Application-specific Custom Operations For A VLIW Soft-Core

Alessandro Lo-Presti (0795648)
alessandro@lopresti.nl

October 23, 2011

THESIS
Program '*Technical Informatics*'
Subclass of Computer Engineering
Rotterdam University of Applied Sciences

Abstract

This thesis describes the design and implementation of a custom CPU operation used to accelerate software execution. The Delft reconfigurable and parameterizable VLIW soft-core processor, ρ-VEX, once supported custom CPU operations but support for this feature has faded in recent versions. The desire grew to restore this feature, create a methodology for using custom CPU operations and show that implementing such operations in ρ-VEX is beneficial to improving execution time of an application. A target application was chosen, analyzed and a candidate set of multiple operations was identified as a multiply-accumulate operation. The ρ-VEX processor architecture was modified to allow a third operand to be spatially forwarded from a parallel issue lane. The custom multiply-accumulate operation was implemented in the ρ-VEX processor, toolchain and the target application. Simulation results show a performance improvement of approximately 1.21% was achieved.

Introduction

Custom operations have traditionally been used in commercial complex instruction set computer (CISC) and digital signal processors (DSP) to accommodate operations that consist of more complexity; operations such as floating point or complex arithmetic operations. By extending the instruction set with a more complex operation, fewer operations are needed to achieve the same result leading to a more compact set of operations in an application with potentially faster execution time as a result.

The Delft reconfigurable and parameterized very long instruction word (VLIW) soft-core processor called ρ-VEX[1] was built by the Delft University of Technology (TU Delft). Its instruction set architecture (ISA) is based on HP's VEX ISA. The processor was designed to exploit instruction-level parallelism (ILP) in an application and make its execution faster compared to a reduced instruction set computer (RISC) processor system.[1] Many architectural and organizational features of this processor have been parameterized so that it can be customized to a specific application. One added feature to this processor was the ability to embed custom operations to further optimize the execution time of a target application. However, this feature has not been used extensively and has faded away in recent versions due to active development and ever changing architectural designs.

The desire grew to restore support for custom CPU operations, create a methodology for using custom CPU operations and show that implementing such operations in ρ-VEX is beneficial to improving execution time of a larger application. This means an application will have to be chosen to port[2] to ρ-VEX. Analysis will be carried out to select a candidate set of frequently used operations to implement as one or more complex custom operations. The toolchain and ρ-VEX will need to be modified to support these new custom operations. Finally, results will have to show that optimizing through custom operations directly leads to an improvement in execution time for the selected application.

Related Work

Improving software execution performance can be achieved through many optimization methods ranging from specific compiler switches to

[1]Pronounced "rho-vex"

[2]Oxford dictionary (oxforddictionaries.com) defines "port" (*verb*) as:"*Computing* transfer (software) from one system or machine to another"

modifying/in-lining assembly code. However, a point will eventually be reached when it will be difficult to find new ways to optimize software through the use of these techniques. One way to then further optimize software execution performance is by delegating a set of multiple frequently used hardware operations into less (usually one) complex custom operations.

Accommodating complex operations in hardware to improve execution performance has been debated in complex instruction set computer (CISC) vs reduced instruction set computer (RISC) debates for years[2, 3]. Several examples of complex operations can be found in commercial processors: the `PMADDWD` and `MOVQ` instructions in the Intel MMX instruction set[5, 6] and `LDMXCSR` and `SQRTPS`/`SQRTSS` instructions in the Intel SSE instruction set[6]. Moreover, several instruction-set extensible processors have been created[7, 8, 10, 11] that allow custom operations to be added to the instruction set architecture.

In the case of ρ-VEX, custom operations were introduced in [11] together with a proof-of-concept application that calculates the 45th Fibonacci number using two custom operations. In recent versions of ρ-VEX, however, support for custom operations has been lost. This thesis aims at re-enabling support for custom operations in ρ-VEX and create a methodology for using this type of operation so that it can be implemented in future applications. A target application will be chosen, ported and a custom operation will be implemented to show the benefit of this type of operation regarding execution performance.

ρ-VEX

To understand ρ-VEX it is important to first look at instruction-level parallelism (ILP), very long instruction word (VLIW) and HP's VEX system. These are the techniques and architectures on which ρ-VEX is based and are therefore worth investigating.

ILP and VLIW

Instruction-level parallelism (ILP) is a measurement of how many operations can be scheduled in the same instruction (and thus executed in parallel) within a single stream of instructions. Because ILP is an architectural technique, it is independent on changes in technology, like circuit speed.[11] ILP is sometimes compared, or confused, with other types of processor parallelism such as: vector processing, multi-processing, multi-threading or micro-SIMD.[11]

Very long instruction word (VLIW) is an architecture design philosophy; an informal set of guidelines and principles.[4] One key aspect of VLIW is that a complex compiler exposes the available ILP in such a manner that it can be exploited using simple hardware. VLIW is sometimes compared to superscalar architecture[11, 4] because both are quite similar in their quest to exploit ILP. However, the most important difference lies in the fact that superscalar architectures have delegated scheduling to complex hardware, and therefore has a less complex compiler. In VLIW, this scheduling is done at compile time by a complex compiler so as to reduce the complexity of the hardware. Using this complex compiler, multiple operations will be scheduled into one instruction, thereby accelerating applications beyond conventional reduced instruction set computer (RISC) processors.[1]

The type of parallelism in VLIW is inherently different to multi-core processors. Multi-core processors consist of multiples of the same core that each run a separate program, or parts of a program, whereas VLIW processors have multiple functional units in one core which lie in parallel.

