POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Computer Engineering

Master Degree Thesis

ASIP Design for Motion Estimation in Video Compression Algorithms

Supervisiors: prof. Maurizio Martina, prof. Guido Masera

Author: Simone Machetti

October, 2018

Summary

Motion Estimation is one of the most complex and time consuming functions of any video compression algorithm. This function requires a huge amount of computations to evaluate the expected results. Generally, the Motion Estimation is implemented in software by using specific video compression programs able to run on a General Purpose Processor (GPP). This approach is characterized by high flexibility but very low performance. On the other hand, some hardware implementations, called Application Specific Integrated Circuits (ASICs), exist, aimed at very special applications that require high performance at the cost of very low flexibility.
ASIP design is an intermediate solution between the pure software and the pure hardware methods. This approach is implemented using an existing general purpose processor as a starting point and optimizing its instruction set and architecture to make it suitable for a specific application. The main advantage of ASIP is the possibility to freely decide which function of the algorithm has to be implemented in software and which in hardware; this allows the designer to explore the whole design space and find the best trade off between flexibility and performance.
This thesis describes in detail the procedure to implement an ASIP processor, called MECORE, optimized for Motion Estimation applications, following the design steps provided by the Synopsys training manual [1]. Three versions of the ASIP have been realized, starting from the existing TMICRO processor model, found in the ASIP Designer software library. Each version provides an increase in performance with respect to the previous one.
The first version introduces new instructions, used to support single byte load and store operations from and to the data memory, not supported by the original architecture of TMICRO.
In the second version, two special purpose instructions are added to the MECORE instruction set: one computes the absolute value of the difference of two pixel values and accumulates the result; the other combines the absolute difference and accumulation with two parallel load operations. These two additional instructions exploit a new designed scalar accelerator, and are used to speed up the execution of the Sum of Absolute Differences (SAD), that is the most expensive function of any ME algorithm.
The third version extends the MECORE instruction set with new SIMD instructions. SIMD stands for Single Instruction Multiple Data and is a form of data parallelism: a SIMD instruction processes the elements of a vector simultaneously and in an identical way, specified by that single instruction. These additional features are supported by a new designed SIMD hardware accelerator, and allow to reach very high performance with good flexibility.
Each version has been designed using the nML description language and simulated making use of the ASIP Designer tools. The code, used for testing the processor, computes the Motion Vector of a single Macro Block of size 16x16 pixels over a Search Window of 48x48 pixels, and is executed in only 9.46 us by the last version of MECORE. Each version has also been synthesized, using a 65 nm CMOS library of cells, exploiting Synopsys Design Compiler.
Finally, the MECORE processor is compared to other existing approaches and architectures, found in the literature, to highlight differences and similarities.

Notice

The three versions of the MECORE microprocessor core, described in chapter 3, have been implemented following the steps provided by the Synopsys training manual [1]. The sections of the chapter describe in detail the design steps performed by the author of this thesis to implement each version of MECORE, but are not proposed as original implementations. The purpose of the chapter is to provide a deeper and more accurate description of the design procedure to guide and instruct the user on how to implement the three versions of the processor from scratch and how to test them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction to Motion Estimation

The Motion Estimation is one of the most complex and time consuming functions of any video compression algorithm. Both old and recent standards use the Motion Estimation to perform video compression. The algorithm may change, from one standard to another, but the main working principle remains the same across any compression standard. In this section, the Motion Estimation is described and analyzed: first, highlighting the main algorithm behind it and then, focusing on the most common and powerful cost functions.

1.1 Algorithm

A video is a sequence of frames, still images, each with the same number of pixels. The Motion Estimation is used to remove the temporal redundancy between frames by exploiting temporal correlation, i.e. each block of pixels in a frame will move in the following frame to a position at a certain distance with respect to the original one. This principle is adopted to compress a video sequence, storing only the information regarding the motion of each block from one frame to another.

The algorithm consists of the following steps:

1. Block Partitioning,
2. Search Window construction,
3. Search Window exploration,
4. Motion Vector evaluation,
5. Iteration.

Block Partitioning

The current frame, i.e. the frame that is currently being encoded, is divided into a number of blocks, called Macro Blocks (MB), of specific dimensions. This process is called Block Partitioning. Figure 1.1 shows the partitioning of a frame with 16x16 pixel blocks. Each MB will be encoded independently from the others, as you will analyze in the following steps.

Note that, the same dimension for all blocks has been selected to simplify the explanation, but in practice it may change depending on the adopted algorithm.

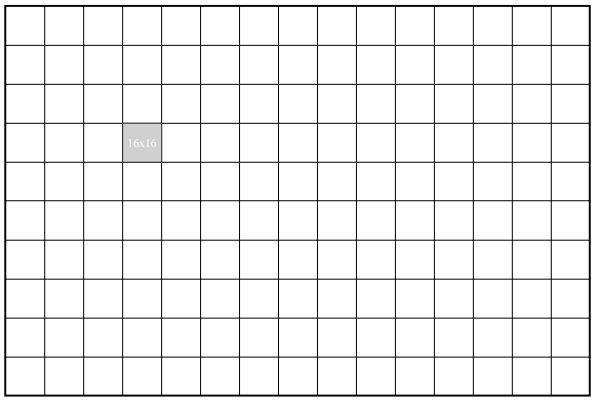


Figure 1.1: Block Partitioning of the current frame.

Search Window construction

Once a frame is partitioned, a Macro Block in the current frame is selected and an area, called Search Window, is defined around the corresponding position of the Macro Block in the reference frame, i.e. the previously encoded frame. This area is generally squared and is constructed starting from the position of each corner of the a selected block in the reference frame and adding or subtracting a previously defined value. See figure 1.2.

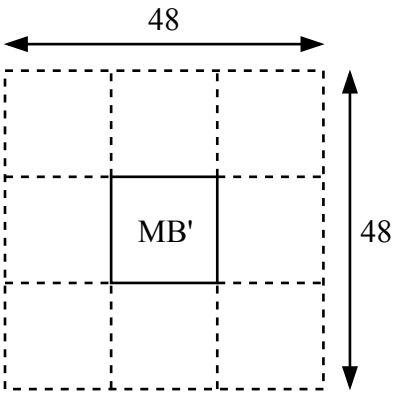


Figure 1.2: 48x48 Search Window of a 16x16 Macro Block.

Note that, the area is located in the reference frame and not in the current frame, and is constructed around the block of pixels, that we call MB', at the same position in the frame with respect to the position of the Macro Block in the current frame.

Search Window exploration

After that, the Search Window is explored to find the Best Matching Block (BMB), i.e. the block that best matches the selected Macro Block in the current frame. This implies starting from the left upper corner of the window, taking the block of pixels that starts in that position and with the same size of the selected MB and evaluating a cost function between them. Then, the block is moved one sample column toward right and a new comparison is performed between the MB and the new obtained block. This is repeated until the block reaches the right edge of the Search Window.

Then, the block is moved back to the left side and down by one sample row and the same actions and motion along the row are carried out. This procedure is repeated until the block reaches the right lower corner of the Search Window. A full search has been computed.

Motion Vector evaluation

Once the whole Search Window has been explored, the block with the highest value in the cost function is selected and the Motion Vector (MV) is computed. This vector, composed of a couple of values (ΔX, ΔY), specifies the horizontal and vertical displacements of the Best Matching Block in the reference frame with respect to the position of the original Macro Block in the current frame. In figure 1.3, you can see a sample Motion Vector: the vector starts from the top left pixel of MB' and ends at the top left pixel of the Best Matching Block; the starting position of the vector is considered the origin (0, 0) of the reference system.

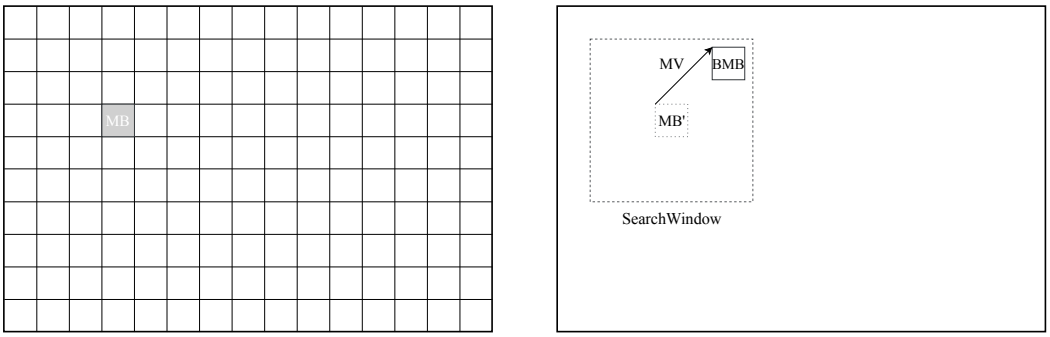


Figure 1.3: Motion Vector.

Iteration

Steps 2, 3 and 4 are repeated for each Macro Block in the current frame. In this way, a Motion Vector for each block is generated. See figure 1.4.

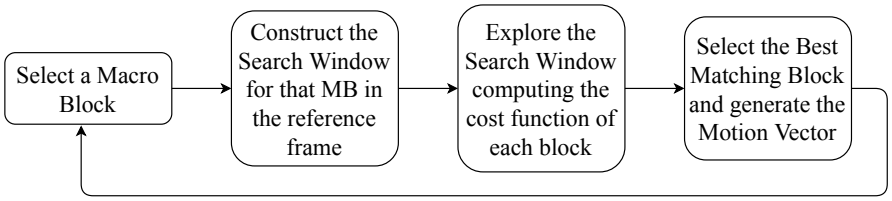


Figure 1.4: Motion Estimation FSM.

1.2 Cost functions

During the Search Window exploration, a cost function is used in order to find the block that best matches the selected Macro Block. The function is evaluated for each block in the Search Window and, at the end, the block that has obtained the highest value is taken.

In this section, the most common and powerful cost functions are analyzed, with a particular focus on the SAD technique.

1.2.1 SAD

SAD stands for Sum of Absolute Differences and is the most adopted cost function for Motion Estimation applications. It computes the absolute value of the difference of each pair of pixels in the same position in the two blocks and then sums all the resulting values together. The computation is performed by applying the following formula:

$$\texttt{SAD} = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} |C_{ij} - R_{ij}|$$

where n and m are the number of rows and the number of columns of a Macro Block respectively, C_{ij} is the selected pixel in the current frame and R_{ij} is the selected pixel in the reference frame.

1.2.2 Other techniques

This subsection analyses other two common cost functions: MAD and MSE.

MAD

MAD stands for Mean of Absolute Differences and is computed by applying the following formula:

$$\texttt{MAD} = \frac{1}{N^2} \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} |C_{ij} - R_{ij}|$$

where N is the size of a Macro Block, i.e. the total number of pixels (nxm).

MSE

MSE stands for Mean Squared Error and is computed by applying the following formula:

$$\texttt{MSE} = \frac{1}{N^2} \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (C_{ij} - R_{ij})^2$$

where N is the size of a Macro Block, i.e. the total number of pixels (nxm).

Chapter 2

ASIP design

This chapter describes the meaning of ASIP design and the reason for which this approach has been selected to implement the Motion Estimation, with respect to other traditional methods.
The first sections explain the design approach and perform a comparison with the two borderline methods: the ASIC design and the General Purpose uP design. Then, the last section gives an overview of the software ASIP Designer by Synopsys, that has been used to implement the MECORE microprocessor described in chapter 3.

2.1 Design approach

ASIP stands for Application Specific Instruction-set Processor. This approach is implemented using as a starting point a general purpose processor and optimizing its instruction set and architecture in order to make it suitable for a specific application. Usually, this is performed in some steps:

1. choose an appropriate microprocessor to be used as a starting point;
2. remove unnecessary functional units;
3. remove unnecessary instructions;
4. add new specific functional units used to implement in hardware the most time expensive functions, e.g. the Motion Estimation;
5. add new instructions to use the new functional units;
6. check if your design satisfies the requirements.

A very important part in the development of an ASIP is the selection of the general purpose processor that has to be used as a starting point for the design. Two main aspects have to be considered: first, it should be as simple as possible to avoid wasted area and power; second, it should be powerful enough to allow the designer to implement the minimum number of changes in order to make the design satisfying the requirements.

As you will see in chapter 3, the processor chosen to implement the Motion Estimation is called TMICRO. It has a simpler architecture with respect to other processors, such as the DLX, and this ensures a lower area and power consumption. At the same time, it is powerful enough to support all the changes done, from version one to three, to archive a much faster Motion Estimation.

2.2 Comparison with other approaches

ASIP design is an intermediate approach between the two borderline design methods: ASIC and General Purpose uP. This section briefly analyzes the two traditional approaches and then collocates the ASIP approach among them, highlighting differences and similarities, see figure 2.1.

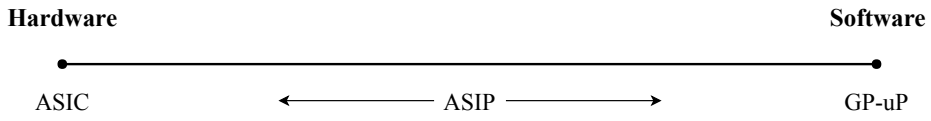


Figure 2.1: Design approaches.

ASIC approach

ASIC stands for Application Specific Integrated Circuit and corresponds to the left most solution in figure 2.1. In fact, ASIC design is a complete hardware solution, where algorithms are fully described at the RTL level, generally in VHDL or Verilog, and implemented in silicon.

Advantages:

- best performance,
- smallest area,
- lowest power consumption.

Disadvantages:

- high complexity,
- long design time,
- high design cost,
- no flexibility.

One of the main disadvantages of the ASIC approach is the zero flexibility of the design: once the functions are implemented in hardware there is no way of performing changes to the algorithm; the only way is to re-design a new integrated circuit, with a very high additional cost.
For this reason, ASIC design is only used for specific applications with very restrictive requirements that can not be met with other design methods.

GP-uP approach

The general purpose uP approach corresponds to the right most solution in figure 2.1. In this method, algorithms are implemented in software, such as C or Assembly, and executed on a general purpose microprocessor. No hardware optimization is performed and for this reason, this solution is known as the pure software solution.

Advantages:

- low design complexity,

- short design time,
- low design cost,
- high flexibility.

Disadvantages:

- low performance,
- big area,
- high power consumption.

Differently from the ASIC approach, this solution is very flexible and the implemented algorithms can be changed as many times the designer wants, during the circuit lifetime; this at the cost of lower performance and higher area and power consumption.
This approach is generally used for those applications that do no have strict requirements and that need the code to be updated regularly.

ASIP approach

The ASIP approach is an intermediate solution, as you can see in figure 2.1, between the ASIP and the GP-uP methods.
The main advantage of this solution is the possibility of moving along the line to obtain the most suitable location for your design. The designer is free to choose which functions have to be implemented in hardware and which in software. Normally, this mapping is performed considering the complexity and the time required by each function: the most time consuming functions are mapped to hardware modules, while the less expensive functions to software modules.
This approach allows you to find the best compromise, between hardware and software, that meets the requirements of your design.

2.3 ASIP Designer

ASIP Designer by Synopsys is a software that allows you to design ASIP processors. This is performed using a proprietary description language, called nML, that you can use to describe the instruction set and architecture of your own processor. After the design phase, a simulation can be carried out, running your own code on the designed ASIP. The VHDL/Verilog files of the design, an Assembler and a C compiler for that specific instruction set are also generated and provided to the user.

2.3.1 Main features

In this subsection, the main features of ASIP Designer are listed and analyzed:

- the instruction set and the architecture are described through the nML description language, examined in the following subsection.
- The VHDL or Verilog files of the design can be generated by the software and used to synthesize or place and route the processor; note that, the tools to perform the synthesis and physical design are not included in ASIP Designer.
- Two compilers are provided to the user: an Assembly compiler, used to compile the specific assembly code to the specific machine language of your processor; a C compiler, to compile the traditional C code to the specific machine language required by your processor.

- A powerful simulation tool allows you to compile your own code (Assembly or C) and to perform a simulation running the code on the designed processor; statistical information about the simulated code, such as the number of clock cycles required to perform the simulation, the number of executed instructions, the execution time, profiling information and many others, are provided by the tool.
- It provides a easy to use and intuitive graphical interface that guides the user along the different functions offered by the tool.
- An extensive documentation, describing each tool of the software in detail, is included in the ASIP Designer distribution, see manual [2].
- A large variety of processor models, to be used as a starting point for new designs, can be found in the ASIP Designer library, see manual [3].

2.3.2 nML language

This subsection gives an introduction of the nML description language used to describe processor models; the complete documentation can be found in the nML reference manual [4].

Several files are needed to exhaustively describe a processor model. The main ones are listed here:

- *<processor>.n*
- *<processor>.h*
- *<processor>.p*
- *<processor>_chess.h*

<processor>.n

The nML description is located in the file *<processor>.n*, with <processor> the name of the target processor. This file contains:

- code and data memory declarations,
- register file declarations,
- register declarations,
- constant declarations,
- instruction set rules.

