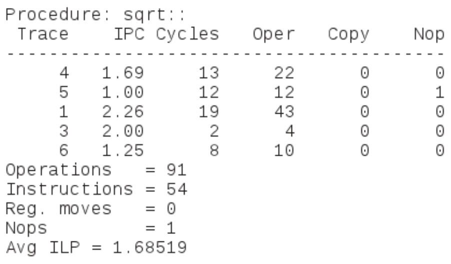


Figure 2.2: Operations dependence graph.

Another VEX profiling utility is `pcntl`. It can be invoked as `pcntl fir.s` from the simulation output directory. As a result, it will produce the output in Figure 2.3. As we can see, this utility analyzes information for every function



```
Procedure: sqrt::
 Trace     IPC Cycles    Oper    Copy     Nop
------------------------------------------------
     4    1.69     13      22       0       0
     5    1.00     12      12       0       1
     1    2.26     19      43       0       0
     3    2.00      2       4       0       0
     6    1.25      8      10       0       0
Operations    = 91
Instructions = 54
Reg. moves    = 0
Nops          = 1
Avg ILP = 1.68519
```

Figure 2.3: `pcntl` output log segment.

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

Table 2.1: Area estimates for default VEX architecture.

A_{1_ALU}	A_{1_Mult}	$A_{1\_\text{LW/SW}}$	A_{64_GPR}	A_{8_BR}	A_{misc}	$A_{1\_\text{Connection}}$
3273	40614	1500	26388	258	6739	1000

- 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.

2.6 Resource Utilization

As stated, a reasonable compromise between performance and area utilization must be reached, but the tools do not give any area numbers. This makes sense, because at this point in the design-space exploration process there is no processor implementation yet! Instead, an area estimation model has to be devised, based on the resources employed in the machine configuration file. This model gives a result (a number) which will be approximately proportional to the area of the implemented processor.

In Table 2.1 we present area estimates for the FPGA mapped default VEX configuration (issue width = 4), which consists of four ALU units, two 16×32 Multiply units, one Load/Store unit, 8 1-bit Branch Registers (BR) and 64 32-bit General Purpose Register (GPR), and *misc*, i.e., the remaining logic and interconnects, which can be assumed to be constant in Assignment 1 and not included in the area estimation model. $A_{1\_\text{Connection}}$ is the estimated area for one 32-bit connection to data cache. The subscript “1” in A_{1_ALU}, A_{1_Mult} and $A_{1\_\text{LW/SW}}$ means one functional unit.

We note that, the register file is a major contributor to the area utilization. This is because a VLIW with many functional resources also needs 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), as stated in Equation 2.1. 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, as suggested in Equation 2.2.

$$A_{\text{REGS}} \sim No_{\text{regs}} * (\#\text{read ports}) * (\#\text{write ports}). \tag{2.1}$$

$$A_{\text{REGS}} \sim No_{\text{regs}} * 2 * (\text{IssueWidth})^2 . \tag{2.2}$$

For the GPR area (A_{GPR}) estimation, we have to consider the issue width and the number of registers that we set in the configuration file. Due to the fact that the register file area is proportional to the number of registers $\times$ the square of the issue width, the area estimation model computes the GPR area as follows:

$$A_{\mathrm{GR}} = \frac{A_{64\_\mathrm{GR}}}{64} * (\#\mathrm{regs}) * \left(\frac{\mathrm{IssueWidth}}{4}\right)^2. \tag{2.3}$$

The BR area should be computed in a similar way as the GPR area.

The cache area (Data Cache and Instruction Cache) is also a major factor for the area estimation model. A 32 KB cache, which is the VEX default configuration requires about 568882 area units. The cache area is dominating the processor area, thus including it in the area estimation model, would make any changes of the user-defined processor insignificant. So, for the purpose of Assignment 1, where the focus is on optimizing the processor configuration, we disregard the cache area from the total area model.

When increasing the number of connections to the data cache, it means (1) the number of ports to the data cache and the cache complexity (multi-port cache) will increase, and (2) the number of outputs and inputs to the lw/sw functional unit and its associated additional logic will increase. Increasing the number of 32-bit connections will be accounted for, only in (2).