HP VEX

VEX stands for "VLIW example", it is a system that is developed by HP based on the VLIW design philosophy. The VEX system consists of three components[4]:

1. *The VEX instruction set architecture (ISA)* that is loosely modeled on the Lx/ST200 ISA by HP and STMicroelectronics. It is a 32-bit clustered VLIW ISA in which several parameters can be modified, such as the number of clusters, execution units, registers and latencies.
2. *The VEX C compiler* that is derived from the Lx/ST200 C compiler. It is a C89 compiler that uses *trace scheduling* as its global scheduling engine. A flexible machine file is used to describe the target machine for which executables are created, in which several parameters can be adjusted without having to recompile the compiler. Some of the parameters that can be adjusted are: issue width, number of clusters, execution units and operation latencies.

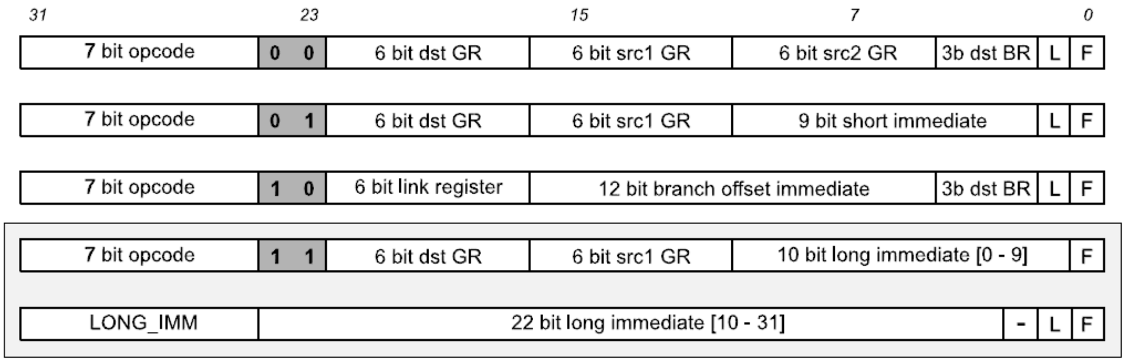

Figure 1: Syllable layout

3. *The VEX simulation system* is an architecture-level cycle accurate (functional) simulator. It comes with a POSIX-like *libc* and *libm* library set (based on the GNU *newlib* libraries), a built-in level-1 cache simulator and an application programming interface (API) that enables other plug-ins used for modeling the memory system.

For this project, a machine configuration file was provided[3]. The machine configuration file specifies a 4-issue machine with forwarding that consists of several set parameters such as: four ALUs, two multiplication units, and one load/store unit for each issue slot; 62 general registers; 8 branch registers.

ρ-*VEX*

ρ-VEX is a reconfigurable (at design-time) and extensible VLIW soft-core processor and was originally designed and implemented by Thijs van As[11] of the Delft University of Technology. It was built in order to achieve higher performance within the application-specific domain by combining several technologically proven paradigms to provide a new architectural solution for a general-purpose computing machine.[11] The creation of ρ-VEX acted as a foundation to facilitate future scientific research, that is currently still ongoing, with students of the Delft University of Technology working on projects such as: Level-1 (L1) caching, a network on chip (NoC) implementation of ρ-VEX and a runtime reconfigurable issue width.

The design of ρ-VEX is based on Harvard architecture which describes physically separate instruction- and data-memory. A 4-issue ρ-VEX processor is depicted in figure 2. The ρ-VEX ISA is based on the VEX ISA and exhibits the same characteristics with regard to parameterization. Several of the parameters can be modified (at design-time), such as: the number of functional units, the amount of general purpose and branch registers, number and type of available functional units per syllable.

The ρ-VEX implementation by Thijs van As supported so-called "ρ-OPS" which were a way of embedding custom operations into ρ-VEX as extensions to the ALU. However, since the implementation of ρ-VEX by Thijs van As, a lot of improvements have been made to the architecture such as: dynamically adjustable issue slots[12], dynamically reconfigurable register File[13] and the addition of "pipe lanes"[14] (i.e., issue lanes). Pipe-lanes now accommodate the decode, execute and writeback stages of the processor. Because of these architectural changes, the way ρ-OPS describes embedding custom operations no longer stands and new work will need to be done to embed custom operations in the ρ-VEX architecture.

The instruction layout (figure 1) consists of four 32-bit syllables[4] to create one very long instruction

[3]The machine configuration file (Appendix A) was provided to me by the project stakeholder at the Delft University of Technology, Ir. Fakhar Anjam

[4]For sake of consistency, the same naming conventions for *operation*, *instruction* and *syllable* are used as in [11]:

- An **operation** is defined as an atomic command for the processor to be executed, e.g. the addition of two operands.
- An **instruction** is defined as the data fetched from the instruction memory, in which a number of operations is defined together with their operands and destinations.
- A **syllable** is defined as a combination of a single operation together with its operands and destination. A syllable is the same as an instruction in RISC machines.