The instruction set rules are used to define the instructions, with the corresponding syntax and encoding, composing the instruction set of the target processor. The description always starts with the start symbol of the grammar:

```
start <processor>
```

These sets of rules are generally grouped into several files, included in the *<processor>.n* file through the following code line:

```
#include "<file_name>.n"
```

<processor>.h

This file is the primitive processor header file. It is a C++ header file that declares the data types and the operations used in the nML description. The primitive data types are modeled using C++ classes, while the primitive operations are modeled using C++ functions and operators.

All the declarations are done inside the following *namespace*:

```
namespace <processor>_primitive
{
/* Declarations */
}
```

<processor>.p

The behavior of the primitive functions, declared in the processor header file, is specified in the *<processor>.p* file using a description language called PDG. PDG stands for Primitives Definition and Generation language and its complete documentation can be found in the reference manual [5].

<processor>_chess.h

It is the compiler processor header file and describes the translation process from the C build-in types and operators to the processor primitive data types and operations. This allows the primitive types and functions to be used and called as intrinsic data types and functions inside the C programs to be run on that specific processor model.
The translations are generally divided in groups and stored into separated header files; each file is then included in the *<processor>_ chess.h* file through the following code line:

```
#include "<processor>_<group>.h"
```

2.3.3 Quick start guide

This subsection contains a quick guide of ASIP Designer, that the user can use to learn the most important actions that can be performed with the software.

The presented topics are:

1. how to launch ASIP Designer;
2. how to compile a processor model;
3. how to create a new project;
4. how to perform a simulation;
5. how to figure out the profiling information;
6. how to generate the VHDL code of a processor model.

How to launch ASIP Designer

ASIP Designer is launched issuing on the command line the following commands one after the other:

```
source /software/synopsys/asip_designer_vN-2017.09-SP2/linux64/chess_env_LNa64.sh
source /software/scripts/init_asip_designer
chessde &
```

Note that, this commands may be subjected to change depending on the version of the software that you have installed and on the location of the main software folder on your file system. I am currently using ASIP Designer N-2017.09-SP2.

Once ASIP Designer is launched, you can start using the graphical interface, shown in figure 2.2.

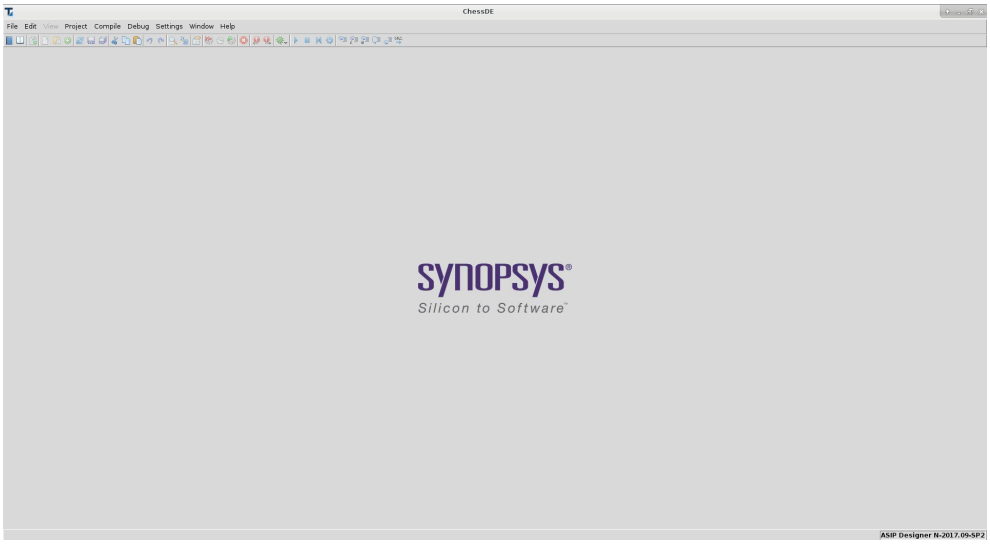


Figure 2.2: ASIP Designer graphical interface.

How to compile a processor model

An existing processor model can be compiled in three steps:

1. from the main window top menu select *File* → *Open* → *Project...* (figure 2.3).

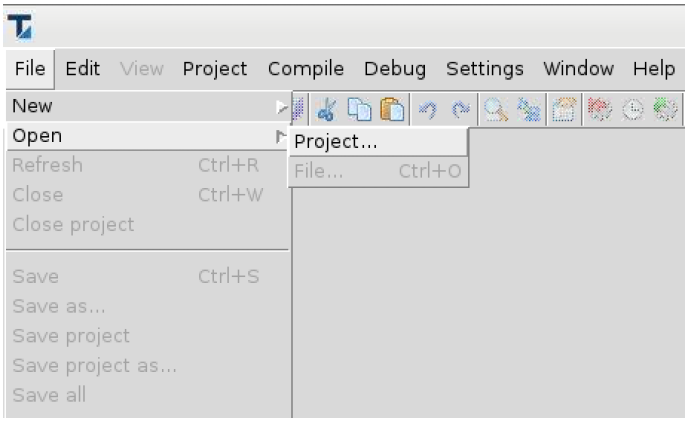


Figure 2.3: Compilation, Step 1.

2. A window opens; navigate in the processor folder and select the processor project file *model.ptx*, then click *Open*. See figure 2.4.

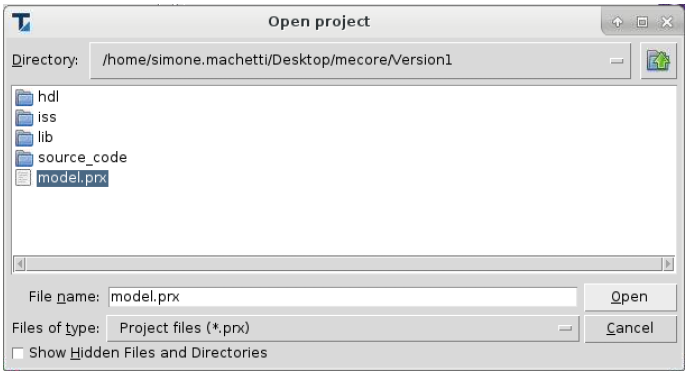


Figure 2.4: Compilation, Step 2.

3. The processor model has been loaded; in the top menu, click *Make* to compile all the processor model files (figure 2.5).

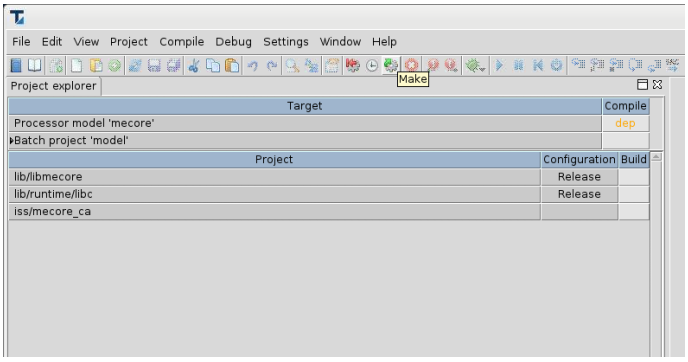


Figure 2.5: Compilation, Step 3A.

In case of a successful compilation, a green OK appears on each row of the *Compile* column. See figure 2.6.

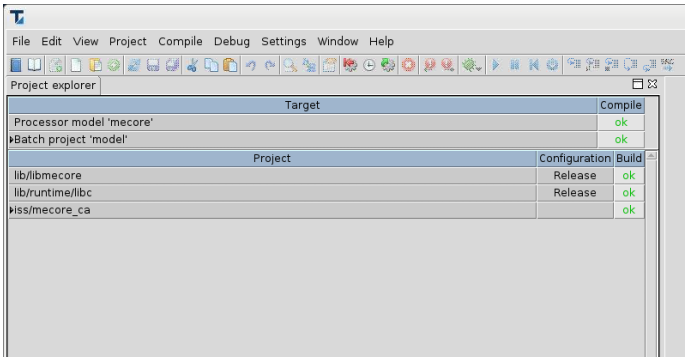


Figure 2.6: Compilation, Step 3B.

How to create a new project

The creation of a project is used to associate a C program with the processor model intended to execute it. In this way, the program can be compiled for that particular processor and a simulation can be run to obtain the required statistics.
A new project is created in five steps:

1. from the main window top menu select *File* → *New* → *Project...* (figure 2.7).

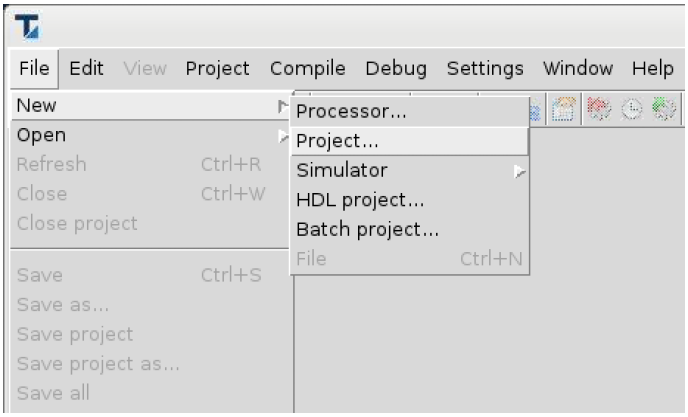


Figure 2.7: Project creation, Step 1.

2. A window opens; write a name for your project, in the *Project name* field, then browse and select a directory where your project will be created, in the *Project directory* field; after that, set *Executable* as *Processor type*, write the existing processor name, in the *Processor name* field, and browse and select the directory containing the processor files (*.../lib*), in the *Processor model directory* field; finally, click *Ok*. See figure 2.8.

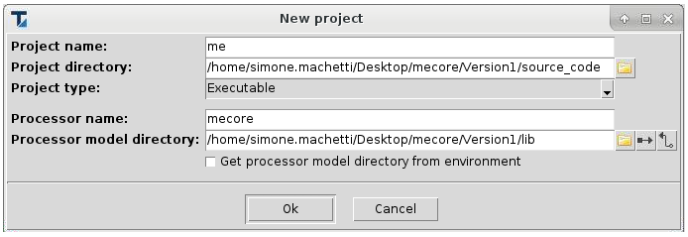


Figure 2.8: Project creation, Step 2.

3. The processor model has been loaded; in the top menu click on the *Add source files...* icon to import into the project the C source files containing your program (figure 2.9).

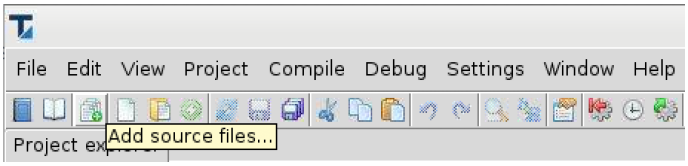


Figure 2.9: Project creation, Step 3.

4. A window opens; browse and select the C source files to be imported; then, click *Open*. See figure 2.6. Note, you do not need to import the input and output files used by the C program.

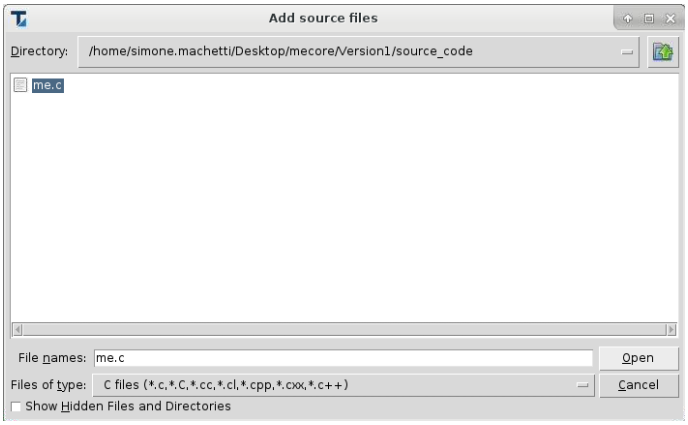


Figure 2.10: Project creation, Step 4.

5. The source files have been imported; in the top menu click on the *Make* icon to compile your program for that specific processor model (figure 2.11).

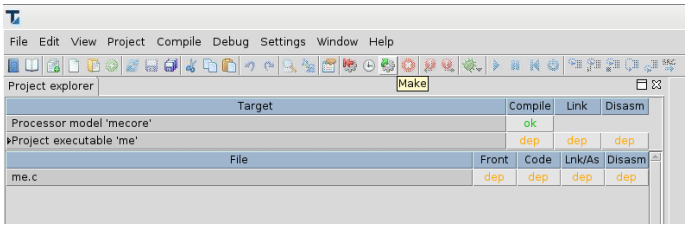


Figure 2.11: Project creation, Step 5A.

In case of a successful compilation, green Oks appear on the right side of each row.
See figure 2.12.

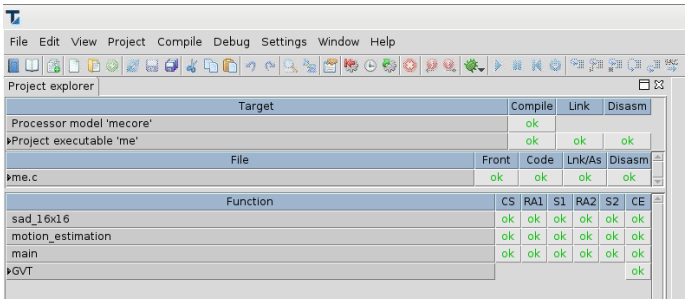


Figure 2.12: Project creation, Step 5B.

How to perform a simulation

Once a new project has been created, you are able to open it whenever you need, following steps 1 and

2 of the guide *How to compile a processor model*, but selecting the project file *<project_name>.ptx* instead of the processor model file *model.ptx*.

A simulation is performed in four steps:

1. the project is open; in the top menu click on the *Start debugging* icon to launch the simulation environment (figure 2.13).

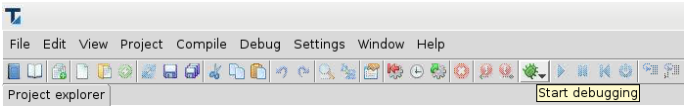


Figure 2.13: Simulation, Step 1.

2. A new graphical interface appears. See figure 2.14.

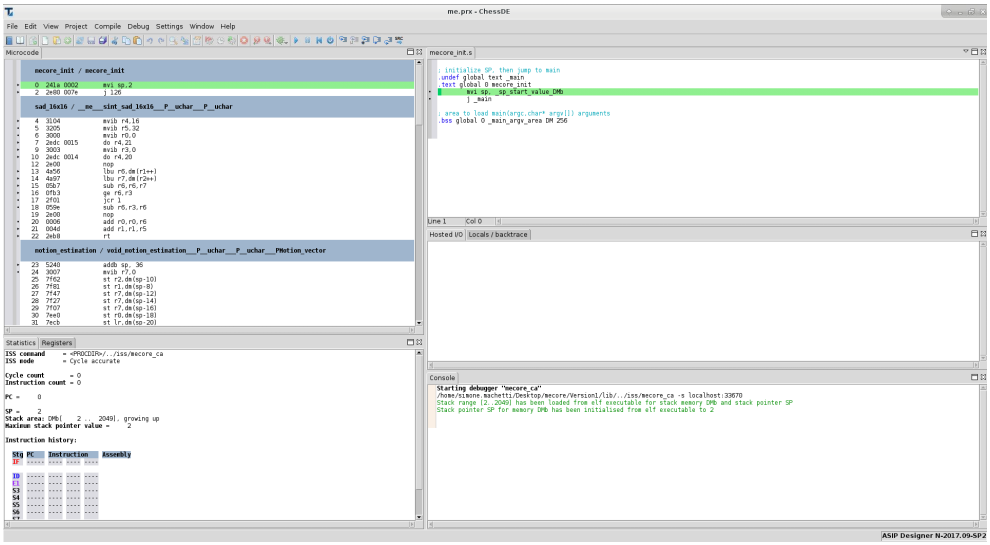


Figure 2.14: Simulation, Step 2A.

In the top menu click on the *Go/Continue* icon to run a simulation, i.e. to let the compiled source code run on the selected processor (figure 2.15).

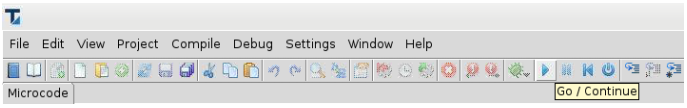


Figure 2.15: Simulation, Step 2B.

Once the simulation is completed, you can check the results printed to the standard output (if any) in the window *Hosted I/O*, or analyze some simulation statistics in windows *Statistics*, *Registers* and *Console*; window *Microcode* shows the assembly code with the corresponding encoding. See figure 2.16.

