# Report Lab Assignment
# Running x264 on Host Processor
# Extracting and Accelerating Kernel on Co-processor

Imara Speek (1506374)
Aimee Ferouge (4014030)

11 november 2013

| | |
|---|---|
| Date Performed: | October, 2013 |
| Instructor: | Ir. A. Brandon |

**Samenvatting**

During this lab, we are speeding up the `x264` application by extracting the `pixel_satd_8x4` and the `pixel_satd_4x4` kernel for parallel processing on a $\rho$-VEX co-processor. This involves running the `x264` application on a FPGA which runs a MicroBlaze operating system. The required instructions and data for the extracted kernels is being written to the instruction- and data memory of the $\rho$-VEX. This lab report describes in detail our chosen approach, problems we encountered along the way, test results measuring the execution time before and after extracting the kernels and an overal evaluation.

## 1 Introduction

For the lab a computational intensive kernel has to be extracted from the `x264` software application: a free software library for encoding video stream into H.236/MPEG-4 AVC format. It is able to use Periodic Intra Refresh instead of keyframes as used by h.264, refreshing the image of the video by moving a column of intra blocks from one side of the screen to the other. This hides the refreshing effect from the user while the frame loads. The `x264` application is executed on a FPGA running a MicroBlaze host processor. By running the particular kernel on the $\rho$-VEX co-processor, the execution time can be decreased by making use of parallel processing.

## 2 Exploring the x264 application

Duing the first lab we got familiar with the x264 application, a powerfull coding tool to remove temporal redundancy by using motion estimation. In Section 2.1 we will discuss the various computational intenstive kernels within the x264 algorithm. The choosen kernel will be explained in Section 2.2, the provided memory layout of the FPGA architecture in Section 2.3 and the actual implementation will be discussed in Section 2.4.

### 2.1 Detecting the Computationally Most Intensive Kernel

A few input files are provided in the first lab to demonstrate the compile and run commands of the x264 application. The `.y4m` input files are decoded to create a short `.mkv` movie using this command in the terminal: `./x264 eledream_64x32_3.y4m —o testframe.mkv`. By adding the `gprof` flag to the compile command, a list is created of all functions ordered by their share of the total execution time (in percentage).

The number of function calls is also shown, as well as the total execution time for each input file. These five input `.y4m` files, vary in resolution (either 64x32 or 640x320 pixels) and amount of frames (either 1, 3, 8, 32 or 128). Profiling the x264 execution for the `.y4m` files that are provided by the lab leads to the ranking show in table 1.

| Input file | Time (sec) | Share (%) | Calls | Kernel name |
|---|---|---|---|---|
| eledream_32x18_1.y4m | 0.02 | 100.00 | 71 | x264_analyse_init_costs |
| | | 0.00 | 1646 | x264_free |
| | | 0.00 | 784 | x264_cabac_encode_desicion_c |
| eledream_64x32_3.y4m | 0.03 | 66.67 | 71 | x264_analyse_init_costs |
| | | 33.33 | 1971 | x264_pixel_satd_4x4 |
| | | 0.00 | 4830 | x264_pixel_satd_8x4 |
| eledream_640x320_8.y4m | 1.61 | 14.29 | 1599044 | x264_pixel_satd_8x4 |
| | | 11.80 | 570708 | x264_get_ref |
| | | 4.97 | 38770 | x264_pixel_satd_x4_16x16 |
| eledream_640x320_32.y4m | 8.54 | 20.61 | 7228633 | get_ref |
| | | 13.23 | 15386501 | x264_pixel_satd_8x4 |
| | | 4.57 | 1009690 | x264_pixel_satd_x4_8x8 |
| eledream_640x320_128.y4m | 29.86 | 17.48 | 21956292 | get_ref |
| | | 14.17 | 49862831 | x264_pixel_satd_8x4 |
| | | 6.56 | 1023315 | x264_pixel_satd_x4_16x16 |

Tabel 1: Chart with computationally most intensive kernels for each input stream.