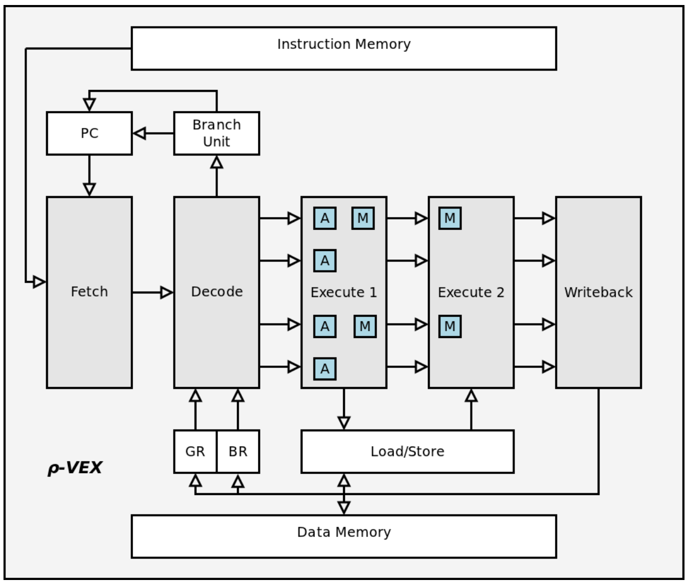

Figure 2: 4-issue ρ-VEX VLIW Processor

word of 128 bits. VEX operands can be general-purpose registers (GR), branch registers (BR), link registers (LR), or immediates (constants). The three types of immediate are: *short immediate* operands, *branch offset immediate* operands and *long immediate* operands.[4] Every syllable has an immediate switch field consisting of 2 bits that describe the type of immediate operand within the syllable. ρ-VEX syllables include two bits with syllable meta-data, the L and F bits. The L bit denotes whether a syllable is the last syllable in an instruction and the F bit denotes whether it is the first syllable in an instruction.[11]

Development Environment

The development environment consists of several components: the toolchain for building VEX and ρ-VEX executable binary files, and several simulators used to test these binaries.

Toolchain

The VEX C compiler, derived from the Lx/ST200 C compiler, is a C89 compiler that uses trace scheduling to create a VEX simulation system binary. The VEX compiler is used to generate assembly source code files for ρ-VEX using a supplied machine description file which describes micro-architectural characteristics such as issue width, latency and resources. The assembly files are assembled and linked using a toolchain —based on GNU Binutils— that was developed by Ir. Anthony Brandon at the Delft University, to form a binary executable file. This resulting binary is then run through an `elf2vhdl` tool from the toolchain to create several output files, including a test bench in the hardware description language VHDL that can be used for simulation and synthesis. Inside this test bench, the binary is embedded as data- and instruction memory.

The ρ-VEX soft-core is implemented in the hardware description language VHDL and synthesis is performed with XST from the Xilinx ISE suite for the ViRTEX-6 field-programmable gate array (FPGA) ML605 kit.

Simulators

The VEX simulation system is a simulator containing the program that was compiled by the VEX compiler, complete with a set of POSIX-like libraries. When the VEX simulator is run, log files are generated containing several run-time data that can be used to analyze various aspects of execution.

A flexible simulator called xSTsim by STMicroelectronics is used to simulate ρ-VEX binaries before simulating it on the ρ-VEX soft-core. This simulator has a lot of built-in features useful for debugging, such as: tracing debugging, dumping of memory content and resizing of available memory.

Modelsim SE 10.0b from Mentor Graphics is used to simulate the ρ-VEX soft-core and the supplied test bench which houses a ρ-VEX executable binary.

Applications

A software application had to be chosen to port to ρ-VEX. A list of several pre-selected[5] applications was provided[6] to pick from. This list of applications consists mostly of digital signal processing (DSP) benchmark applications, all suitable candidates for optimization through extension of the ISA because they will all be issuing multiple operations while processing input data. Thus ILP count should be sufficiently high with room for optimization by introducing a custom, more complex, operation.

Having assessed the complexity of the listed applications, with regard to attainability within the required time frame, the least complex application called *SUSAN* was chosen to port to ρ-VEX.

SUSAN

SUSAN[15] is a low-level image processing application. It is a multi-platform self-contained command-line C application consisting of three main algorithms: noise reduction filtering (i.e., 'smoothing'), edge detection and corner detection.

Functionality

SUSAN reads an input portable graymap format (PGM) image file and will write a resulting PGM file. Additionally, several options and parameters can be supplied to SUSAN. The main parameters that have been evaluated are: the three modes (edge detection, corner detection and smoothing), a fixed brightness threshold of 10 for all three modes and the default distance threshold of 4 for smoothing mode. These values were chosen based on the test results as shown in [16] which were used as references. The images used to test SUSAN are the supplied image in [17] for edge and corner detection and the image of the human brain on [18] for smoothing.

Analysis

In order to improve performance of SUSAN, several operations can be embedded into a more complex custom CPU operation. The aim is to analyze the execution flow of SUSAN and find candidate custom operations. When using the HP VEX simulator, reports logs are generated (see appendix C) showing various run-time data. The most important data to analyze are: the total amount of cycles taken, execution time and a profile of functions ordered by the amount of time it took to execute those functions.

Upon examining the edge and corner detection mode functions, it became apparent that the majority of repeated operations were of the form:

```
n += *(cp - *p++);
```

This C code is scheduled by the VEX compiler into three separate instructions (in order): a load, a subtraction and a final load. It is also important to remember that a load requires two cycles to finish, thus it takes 5 cycles for these operations to become finalized. The possibility of combining the first load and the subsequent subtraction as one custom operation was investigated. However, this candidate was quickly discarded after finding out it is not possible to implement due to the limitations of custom operation support in the VEX compiler[20] which states that custom operations are limited to operations that do not access memory.

Examination of the generated log file for edge detection mode function showed that the most time-consuming function `_i_udiv` comprises 24.09% of the total execution time. This function is part of a division library (`divrem.c`) as part of the HP VEX library set. Analyzing the execution flow of this function showed that there was one particular `while`-loop that, when called from SUSAN's edge detection function, is executed 811.332 times:

```
while ( den < num &&
        bit &&
        !(den & (1L << 31)) ) {
  den <<= 1;
  bit <<= 1;
}
```

This while condition is scheduled by the VEX compiler into six operations and can be implemented as one custom operation. The result of this would yield a decrease of the number of instructions within this loop (unrolled four times by the compiler) from 10 to 8 instructions. However, when this was discussed[7] it was revealed that there is an inline option for integer divisions that can be passed to the compiler that will calculate integer divisions in 35 cycles, which is 2 cycles faster than calling the `_i_udiv` library function. Even with a custom operation, there would be no significant increase over inlining integer divisions, so this candidate was therefore discarded.