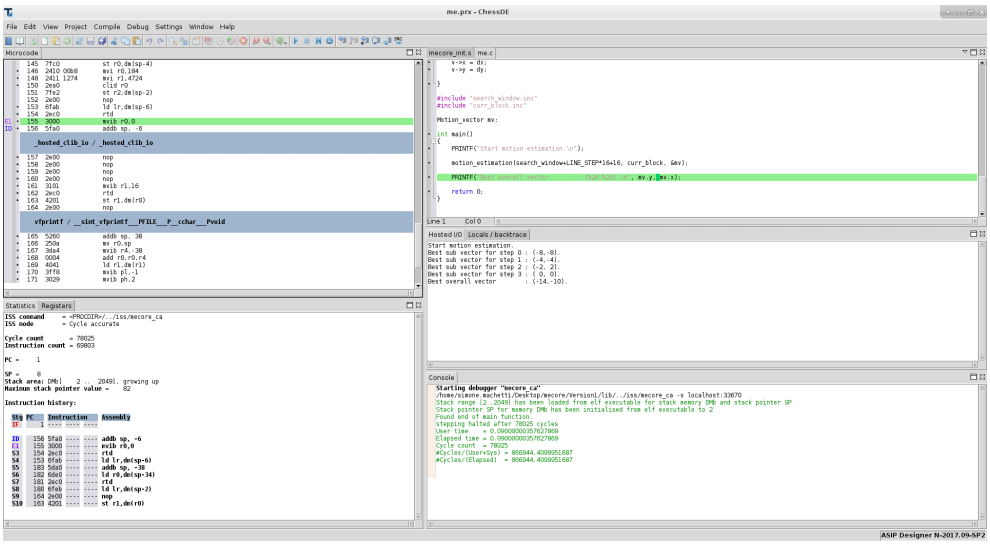


Figure 2.16: Simulation, Step 2C.

How to figure out the profiling information

The instruction profiling information can be obtained in two steps:

1. just after the simulation termination, in the top menu select *View* → *Profiling* (figure 2.17).

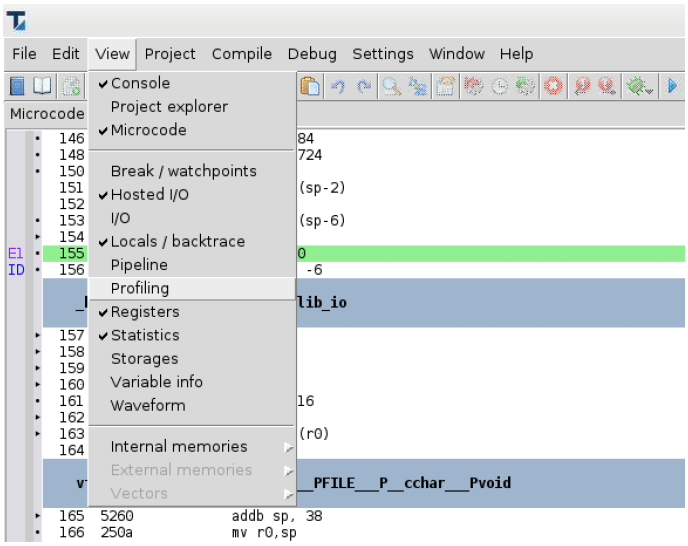


Figure 2.17: Figure out statistics, Step 1.

2. A new window, called *Profiling*, has been added to the graphical interface; in this window click on *Instruction report*; then, set *On* in the *Show user cycle count* field and click *Create report*. See figure 2.18. A file, whose name is specified in the *File name* field, is created and located into the project main directory. This file contains many useful statistics about the simulation.

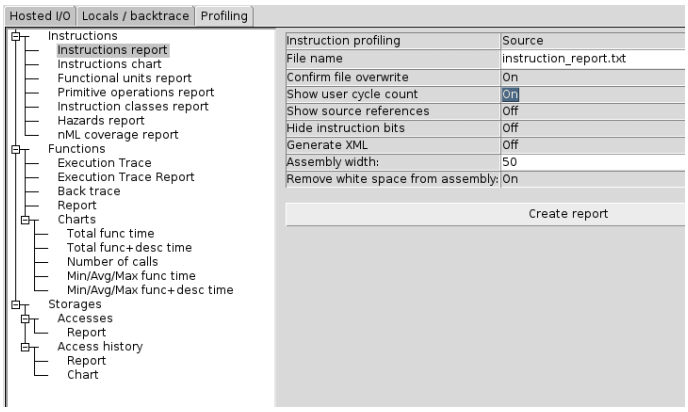


Figure 2.18: Figure out statistics, Step 2.

How to generate the VHDL code of a processor model

The VHDL code is generated in two steps:

1. open the VHDL project called *<processor_ name>_ vhdl.ptx*, located in the folder *.../hdl*, following steps 1 and 2 of the guide *How to compile a processor model*.

2. The project has been loaded; in the top menu click on *Make* to compile the project and generate the VHDL code of the processor model. See figure 2.19.

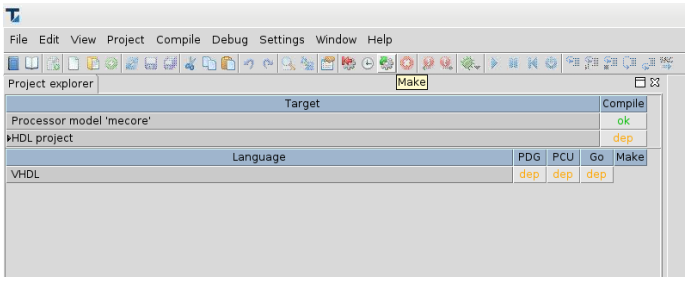


Figure 2.19: VHDL generation, Step 2.

The VHDL files have been generated and saved into the folder *hdl* → *<processor_ name>_ vhdl* → *<processor_ name>*.

Note that, the same procedure can be adopted to generate the Verilog files, with the only change of opening the Verilog project file, called *<processor_ name>_ vlog.ptx*, instead of the VHDL project file.

Chapter 3

Implementations

This chapter describes three versions of the MECORE microprocessor core optimized for Motion Estimation. In version 1 to 3, the performance of the core is improved by adding special features used to speed up the SAD computation.
The first section gives a general introduction of the MECORE core, while the following ones describe in detail each version. A final section is added in order to analyze and compare the obtained results.

NOTICE: The three versions of MECORE are implemented following the design steps provided by the Synopsys training manual [1] and they are not presented as original implementations.

3.1 mecore

MECORE is a 16 bit microprocessor core based on the same structure and instruction set of the TMICRO core, present in the ASIP Designer library, plus some additional features.
This section gives first a general overview of the TMICRO core, and then focuses on the new features supported by MECORE.

Tmicro

TMICRO is a general purpose 16 bit microcontroller designed by Synopsys and intended to be used as a starting point for the development of application specific instruction set processors. It contains a collection of architectural features that are common to most of the 16 bit microcontrollers, such as:

- 16 bit integer arithmetic, bitwise logical and compare instructions, executed on a 16 bit ALU and operating on an 8 field register file.
- Integer multiplication instructions with 16 bit operands and 32 bit results.
- 16 bit shift instructions.
- A 16 bit division instruction.
- Load and store instructions from and to a 16 bit data memory with an address space of 64k words, using indirect addressing and little endian format.
- Instruction fetch from a separate code memory with an address space of 64k words and little endian format.

- Various control instructions such as jumps, subroutine call and return.
- Pipelined architecture with three pipeline stages: IF, ID and E1.
- Zero overhead loops.
- Support for interrupts.
- Support for on chip debugging.

For more information on the TMICRO core, consult the reference manual [6].

Additional features

In video processing, pixel values are represented as unsigned bytes. As the original TMICRO does not support loading and storing of bytes in memory, this functionalities have been added in order optimize the memory usage and the computation process.

Nine new instructions have been added to the TMICRO instruction set. Figure 3.1 gives an overview of their assembler syntax and corresponding binary encoding.

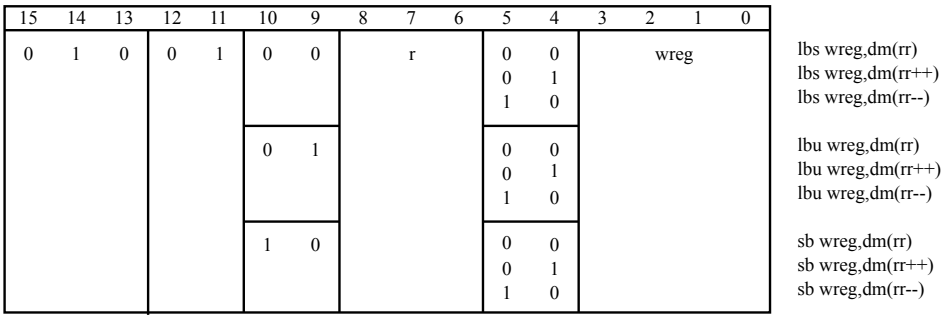


Figure 3.1: Byte load and store instructions.

The instructions use indirect addressing: the address is read from a field of the R register file and optionally post-incremented or post-decremented by one; then, the final address is used to access, i.e. read or write, the DMb data memory.

The *lbs* instruction performs a signed byte load operation. The byte value is sign extended at the MSB side, by the function *extend_sign*, before it is stored in the 16 bit destination register.

```
lbs : wreg ← extend_sign(DMb[R[r]])
```

The *lbu* instruction performs an unsigned byte load operation. The byte value is zero extended at the MSB, by the function *extend_zero*, before it is stored in the 16 bit destination register.

```
lbu : wreg ← extend_zero(DMb[R[r]])
```

The *sb* instruction performs a byte store operation. The eight least significant bits of the source register are extracted, by the function *extract_low*, and stored in the selected 8 bit memory location.

```
sb : DMb[R[r]] ← extract_low(wreg)
```

A new functional unit, called XS, is added to the processor architecture in order to implement the three functions *extend_sign*, *extend_zero* and *extract_low*.

From now on, this processor will be called MECORE and will be used as a starting point for the implementations presented in the following sections.
The zip file containing the three versions of the processor can be downloaded using the following link: ReferenceMaterial. Each version directory contains the following subdirectories:

- lib: contains the processor model files.
- iss: directory in which the ISS, Instruction Set Simulator, is built.
- hdl: directory in which the RTL model is generated.
- source_code: contains the C code implementing the Motion Estimation.

3.2 Version 1: pure software implementation

This version is a pure software implementation of the Motion Estimation function. The source code is executed on the MECORE processor making use of the existing instructions only. No additional hardware support is needed to implement this version.

3.2.1 Source code

The C source code is located in the folder *Version1* → *source_code* of the reference material. Inside this folder you can find three files:

- *curr_block.inc*: contains a sample 16x16 pixel block of a current frame.
- *search_window.inc*: contains a sample 48x48 pixel Search Window of a reference frame.
- *me.c*: contains the C code implementing the Motion Estimation algorithm.

The C program is based on two functions: *motion_estimation* and *sad_16x16*.

The *motion_estimation* function is used to explore the Search Window selecting, one by one, each block that has to be compared to the current one in order to find the best match.
The optimal way to determine the best matching block is to adopt the full search algorithm, previously described. However, the high computational complexity of this algorithm, makes it unreliable for most real-time applications.
For this reason, an approximate search algorithm, called *Three Step Search* and developed by Koda e.a. [7], has been used in order to speed-up the execution process. This algorithm searches for the best matching block in a coarse to fine search pattern.
The algorithm works as follows:

1. an initial step size is picked. Eight blocks at a distance of step size from the center block as well as the center block are picked for comparison.
2. The step size is halved and the center is moved to the point that has obtained the best match in Step 1.
3. Steps 1 and 2 are repeated until the step size becomes 1.

In the original description [7], steps 1 and 2 were repeated three times, hence the name of the algorithm. We will use a variant that computes the best motion vector in four iterations, using the following step sizes: 8, 4, 2, 1.
The C code that implements the algorithm is shown in figure 3.2.

```
static int x_steps[9] = { 0, 8, 8, 0, -8, -8, -8, 0, 8 };
static int y_steps[9] = { 0, 0, 8, 8, 8, 0, -8, -8, -8 };

struct Motion_vector { int x, y; };

void motion_estimation(unsigned char* search_window,
                       unsigned char* curr_block,
                       Motion_vector* v)
{
    int sad, min_sad;
    int dx=0, dy=0;

    for (int j = 0; j < 4; j++) { // displacements of 8, 4, 2, 1

        min_sad = 0x7fff;
        int best_x = 0;
        int best_y = 0;

        for (int i = 0; i < 9; i++) { // center and 8 surrounding points

            int x = x_steps[i] >> j;
            int y = y_steps[i] >> j;

            sad = sad_16x16(search_window + LINE_STEP * (y+dy) + (x+dx), curr_block);

            if (sad < min_sad)
            {
                min_sad = sad;
                best_x = x;
                best_y = y;
            }
        }

        PRINTF("Best sub vector for step %d : (%2d,%2d).\n", j, best_y, best_x);

        dx += best_x;
        dy += best_y;

    }

    v->x = dx;
    v->y = dy;
}
```

Figure 3.2: The *motion_ estimation* function.

The arguments of the function are:

- *search_ window*: a pointer to the first pixel, i.e. the top left pixel, of the central 16x16 block in the Search Window. The pixels of the Search Window are stored, row by row, in a linear array of size 48x48, defined in file *search_ window.inc* through the following data structure:

```
unsigned char search_window[48*48];
```

- *curr_ block*: a pointer to the first pixel of the current block. The pixels of the current block are stored, row by row, in a linear array of size 16x16, defined in file *curr_ block.inc* through the following data structure:

```
unsigned char curr_block[16*16];
```

- *v*: output data structure. The coordinates of the resulting Motion Vector are assigned to variables x and y, defined by the structure.

The *motion_ estimation* function is called by the main function as follows:

```
motion_estimation(search_window+LINE_STEP*16+16, curr_block,&mv);
```

Note that the first argument does not point to the beginning of the search_window array, but to the first pixel of the central block.
The code of the main function is shown in figure 3.3.

```
#include "search_window.inc"
#include "curr_block.inc"

Motion_vector mv;

int main()
{
    PRINTF("Start motion estimation.\n");

    motion_estimation(search_window+LINE_STEP*16+16, curr_block, &mv);

    PRINTF("Best overall vector     : (%2d,%2d).\n", mv.y, mv.x);

    return 0;
}
```

Figure 3.3: The *main* function.

The *sad_ 16x16* function is called by the *motion_ estimation* function through the following call:

```
sad = sad_16x16(search_window+LINE_STEP*(y+dy)+(x+dx), curr_block);
```

The first argument is the pointer to the top left pixel of the selected block in the *search_ window*, while the second argument is the pointer to the first pixel of the current block.
This function performs the sum of absolute differences between the two blocks and returns the result.

You can analyze its code in figure 3.4.

```
inline int ABS(int x) { return (x<0)?-x:x; }
const int LINE_STEP = 48;

static int sad_16x16(unsigned char* search_window, unsigned char* curr_block)
{
    int sad = 0;
    int s, c, diff;
    for (int i = 0; i < 16; i++) {
        for (int j = 0; j < 16; j++) {
            s = search_window[j];
            c = curr_block[j];
            diff = s - c;
            sad += ABS(diff);
        }
        curr_block += 16;
        search_window += LINE_STEP;
    }
    return sad;
}
```

Figure 3.4: The *sad_ 16x16* function, Version 1.

As you can see, the SAD computation is implemented in a pure software way, using two *for* loops to select each pair of pixels and compute the absolute difference between them.

3.2.2 Compilation and simulation

The processor model and the C program are compiled and a simulation is performed. The following results are printed to the standard output:

```
Start motion estimation.
Best sub vector for step 0 : (-8,-8).
Best sub vector for step 1 : (-4,-4).
Best sub vector for step 2 : (-2, 2).
Best sub vector for step 3 : ( 0, 0).
Best overall vector        : (-14,-10).
```

The results are compared with the expected ones in order to verify the correctness of the design.
Using the profiling tool, provided by ASIP Designer, important statistics are extracted.
The whole program contains 167 instructions, i.e. 386 bytes in program memory. It is simulated in 78025 clock cycles (cycle count) and the number of executed instructions (instruction count) is 69803. These global numbers can be partitioned among the three main functions of the program, see table 3.1.

FUNCTION	INSTR COUNT	% OF TOTAL	CYCLE COUNT	% OF TOTAL
sad_16x16	68388	97.97%	76356	97.86%
motion_estimation	1193	1.71%	1417	1.82%
main	20	0.03%	29	0.04%
others	200	0.28%	220	0.29%

Table 3.1: Functions profiling, Version 1.

As you can notice, almost the 98% of the total number of clock cycles is exploited to perform SAD computations. For this reason, the objective of the next design versions will be that of optimizing this function as much as possible: first, implementing the SAD in hardware, and then increasing the data parallelism through SIMD instructions.

The *sad_ 16x16* function is compiled into 17 assembly instructions, see table 3.2.

PC	ASSEMBLY	INSTRUCTION COUNT	CYCLE COUNT
4	mvib r4,16	36	36
5	mvib r5,32	36	36
6	mvib r0,0	36	36
7	do r4,21	36	72
9	mvib r3,0	36	36
10	do r4,20	576	1152
12	nop	576	576
13	lbu r6,dm(r1++)	9216	9216
14	lbu r7,dm(r2++)	9216	9216
15	sub r6,r6,r7	9216	9216
16	ge r6,r3	9216	9216
17	jcr 1	9216	16500
18	sub r6,r3,r6	5574	5574
19	nop	5574	5574
20	add r0,r0,r6	9216	9216
21	add r1,r1,r5	576	576
22	rt	36	108

Table 3.2: Assembly instructions of the *sad_ 16x16* function, Version 1.