## 2.2 The `pixel_satd_8x4` kernel

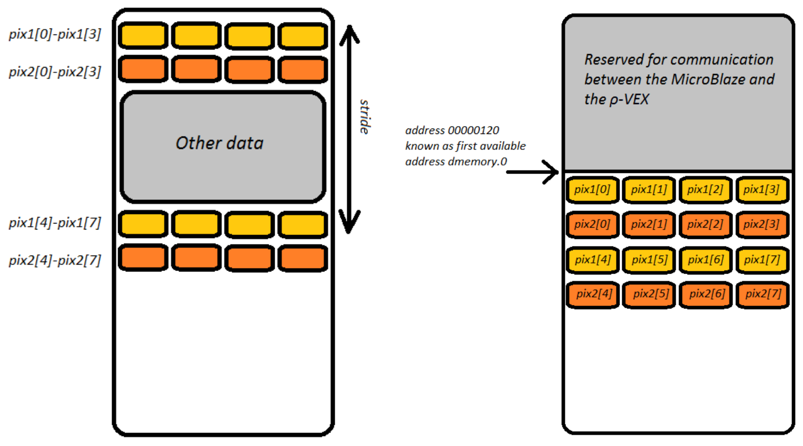
Given these statistics, we decide to extract the `x264_pixel_satd_8x4 kernel`, since its share in execution time increases as the files become larger. This kernel evaluates the Sum of Transformed Differences (SATD) between a 8x4 pixel block from the input stream and reference blocks. The SATD is a metric used for video compression where the differences between the pixels are taken and put into a frequency transform, usually a Hadamard transform. These blocks can have various sizes depending on the inputs and constraint of the system varying from 4x4 pixel blocks to 8x4 blocks and even 16x16 blocks. The reference block with the lowest SATD value can be used to estimate motion in video coding to remove temporal redundancy.

## 2.3 Memory layout of the MicroBlaze

When writing pixels to the $\rho$-VEX for SATD calculation, these pixels need to be picked from the MicroBlaze memory. However, these pixels are not written as one block of contiguous bytes. Rows of four bytes (MicroBlaze has 32-bit registers) are interrupted by a fixed amount of memory called a stride. This value is often given in bytes to be skipped in order to continue reading the particular data file. Figure 1a shows the concept of strides for two pixels from the MicroBlaze memory, which are splitted into two rows of 4 bytes since a pixels consists of eight bytes. When writing data to the $\rho$-VEX memory we use a stride of four bytes, resulting in a configuration as shown in 1b

## 2.4 Executing an Application on the Development Board and $\rho$-VEX

When executing `./configure`, a file is created for configuration of the application. However, the resulting `config.mak` file is made for applications running on the guest (Ubuntu). In order to configure for MicroBlaze, the `config.mak` file has to altered. All references to `m32` have to be removed and the `--DWORDS_BIGENDIAN` flag has to be added to the `CFLAGS` variable. This has to be done everytime when configurating the application for MicroBlaze.

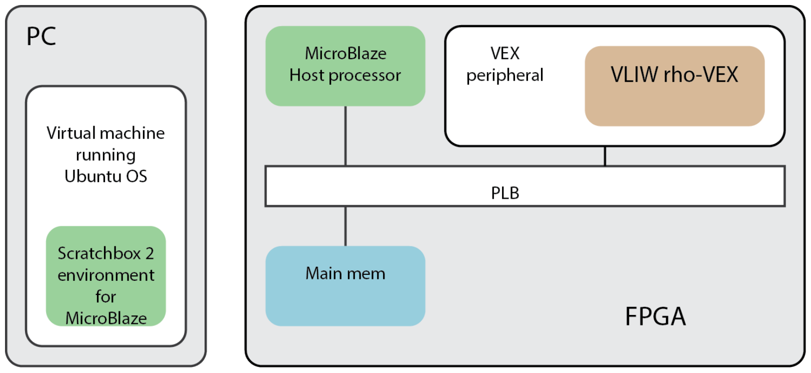

pix1[0]-pix1[3]
pix2[0]-pix2[3]
Other data
stride
pix1[4]-pix1[7]
pix2[4]-pix2[7]
Reserved for communication between the MicroBlaze and the ρ-VEX
address 00000120 known as first available address dmemory.0
pix1[0] pix1[1] pix1[2] pix1[3]
pix2[0] pix2[1] pix2[2] pix2[3]
pix1[4] pix1[5] pix1[6] pix1[7]
pix2[4] pix2[5] pix2[6] pix2[7]

(a) Large unknown stride

(b) Stride of four bytes

Figuur 1: MicroBlaze memory using strides to split up data files

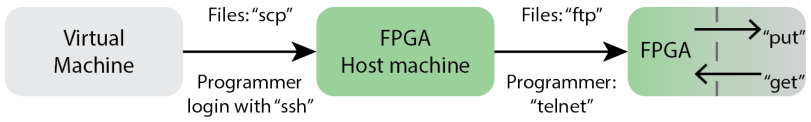
After doing this, the application now can be 'made' for MicroBlaze by first moving to the Scratchbox 2 environment for MicroBlaze and then execute the `make` command. Fig. 2 shows a block diagram of both the platforms.