[5]Pre-selected by the ERA project[9]

[6]The list of applications (Appendix B) was provided to me by the project stakeholder at the Delft University of Technology, Ir. Fakhar Anjam

[7]With Ir. Anthony Brandon of the Delft University of Technology

Examination of the generated log file for the smoothing mode function showed that the most time-consuming function `susan_smoothing` comprises 69.97% of the total execution time. Inspecting this function revealed a quadruple nested `for`-loop in which the inner most code block is executed 1.237.500 times and contains the following line:

```
total += tmp * brightness
```

This is a classic example of a potential optimization candidate known as a "multiply-accumulate" in which the multiplication and addition are combined into one operation. Currently, this code is scheduled into four separate operations: multiplication of high 16-bits, multiplication of low 16-bits, the accumulation of those results and the final addition. Introducing a multiply-accumulate operation (combining one of the multiplications with the addition) could improve the overall execution speed. This was confirmed by a quick modification in SUSAN whereby the multiply-accumulate was introduced and an assembly file was created by the VEX compiler. This showed a decrease in the amount of scheduled instructions. For this reason, this multiply-accumulate was chosen to implement in ρ-VEX.

Porting

SUSAN is required to run on ρ-VEX before a custom operation can be introduced. However, ρ-VEX does not have access to a filesystem, can not print text and there currently exists no floating point unit for ρ-VEX. Changes need to be made to SUSAN before it is able to run on ρ-VEX.

The first simple change was to strip SUSAN of all `printf` function calls. After that, relevant libraries were compiled from the VEX library set and assembled to ρ-VEX object files, namely: `floatlib` (based on SoftFloat[21]), `memcmp` and `ef_exp`. The memory allocation library, for the `malloc`, `free` and `memset` function calls, proved difficult to port. However, since these functions were only called 9, 2 and 7 times respectively, it was manageable to convert these allocated memory regions to static arrays. For the function call to `memcpy`, one function was taken from the implementation in an online article[22]. For the function call to `sqrt`, two functions were taken from the 2.11BSD source tree[23], namely: `sqrt` and `frexp`. For the function call to `abs`, one function was added by writing a simple absolute value function. The `exit` function was also added, since there is no existing system call in ρ-VEX to handle this function. Some Assembly logic is manually added during the process of compilation and assembling which takes the exit argument, writes it to unused register `$r0.63` and issues the ρ-VEX `STOP` operation to halt the processor. With external function dependencies taken care of, the focus changed to SUSAN itself.

The input images[17, 18] to test SUSAN were embedded as C header files into SUSAN and the functions regarding file reading and writing were removed. Finally, the parameters for running SUSAN were hard-coded into the `main` function.

Testing

After porting, the functionality of SUSAN needed to be tested to ensure functionality remained equal to the unmodified version. The input images[17, 18] to test SUSAN proved to be too large for the available stack space in xSTsim. There were also concerns that the total application size would not fit the available 1.8 MB data memory provided by the Field-Programmable Gate Array (FPGA) board on which ρ-VEX is run. For this reason, a smaller size of 100 pixels wide by 55 pixels high was cropped from the input images and used for SUSAN. Figure 3 is taken for usage in smoothing mode and figure 4 was taken for usage in edge and corner detection modes.

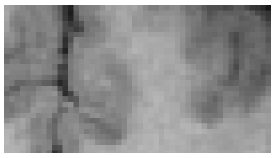
Figure 3: Input image for smoothing

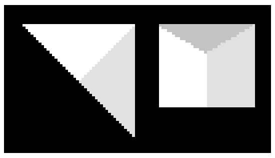
Figure 4: Input image for edge and corner detection

To ensure that the output of these input images match the output images of the unmodified version of SUSAN, these images were run through the unmodified version of SUSAN. The resulting output images (figures 5, 6 and 7) were embedded into SUSAN as C header files and used as reference images.

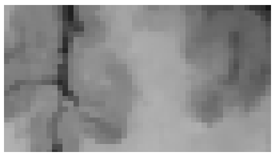
Figure 5: Output image for smoothing

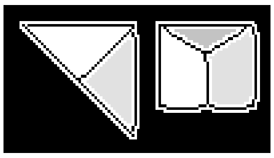

Figure 6: Output image for edge detection

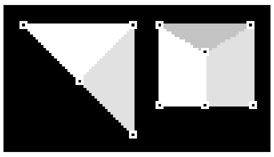

Figure 7: Output image for corner detection

A simple routine was then implemented at the end of the `main` function to check if —based on the selected mode— the modified input image matches the reference image. If they match, SUSAN will `exit` with a return value of 33 (i.e., successful execution), otherwise with a return value of 30 (i.e., unsuccessful execution).

Encountered Problems

Some preliminary changes had to be made to SUSAN that had to do with C89 coding conventions regarding assignment ambiguity. Extra spacing needed to be applied to all compound assignment operators (e.g., `a=-1` was changed to `a = -1`).

When an assembly source file was created with the VEX compiler and it was attempted to assemble, some errors arose that had to be dealt with. The VEX compiler tries to schedule what is known as *dismissible loads* but these types of loads are currently not supported by ρ-VEX. All dismissible modifiers in the assembly source can therefore be stripped from their loads (e.g., the `ldb.d` operation becomes `ldb`). Other unsupported features that arose from assembling are the pseudo-operations *.real4* and *.real8*. Support for these pseudo-operations was added to the toolchain.