Both *for* loops are mapped onto zero overhead hardware loops.
The outer loop, initiated by the *do* instruction at address 7, starts at instruction 10 and ends at 21. The *do* instruction has one delay slot (instruction 9), which the compiler has utilized to schedule a load byte immediate instruction.
The inner loop, initiated by the *do* instruction at address 10, starts at instruction 13 and ends at 20. Note that the compiler could not find a useful operation to schedule in the delay slot at address 12, and inserted a *nop*.
The absolute value computation is implemented using a conditional jump. Instruction 15 computes the difference between the current block and Search Window pixels, while instruction 16 compares the difference against zero: when the difference is positive or zero, the conditional jump to instruction 20 is taken; in case the difference is negative, the jump is not taken and instruction 18 negates the difference.
The *nop* at address 19 is inserted due to a software stall rule that states that a conditional jump must be located at least 4 instructions before the hardware loop end address. This stall rule ensures that the jump is taken prior to the end of loop tests.

As you can notice, the assembly instructions that are executed an higher number of times are those involved in the load of each pair of pixel values and in the computation of the sum of absolute difference between them, i.e. instructions from address 13 to 20.

3.2.3 Synthesis and physical design

Synopsys Design Compiler has been used in order to synthesize the first version of MECORE. The synthesis has been performed and optimized, to archive the maximum operating frequency, using the script shown in figure 3.5.

```
#Analyze internal modules:
......

#Analyze MECORE top-level entity:
analyze -library WORK -format vhdl {mecore.vhd}

#Elaborate MECORE top-level entity:
elaborate mecore -architecture structural -library WORK

#Create clock signal and set dont touch on it:
create_clock -name MY_CLOCK -period 0.00 clock
set_dont_touch_network MY_CLOCK

#Compile design:
compile -map_effort high

#Flat architecture using Verilog rules:
ungroup -all -flatten
change_names -hierarchy -rules verilog

#Write architecture in Verilog:
write -hierarchy -format verilog -output ./mecore.v

#Write .sdc file:
write_sdc ./mecore.sdc

#Write reports:
report_timing >> timingReport.txt
report_area >> areaReport.txt
report_power >> powerReport.txt
```

Figure 3.5: Synthesis script of MECORE.

The obtained results are presented here:

```
Critical path   :   1.43 ns
Max. frequency  :   699.30 MHz
Area            :   22671.72 um^2
Dynamic power   :   51.03 uW
Leakage power   :   2.06 uW
```

After the synthesis, the design has been placed and routed, using Innovus by Cadence, in order to evaluate the area:

```
Gates   :   22557
Cells   :   13779
```

```
Area    :   18001.00 um^2
```

Note that, the synthesis has been performed using a 65 nm library, called *uk65lscllmvbbr_120c2_tc*, while for the physical design, the *NangateOpenCellLibrary* library has been exploited.

3.3 Version 2: hardware accelerator

In this second version of the design, a special purpose instruction is added to the MECORE instruction set to compute the absolute value of the difference of two values and accumulate the result. Moreover, this absolute difference computation is combined with two parallel load operations in order to speed up the SAD execution.

3.3.1 Hardware support

This subsection describes in detail the procedure used to implement the second version of the design. The development is explained step by step **using the first version as a starting point**.

Add the adiff primitive function

The *adiff* primitive function is added to the header file *mecore.h*.

```
word adiff(word,word) property(commutative);
```

This function receives two input parameters of type *word* (16 bits), computes the absolute value of the difference of the two values and returns a result of type *word*. The *commutative* annotation tells the compiler that it can switch the two inputs of the function as needed, to obtain a better compilation result.

The function implementation is added to the file *mecore.p*, containing all the function definitions.

```
word adiff(word a, word b) {
    word diff = a - b;
    return diff < 0 ? -diff : diff;
}
```

In order to be able to call the *adiff* function as an intrinsic function in the source code, the following promotion is added to the header file *mecore_int.h*.

```
promotion int adiff(int,int) = word adiff(word,word);
```

It allows the function to be called with arguments of type *int*.

Add the new instructions

In order to implement the *adiff* primitive function in hardware and use the implemented functional unit, two new instructions are added to the MECORE instruction set. The first one computes the absolute difference of two values and accumulates the result, while the second one combines the absolute difference computation with two parallel unsigned byte load operations.

In order to add the new set of instructions to the MECORE instruction set, a new entry, called *adiff_instr*, is inserted into the main nML description of the instruction grammar rules, located

in the file *mecore.n*. This piece of code lists the names of the groups of instructions composing the MECORE instruction set.

```
opn mecore( alu_instr
          | shift_instr
          | mult_instr
          | move_instr
          | load_store_instr
          | control_instr
          | adiff_instr //added line
          )
{
    image : alu_instr
          | shift_instr
          | mult_instr
          | move_instr
          | load_store_instr
          | control_instr
          | adiff_instr //added line
          ;
}
```

Each group is then described in a file with the same name, that is included in the design in the last part of the file *mecore.n*. In this case, we include the file *adiff.n*.

```
#include "adiff.n"
```

This file must be added to the folder *Version2* $\rightarrow$ *lib*, in order to be correctly included during the compilation of the processor.

The detailed description of the content of the file is presented here.

The first portion of code tells to the compiler that two instructions, identified by the *adiff_ add_ adiff_ rrr* and the *adiff_ parr_ instr* rule names, are defined in this group.

```
opn adiff_instr(adiff_add_adiff_rrr | adiff_parr_instr)
{
       image
      : "100"::adiff_add_adiff_rrr
      | "100"::adiff_parr_instr
      ;
}
```

The *image* attribute is then used to define the binary encoding for the first part, i.e. the most significant three bits, of the corresponding instruction. In this case, both instructions share the same binary code "100".

Each instruction is now analyzed separately.

SAD computation

The functional unit implementing the *adiff* primitive function is now defined, together with the input and output signals used to connect the unit to the other modules of the datapath.

```
fu adiff;
trn adiffr<word>; //input signal
trn adiffs<word>; //input signal
trn adifft<word>; //input signal
trn adiffu<word>; //internal signal
trn adiffv<word>; //output signal
```

The instruction behavior is described using the following nML code:

```
opn adiff_add_adiff_rrr(t: c3u, r: c3u, s: c3u)
{
    action {
        stage E1:
           adiffu = adiff(adiffr=rre1=R[r],adiffs=rse1=R[s]) @adiff;
           R[t] = rte1 = adiffv = add(adiffu,adifft=rue1=R[t]) @adiff;
    }
    syntax : "aad r"t ",r"r ",r"s;
    image  : "0000"::t::r::s;
}
```

Three parameters, called t, r and s, are needed by this instruction. They are three addresses: the last two (r and s) are used to address the register file and read the two input values, while the first one (t) is used both to address and read the accumulation value from the register file, and to write back the final result of the SAD to the same register.
The *action* attribute specifies the sequence of operations performed by the instruction. They are also graphically explained in figure 3.6.

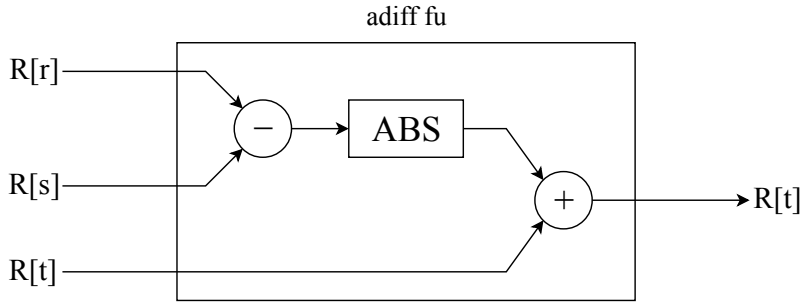


Figure 3.6: Behavior of the *adiff_ add_ adiff_ rrr* rule.

As you can see, the values stored in the two selected registers (R[r] and R[s]) are subtracted and the absolute value of the result is computed. After that, the resulting value is added to the accumulation value, stored in register R[t], and the result is saved back to the same register (R[t]) to continue the accumulation. The primitive functions *adiff()* and *add()* are used to implement the operations. This sequence of computations is assigned to the ADIFF functional unit (fu) using the *@adiff* code.

In order to execute this instruction, an additional read port has to be added to the register file to allow three reading operations and one writing operation in the same clock cycle. The additional read port, called *rue1*, is added to the following piece of code, located in the file *mecore.n*.

```
reg R[8]<word,uint3>
syntax ("R")
    read( rrid        //ID stage read port
```

```
rre1 rse1 rue1)   //E1 stage read ports - added rue1
write(rtid        //ID stage write port
rte1);            //E1 stage write port
```

This nML code declares the register file R, containing eight fields of type *word* (16 bits), and specifies its read and write access ports. Port *rue1* is added to the existing ports *rre1* and *rse1*. These ports can be only used in the E1 pipeline stage, as the *adiff* instruction does.

The *syntax* attribute specifies the assembly syntax for the instruction, where *rt*, *rr* and *rs* are the names of the corresponding source and destination registers, e.g. r0, r1, r2, ecc.

```
aad rt,rr,rs
```

The *image* attribute defines the binary encoding for the second part, i.e the least significant thirteen bits, of the instruction. The :: operator is used to concatenate the different bit strings. The encoding is so defined:

```
"0000" :: t :: r :: s
```

where *t*, *r* and *s* are 3 bit addresses used to specify the corresponding source or destination registers. The complete encoding of the instruction, taking into account the additional bit part specified in the *image* attribute of the *adiff_instr* rule, is presented here:

```
"100" :: "0000" :: t :: r :: s
```

The first new instruction has been fully described.

Parallel loads and SAD computation

Two additional functional units, with the corresponding input and output signals, are defined.

```
fu ag2;
trn ag2p<word>;
trn ag2m<word>;
trn ag2q<word>;

fu xs2;
trn wbus2<word>;
```

The AG2 fu is a second address generation unit (the first one, called AG1, is already declared in the file *load_store.n*). We need this additional unit to implement two parallel byte load operations from the data memory: each address generation unit takes care of computing the right read address, post-incrementing the value coming from the selected source register.
The XS2 fu is a second extension unit (the first one, called XS1, is already declared in the file *load_store.n*). Once the two parallel byte load operations are executed, the two fetched bytes are zero extended from 8 to 16 bits in order to be stored in the corresponding destination registers. This extensions are performed by the two extension units XS1 and XS2.

In order to access the data memory two times in the same clock cycle, an additional read port is needed. The new port is added to the nML data memory declaration in the file *mecore.n*.

```
def  dm_size = 2**16;  // data memory
mem  DMb[dm_size, 1]<w8,  addr> access {
    dmb_ld‘ID‘: dmb_read‘E1‘ = DMb[dm_addr‘ID‘]‘ID‘;
```

```
    dmb_ld2‘ID‘: dmb_read2‘E1‘ = DMb[dm_addr2‘ID‘]‘ID‘; //added line
    dmb_st‘E1‘: DMb[dm_addr‘E1‘]‘E1‘ = dmb_write‘E1‘;
};
```

This piece of code is used to declare the data memory DMb, of size 2^{16} bytes, and its read and write ports. As you can see, a new port, called *dmb_ld2*, in added to the already existing ports. The signal *dm_addr2* is the address signal used to access the data memory, while the signal *dmb_read2* is the output signal of the memory containing the read value. The *ID* and *E1* control words are used to specify the timing of a reading operation:

```
dmb_ld2‘D‘: dmb_read2‘C‘ = DMb[dm_addr2‘A‘]‘B‘;
```

A specifies in which cycle the address is on the address bus, B states in which cycle the memory is accessed, C specifies in which cycle the data is on the read bus and D states in which cycle the reading operation starts. In this case, in the ID stage the address is on the address bus and the memory is accessed, while in the E1 stage the fetched data is on the read bus.

The instruction behavior is described by three nML rules: *load_R0*, *load_R1* and *add_adiff_opn*.

To simplify the code, an alias is associated to each of the used registers:

```
reg R0<word> alias R[0];
reg R1<word> alias R[1];
reg R2<word> alias R[2];
reg R3<word> alias R[3];
reg R4<word> alias R[4];
```

The *load_R0* rule has the following code:

```
opn load_R0()
{
    action {
        stage ID:
            R3 = rtid = ag1q = add(ag1p=rrid=R3,ag1m=1) @ag1;
        stage ID..E1:
            dm_addr‘ID‘ = ag1p‘ID‘;
            dmb_read‘E1‘ = DMb[dm_addr‘ID‘]‘ID‘;
            R0‘E1‘ = rle1 = wbus‘E1‘ = extend_zero(dmb_read‘E1‘) @xs;
    }
    syntax : "ld r0,dm(r3++)";
}
```

It is used to implement one of the two load operations.
The *action* attribute specifies the sequence of performed operations. In the ID stage, the value stored in the register R3 is sent to the address bus and the data memory is accessed. In the same clock cycle, the value is post-incremented by the first address generation unit AG1 and written back to the same destination register R3, ready for the next reading operation in the following clock cycle. In the E1 stage, the fetched byte is zero extended by the extension unit XS and stored into the register R0.
As you can notice, the number of possible operands is restricted to a single source register, R3, and a single destination register, R0. The reason for that is the need of encoding three parallel actions in a single 16 bit instruction word, as you will see at the end of this subsection.
The operations are graphically explained in figure 3.7.

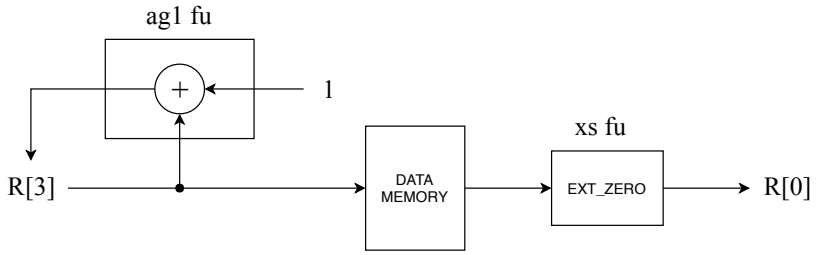


Figure 3.7: Behavior of the *load_R0* rule.

The *syntax* attribute specifies the assembly syntax for the rule:

```
ld r0,dm(r3++)
```

The second load operation *load_R1* is described by the following code:

```
opn load_R1()
{
    action {
        stage ID:
           R4 = rt2id = ag2q = add(ag2p=rr2id=R4,ag2m=1) @ag2;
        stage ID..E1:
           dm_addr2‘ID‘ = ag2p‘ID‘;
           dmb_read2‘E1‘ = DMb[dm_addr2‘ID‘]‘ID‘;
           R1‘E1‘ = rke1 = wbus2‘E1‘ = extend_zero(dmb_read2‘E1‘) @xs2;
    }
    syntax : "ld r1,dm(r4++)";
}
```

The *action* attribute specifies the same sequence of operations performed by the previous load rule, but using the second address generation unit AG2 and the second extension unit XS2. The source and destination registers are also changed: the load address is read from the register R4 and the fetched value is stored into the register R1.
In figure 3.8, you can analyze the behavior of the rule in detail.

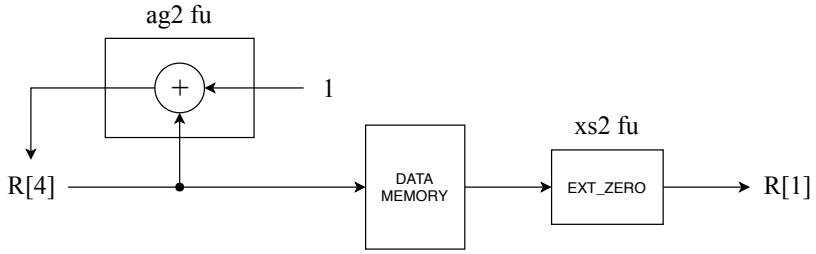


Figure 3.8: Behavior of the *load_R1* rule.

The assembly syntax of the rule, specified by the *syntax* attribute, is the following:

```
ld r1,dm(r4++)
```

As you can see, the same syntax of the previous rule is used, but changing the source and destination registers to R4 and R1, respectively.

The third nML rule is the *add_ adiff_ opn* described by the following code:

```
opn add_adiff_opn()
{
    action {
        stage E1:
            adiffu = adiff(adiffr=rre1=R0,adiffs=rse1=R1) @adiff;
            R2 = rte1 = adiffv = add(adiffu,adifft=rue1=R2) @adiff;
    }
    syntax : "aad r2,r0,r1";
}
```

This rule computes the same operations performed by the *aad* instruction, previously described, but restricting the possible operands and result registers to single registers.
The *action* attribute describes the rule behaviour: two operands, stored in the registers R0 and R1 of the register file R are read and the absolute value of their difference is evaluated; after that, the result is added to a third operand, stored in the register R2, and the final value is written back to the same register, R2.
Figure 3.9 explains the behavior in detail.