PC
Virtual machine running Ubuntu OS
Scratchbox 2 environment for MicroBlaze
MicroBlaze Host processor
VEX peripheral
VLIW rho-VEX
PLB
Main mem
FPGA

Figuur 2: Block diagram of the Virtual Machine and the ERA platform

In order to run the application on the MicroBlaze host processor, the `x264` executable must be compressed along with an input stream in a `tar.gz` file. This file can be put, via the FPGA host machine, on one of the three FPGAs using the `scp`, `ftp` and `put` command. For the programmer to connect to the development board, one must use the `ssh` and `telnet` command. See also fig 3.
For the first lab, the `satd_8x4` kernel has to be:

- extracted into a separate file
- supplemented to a proper .c file which can be compiled and run using the ρ-VEX
- included in the makefile that creates a bytecode and bytedata file of the kernel
- debugged



Figuur 3: Ubuntu commands for putting files on and navigating to the FPGAs

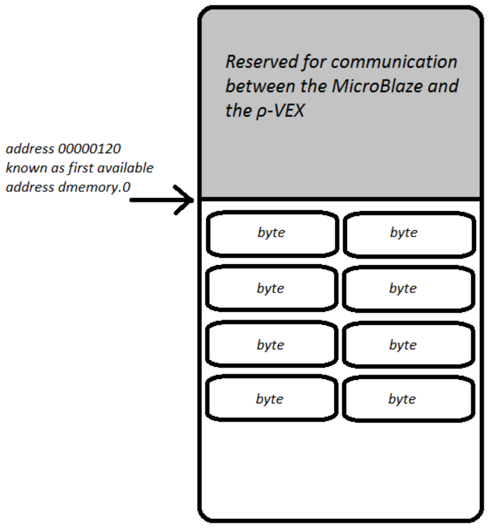
Looking back, the debugging part has been chasing us all the way through the lab. Not only did we suffer from bugs in our source code, the $\rho$-VEX itself did also have shortcomings that had to be circumvented by downloading several fixes, competing for FPGA availability and break downs of the entire host machine due to unsufficient capacity for the amount of students.

## 3 Approach

The kernel can run in the $\rho$-VEX by putting the bytecode generated by the makefile into the instruction memory of the $\rho$-VEX and placing all required data in its data memory dynamically. By executing `make byte−` command two files are created:

- `bytecode`, containing all the intstructions to be executed by the $\rho$-VEX
- `bytedata`, containing the pixels of the input stream

In order to make the extracted kernel qualified for compilation and execution, a few things have to be altered. First, the new file has to be recognized by the `makefile` in the rovex-examples directory. Second, some type definitions have to be made. Originally, `pixel_satd_8x4` resides in the `pixel.c` file of the x264 application. When extracting this kernel, all prior knowledge is lost and has to be defined again. Then, in order to make the kernel compile and run on the $\rho$-VEX, the development board has to be reset and started. `Bytecode` has to be written to the instruction memory (`rex−imemory`) and `bytedata` to the data memory (`rvex−dmemory`). Finally, the calculated result should be returned to the host. Figure 4 shows the $\rho$-VEX memory layout for our kernel.


Reserved for communication between the MicroBlaze and the ρ-VEX
address 00000120 known as first available address dmemory.0
byte
byte
byte
byte
byte
byte
byte
byte

Figuur 4: Memory layout of the $\rho$-VEX

## 3.1 Adjusting the `makefile`

We added the kernel to the `EXECUTABLES` defined in the `makefile`. Other files defined here can be removed since we will not need them for our application. An important issue is the difference between logical memory and physical memory. While the first address of a logical memory is obviously zero, the physical memory can have the first part of the register being occupied by the operating system. For the $\rho$-VEX, the first address which is allowed to be written to is 120. Thus, when writing the result of the kernel to the logical address '0', it is actually writing to the physical address '120' of the data memory.
Despite the fact that this is a rather simplistic operation, it took us some struggling to have this action confirmed as correctly. For example, we were told to remove the `autoinline` flag which resulted in a lot of wrong hexdumps. Also, half way the lab a fix had been made to eliminate the issue of the logical and physical addresses. Since the purpose of `__DATA_START` was now unclear we decided to remove all references to it, which felt like we had wasted lots of time. As it turned out, the fix had only affected the 'home' directory. In order to avoid conflicting files, we had created a separate folder named 'lab2' from where we executed the application. Due to this, the fix did not reach our code and thus did not eliminate the start address issue untill this problem was discovered very late into the project.