Therefore, the total area estimation model for Assignment 1 at the soft simulation level can be described as follows:

$$\begin{aligned} A =& \#\mathrm{ALU} * A_{1\_\mathrm{ALU}} + \#\mathrm{Mult} * A_{1\_\mathrm{Mult}} + \#\mathrm{LW/SW} * A_{1\_\mathrm{LW/SW}} + A_{\mathrm{GR}} + A_{\mathrm{BR}} \\ &+ \#(\mathrm{Connections\ to\ data\ cache}) * A_{1\_\mathrm{Connection}}. \end{aligned} \tag{2.4}$$

2.7 Changing the Machine Configuration

The goal of this assignment is to optimize the processor configuration 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. One machine configuration file is given as an example: a basic 4-way VLIW. Making a copy of the configuration directory before changing its content is more than advisable.

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 configuration 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 (see the first ET4074 lecture). 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.

Note that the delay (`DEL`) values should not be changed; they match the ρ-VEX implementation you will be using in Assignment 2.

2.8 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. Adding `#pragma` statements to the benchmark C files to give more fine-grained information to the compiler is allowed, as long as the actual benchmark calculation code is not changed. Doing these things is optional.

2.9 What to Turn In

In order to successfully complete this assignment, the following need to be turned in:

- An archive with the machine configuration file and optionally the source files of the benchamrks (if you changed them); and
- A report.

The naming convention is `ET4074_2019_A1_report_group#.pdf` for the report and `ET4074_2019_A1_src_group#.zip` for the source files, where # is your group number.

The report should not be longer than 4 pages and should contain at least the following:

- 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);
- The profiling analysis.
- 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 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 the design which you think is the most well-balanced and explain why. The VEX toolchain, `static_prof_file.txt` and `dyn_prof_file.txt`, and the `ta.log` file give a lot of information. Note that part of the assignment is to demonstrate your ability to identify what information is important and to base your design decisions on it. So do not produce graphs of all the numbers you can find, but only present the data that you think is the most relevant!

Chapter 3

Assignment 2

3.1 Overview

This chapter describes the second part of the lab as follows:

- Section 3.2 presents the goals for this Assignment.
- Section 3.3 introduces the platform that will be used.
- Section 3.4 describes how to configure the hardware and what the generated directory structure looks like.
- Section 3.5 describes how to configure and modify the C source code for your platform.
- Section 3.6 describes the commands that are at your disposal for running your design using Modelsim simulation and the boardserver.
- Section 3.7 describes how to interpret the results that are generated by synthesis and the boardserver.
- Section 3.8 describes what needs to be turned in at the end of this Assignment.

3.2 Goals and Definitions

In the first part of the lab, you learned how to analyze a program and select well-balanced VEX processor design parameters for it. In the second part, an actual SoC design with one or more ρ-VEX cores will be used to run a small workload consisting of 4 Powerstone benchmarks, 2 of which are the benchmarks used in Assignment 1. Your task is to create an 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 2 new benchmarks in the same way as was done for Assignment 1, in order to determine what kind of processor would execute them efficiently. To this end, the workspace for Assignment 1 can be used. The goal is now to run the benchmarks on a real ρ-VEX core (implemented on an FPGA board). The ρ-VEX core does not have quite as much configuration options as the VEX compiler can handle. In fact, the only things that can be varied 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 need to be used 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. Results about your design in terms of performance (delay), area (resource usage on the FPGA), and energy (measured on the FPGA) will be generated. 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 (e.g., the least area and power per performance unit) design. Plot your results of these 3 designs in the 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.

3.3 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 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

[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.

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 Assignment 2. 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` for more details about the ρ-VEX processor.

3.4 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.A commented example configuration file for a platform with a single, basic 4-way ρ-VEX processor can be found in the `configurations/example` folder within the `Assignment 2` workspace. Like in Assignment 1, you can modify the file in-place, or copy the `example` folder for as many configurations as desired.

When having 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 if you 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 3.5.
- `results`: this folder will contain the results as they are generated. More information about the results can be found in Section 3.7.
- `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.

3.5 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 executed on 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 the benchmarks in the `main-core*.c` files are called.

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 `fir.c` as an example, this would cause the `main()` function defined therein to be known as `fir_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 removing 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, communication is needed (for elaborate reconfiguration schemes for instance), you can use the `struct` defined in `intercore.h`.

3.5.1 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. For 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, both benchmarks could be mapped 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.