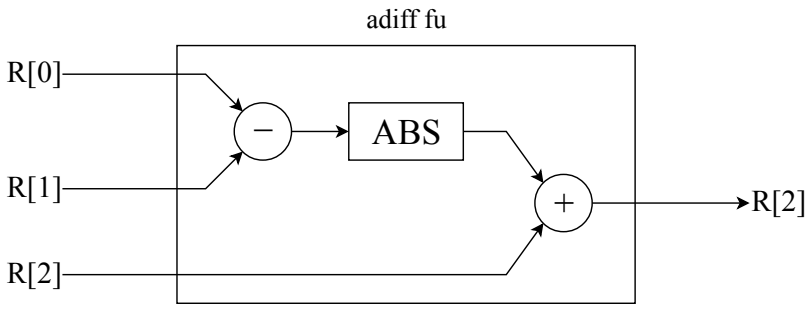


Figure 3.9: Behavior of the *add_ adiff_ opn* rule.

Finally, the rule *adiff_ parr_ instr* is used to group the previous three rules in a single instruction. This is performed by the following piece of code:

```
opn adiff_parr_instr(ad   : add_adiff_opn,
                     ld0  : load_R0,
                     ld1  : load_R1)
{
    action { ad; ld0; ld1; }
    syntax : ad " | " ld0 " | " ld1;
    image  : "1111"::"000"::"000"::"000";
}
```

The *action* attribute specifies that the three rules, *add_ adiff_ opn*, *load_ R0* and *load_ R1*, have to be executed together. So, their behaviours will be combined: two load operations are performed in parallel, using as addresses the values stored in the registers R3 and R4; then, the two fetched values are zero extended and written to the registers R0 and R1, respectively; after that, the two registers and the accumulation register R2 are read and their sum of absolute difference is computed, as described previously; the final result is then written back to the register R2.
Analyze figure 3.10.

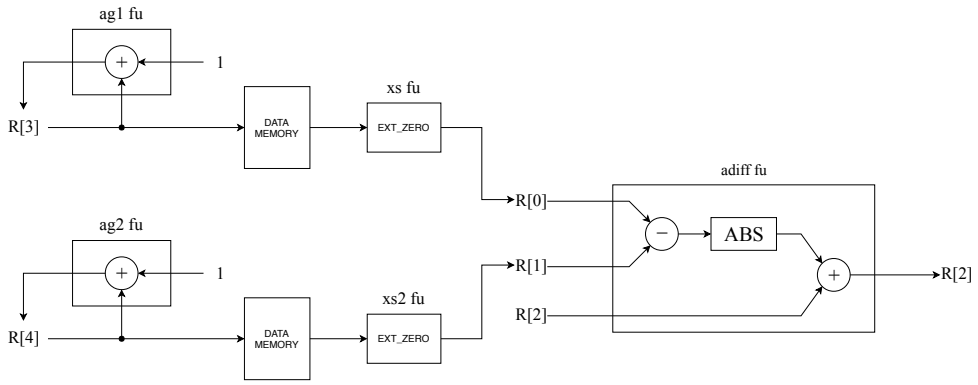


Figure 3.10: Behavior of the *adiff_parr_instr* rule.

In order to execute this instruction, some additional ports have to be added to the register file R in order to allow:

- 2 reading operations in the ID stage → R3 and R4
- 3 reading operations in the E1 stage → R0, R1 and R2
- 2 writing operations in the ID stage → R3 and R4
- 3 writing operations in the E1 stage → R0, R1 and R2

The additional ports are specified in the register file description, located in the file *mecore.n*.

```
reg R[8]<word,uint3>
syntax ("R")
    read( rrid rr2id  //ID stage read port - added rr2id
    rre1 rse1 rue1)   //E1 stage read ports
    write(rtid rt2id  //ID stage write port - added rt2id
    rte1 rle1 rke1);  //E1 stage write port - added rle1 and rke1
```

As you can see, four ports have been added: *rr2id*, *rt2id*, *rle1* and *rke1*.

The *syntax* attribute specifies that the assembly syntax of the instruction is obtained combining the assembly syntaxes of the three rules: *add_adiff_opn*, *load_R0* and *load_R1*. So, the following result is obtained:

```
aad r2,r0,r1 | ld r0,dm(r3++) | ld r1,dm(r4++)
```

The *image* attribute defines the binary encoding for the second part, i.e. the least significant thirteen bits, of the instruction.

```
"1111" :: "000" :: "000" :: "000"
```

Now, the bits of the first part are added and the following result is obtained:

```
"100" :: "1111" :: "000" :: "000" :: "000"
```

The second new instruction has been fully described.

3.3.2 Source code

The C source code is located in the folder *Version2* → *source_ code*.
In order to use the new instructions, the function *sad_ 16x16* has to be properly modified: in the body of the inner for loop, the two lines of code implementing the sum of absolute difference are substituted with a single line calling the *adiff* primitive function, previously described.
The new version of the code can be analyzed in figure 3.11.

```
const int LINE_STEP = 48;

static int sad_16x16(unsigned char* search_window, unsigned char* curr_block)
{
    int sad = 0;
    int s, c;
    for (int i = 0; i < 16; i++) {
        for (int j = 0; j < 16; j++) {
            s = search_window[j];
            c = curr_block[j];
            sad += adiff(s,c);
        }
        curr_block += 16;
        search_window += LINE_STEP;
    }
    return sad;
}
```

Figure 3.11: The *sad_ 16x16* function, Version 2.

3.3.3 Compilation and simulation

The processor model and the C program are compiled and a simulation is performed. The obtained results are equal to those of the first version, and this guarantees the correctness of the design.

Using the profiling tool, provided by ASIP Designer, important statistics are extracted. The compiled program contains 165 instructions, i.e. 384 bytes in program memory. The simulation is performed in 14161 clock cycles (cycle count) and the number of executed instructions (instruction count) is 13259. In table 3.3, these global numbers are partitioned among the three main functions of the program.

FUNCTION	INSTR COUNT	% OF TOTAL	CYCLE COUNT	% OF TOTAL
sad_16x16	11844	89.33%	12492	88.21%
motion_estimation	1193	9.00%	1417	10.01%
main	20	0.15%	29	0.20%
others	200	1.51%	220	1.56%

Table 3.3: Functions profiling, Version 2.

The sad_16x16 function is compiled into 15 assembly instructions, see table 3.4.

PC	ASSEMBLY	INSTRUCTION COUNT	CYCLE COUNT
4	mv r4,r2	36	36
5	mv r3,r1	36	36
6	mvib r2,0	36	36
7	mvib r5,15	36	36
8	mvib r6,32	36	36
9	doi 16,18	36	108
12	lbu r1,dm(r4++)	576	576
13	do r5,16	576	1152
15	lbu r0,dm(r3++)	576	576
16	aad r2,r0,r1 |		
	ld r0,dm(r3++) |		
	ld r1,dm(r4++)	8640	8640
17	add r3,r3,r6	576	576
18	aad r2,r0,r1	576	576
19	rtd	36	36
20	mv r0,r2	36	36
21	nop	36	36

Table 3.4: Assembly instructions of the *sad_16x16* function, Version 2.

Both for loops are mapped onto zero overhead hardware loops.
The outer loop, initiated by the *doi* instruction at address 9, starts at instruction 12 and ends at 18. Note that no delay slot is generated by the *doi* instruction.
The inner loop, initiated by the *do* instruction at address 13, starts and ends at instruction 16, i.e. it is compiled into a single parallel instruction. An optimization, known as *loop folding*, has been used by the compiler to archive this result. This technique created a preamble consisting of two load instructions, 12 and 15, and a post amble consisting of instruction 18. Note that instruction 15 is scheduled in the delay slot of the *do* instruction.

3.3.4 Synthesis and physical design

The second version of MECORE has been synthesized using Synopsys Design Compiler. The synthesis has been performed and optimized, to archive the maximum operating frequency.

The obtained results are presented here:

```
Critical path   :   1.44 ns
Max. frequency  :   694.44 MHz
Area            :   24453.00 um^2
Dynamic power   :   52.92 uW
Leakage power   :   2.14 uW
```

After the synthesis, the design has been placed and routed, using Innovus by Cadence, in order to evaluate the area:

```
Gates   :   23794
Cells   :   14715
Area    :   18988.10 um^2
```

3.4 Version 3: SIMD architecture

In this third version of the design, the MECORE processor is extended with SIMD instructions. SIMD stands for Single Instruction Multiple Data and is a form of data parallelism. A SIMD instruction processes the elements of a vector simultaneously. All elements are processed in an identical way, as specified by the single instruction. In order to be processed, the vectors are fetched from the data memory and stored in a vector register file.

3.4.1 Hardware support

This subsection describes in detail the procedure used to implement the third version of the design. The development is explained step by step **using the first version as a starting point**.

Add the v16w8 primitive data type

The *v16w8* data type is added to the header file *mecore.h*.

```
class v16w8 property( vector w8[VSIZE] );
```

This new data type is defined to model a SIMD vector composed of 16 words of 8 bits each (128 bits in total). This size has been chosen in such a way that a complete line of the current block (16 bytes) can be captured in a single vector.
The value of the constant VSIZE must be specified in the file *mecore_config.h*, adding the following line of code:

```
#define VSIZE 16
```

In order to be able to use this definition, the header file (*mecore_config.h*) must be included in all the files using the constant. So, insert the following line at the beginning of the files *mecore.h* and *mecore.p* (used later on):

```
#include "mecore_config.h"
```

Add an alias data memory for vectors

An alias of the data memory is defined in order to be able to fetch or store 16 bytes (a whole vector) with a single load or store operation. The alias is specified in the memory declaration section of the the file *mecore.n*.

```
mem DMv[dm_size, 16]<v16w8,addr> alias DMb access {
dmv_ld‘ID‘: dmv_read‘E1‘ = DMv[dm_addr‘ID‘]‘ID‘;
dmv_st‘E1‘: DMv[dm_addr‘E1‘]‘E1‘ = dmv_write‘E1‘;
};
```

This piece of code declares the memory DMv as an alias of the data memory DMb, and allows to read and write to the memory 16 bytes at the time (a vector).
As you can see, two ports are specified: *dmv_ld* and *dmv_st*. The former port is used for loading operations: in the ID stage, the read address is on the address bus and the memory is accessed, while in the E1 stage, the read vector is on the data bus. The latter port is used for storing operations and the writing process is performed in the E1 stage only.
The *dm_addr* signal is the address signal, while the *dmv_read* and *dmv_write* signals are the read data signal and the write data signal, respectively.

Add a vector register file

Once vectors are fetched from the data memory, they need to be stored in a appropriate data structure. For this reason, a vector register file is defined. Its definition is located in the file *mecore.n*.

```
reg V[4]<v16w8,uint2>;
```

As you can see, the register file V is able to store four vectors, 128 bits each. A two bit signal is used to address the register file and select the proper vector.

Add the needed primitive functions

In order to properly manipulate vectors and compute the SAD, some important primitive functions have to be specified (their usage will be explained later on).

The new primitive functions are defined in the file *mecore.h*.

```
v16w8 vadiff(v16w8,v16w8) property(commutative);
word  vsum(v16w8);
addr  force_align(addr);
v16w8 valign(word,v16w8,v16w8);
```

The *vadiff* function receives, as input arguments, two vectors and computes the absolute difference of each pair of elements in the same position along them. The result of each computation is stored in an output vector of the same size, see figure 3.12.

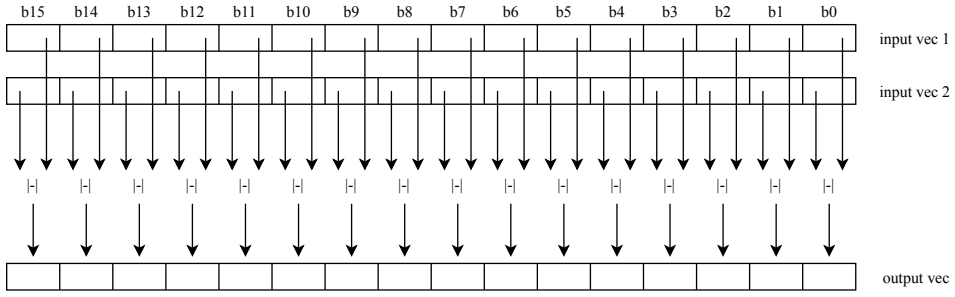


Figure 3.12: Behavior of the *vadiff* primitive function.

The function behavior is defined in the file *mecore.p* through the following code:

```
uint8_t absdiff(uint8_t a, uint8_t b)
{
    return a<b ? b-a : a-b;
}
v16w8 vadiff(v16w8 a,v16w8 b)
{
    v16w8 result;
    for (int32_t i = 0; i < VSIZE; i++)
    result[i] = absdiff(a[i], b[i]);

    return result;
}
```

The *vsum* function sums the elements of a vector, passed as argument, and returns a result of type *word*. The behavior can be analyzed in figure 3.13.

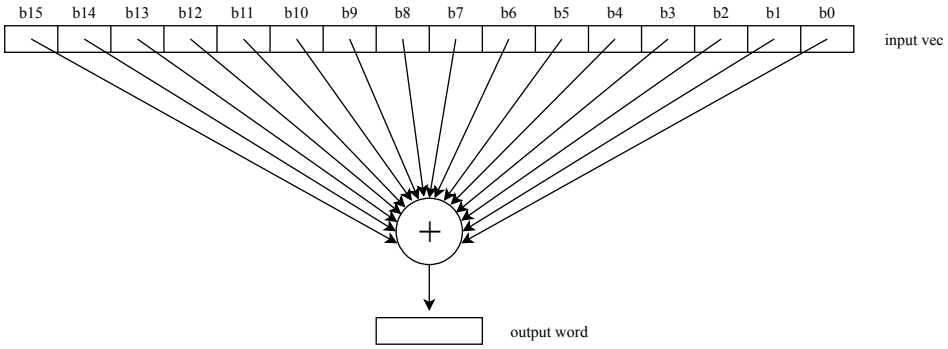


Figure 3.13: Behavior of the *vsum* primitive function.

The function implementation is added to the file *mecore.p*.

```
word  vsum(v16w8 a)
{
    return (word)((uint12_t)((uint11_t)((uint10_t)((uint9_t)a[15] + (uint9_t)a[14]) +
                  (uint10_t)((uint9_t)a[13] + (uint9_t)a[12])) +
                  (uint11_t)((uint10_t)((uint9_t)a[11] + (uint9_t)a[10]) +
                  (uint10_t)((uint9_t)a[ 9] + (uint9_t)a[ 8]))) +
                  (uint12_t)((uint11_t)((uint10_t)((uint9_t)a[ 7] + (uint9_t)a[ 6]) +
                  (uint10_t)((uint9_t)a[ 5] + (uint9_t)a[ 4])) +
                  (uint11_t)((uint10_t)((uint9_t)a[ 3] + (uint9_t)a[ 2]) +
                  (uint10_t)((uint9_t)a[ 1] + (uint9_t)a[ 0])));
}
```

The *force_ align* function sets the 4 least significant bits of an address, passed as argument, to zero and returns the result. It is generally used to force an aligned load operation from the data memory when the address is not aligned. The following line of code is added to the file *mecore.p* and defines the function implementation:

```
addr  force_align(addr a)
{
    return a[15:4] :: "0000";
}
```

The *valign* function is used to solve the problem of not aligned memory load operations. When loading a vector of pixels from the Search Window, it is often the case that this vector is not aligned to a 16 bit address boundary. The reason is that step values of 8, 4, 2 and 1 can create unaligned addresses, even though the *search_ window* array itself is properly aligned.

An example will clarify the concept.

Suppose you need to load a vector that starts at address 4 and ends at 19 (see figure 3.14). To do that, two aligned loads of adjacent vectors have to be performed:

- of V0 at the aligned address 0,
- of V1 at the aligned address 16.

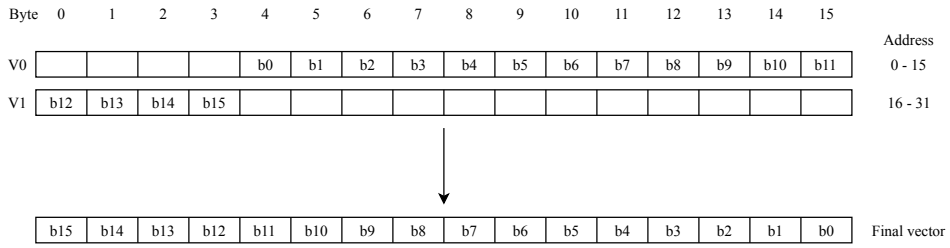


Figure 3.14: Example of the *valign* primitive function.

The vector is then reconstructed as follows:

- elements b0...b11 are extracted from V0 and copied to the lower part of the vector;
- elements b12...b15 are extracted from V1 and copied to the higher part of the vector.

The primitive function *valign* is used to model the re-packing of the final vector. The function has three input arguments: two vector operands of type *v16w8* and a value of type *word*. The latter is used to determine how the operands are re-packed, i.e it specifies the position of the least significant byte of the vector that has to be re-packed, in the first aligned vector (in the previous example, the first byte (b0) is located in position 4 in the vector V0 and so the third argument would be 4).