## 3.2 Constraints of the $\rho$-VEX

Using the $\rho$-VEX as a co-processor comes with some limitations. First, the $\rho$-VEX supports a very old and basic C compiler, so using `printf` statements to find bugs in the extracted kernel file is not possible. Everytime the application is being run concurrently on the $\rho$-VEX, the status is being inspected by placing `printf` statements in the `pixel.c` file. Also, the $\rho$-VEX is very strict in the order of variable declarations and initialization. One of our bugs involved `int i` being declared and initialized, followed by a declaration of `int result`. This is not supported by the $\rho$-VEX, since it wants to have `int i, result;` both to be declared first, before initializing `i = 4;`.
When running an extracted kernel on the $\rho$-VEX, the kernel is unable to obtain information from previous code. Consequently, all parameters have to be re-defined. Table 3 shows all required type definitions.

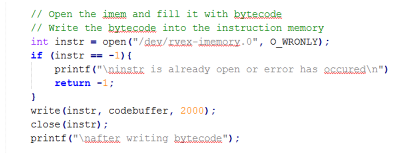
| **Parameter** | **Type definition** | **Motivation** |
|---|---|---|
| `intptr_t` | `unsigned int` | This parameter represents the stride, which cannot be <0 |
| `pixel` | `unsigned char` | Pixels are made out of bytes, which have 8 bits (like a `char`) |
| `sum_t` | `short int` | Same type definition as in the source code (16 bits) |
| `sum2_t` | `long int` | Same type definition as in the source code (32 bits) |
| `BIT_DEPTH` | defined as 8 | Also in the source code, plus the kernel handles pixels (bytes) |
| `BIT_PER_SUM` | (8 * `sizeof(sum_t)`) | Also predefined as in the source code, being 16 |

Tabel 2: Type definitions in the extracted kernel file.

## 3.3 Communication between the MicroBlaze and the $\rho$-VEX

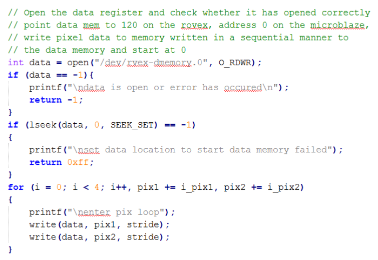
In order to delegate the `satd_8x4` kernel from the MicroBlaze to the $\rho$-VEX, both the environments need to communicate with each other. When executing the `x264` application, the MicroBlaze has to load the instructions of the extracted kernel into the instruction memory of the $\rho$-VEX and the data (for which the SATD has to be evaluated) into the data memory. This is when the source code of `x264` becomes involved. The `pixel_satd_8x4 kernel`, which is residing in the `pixel.c` file of the application, needs to be adjusted. Instead of calculating the SATD, it should send the instructions and input to the $\rho$-VEX. The commands 5a and 5b are therefore added to the `pixel.c` file of the `x264` application. These pixels are written at physical address 120, the first address on the $\rho$-VEX that is not set apart for communication between the driver and the $\rho$-VEX, defined as address 0. To be sure that the data is always written to the same location we use the

`lseek()` function to specify the start location of data as 0x00 from the `SEEK_SET`, or the beginning of the file.
*HOEVEEL TOEGEVOEGDE WAARDE HEEFT DIT? IS NAUWELIJKS LEESBAAR*



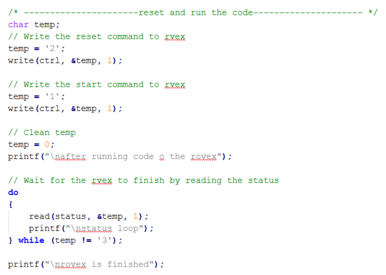
```c
// Open the imem and fill it with bytecode
// Write the bytecode into the instruction memory
int instr = open("/dev/rvex-imemory.0", O_WRONLY);
if (instr == -1){
    printf("\ninstr is already open or error has occured\n")
    return -1;
}
write(instr, codebuffer, 2000);
close(instr);
printf("\nafter writing bytecode");
```