An implementation issue in the VEX compiler[24] causes link register `r0.63` to sometimes be used, even when the machine configuration file explicitly states that there are only 62 registers. This could lead to problems in execution flow. Thankfully, the few instances register `r0.63` is used as a link register in SUSAN caused no problems, so a simple search and replace removes all references to register `r0.63`.

There was some concern whether or not symbolic labels in multiple object files were correctly referenced by the compiler when jumping to those labels, since the same symbolic label names were used in different object files. For this reason, a simple test application was created to test whether or not this would occur. Analysis showed that there were no problems referencing the same label name in two object files.

When SUSAN was just ported and testing began using the HP VEX, xSTsim and Modelsim simulators, it seemed that SUSAN was functioning well in HP VEX but in the other simulators it was getting stuck in an endless loop[19]. After debugging the application by injecting `exit` routines at various places, to see which points in the application were reached, the problem was narrowed down to a floating point division. The floating point division was extracted to a test application to make it easier to debug the problem. After looking at the routines that were called in floatlib, there did not appear to be any problems. After more than a week of debugging, the problem was finally discussed[8] and it became apparent that the problem might be caused by endianness. HP's VEX —including its library set— is big endian, whereas xSTsim and ρ-VEX are little endian. To try and solve the problem, SoftFloat[21] (where floatlib of HP VEX was derived from) was downloaded and floatlib was modified, using SoftFloat as reference, to make it little endian. However, that still did not solve the problem. The decision was made to port SoftFloat to ρ-VEX and call its routines directly. However, porting SoftFloat *still* did not solve the problem. After spending more days debugging, a meeting[8] was called to have a look at this problem. Extensive debugging then finally identified the problem to a faulty implementation of `DIVS` and `ADDCG` in the toolchain and ρ-VEX. After patching these implementations further testing could continue.

When smoothing mode in SUSAN was tested using the xSTsim simulator, the resulting input image did not match the reference image. After days of debugging and some discussion[8], one problem was finally identified to reside in the application loader `_start.s` that was used to set up the stack size. This loader still had some legacy code that was no longer required, so it was removed. However, this still did not give a correct result. More days were spent debugging when finally a problem was discovered in the implementation of a `double` exponent function that was taken from 2.11BSD sources. The exact problem was identified[8] in the fact that double precision (i.e., 64-bit) numbers were not ordered correctly by the compiler, due to its intrinsic big endian nature. After inspecting the calls that were made to the exponent function, it was quickly established that double precision was never required and float-

[8]With Ir. Anthony Brandon of the Delft University of Technology

ing point numbers would suffice. With this in mind, the double precision exponent function was removed and substituted by the floating point `ef_exp` function of the VEX library set.

At this point SUSAN was running successfully in the VEX and xSTsim simulators. However, when running in Modelsim's simulator the modified input image never matched the reference image. Since xSTsim and Modelsim's simulator should equal in functionality, the problem was suspected to reside in either the toolchain or ρ-VEX. Having spent days debugging the problem, it was finally discussed[8]. The problem was eventually identified to be a faulty implementation of an unsigned integer `LDBU` load operation. After this implementation was patched, all modes of SUSAN were finally working successfully in all simulators.

Custom Operation

After a candidate set of operations was found as a multiply-accumulate custom operation, an implementation can be made in SUSAN. The toolchain and ρ-VEX also need to be modified to support this new operation.

Implementation

The VEX compiler already has support for dealing with custom instructions via pragmas inside the application code[20]. In order to use a custom multiply-accumulate operation, manual changes are required in SUSAN to allow the compiler to schedule a new ρ-VEX operation based on the chosen candidate:

```
total += tmp * brightness
```

Multiplication of two signed integers is scheduled by the VEX compiler into three operations: an `MPYHS`, multiplying signed operand 1 with the 16 most significant bits of signed operand 2; an `MPYLU`, multiplying unsigned operand 1 with the 16 least significant bits of unsigned operand 2; and an addition of these results. The multiplications are scheduled in parallel and will take two cycles to finalize, the addition will then be scheduled in a separate instruction. Afterwards, the extra addition is scheduled in a separate instruction.

In order for the custom operation to be scheduled with a default two-cycle latency, the assembly opcode number for at least one multiplication needs to be in region 64–127 (i.e., group 'B')[20]. The `MPYLU` operation was chosen to house the third operand used for the extra addition of the multiply-accumulate. A C-macro designed for a multiply-accumulate with three register operands —as opposed to any immediate operand— is created for easy usage:

```
#define MAC(a, b, c) \
  ( (int)_asm1(80, (a), (b), (c)) \
  + (int)_asm1(11, (a), (b)) )
```

The VEX compiler does not associate custom operation opcodes with operation names, even when using existing opcodes, so the assembly file now has operation names `asm,80` and `asm,11`. Therefore, a search and replace before assembling changes `asm,80` to `MACLU` (the operation name chosen for the multiply-accumulate) and `asm,11` to `MPYHS`.

Modification

Implementation of a custom operation is comprised of modification in two separate parts, namely the ρ-VEX soft-core and toolchain.

ρ-VEX

Modifying ρ-VEX for this new custom operation proved to be tricky, since ρ-VEX does not currently support operations with three register input operands. One solution to add support for a third operand could be by changing the instruction layout and resizing the syllables to introduce a third register operand. However, due to the large amount of extra hardware complexity that this would yield throughout the entire architecture, this would greatly affect the speed of the entire processor.