The function definition is added to the file *mecore.p*.

```
v16w8 valign(word sh,v16w8 aa,v16w8 bb)
{
    uint128_t a = (uint128_t)aa;
    uint128_t b = (uint128_t)bb;
    uint4_t sh4 = sh;
         if (sh4== 0) return aa;
    else if (sh4== 1) return (v16w8)(b[ 1*8-1:0]::a[127:1*8]);
    else if (sh4== 2) return (v16w8)(b[ 2*8-1:0]::a[127:2*8]);
    else if (sh4== 3) return (v16w8)(b[ 3*8-1:0]::a[127:3*8]);
    else if (sh4== 4) return (v16w8)(b[ 4*8-1:0]::a[127:4*8]);
    else if (sh4== 5) return (v16w8)(b[ 5*8-1:0]::a[127:5*8]);
    else if (sh4== 6) return (v16w8)(b[ 6*8-1:0]::a[127:6*8]);
    else if (sh4== 7) return (v16w8)(b[ 7*8-1:0]::a[127:7*8]);
    else if (sh4== 8) return (v16w8)(b[ 8*8-1:0]::a[127:8*8]);
    else if (sh4== 9) return (v16w8)(b[ 9*8-1:0]::a[127:9*8]);
    else if (sh4==10) return (v16w8)(b[10*8-1:0]::a[127:10*8]);
    else if (sh4==11) return (v16w8)(b[11*8-1:0]::a[127:11*8]);
    else if (sh4==12) return (v16w8)(b[12*8-1:0]::a[127:12*8]);
    else if (sh4==13) return (v16w8)(b[13*8-1:0]::a[127:13*8]);
    else if (sh4==14) return (v16w8)(b[14*8-1:0]::a[127:14*8]);
    else /*(sh4==15)*/return (v16w8)(b[15*8-1:0]::a[127:15*8]);
}
```

Add the new instructions

In order to use the new primitive functions, five new instructions are added to the MECORE instruction set. To do that, a new entry, called *vector_ instr*, is inserted into the nML description of the instruction grammar rules, located in the file *mecore.n*.

```
opn mecore( alu_instr
          | shift_instr
          | mult_instr
          | move_instr
          | load_store_instr
          | control_instr
          | vector_instr //added line
          )
{
    image : alu_instr
          | shift_instr
          | mult_instr
          | move_instr
          | load_store_instr
          | control_instr
          | vector_instr //added line
          ;
}
```

The new group of instructions is defined in the file *vector.n*, that has to be added to the directory *Version3* $\rightarrow$ *lib* and included in the design at the end of the file *mecore.n*, using the following line of code:

```
#include "vector.n"
```

The detailed description of the content of the file is presented here.

The first portion of code defines two sub-groups of instructions, identified by the *load_ store_ vreg_ sp_ indexed* and the *vector_ single_ instr* rules.

```
opn vector_instr(load_store_vreg_sp_indexed | vector_single_instr)
{
       image
       : "111"::load_store_vreg_sp_indexed
       | "110"::vector_single_instr
       ;
}
```

The *image* attribute specifies the binary encoding for the first part, i.e. the most significant three bits, of the corresponding instruction group.

Each instruction group is now analyzed separately.

Vector register spilling and filling

The first group is composed of a single instruction, used to spill and fill the vector registers of V to and from the stack memory, respectively.
This instruction is defined by the following nML code:

```
trn ag1_addr<addr>;
opn load_store_vreg_sp_indexed(ls: load_store_op, v: c2u, offs: c9n)
{
    action {
        stage ID:
```

```
            ag1_addr = ag1q = add(ag1p=SP,ag1m=offs) @ag1;
        stage ID..E1:
            switch (ls) {
                case ld:
                    dm_addr‘ID‘ = ag1_addr‘ID‘;
                    V[v]‘E1‘ = dmv_read‘E1‘ = DMv[dm_addr‘ID‘]‘ID‘;
                case st:
                    dm_addr_pipe‘ID‘ = ag1_addr‘ID‘;
                    dm_addr‘E1‘ = dm_addr_pipe‘E1‘;
                    DMv[dm_addr‘E1‘]‘E1‘ = dmv_write‘E1‘ = V[v]‘E1‘;
            }
    }
    syntax : ls " v" v ",dm(sp" offs ")" ;
    image  : ls::offs[one 7..0]::"00"::v;
}
```

Three parameters are needed: *ls*, *v* and *offs*. *ls* is used to select the type of operation, i.e. load or store, through the switch statement. *v* is adopted to address the vector register file V and read or write the needed vector. *offs* is the offset that is added to the Stack Pointer (SP) in order to obtain the required address.

The *action* attribute defines the performed operations. In case a load from the stack has to be performed, the Stack Pointer and the offset are added by the address generation unit AG1 and the resulting address is used to access the stack, located in the data memory; this is performed in the ID stage. In the E1 stage, the read vector is propagated on the memory data bus and written into the vector register file, using the parameter *v* as address. In case a store to the stack has to be performed, the obtained address (SP + offset) is used to address the stack and to write the vector read from the register file at the address specified by the parameter *v*; this is performed in the E1 stage.
As you can notice, to fill or spill the whole vector register file from the stack, the instruction must be executed multiple times (four in this case).

The *syntax* attribute specifies the assembly syntax of the instruction:

```
ex load    :   ld v0,dm(sp-32)
ex store   :   st v1,dm(sp-44)
```

The *image* attribute tells the compiler how to encode the second part, i.e. the least significant thirteen bits, of the instruction:

```
ls :: offs :: "00" :: v
```

where *ls* is a 1 bit selection value, *offs* is a 8 bit offset and *v* is a 2 bit address.
Adding the bits of the first part, i.e. the group code, the following encoding is obtained:

```
"111" :: ls :: offs :: "00" :: v
```

SAD computations

This second group contains four instructions used to load aligned vectors from the data memory, perform the sum of absolute differences, align not aligned vectors and perform move operations.

To implement these instructions, three new functional units are defined: VEC, used to compute the SAD; VAL, used to implement the *valign* primitive function; FA, used to implement the *force_ align*

primitive function. The three functional units, with the corresponding input and output signals, are described by the following code:

```
fu vec;
trn vecr<v16w8>;
trn vecs<v16w8>;
trn vect<v16w8>;
trn vecw<word>;

fu val;
trn valr<v16w8>;
trn vals<v16w8>;
trn valt<v16w8>;
trn valu<word>;

fu fa;
```

Some register aliases are also defined in order to simply the code:

```
reg VA[2]<v16w8,uint1> alias V[0];
reg VB[2]<v16w8,uint1> alias V[2];
property unconnected: VA, VB;

reg R03[4]<word,uint2> alias R[0]; //R03 :used to access the
reg R4<word> alias R[4];           //first 4 registers of R
reg R5<word> alias R[5];
reg R6<word> alias R[6];
property unconnected: R4, R5, R6;
```

The `first instruction` is described by the *vec_ vsad_ opn* rule with the following nML lines:

```
opn vec_vsad_opn(r: c3u, s: c2u, t: c2u)
{
    action {
        stage E1:
            vect = vadiff(vecr=V[s],vecs=V[t]) @vec;
            R[r] = rte1 = vecw = vsum(vect) @vec;
    }
    syntax : "vsad r" r ",v"s ",v"t;
    image  : r::s::t;
}
```

Three parameters r, s and t are used.
The *action* attribute defines the behavior of the instruction, see figure 3.15.

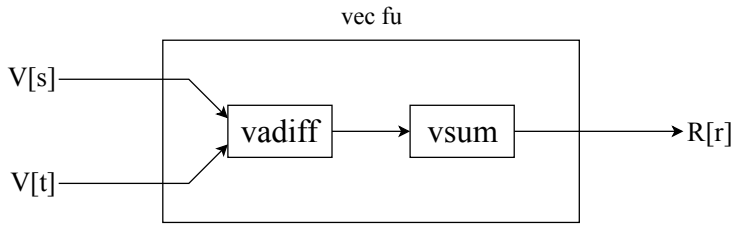


Figure 3.15: Behavior of the *vec_ vsad_ opn* primitive function.

Two vectors are read from the register file V, using the parameters s and t as addresses, and the absolute difference between each pair of elements of the two vectors is computed and stored in an output vector. Then, all the elements of the output vector are added and the final result is stored back into the register file R, at the address specified by the parameter r. These operations are performed by the VEC functional unit.

The *syntax* attribute specifies the assembly syntax:

```
ex  :   vsad r1,v2,v3
```

The *image* attribute defines the encoding of the last part, i.e. the least significant seven bits, of the instruction:

```
r :: s :: t
```

where r is a 3 bit address, while s and t are 2 bit addresses.

The `second instruction` is described by the *vec_ load_ opn* nML rule:

```
enum vec_ld_op { in "", pp "++", pm "+=r4" };
trn ag1_addr_algn<addr>;
opn vec_load_opn(op: vec_ld_op, v: c2u, r: c2u)
{
    action {
        stage ID:
            switch (op) {
                case in: ag1p = R03[r];
                case pp: R03[r] = ag1q = add(ag1p=R03[r],ag1m=16) @ag1;
                case pm: R03[r] = ag1q = add(ag1p=R03[r],ag1m=R4) @ag1;
             }
            ag1_addr = ag1p;
            dm_addr = ag1_addr_algn = force_align(ag1_addr) @fa;
        stage ID..E1:
            V[v]‘E1‘ = dmv_read‘E1‘ = DMv[dm_addr‘ID‘]‘ID‘;
    }
    syntax : "ld v" v ",dm(r" r op ")";
    image  : op::v::r;
}
```

Three parameters are needed: *op*, *v*, *r*.
This instruction performs an aligned vector load operation from the data memory, using the *force_ align* primitive function, and stores the obtained vector into the register file V.

The *action* attribute describes the behavior in detail: the value stored in the register file R at the address specified by the parameter r is read and aligned by the FA functional unit. Then, it is sent on the address bus and the data memory is accessed. At the same time, the value is post-incremented by the address generation unit AG1 and stored back to the register file, at the same location, ready for the next load operation. The post-incrementation is performed adding to the read value:

- 0, the value is not incremented.
- 16, the value is incremented of the size of a vector.
- the number stored in the register R[4].

The parameter *op* is used to select the proper incrementation value. These operations are performed in the ID stage. In the E1 stage, the fetched vector is on the memory data bus and is stored into the register file V, at the address specified by the parameter *v*.
The instruction behavior is also described in figure 3.16.

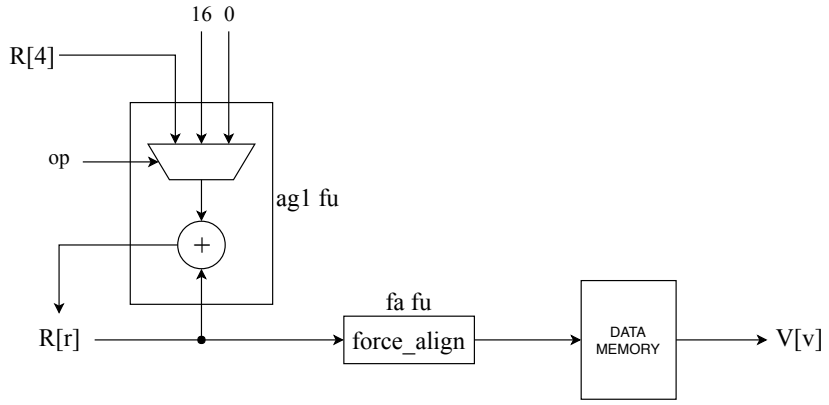


Figure 3.16: Behavior of the *vec_load_opn* primitive function.

The *syntax* attribute defines the assembly syntax:

```
ex op = in   :   ld v2,dm(r1)
ex op = pp   :   ld v1,dm(r3++)
ex op = pm   :   ld v0,dm(r3+=r4)
```

The *image* attributes describes the bit encoding of the last part, i.e. the least significant six bits, of the instruction:

```
op :: v :: r
```

where *op* is a 2 bit selection value, while *v* and *r* are 2 bit addresses. Note that the parameter *r* is a 2 bit and not a 3 bit address due to the fact that the number of possible used registers has been restricted to the first four only, in the register aliases declarations (already presented).

The *force_align* primitive function is hidden from the CHESS compiler by mean of a *chess_view* rule:

```
chess_view() {
    ag1_addr_algn = force_align(ag1_addr);
} -> {
    ag1_addr_algn = ag1_addr;
}
```

With the rule in place, the CHESS compiler can map the dereferencing of the vector pointer or the indexing of a vector array onto a vector load. It is the responsibility of the programmer to handle the alignment.

The `third instruction` is defined by the *vec_align_opn* rule:

```
opn vec_align_opn(r: c2u, s: c2u, u: c2u)
{
    action {
```

```
        stage E1:
          V[s] = valt = valign(valu=R03[u],valr=V[r],vals=V[s]) @val;
    }
    syntax : "valign v"s ",r" u ",v"r ",v"s;
    image  : u::r::s;
}
```

Three parameters are present: r, s and u. This instruction is used to re-pack a not aligned vector, starting from two aligned vectors already fetched from the data memory. This is performed by the *valign* primitive function.

The *action* attribute describes the operations: two vectors are read from the register file V, using as addresses the parameters r and s, and sent to the VAL functional unit. Here, the final vector is generated, using the value stored in the register R03[u] to determine how the operands have to be re-packed. After that, the resulting vector is stored back into the register file, at the location specified by the parameter s.
The instruction behavior is also described in figure 3.17.

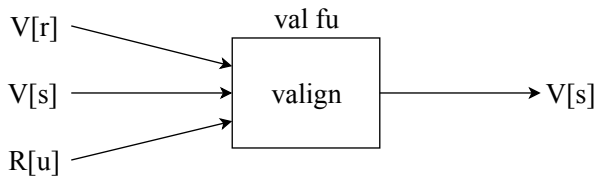


Figure 3.17: Behavior of the *vec_ align_ opn* primitive function.

The *syntax* attribute specifies the following assembly syntax:

```
ex  :  valign v0,r1,v1,v0
```

The *image* attribute shows the bit encoding of the last part, i.e. the least significant six bits, of the instruction:

```
u :: r :: s;
```

where u, r and s are 2 bit addresses.

The `fourth instruction` is described by the *vec_ vmove_ opn* rule:

```
opn vec_vmove_opn(t: c2u, s: c2u)
{
    action {
        stage E1: V[t] = vect = vecs=V[s];
    }
    syntax : "vmove v" t ",v"s;
    image  : t::s;
}
```

This instruction has two parameters only, t and s, and it is used to perform move vector operations. Note that, the instruction is not strictly required to perform a Motion Estimation, but it has been equally added to the MECORE instruction set for completeness and to support future developments.

The *action* attribute describes the behavior of the instruction: the vector stored in the register file at

the location specified by the parameter *s*, is read and immediately stored back into the register file at the location *t*.

The *syntax* attribute defines the assembly syntax:

```
ex  :   vmove v1,v2
```

The *image* attribute specifies the following encoding for the last part, i.e. the least significant four bits, of the instruction:

```
t :: s;
```

where *t* and *s* are 2 bit addresses.

To complete the instruction description of this group, the following piece of code has to be added:

```
opn vector_single_instr( vec_vmove_opn
                | vec_align_opn
                | vec_load_opn
                | vec_vsad_opn
                )
{
        image : "000000001"::vec_vmove_opn
                | "0000100"::vec_align_opn
                | "0000001"::vec_load_opn
                | "000001"::vec_vsad_opn
                ;
}
```

This code is used to specify the remaining encoding bits of the instructions, see the *image* attribute.

```
move instr     :   "000000001"
align instr    :   "0000100"
load instr     :   "0000001"
sad instr      :   "000001"
```

Finally, the whole instruction encodings can be obtained:

```
move instr     :   "110" :: "000000001"  :: t    :: s
align instr    :   "110" :: "0000100"    :: u    :: r    :: s
load instr     :   "110" :: "0000001"    :: op   :: v    :: r
sad instr      :   "110" :: "000001"     :: r    :: s    :: t
```

Add constant c2u

The constant *c2u* is added to the immediate constant declaration section in the file *mecore.n*:

```
cst c2u<uint2>;
```

It is used to implement the 2 bit parameters of the previously described instructions.

Add the promotion definition file

In order to use the new primitive function *vsum*, *vadiff* and *valign* as intrinsic functions in the C code, some promotion definitions have to be defined. These definitions are located in a file, called

mecore_ vector.h, that has to be added to the directory *Version3* → *lib*. The file has also to be included into the main CHESS header file *mecore_ chess.h*, adding the following line before the *mecore_ rewrite.h* include line:

```
#include "mecore_vector.h"
```

A detailed description of the file *mecore_ vector.h* will follow.