(a) Bytecode



```c
// Open the data register and check whether it has opened correctly
// point data mem to 120 on the rovex, address 0 on the microblaze,
// write pixel data to memory written in a sequential manner to
// the data memory and start at 0
int data = open("/dev/rvex-dmemory.0", O_RDWR);
if (data == -1){
    printf("\ndata is open or error has occured\n");
    return -1;
}
if (lseek(data, 0, SEEK_SET) == -1)
{
    printf("\nset data location to start data memory failed");
    return 0xff;
}
for (i = 0; i < 4; i++, pix1 += i_pix1, pix2 += i_pix2)
{
    printf("\nenter pix loop");
    write(data, pix1, stride);
    write(data, pix2, stride);
}
```

(b) Bytedata



```c
/* ---------------------reset and run the code--------------------- */
char temp;
// Write the reset command to rvex
temp = '2';
write(ctrl, &temp, 1);

// Write the start command to rvex
temp = '1';
write(ctrl, &temp, 1);

// Clean temp
temp = 0;
printf("\nafter running code o the rovex");

// Wait for the rvex to finish by reading the status
do
{
    read(status, &temp, 1);
    printf("\nstatus loop");
} while (temp != '3');

printf("\nrovex is finished");
```

(c) Reset, set and get the status of the $\rho$-VEX

Figuur 5: Commands for reading from and writing to the $\rho$-VEX

To control the $\rho$-VEX from the x264 application, the control registers can be written to. By writing `'2'` to the control registers, the $\rho$-VEX is being reset. It can then be started by writing `'1'` to the register, telling the $\rho$-VEX to start running the instructions available in the instruction memory, calculating the satd of the two pixels. While calculating, the status of the $\rho$-VEX can be checked in a `while`-loop by reading the status variable of the status memory (`rvex−smemory`). When the `satd` kernel is finished, the result can be read from the data memory. See also figure 5c.

## 3.4 Result Hyphothesis

By having the computational intensive kernel being run concurrently on a co-processor, one would expect an overal speedup of the application. A problem is, however, that the `bytecode` and new data have to be sent to the $\rho$-VEX every time the kernel is being called. The data contains two new pixels of which the SATD has to be calculated, `bytecode` the kernel instructions. Because of this constant data traffic between the MicroBlaze and the $\rho$-VEX we don't think the intended speed up is achieved.
The reason that `bytecode` has to be sent every time the kernel is called, is that the three FPGAs are shared among a lot of students. When running their application concurrently, the instruction memory is constantly overwritten by another group. If there was a one-to-one setup, `bytecode` could have to been placed in `main.c`, written tot the `imemory` of the $\rho$-VEX once when starting the application.

## 4 Implementation

This section will cover some difficulties that came along during the lab. Section 4.1 will discuss our method for debugging and testing the working manner of the $\rho$-VEX, we will discuss some limitations we came accross in Section 4.2 and **??**. Important features to be considered were the difference in Endianess as will be discussed in Section 4.3 and the reliability of the code, discussed in Section 4.4.

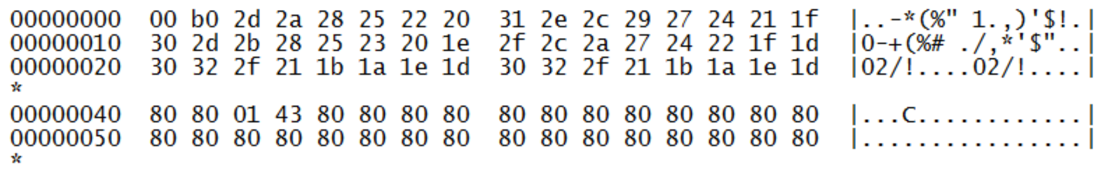
The approach was to first let the `pixel.c` write mock up pixel data to the data memory. The instruction memory would already contain the bytecode of our extracted kernel as it was put in manually. The kernel would only contain a single public declared variable `pixel datamem[128];`. This array of pixels points towards the first address of the data memory as no other variables are declared and thus there are no competing variables after this data memory location. The pixel.map file and the hexdump proved this to be correct. By declaring a pointer to this data memory location we could determine the result on the $\rho$-VEX and write the result into this array at a set location where the `pixel.c` can read and return it.