A more intricate solution was devised based on the current implementation of long immediates. The instruction layout consists of four 32-bit syllables to create one very long instruction word of 128 bits. This layout restricts a large number such as a long immediate (i.e., a 32-bit number) from fitting in one syllable and is therefore spread over two syllables. The second syllable's opcode denotes that it is an extension of the previous syllable. This way, the entire 32-bit number is passed along and pieced together inside of ρ-VEX. This same principle can be applied to the custom operation but instead of extending the bits of a large number, a third register operand is placed in a separate syllable by way of an extra operation added to the instruction. Since these operations are run in parallel, the way ρ-VEX can then access this third operand is by forwarding (spatially) the third operand to the pipe lane (i.e., issue lane) that is handling the multiply-accumulate operation. Figure 8 demonstrates how forwarding is applied.

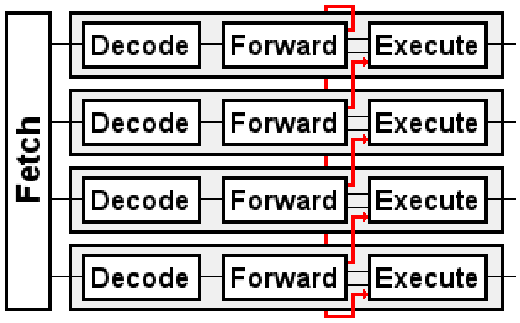

Figure 8: Pipe lanes with third operand forwarding

With a solution for a third register operand in place, the extra multiply-accumulate operation is added to the instruction set architecture of ρ-VEX and uses the first syllable layout type as depicted in figure 1.

Toolchain

Now that the custom multiply-accumulate `MACLU` operation has been implemented in SUSAN and added to ρ-VEX, the assembler needs to be modified to interpret this new operation and supply ρ-VEX with the third operand. This third operand will be supplied to ρ-VEX by an `ADD` operation and is added into the same instruction just after `MACLU`.

Testing

The custom multiply-accumulate operation was implemented in SUSAN, ρ-VEX was modified to be able to execute this new operation and the toolchain was modified to recognize the new operation and be able to assemble a binary with multiply-accumulate operations with their required third operand. All of these steps now required testing.

For the initial test a simple application was created that only executed a multiplication and accumulation. An assembly source file was created by the HP VEX compiler, assembled using the modified assembler and linked with the necessary object files to form an executable binary file. The `elf2vhdl`[9] tool then converted this binary to a `VHDL` file that can be run on ρ-VEX using Modelsim's simulator. When the simulation ended the result of the multiplication and accumulation showed an incorrect result. When analyzing different signals within the simulation the forwarding seemed to work correctly and was passing along the correct values, but still the result was incorrect. After finally discussing[10] this problem, the cause was eventually identified in the toolchain: the operations were not sorted correctly. Not all issue slots in ρ-VEX have the same execution units. The `MACLU` operation was misaligned to a slot that had no multiplication unit and was mistaken for an addition operation. After correctly aligning the `MACLU` operation this test showed that the multiply-accumulate was working well on all levels and showed a correct result in Modelsim's simulator (figure 9).

[9]`elf2vhdl` is one of many tools in the toolchain

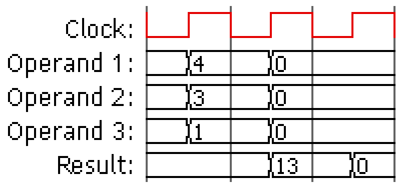

Figure 9: A successful multiply-accumulate operation based on: `(operand 1 * operand 2) + operand 3`

Results

The toolchain and architecture were modified to support a new custom multiply-accumulate operation. The target application, SUSAN, was modified to make use of this new custom operation. Analyzing the assembly file at the nested `for`-loop block, where the multiply-accumulate operation was introduced, shows a reduction of approximately 7.1% in the amount of instructions (from 30 to 28 instructions). One extra instruction is also removed just after this loop. A simulated run in Modelsim showed that without the custom multiply-accumulate operation, 9.262.996 cycles were executed before SUSAN finished. With the custom multiply-accumulate operation, a total of 8.767.996 cycles were executed. Without a change in clock frequency, the total improvement in speed is approximately 5.6%.

Synthesizing ρ-VEX without the custom operation logic resulted in a maximum attainable clock frequency of 119.12 MHz. With the increase of hardware complexity to accommodate the custom multiply-accumulate operation (see table 1), the critical path is increased resulting in a maximum attainable clock frequency of 114.12 MHz. These clock frequencies tell us[11] that the total improvement in execution speed is approximately 1.21%.

[10]With Ir. Anthony Brandon of the Delft University of Technology

[11]by calculating: ${}^{(9262996/119)}/_{(8767996/114)} \times 100$

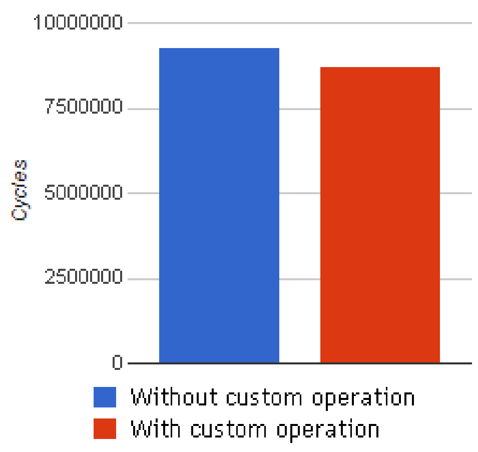

Figure 10: Modelsim simulator results

Property	Without MAC	With MAC
Slice Registers	1,081	1,109
Slice LUTs	7,673	7,798
..used as logic	6,255	6,381
..used as Memory	1,408	1,408
DSP48E1s	4	8

Table 1: Comparison of used area size

Rework and Further Experiments

In order to show that a multiply-accumulate operation can be used to improve execution time in more than one application the decision was made to port another software application, consisting of a function that would frequently make use of such an operation, to ρ-VEX.