A new data type, called *vpix*, is defined as a vector of size VSIZE (16 bytes) and it is associated to the MECORE specific data type *v16w8*. It can now be called as an intrinsic data type in the C code.

```
class vpix property(vector unsigned char[VSIZE]);

chess_properties {
    representation vpix : v16w8;
}
```

Now, the promotion definitions for the new primitive functions are added:

```
promotion int vsumi(vpix) = word vsum(v16w8);
promotion vpix vadiff(vpix, vpix) = v16w8 vadiff(v16w8, v16w8);

inline   vpix* valign(vpix* a) { return a; }
promotion vpix valign(vpix*,vpix,vpix) = v16w8 valign(word,v16w8,v16w8);
```

As you can see, the functions with the MECORE specific data types are associated to functions with the C specific data types, and this allows them to be called directly in the C code. Obviously, the function behaviors do not change.
An inline function is also defined and used to perform a direct pointer alignment: a pointer to a vector, passed as parameter, is read and the pointer to the first element of the same vector is returned.

Modify the linker configuration file

To conclude the hardware support, the default linker configuration file *mecore.bcf* has to be modified as follow: comment the whole code, present in the file, and add the following lines:

```
_symbol _main 0
_stack DMb 0 0x0800
```

The first line defines the *main* function as the entry point for the code, while the second line, specifies the size and location of the stack in the data memory (from address 0x0000 to 0x0800).

3.4.2 Source code

The C source code is located in the folder *Version3* → *source_ code*.
To use the new instructions, the function *sad_ 16x16* has been properly modified, see figure 3.18.
The char pointers *search_ window* and *curr_ block* are cast to vector pointers of type *vpix*, called *pvss* and *pvc* respectively. The pointer *pvss* is then aligned, using the *valign* inline function, and the result saved into the pointer *pvs*. As you can notice, a single *for* loop is needed due to the use of vector instructions able to process all the element of a vector, i.e. a complete line of the current block (16 bytes), simultaneously. For this reason, only 16 loop iterations are needed.
The body of the loop is composed of five lines of code implementing the SAD computation between two vectors. The pointer *pvc* is used to load a vector from the current block and save it into *vc*, while the pointer *pvs* is used to load two aligned vectors from the Search Window, **pvs* and **(pvs+1)*.

Then, the two vectors (**pvs* and **(pvs+1)*) are re-packed into a single vector by the *valign* function, using the four least significant bits of the argument *pvss* to specify the alignment; the resulting vector is then stored into *vs*.
The functions *vadiff* and *vsumi* are used to compute the sum of absolute differences of the two vectors, *vs* and *vc*, and accumulate the result into the integer variable *sad*.
Finally, the pointer *pvs* is incremented by 3 (48 bytes) in order to be ready for the next loop iteration.

```
const int LINE_STEP = 48;

static int sad_16x16(unsigned char* search_window, unsigned char* curr_block)
{
    int sad = 0;
    vpix* pvc = (vpix*)curr_block;
    vpix* pvss = (vpix*)search_window;
    vpix* pvs = valign(pvss);
    for (int i = 0; i < 16; i++) {
        vpix vc = *pvc++;
        vpix vs = valign(pvss,*pvs,*(pvs+1));
        vpix adiff = vadiff(vs,vc);
        sad += vsumi(adiff);
        pvs += LINE_STEP/16;
    }
    return sad;
}
```

Figure 3.18: The *sad_16x16* function, Version 3.

3.4.3 Compilation and simulation

The processor model and the C program are compiled and a simulation is performed. The obtained results are equal to those of the first version, and this guarantees the correctness of the design.

Using the profiling tool, provided by ASIP Designer, important statistics are extracted. The compiled program contains 160 instructions, i.e. 366 bytes in program memory. The simulation is performed in 5285 clock cycles (cycle count) and the number of executed instructions (instruction count) is 4989. In table 3.5, these global numbers are partitioned among the three main functions of the program.

FUNCTION	INSTR COUNT	% OF TOTAL	CYCLE COUNT	% OF TOTAL
sad_16x16	3636	72.88%	3780	71.52%
motion_estimation	1133	22.71%	1257	23.78%
main	20	0.40%	29	0.55%
others	198	3.96%	216	4.09%

Table 3.5: Functions profiling, Version 3.

The sad_16x16 function is compiled into 11 assembly instructions, see table 3.6.

PC	ASSEMBLY	INSTRUCTION COUNT	CYCLE COUNT
31	mvib r0,0	36	36
32	mvib r4,32	36	36
33	mv r3,r1	36	36
34	doi 16,42	36	108
37	ld v1,dm(r3++)	576	576
38	ld v0,dm(r3+=r4)	576	576
39	ld v2,dm(r2++)	576	576
40	valign v0,r1,v1,v0	576	576
41	vsad r5,v0,v2	576	576
42	add r0,r0,r5	576	576
43	rt	36	108

Table 3.6: Assembly instructions of the *sad_ 16x16* function, Version 3.

A single loop is present and mapped onto zero overhead hardware loop.
The loop, initiated by the *doi* instruction at address 34, starts at instruction 37 and ends at 42. Note that, no delay slot is generated by the *doi* instruction.
The three load instructions are used to load the three operand vectors from the data memory. Then, vector *v1* and *v0* are re-aligned into a single vector, stored into *v0*, and the sum of absolute differences plus accumulation is computed by instructions 40, 41 and 42 of the loop body.

3.4.4 Synthesis and physical design

The third version of MECORE has been synthesized using Synopsys Design Compiler. The synthesis has been performed and optimized, to archive the maximum operating frequency.

The obtained results are presented here:

```
Critical path   :   1.79 ns
Max. frequency  :   558.66 MHz
Area            :   47858.40 um^2
Dynamic power   :   56.28 uW
Leakage power   :   3.69 uW
```

After the synthesis, the design has been placed and routed, using Innovus by Cadence, in order to evaluate the area:

```
Gates   :   49789
Cells   :   31325
Area    :   39732.20 um^2
```

3.5 Results comparison

This section compares the three obtained versions of MECORE from an hardware and software point of view, highlighting how the selected parameters change moving from the first version to the last.
The hardware view will focus on parameters that are related to the hardware implementation of the three versions only, without taking into account the final application. Such parameters are the area, the dynamic power, the leakage power and the maximum operating frequency.
On the other hand, the software view will focus more on the application side, evaluating the compiled

code length, the execution time and performing an analysis on the execution density, i.e. how the execution time is distributed over the main functions of the program and how this distribution changes from the first version to the last.

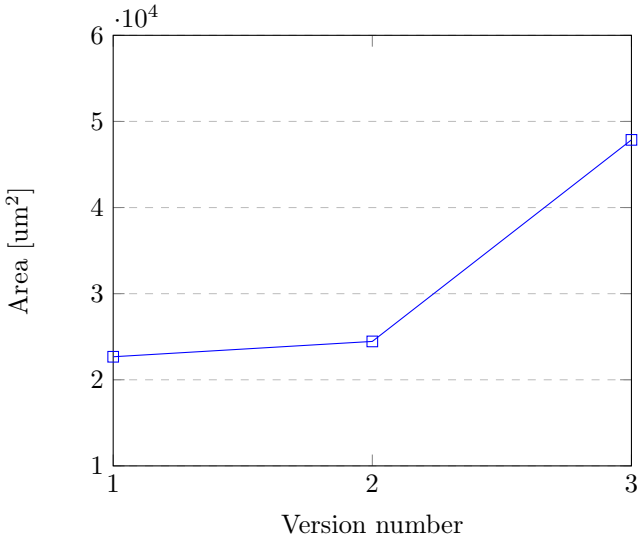
3.5.1 Hardware

Four parameters will be analyzed in this subsection: the area, the dynamic power, the leakage power and the maximum operating frequency of each version of MECORE. The comparisons will be figured out using graphs; in this way, the trend of each parameter, from the first version to the last, can be better analyzed and commented.

Area

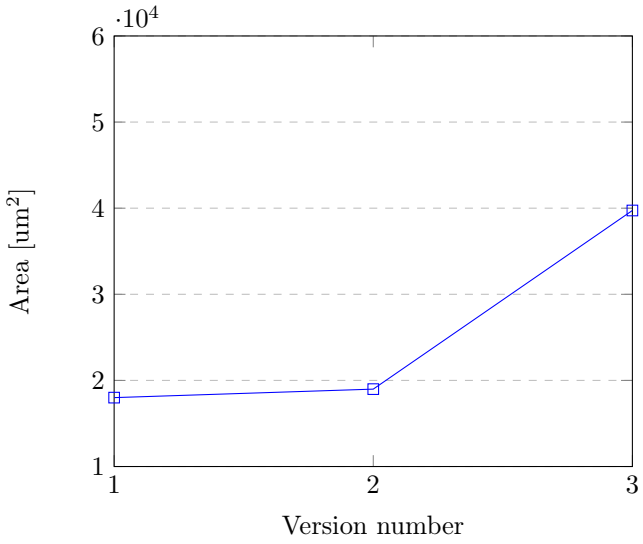
In the following graph, figure 3.19, you can analyze how the area evaluated after synthesis changes from one version to another.
As you can see, there is a slight increase between the first and the second versions, moving from an area of 22671.72 um^2 to 24453.00 um^2. The reason for that, is the additional hardware used to implement the scalar SAD accelerator.
The increase in area is much more evident considering the transition from the second version to the third, where there is an increase of the 95.72%, moving from an area of 24453.00 um^2 to 47858.40 um^2. This can be easily explained by the fact that, the third version is a SIMD, Single Instruction Multiple Data, processor that elaborates not single data but vectors of data in parallel and this is clearly supported by a more complex and area expensive hardware.

Figure 3.19: Area of the three versions after synthesis.



The same trend can be observed analyzing the graph in figure 3.20, obtained after the physical design. You can note that, even though the values are smaller due to the use of a different library of cells, their relationship remains almost the same: a slight growing moving from the first version to the second (from 18001.00 um^2 to 18988.10 um^2) and a considerable growing from the second version to the third (from 18988.10 um^2 to 39732.20 um^2).

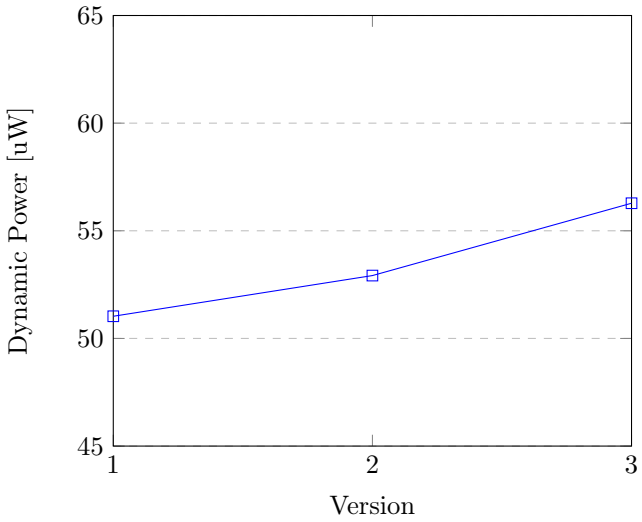
Figure 3.20: Area of the three versions after physical design.



Power

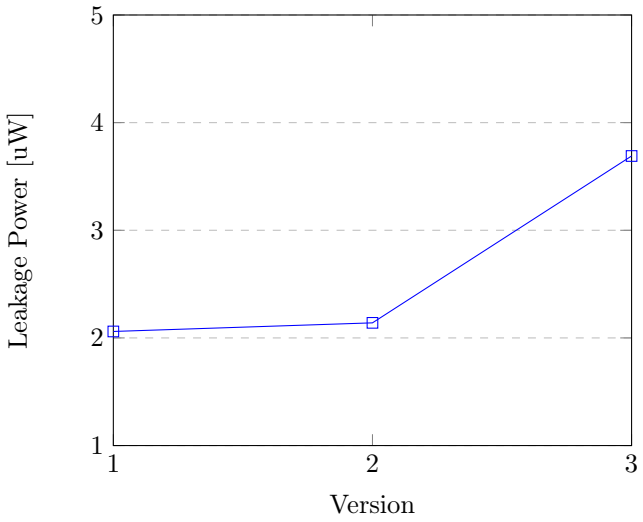
The graphs in figure 3.21 and 3.22 show the variation of the dynamic power and the leakage power, respectively.
As you can see, the trend is very similar to that obtained for the area. The reason for that is the strict relationship existing between area and power, both dynamic and leakage. An increase in the area leads to an increase in the number of gates and so to an increase in the power dissipated by the design, due to the higher number of working units (gates). In the case of dynamic power, the power is dissipated during the gate switching, while, in the case of leakage power, the power is dissipated in a static way when the gates are in the rest condition.
Looking at numbers, in the dynamic power graph you can notice a small rise from the first version to the second (from 51.03 uW to 52.92 uW), followed by a slightly bigger rise from the second version to the third (from 52.92 uW to 56.28 uW).

Figure 3.21: Dynamic power of the three versions.



For what regards the leakage power graph, a different trend is observed, with a slight increase from the first version to the second (from 2.06 uW to 2.14 uW) and a much more considerable increase from the second version to the third (from 2.14 uW to 3.69 uW).

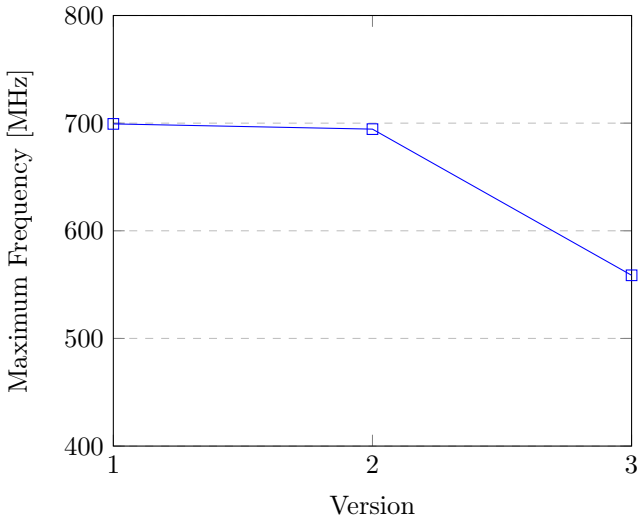
Figure 3.22: Leakage power of the three versions.



Frequency

The additional hardware support used to implement the second version did not have a big impact on the maximum frequency of the design that remains almost constant (from 699.30 MHz to 694.44 MHz). On the other side, the implementation of the third version leads to a decrease of the 19.6% of the maximum operating frequency (from 694.44 MHz to 558.66 MHz). See graph in figure 3.23.

Figure 3.23: Maximum frequency of the three versions.



3.5.2 Software

Three parameters will be analyzed in this subsection: the code length, the execution time and the execution density of each version of MECORE.

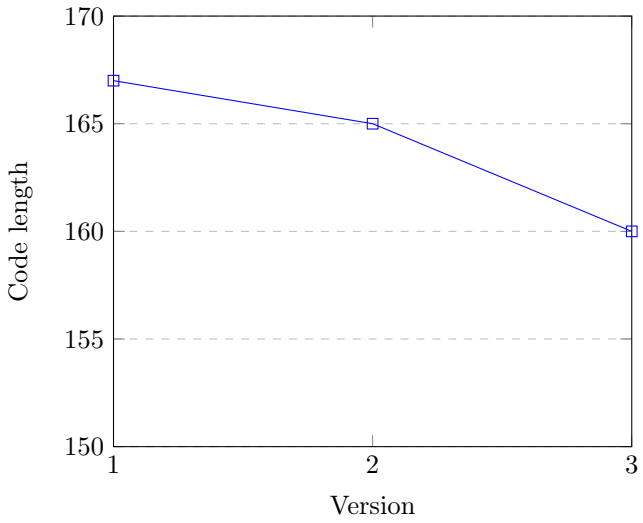
Code length

The code length has been evaluated in two ways:

- counting the number of assembly instructions obtained after compilation;
- calculating the space in memory occupied by the program.

The following graph, figure 3.24, shows the results obtained with the first approach:

Figure 3.24: Code length of the three versions.



As you can see, there is a decrease in the number of instructions moving from the first version to the second (from 167 to 165) and from the second version to the last (from 165 to 160). This can appear as a small reduction, but actually it has a big impact on the execution time of the program, that will be analyzed later on this section.
Note that, the decrease only affects the instructions of the *sad_16x16* function, that is the only part of the code that is modified from one version to another; the other functions of the program never change and are compiled to the same set of assembly instructions.

The occupied space in memory has the same trend, with a small drop from the first version to the second (from 386 bytes to 384 bytes) and a bigger drop from the second version to the third (from 384 bytes to 366 bytes).

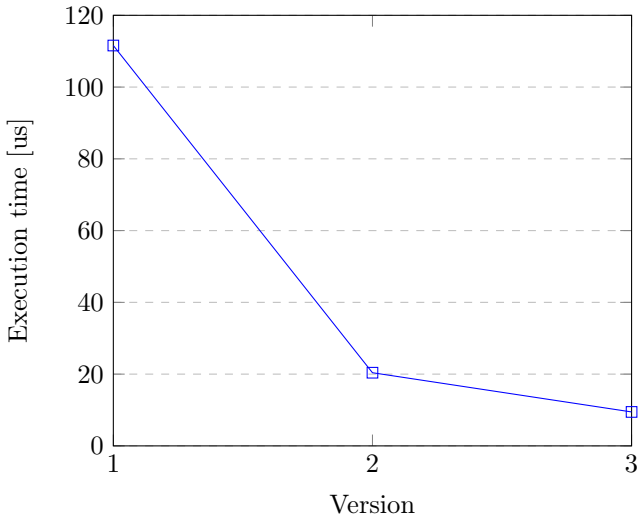
Execution time

This parameter shows the time, in microseconds, needed to perform a complete simulation of the *me.c* program, i.e. to compute the Motion Vector of a single Macro Block of size 16x16 pixels over a Search Window of 48x48 pixels.