## 4.1 Mock up pixels

In order to get some insight in how to communicate with the $\rho$-VEX we made `hexdumps` of the data memory and created a mock up version of the satd function using 2 pixels we took from the original application. In order to check where the $\rho$-VEX is writing the data, a debugging file `Debugging_pixel.c` is created. When using this debugging file, `bytecode` is manually put into the instruction memory using `scp`, `ftp` and `put`. The data that is being sent from the MicroBlaze are two pre-defined pixels. By knowing the expected result and the possibility to write easy recognizable hexadecimal numbers such as `0xdead`, we can use the `hexdump` to examine the registers. The hexdump in figure 6 portrays all but the first two bytes or our mock up bytes. These first bytes display the correct result written to a convenient addres `0x00`.

## 4.2 Issues with the fix

Halfway the lab, a fix was introduced. The physical address was supposed to be set to 0x120 and did not have to be passed on using `__DATA_START`. In order to make the $\rho$-VEX read the pixels in the data memory correct a pointer to the first logical address 0 would be enough. However as we would later discover this was not the case. The fix we applied did not adjust our working directory as it was designed to alter the tools folder in the home directory and we were working in a seperate folder. When we did find out, we changed the script of the fix to match our own tools folder, but it did cause some days delay.



```
00000000  00 b0 2d 2a 28 25 22 20  31 2e 2c 29 27 24 21 1f  |..-*(%" 1.,)'$!.|
00000010  30 2d 2b 28 25 23 20 1e  2f 2c 2a 27 24 22 1f 1d  |0-+(%# ./,*'$"..|
00000020  30 32 2f 21 1b 1a 1e 1d  30 32 2f 21 1b 1a 1e 1d  |02/!....02/!....|
*
00000040  80 80 01 43 80 80 80 80  80 80 80 80 80 80 80 80  |...C............|
00000050  80 80 80 80 80 80 80 80  80 80 80 80 80 80 80 80  |................|
*
```

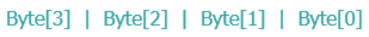
Figuur 6: Hexdump of the kernel run with two mock up pixels

## 4.3 Endianness

The MicroBlaze and the $\rho$-VEX understand basic operations like `open()`, `close()`, `read()` and `write()`. However, they differ in Endiannes. $\rho$-VEX operates in Little Endian, meaning the bytes are read from right to left, as seen in figure 7a. MicroBlaze, on the other hand, operates in Big Endian and thus reads the bytes from left to right (see figure 7b ). This becomes an issue when reading the `result` variable from the data memory, as we will get the bytes in reversed order. This problem is solved by manually reversing the order of the `result` bytes, realized by the piece of code in figure 7c.

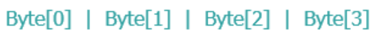
## 4.4 Reliability of the code

When writing pixels to the data memory of the $\rho$-VEX, the data pointer is automatically increased. To ensure that `pixel.c` reads the correct result from the memory location we can use `lseek`, a system call that is used to change the location of the read/write pointer of a file descriptor. Using `lseek(data, 0, SEEK_SET)`, the pointer in the $\rho$-VEX moves to the address that is the start address from his point of view (thus, physical address 0x120). Translating the logical address 0 to the physical address 0x120 is done by the linker and is not of our concern. We also added some if statements that ensure that if memory locations are not opened correctly, they return `0xff` and print an error to the screen. These measures increase the reliability of the code.



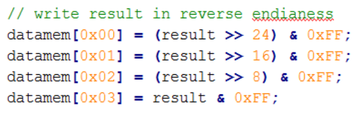
Byte[3] | Byte[2] | Byte[1] | Byte[0]

(a) Little Endian



Byte[0] | Byte[1] | Byte[2] | Byte[3]

(b) Big Endian



```
// write result in reverse endianess
datamem[0x00] = (result >> 24) & 0xFF;
datamem[0x01] = (result >> 16) & 0xFF;
datamem[0x02] = (result >> 8) & 0xFF;
datamem[0x03] = result & 0xFF;
```