Research was done to determine a good candidate function, a function which uses a multiply-accumulate operation frequently and is widely used in specific application domains, so as to validate its usefulness. This led to the selection of a matrix multiplication function — a function which is used in the domains of DSP, scientific calculations, numerical analysis and reportedly[12] also in aerospace.

The C-code for a matrix multiplication is simple and immediately shows a multiply-accumulate operation in the same fashion as displayed in SUSAN:

```
for (i = 0; i < 10; i++)
  for (j = 0; j < 10; j++)
    for (k = 0; k < 10; k++)
      result[i][j] += a[i][k] * b[k][j];
```

However, after implementing the C-macro `MAC()` and analyzing the generated assembly code, the amount of instructions for this loop increased — rather than decreased— from 25 instructions to 29 instructions.

After analyzing the possible cause, it seems likely that the compiler schedules the code without multiply-accumulate operation more efficiently using intrinsic optimizations. The rationale is that the compiler does not know anything about the new multiply-accumulate operation and therefore cannot schedule it as efficiently. However, this is difficult to prove, since the compiler is proprietary and its source code is not freely available to examine.

Conclusions and Recommendations

This thesis described the design and implementation of a custom multiply-accumulate CPU operation to accelerate execution time of a selected application, SUSAN. The toolchain and ρ-VEX were modified to support a custom multiply-accumulate operation. In the process, several bugs in the toolchain and ρ-VEX were uncovered and reported. One extra addition to ρ-VEX is the ability to now support operations with three input register operands. Future projects demanding custom operations and/or operations that require three input register operands, can use this version of SUSAN as a reference for a foundation on how to approach implementing such operations.

Ultimately, the resulting improvement in execution speed is rather low. There are two reasons for this:

1. The instruction-level parallelism (ILP) count is high for the specific `for`-loop block. Removing one operation will only be beneficial when the ILP is sufficiently low so that one instruction is removed entirely. Since this is not the case, in some instances only one operation of the instruction is removed (replaced by a `NOP`[13]).

2. Extra hardware complexity is added to ρ-VEX to support a third register operand for one operation. If ρ-VEX would have had support for this, extra hardware complexity would not be required, the critical path would not have increased and a higher clock frequency would have been attained, resulting in a higher increase in performance.

[12]By lector dr. Sunil Choenni and supervisor Fakhar Anjam

[13]`NOP` stands for 'No Operation'; it is an operation that does nothing

In an attempt to prove that a low ILP count would potentially lead to an improvement, a test had been devised whereby the issue width in the machine description file for the VEX compiler was adjusted to 2 operations. However, in this case, the custom multiply-accumulate operation was not supported by the compiler and extra work would have to have been carried out to support a third register operand; since two multipliers are scheduled in parallel and the extra register operand is currently implemented as a separate parallel operation.

Extra work was done to try and show that a multiply-accumulate CPU operation can be used to improve execution time in more than one application. For this a matrix multiplication function, used in multiple application domains, was ported to ρ-VEX. However, using the VEX compiler to generate assembly code showed an increase of the number of instructions, thus effectively slowing down the execution speed of the application. It seems likely that the compiler schedules the code without multiply-accumulate operation more efficiently using intrinsic optimizations. The rationale is that the compiler does not know anything about the new multiply-accumulate operation and therefore cannot schedule it as efficiently. This extra work is important because it shows that implementing a custom operation does not always lead to an improvement in execution time when the HP VEX compiler is used. Custom operations need to be customized for a specific application in order for them to truly be beneficial.

Future work on ρ-VEX can be done for better support of more than two register operands within a syllable. One proposition in which more register operands can be added to a single syllable would be to restrict the amount of bits denoting the selected registers. This way the available registers are limited, but bits in the instruction layout are freed up and can be rearranged (i.e., fewer bits are needed to encode register operands). The VEX compiler is not flexible enough to restrict registers to specific operations, but with the development of the `GCC` compiler for ρ-VEX, the flexibility required can be programmed into `GCC`. Another way to support three input register operands within a syllable, in the case of a multiply-accumulate operation, is by re-ordering the current syllable layout. Since the 'L' and 'F' bits are only used on multiple clusters and the two immediate flag bits and three branch register destination bits are not used in this case, six of these seven bits can be used to form another register operand.

Acknowledgements

First of all, I want to thank dr. Stephan Wong for offering me the opportunity to work on this graduation project and PhD student Fakhar Anjam for his supervision throughout the project. I also wish to give my utmost sincere thanks to PhD student Anthony Brandon for his continued help and support whenever I needed it, even in the late hours of the day(!). I thank Wessel Oele, Adrie van der Padt and dr. Sunil Choenni for their supervision on behalf of the Hogeschool Rotterdam. Lastly, I thank MSc student Pepijn Westen for proof reading, the many discussions we've had and the suggestions made allowing me to see things in different perspectives.

There are also some external parties that I wish to thank. Namely, I want to thank Nintendo for their Nintendo DS and Pokemon Mini handheld gaming consoles, which were influential in placing me on the path to study the world of embedded systems and were actually the deciding factor for me to pick '*Technical Informatics*' (Computer Engineering) over '*Informatics*' (Computer Science). I also want to thank Rafael Vuijk (*DarkFader*) for his work in reverse engineering the Nintendo DS which, to me, was a real inspiration and insight in to how the world of embedded systems worked and could be exploited. Lastly, I want to thank Marius Schmidt (*Lupin*) on behalf of the Pokemon Mini scene[25] for supplying me with his last remaining flash cart for the Pokemon Mini, which led to enabling me to pick up development for the device and further expand my passion for embedded handheld devices.