For 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.

3.6 Running Your Design

When having finished configuring your design and you would like 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:

- `make compile`: compiles your software.
- `make sim`: simulates the FPGA platform with the current software in Modelsim. This command requires to be connected to the campus network or use the VPN connection. Modelsim simulation is described in more detail in Section 3.6.1.
- `make synth`: synthesizes the design to allow to run it on the FPGA using the boardserver. This command requires to be connected to the campus network or use the VPN connection. 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 to be connected to the campus network or use the VPN connection. This will run `make compile` first if the source files have been changed. If you have not synthesized yet, the command will ask if you want to do that (and cancel if not). The functionality of the boardserver is described in more detail in Section 3.6.2.

- `make clean`: deletes all intermediate files. If you get 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 there are 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 3.6.1). This is a relatively accurate simulation that simulates your entire platform, albeit with a simplified model for the DDR memory, so the numbers obtained 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 the input size can relatively easily decrease, which allows to simulate them faster.
- The boardserver (Section 3.6.2). This will run your complete design on actual hardware, to provide the results that need to be reported. 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.

3.6.1 Modelsim Simulation

To simulate your design using Modelsim, run `make sim` in the directory that contains the `configuration.rvex` file. If not present on the TU Delft campus, the VPN connection should be first enabled. 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 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 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.

3.6.2 Using the Boardserver

The final performance results should be obtained by running your design on the boardserver. If not present on the TU Delft campus, the VPN connection should be first enabled. The design can then be sent 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 the workload if it is out of date. Next, if the design was not synthesized yet, it will 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 queuing 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.

3.6.3 Debugging

To allow debugging 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 "hheellloĺo" 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 running your software on the boardserver, separate log file for each core will be generated, so then this is not an issue.

The functions that can be used 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 performing the 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, or write your own.

3.7 Results

The `make synth` and `make run` commands provide quite a lot of numbers and other results that give 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 4 categories: correctness, performance, area, and energy.

3.7.1 Correctness

The first thing that needs to be verified 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 expected, own debug prints may be added to the code.

Unfortunately, the HP VEX compiler is not bug-free. Some benchmark-configuration combinations will cause the benchmarks to report failure. In such cases the failure can be ignored (it will not detract from your grade).

3.7.2 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. These numbers may be used 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.

Note 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.

3.7.3 Area

The synthesis command provides 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 would be obtained for an ASIC. For the report, the following should be considered:

- the number of occupied slices,
- 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).

Table 3.1: Area estimates for individual FPGA components in terms of A_{1_CLB}.

A_{1_Slice}	A_{1_RAMB36E1}	A_{1_RAMB18E1}	A_{1_DSP48E1}
0.5	2.4	1.2	0.7

The boardserver uses a Virtex 6 FPGA (XC6VLX240T-1FFG1156), and all area estimates will be expressed in terms of the area of one Configurable Logic Block (CLB) (A_{1_CLB}). Table 3.1 summarizes the area of individual FPGA components. The total area for Assignment 2 can thus be estimated by using:

$$\begin{aligned} A =&(\#\text{Occupied slices}) * A_{1\_\text{Slice}} + (\#\text{RAMB36E1}) * A_{1\_\text{RAMB36E1}} + \\ &(\#\text{RAMB18E1}) * A_{1\_\text{RAMB18E1}} + (\#\text{DSP48E1}) * A_{1\_\text{DSP48E1}} \end{aligned} \tag{3.1}$$

For example, if the area estimates for an SoC design are: 7649 occupied slices, 8 RAMB36E1, 42 RAMB18E1 and 8 DSP48E1, then the total area is estimated as $3899.7 \cdot A_{1\_\text{CLB}}$.

3.7.4 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. One 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 your design-space has been narrowed down to only a few designs and software configurations, you may want to run these setups several times to get better averages.

3.8 What to Turn In

In order to successfully complete this Assignment the following need to be delivered:

- Your report.

- The archive generated by means of the `make pack` command (Section 3.6) for your best design. Discuss in the report why it is the best - is it the fastest or most efficient one?

The naming convention is `ET4074_2019_A2_report_group#.pdf` for the report and `ET4074_2019_A2_src_group#.tgz` for the source files.

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 evaluated ones and in which 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 (e.g., least area and power per performance unit).