(c) Reversing the order of bytes

Figuur 7: Different reading direction due to Endianness

## 5 Extracting a Second Kernel

For lab 3, a second kernel has to be extracted in order to achieve an even higher speed up. For this assignment, we wanted to choose our kernel more wisely. Given the constant competition for the use of the FPGAs, `bytecode` has to be written to the instruction memory everytime the kernel is being called. In combination with the data to be processed by the kernel, this leads to huge data traffic which is actually slowing down the application. This is why we wanted, at first, to pick a kernel that is not being called too often but requires a lot of heavy computation. A candidate for this would be the `halfpel` kernel, which has quite its share in execution time but not that many functions calls. The half- and quarter are the functions that ensure the motion estimation in x264 and can consume up to 90% of the total computing time in HD videos. As these functions are not called that often, they consume relatively less time in communication between the MicroBlaze and the $\rho$-VEX and are therefor far more valuable to optimize using parallel computing. However, considering the time constraints and the set backs in this projects we choose a kernel similar to `pixel_satd_8x4` which also has a large share in the execution time: `pixel_satd_4x4`.

### 5.1 Extract `pixel_satd_4x4`

There is not a lot of difference in `pixel_satd_8x4` and `pixel_satd_4x4`. The main difference concerns the stride when writing the pixel streams to the data memory. Previously the stride was 8 and now it is 4. Because this kernel was relatively easy to implement in the $\rho$-VEX as well, we decided to invest some time into comparing the effect of using a single or two bytecodes. The first approach was using a single bytecode, extended with a second function that implemented `pixel_satd_4x4`. The `pixel.c` would have to let the $\rho$-VEX know which function it has to run. We solved this by reserving a byte in the datamemory at location `0x40` where we either wrote `0x84` or `0x44`. The $\rho$-VEX is then able to read from this data location and determine which function it would have to run.
A big downside on this solution however is that the total bytecode of these combined kernels is 3744 bytes. Loading such a large amount into the instruction memory everytime the function is called can slow down the application a lot. Therefor we decided to divide the two kernels into two bytecodes of less than 2000 bytes. Therefor the `pixel.c` only has to load a different bytecode into the instruction memory when the different kernels are called. Both bytecodes have to be loaded into a codebuffer. This is done in the `x264.c` file and the codebuffers are declared in the common file `common.h`

## 6 Evaluation

As was to be expected in a lab of 60 students and 3 FPGAs there was no speedup compared to the initial `x264` application. However we still stick to our result hypothesis that when choosing a relative kernel that

does not spend too much time in communication a definite speedup can be achieved as discussed in Section 5. We did gain some insight in comparing the time spend when executing the application using a single or using two bytecodes.
`time ./x264 eledream_64x32_3.y4m -o testtime.mkv` provides a nice overview of the time spend in the application. We measured both times and these were the result.

| Time parameter | Single bytecode | Two bytecodes |
|---|---|---|
| real | 7m 2.17s | 6m 15.13s |
| user | 2m 21.16s | 2m 20.81s |
| sys | 1m 17.39s | 1m 14.68s |

Tabel 3: Execution times of the x264 application variating in bytecodes

We would have liked to achieve some sort of speed up that would actually benefit the initial application. However concerning the (amount of) tools and time that was handed to us we considered we did fairly well. We learned a lot from this lab, but some better guiding in the available information about fixes or limitations in compiling would have been appreciated prior to the lab.