References

[1] S. Wong, F. Anjam, *The Delft Reconfigurable VLIW Processor*, <`http://ce.et.tudelft.nl/publicationfiles/1823_14_ADCOM2009.pdf`>.

[2] L. Borrett, *RISC versus CISC* Australian Personal Computer, June 1991, <`http://www.borrett.id.au/computing/art-1991-06-02.htm`>.

[3] T. Jamil, *RISC versus CISC*, Potentials, IEEE, vol. 14, no. 3, Aug/Sep 1995, <`http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=464688`>.

[4] J. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*, Morgan Kaufmann, 2004.

[5] M. Mittal, A. Peleg, U. Weiser, *MMX Technology Architecture Overview*, Intel Technology Journal, Q3 1997, <`ftp://download.intel.com/technology/itj/q31997/pdf/archite.pdf`>.

[6] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*, Order Number 253665-039U, May 2011, <http://download.intel.com/design/processor/manuals/253665.pdf>.

[7] Altera's Nios Embedded Processor, <http://www.altera.com/products/ip/processors/nios/>.

[8] P. Faraboschi, G. Brown, J.A.Fisher, G. Desoli, and F. Homewood, *Lx: A Technology Platform for Customizable VLIW Embedded Processing*, 2000.

[9] ERA Project, <http://www.era-project.org/>.

[10] HP VEX binary distribution, <http://www.hpl.hp.com/downloads/vex/>.

[11] T. van As, MSc Thesis, CE-MS-2008-1, *ρ-VEX: A Reconfigurable and Extensible VLIW Processor*, September 2008, <http://ce.et.tudelft.nl/publicationfiles/1565_910_thesis_tvanas.pdf>.

[12] F. Anjam, M. Nadeem, and S. Wong, *A VLIW Softcore Processor with Dynamically Adjustable Issue-slots*, <http://ce.et.tudelft.nl/publicationfiles/1896_927_>.

[13] S. Wong, F. Anjam, and F. Nadeem, *Dynamically Reconfigurable Register File for a Softcore VLIW Processor*, <http://ce.et.tudelft.nl/publicationfiles/1825_14_>.

[14] R. Seedorf, MSc Thesis, CE-MS-2010-27, *Fingerprint Verification on the VEX Processor*, June 2011.

[15] S.M. Smith and J.M. Brady, *SUSAN — A New Approach to Low Level Image Processing*, Defence Research Agency, Technical Report no. TR95SMS1, 1995, <http://users.fmrib.ox.ac.uk/~steve/susan/>.

[16] Figure 11, figure 20, S.M. Smith and J.M. Brady, *SUSAN — A New Approach to Low Level Image Processing*, Defence Research Agency, Technical Report no. TR95SMS1, 1995, p. 19, p. 31.

[17] Figure 10, S.M. Smith and J.M. Brady, *SUSAN — A New Approach to Low Level Image Processing*, Defence Research Agency, Technical Report no. TR95SMS1, Farnborough, England, 1994, p. 18, Downloadable as PGM file at <http://users.fmrib.ox.ac.uk/~steve/susan/>.

[18] Blurred image of the human brain, viewed 9 July 2011, <http://www.fmrib.ox.ac.uk/analysis/research/susan/egorig.gif>, Source url <http://www.fmrib.ox.ac.uk/analysis/research/susan>.

[19] see [26]

[20] J. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*, Morgan Kaufmann, 2004, pp. 586–588.

[21] SoftFloat library, <http://www.jhauser.us/arithmetic/SoftFloat.html>.

[22] D. Vik, *Fast memcpy in c*, viewed 14 August 2011, http://www.danielvik.com/2010/02/fast-memcpy-in-c.html

[23] 2.11BSD source code files for sqrt and frexp functions, <http://minnie.tuhs.org/cgi-bin/utree.pl?file=2.11BSD/src/usr.lib/libom/sqrt.c>, <http://minnie.tuhs.org/cgi-bin/utree.pl?file=2.11BSD/src/lib/libc/gen/frexp.c>.

[24] *Link register delay*, viewed 26 July 2011, <http://www.vliw.org/vex/viewtopic.php?f=1&t=333>.

[25] Pokemon Mini homebrew community, <http://www.pokemon-mini.net/>.

[26] see [19]

Curriculum Vitae

Alessandro Lo-Presti, born September 5th 1986 in Rotterdam, The Netherlands, is 1.78m tall, outrageously handsome, Dutch and proud of it. In addition to being a hero, trendsetter, and leader of fashion, he is widely regarded as an expert in all aspects of electronics (at least by his mother). He enjoys long romantic walks on the beach and his two favourite things are changing himself and commitment.

After finishing his secondary education in 2003, he attended an intermediate vocational education from 2003–2007. In 2007 he was admitted to the Rotterdam University of Applied Sciences to study "Technical Informatics" (subclass of Computer Engineering) where he is currently working towards attaining his Bachelor's degree.

Notable achievements during his time at the Rotterdam University of Applied Sciences are: a class he gave to his peers, during an academic quarter, in web development, accessibility guidelines/ standards and security; being rewarded a perfect 10 ('A' grade) for a project by introducing his technical specification for a communication protocol with encryption, which was implemented by all other project groups.

During his education he has also worked as a web developer in his spare time, from 2007–2011, at a company called *Liones*. His main activities included collaboratively developing new websites, maintaining existing websites and testing websites for security vulnerabilities and bugs.

His main interests include embedded systems/devices, reconfigurable hardware and computer architecture. Hobbies include a huge and active passion for snowboarding, playing guitar/flute and programming for handheld (gaming) devices.