## A Extracted kernels pixel_satd_8x4 and pixel_satd_4x4

```
// Defining types for the expected input arguments
// changed intptr_t to int instead of unsigned int and sum2 from us long to us int
typedef unsigned int intptr_t;
typedef unsigned char pixel;
typedef unsigned short sum_t;
typedef unsigned long sum2_t;

// Define an inputarray that points to the start of the data memory
pixel datamem[128];

// BIT_DEPTH is 8 always, so if else statement is removed
#define BIT_DEPTH 8

// #define BITS_PER_SUM (8 * sizeof(sum_t))
#define BITS_PER_SUM (8 * sizeof(sum_t))

// HADAMARD4 function exported from the original pixel.c file
#define HADAMARD4(d0, d1, d2, d3, s0, s1, s2, s3) {\
    sum2_t t0 = s0 + s1;\
    sum2_t t1 = s0 - s1;\
    sum2_t t2 = s2 + s3;\
    sum2_t t3 = s2 - s3;\
    d0 = t0 + t2;\
    d2 = t0 - t2;\
    d1 = t1 + t3;\
    d3 = t1 - t3;\
}

// abs function exported from the original pixel.c file
static sum2_t abs2(sum2_t a);

// x264_pixel_satd_8x4 exported from the original pixel.c file
int x264_pixel_satd_8x4();

// x264_pixel_satd_4x4 exported from the original pixel.c file
int x264_pixel_satd_4x4();
```

```
char main()
{
    // run the x264_pixel_satd_8x4 by passing on the pixel values.
    int result, i;
    result = 0;
    i = 0;

    // calculate the result
    if (datamem[0x40] == 0x44)
    {
        result = x264_pixel_satd_4x4();
    }
    else if (datamem[0x40] == 0x84)
    {
        result = x264_pixel_satd_8x4();
    }
    else result = 0xdead;


    // clean result array
    for (i = 0; i < 4; i++)
    {
        datamem[i] = 0x00;
    }

    // write result in reverse endianess
    datamem[0x00] = (result >> 24) & 0xFF;
    datamem[0x01] = (result >> 16) & 0xFF;
    datamem[0x02] = (result >> 8) & 0xFF;
    datamem[0x03] = result & 0xFF;

    // DEBUG
    // datamem[0x40] = 0xfe;
    // datamem[0x41] = 0xed;

    return 0;
}

static sum2_t abs2(sum2_t a)
{
    sum2_t s = ((a>>(BITS_PER_SUM-1))&(((sum2_t)1<<BITS_PER_SUM)+1))*((sum_t)-1);
    return (a+s)^s;
}

int x264_pixel_satd_8x4()
{
        sum2_t tmp[4][4];
        sum2_t a0, a1, a2, a3;
    sum2_t sum = 0;
    int i = 0;

    // Strides are set values and do not have to be given as input arguments
    // Stride is 16 to compute 2 rows of 8 bytes of pixel array elements
    intptr_t stride = 16;

    // Pointer to data memory
    pixel* datapoint = datamem;

    // Adjust the for loop so it can fetch data from data memory
    for(i = 0; i < 4; i++, datapoint += stride)
    {
        a0 = (datapoint[0] - datapoint[8]) + ((sum2_t)(datapoint[4] - datapoint[12]) << BITS_PER_SUM);
        a1 = (datapoint[1] - datapoint[9]) + ((sum2_t)(datapoint[5] - datapoint[13]) << BITS_PER_SUM);
        a2 = (datapoint[2] - datapoint[10]) + ((sum2_t)(datapoint[6] - datapoint[14]) << BITS_PER_SUM);
        a3 = (datapoint[3] - datapoint[11]) + ((sum2_t)(datapoint[7] - datapoint[15]) << BITS_PER_SUM);
        HADAMARD4( tmp[i][0], tmp[i][1], tmp[i][2], tmp[i][3], a0,a1,a2,a3 );
    }
    for(i = 0; i < 4; i++ )
    {
        HADAMARD4( a0, a1, a2, a3, tmp[0][i], tmp[1][i], tmp[2][i], tmp[3][i] );
        sum += abs2(a0) + abs2(a1) + abs2(a2) + abs2(a3);
    }
    return (((sum_t)sum) + (sum>>BITS_PER_SUM)) >> 1;;
}

int x264_pixel_satd_4x4()
{
    sum2_t tmp[4][2];
    sum2_t a0, a1, a2, a3, b0, b1;
    sum2_t sum = 0;
    int i = 0;

        // Strides are set values and do not have to be given as input arguments
    // Stride is 8 to compute 2 rows of 4 bytes of pixel array elements
    intptr_t stride = 8;

    // Pointer to data memory
    pixel* datapoint = datamem;

    // instead of 8x4 method, it writes per 4 bytes
    for( i = 0; i < 4; i++, datapoint +=stride )
    {
        a0 = datapoint[0] - datapoint[4];
        a1 = datapoint[1] - datapoint[5];
        b0 = (a0+a1) + ((a0-a1)<<BITS_PER_SUM);
        a2 = datapoint[2] - datapoint[6];
        a3 = datapoint[3] - datapoint[7];
        b1 = (a2+a3) + ((a2-a3)<<BITS_PER_SUM);
        tmp[i][0] = b0 + b1;
        tmp[i][1] = b0 - b1;
    }
    for( i = 0; i < 2; i++ )
    {
        HADAMARD4( a0, a1, a2, a3, tmp[0][i], tmp[1][i], tmp[2][i], tmp[3][i] );
        a0 = abs2(a0) + abs2(a1) + abs2(a2) + abs2(a3);
        sum += ((sum_t)a0) + (a0>>BITS_PER_SUM);
    }
    return sum >> 1;
}
```