The execution time is evaluated multiplying the critical path to the total number of clock cycles needed to perform the simulation; this will give an accurate value for the required time.

The graph shown in figure 3.25 figures out how the execution time changes.

Figure 3.25: Execution time of the three versions.



As you can see, there is an outstanding decrease from the first version to the second, due to the addition of the special scalar accelerator able to compute the SAD in a single clock cycle. The execution time has been reduced of the 81.73%, moving from 111.58 us to only 20.39 us, and this is a great result and achievement.
Considering the second transition, there is again a huge reduction in the execution time of the 53.60%, moving from 20.39 us to 9.46 us. This thanks to the use of a SIMD architecture able to process vectors of data simultaneously.

Execution density

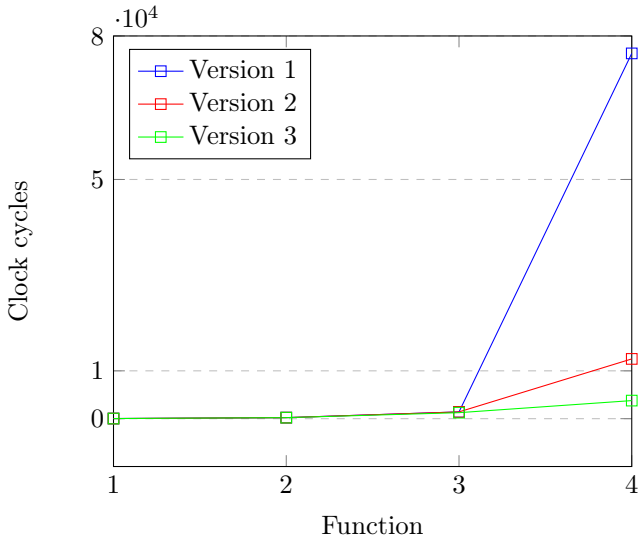
The execution density describes how the time is distributed, i.e. spent by the processor, over the functions composing the program.

The results can be analyzed in the graph shown by figure 3.26, where three lines, corresponding to the three versions of the design, figure out the time distribution. On the horizontal axis, you can find four ordered numbers (1 to 4) corresponding to the four main functions of the C program:

1. the *main* function,
2. other secondary functions,
3. the *motion_estimation* function,
4. the *sad_16x16* function.

On the vertical axis, instead, you can see the number of clock cycles spent by the processor to execute each specific function; this has been evaluated using special tools provided by ASIP Designer.

Figure 3.26: Execution time of the three versions.



You can note that, the distribution tends to become flatter, moving from the first version to the third: the huge disparity between the time needed by the *sad_ 16x16* and by the other functions, in the first version, is greatly attenuated in the second and third versions, strongly reducing the number of clock cycles needed to compute the SAD.

A flatter distribution can lead to great advantages, not only from the performance point of view, but also from the physical implementation point of view: a more regular distribution of activities on the different operating units may generate a better physical layout, able to suffer less of problems such as overheating, electromigration and clock distribution problems, e.g. skew and jitter.

Chapter 4

State of art

This chapter analyses several approaches, found in the literature, adopted to implement Application Specific Instruction-set Processors (ASIPs) for Motion Estimation applications. Each approach is first described from a theoretical point of view and then summaries of several papers are provided to better appreciate the practical implementations.

4.1 Overview

An ASIP is a processor whose architecture and instruction-set have been specialized for a specific application. The design of such kind of processors may follow different approaches depending on the constraints and requirements of the desired application. From an hardware point of view, the *parallelism* has been identified as the main architectural property able to characterize and classify an ASIP processor design.

There are two kinds of parallelism:

- instruction parallelism,
- data parallelism.

Instruction parallelism

This form of parallelism allows the architecture to execute more then one instruction in parallel with a considerable increase in performance. Instruction parallelism can be exploited with three different architecture choices:

- multi-ASIP architecture,
- VLIW architecture,
- super scalar architecture.

The multi-ASIP architecture choice consists in the use of multiple identical ASIP processors to implement a form of instruction parallelism: multiple instructions are executed in parallel, one on each instance of the processor.
On the other side, the VLIW and the super-scalar choices implement the instruction parallelism in the internal architecture of the ASIP processor, exploiting multi-instance functional units.

The Very Long Instruction Word (VLIW) method resorts of a powerful compiler able to combine in the same instruction all the operations that can be executed in parallel, depending on the available functional units. This technique allows to obtain a reasonably complex hardware architecture at the cost of a more sophisticated compiler.
The super scalar method, instead, implements a more complex hardware architecture able to decide in real time the set of instructions that can be executed in parallel, without resorting to a sophisticated compiler.
As you will notice in the following sections, the VLIW is the predominant choice for the designed ASIP processors. The reason for that may be in the field of application for which these architectures have been designed: video compression requires low-power and low-area processors and a VLIW guarantees lower values on these parameters due to a less complex architecture; at the same time, the use of a more sophisticated compiler is not a huge problem because of the fact that, the code running on an ASIP processor for video applications is generally well defined and does not require a lot of upgrades.

Data parallelism

This form of parallelism allows the architecture to process multiple data in parallel. Note that, data parallelism does not decrease the code execution time, but speeds up the data processing.

Considering the two kind of parallelism just examined, four possible architecture design approaches have been highlighted:

- Single Instruction Single Data (SISD),
- Multiple Instruction Single Data (MISD),
- Single Instruction Multiple Data (MIMD),
- Multiple Instruction Multiple Data (MIMD).

The following sections describe each approach in detail and provide several case of studies found in the literature.

4.2 SISD approach

The SISD approach consists of an architecture that does not allow the execution of multiple instructions or the process of several data in parallel.

4.2.1 Case of study 1

Title 'Application Specific Processors for Multimedia Applications' [8]

Summary

This paper mainly describes the design methodology, based on the LISATek software and the LISA 2.0 description language, to design ASIP processors for multimedia applications.
There are two main approaches to design an ASIP. The first approach is to design the processor from scratch: an entirely new instruction set is specifically designed for a target application. The second

approach is to customize the instruction set of an existing general purpose processor. In both cases, the objective of the design is to find the best trade off between flexibility and performance.
LISATek assists the design process by automatically generating the software toolkit (compiler, assembler, linker, simulator) as well as the RTL description of the designed processor. It covers all phases of the design process from the algorithmic specification of the application down to the implementation of the micro architecture (see figure 4.1).

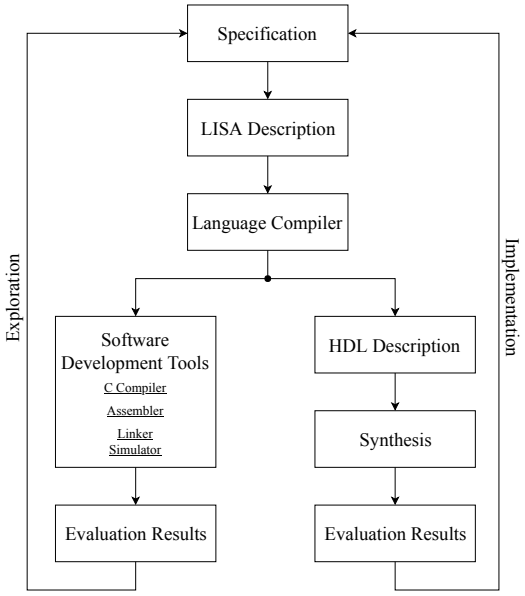


Figure 4.1: LISATek design flow. SISD - Case of study 1.

As a starting point for the processor creation, LISATek provides a library of models which contains processors for different architecture categories like VLIW (Very Long Instruction Word), SIMD (Single Instruction Multiple Data) and RISC (Reduced Instruction Set Computer).
To demonstrate the design flow, a 32 bit RISC processor, present in the library, has been extended and optimized for two main applications: the Motion Estimation algorithms of the H.264 encoding standard and the image compression algorithm of the JPEG standard.
A logic synthesis by Cadence Encounter RTL Compiler has been performed using a CMOS 0.13 um cell library. The obtained results are reported here:

```
Frequency  :  160 MHz
Gates      :  42500
```

4.2.2 Case of study 2

Title 'An ASIP Approach for H.264/AVC Implementation Having Novel Coprocessors' [9]

Summary

This paper proposes a new Motion Estimation ASIP processor using the Modified Normalized Partial Distortion Search (MNPDS) algorithm. The proposed MNPDS can reduce the average computational complexity by 96.2-99.5% compared to the full search algorithm.

In figure 4.2, you can see the overall architecture of the proposed ASIP which consists of a programmable DSP part (blue rectangle) and an hardware accelerating part (red rectangle).

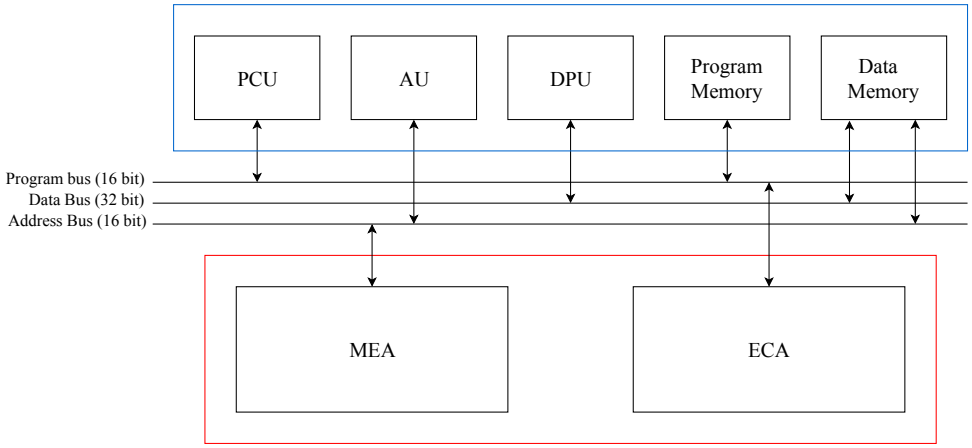


Figure 4.2: Architecture of the proposed ASIP. SISD - Case of study 2.

The programmable DSP part has a Program Control Unit (PCU), an Address Unit (AU) and a Data Processing Unit (DPU). Each of the internal word lengths is 32 bits and the instruction pipeline is composed of six stages: pre-fetch, fetch, decode, execute1, execute2 and execute3.
The hardware accelerating part consists of a ME Accelerator (MEA), used to speed up the execution of the Motion Estimation, and an Entropy Coding Accelerator (ECA).
The ASIP instruction set is composed of 35 arithmetic instructions, 11 logical and shift instructions, 6 program control instructions, 4 move instructions and 16 special instructions, dedicated to the H.264/AVC video compression standard.

4.2.3 Case of study 3

Title 'A 0.3 mW 1.4 mm^2 Motion Estimation Processor for Mobile Video Application' [10]

Summary

This paper describes an Application Specific Instruction set Processor (ASIP) for Motion Estimation processing, in combination with a new Ultra-Low Complexity ME Algorithm (ULCMEA), that can reduce power while keeping high flexibility.
In figure 4.3 you can analyze the processor architecture.
The ASIP has been designed in Verilog HDL starting from a 16 bit pipelined RISC processor model. It has 5 stages of pipeline: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (ME), Write Back (WB).
To enhance the performance of SAD operations and edge detection, six new units (grey in the picture) have been added to the processor model: two Address Generation units (AG1 and AG2), a Template Memory (TM), a Search-area Memory (SM), an Absolute Difference calculation unit (AD) and an ACCumulation unit (ACC). This extension allows to calculate the SAD between two pixel values in a single clock cycle.

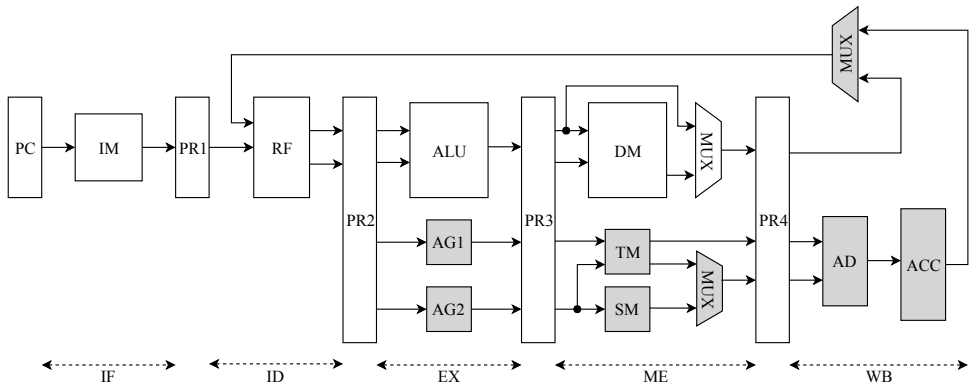


Figure 4.3: Architecture of the proposed ASIP. SISD - Case of study 3.

Finally, the ASIP processor has been synthesized using TSMC 0.18 um CMOS technology and the obtained results can be analyzed in table 4.1.

Process technology	TSMC 0.18 um
Logic gates	about 15K gates
Internal RAM	32Kbits
Area	1.41mm^2
Max frequency	80 MHz
Power (estimated @1.8V)	Dyn: 60.11 uW/MHz
	Leak: 100.5 uW

Table 4.1: Synthesis results of the proposed ASIP. SISD - Case of study 3.

4.3 MISD approach

The MISD approach consists of an architecture that allows the execution of multiple instructions in parallel, but does not allow the process of several data at the same time.

4.3.1 Case of study 1

Title 'A Cost-effective Implementation of Object-based Motion Estimation' [11]

Summary

Emerging applications in the mobile and automotive industries can benefit from a solution which can segment an image into objects. Object-based Motion Estimation (OME) is an algorithm that can provide such a segmentation.
OME consists of two modules that are executed sequentially. These two modules are the Motion Segmentator and the Parameter Estimator. In the Motion Segmentator module the image is segmented using the motion models, which were determined in the Parameter Estimator, to match objects to blocks. The image regions where a certain motion model is valid are then used in the Parameter Estimator to determine a new and more accurate motion model.

Figure 4.4 shows a graphical representation of the just described segmentation process.

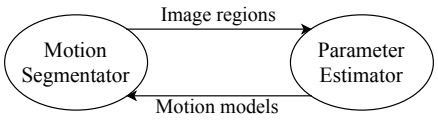


Figure 4.4: Segmentation process diagram. MISD - Case of study 1.

In this paper, the algorithm is implemented on a VLIW Application Specific Instruction-set Processor (ASIP), designed with the A|RT software. The article also proposes a multi-level caching architecture to keep bandwidth and power requirements low and discusses algorithmic changes to the algorithm, which are necessary to map it on the designed ASIP VLIW processor model.

4.4 SIMD approach

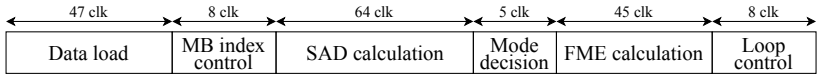
The SIMD approach consists of an architecture that executes of a single instruction at the time, but allows multiple data to be processed in parallel.

4.4.1 Case of study 1

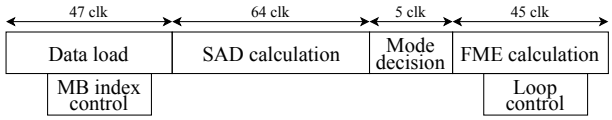
Title 'Efficient Program Control Schemes for Motion Estimation Specific Processor' [12]

Summary

Motion Estimation is the most computation intensive part of any video codec. Many techniques have been investigated to improve its performance. Generally, the ASIP approaches focus on efficient parallel architectures able to speed up the computations. However, to efficiently support complex ME algorithms, it is necessary to investigate non only parallel techniques but also new and more powerful program control schemes. This paper proposes two program control schemes for ME ASIP: a dynamic pipeline control scheme and an hardware loop scheme.
Generally, most of the instructions of an ASIP are executed in the same number of clock cycles except the hardware acceleration instructions which consume a variable number of clock cycles depending on the complexity of the ME operation. These acceleration instructions cause many pipeline stalls and lead to a performance degradation of the ASIP. The proposed dynamic pipeline control scheme can reduce the number of additional cycles and save around the 9.03% of the computation time.
Motion Estimation algorithms also consist of multiple loop operations. Moreover, the algorithms with early termination require additional compare and jump operations. New loop instructions and their specific architectures have been implemented to support multiple nested loops without extra overhead cycles, reducing of about 29.26% the average instruction cycles compared with the existing loop instructions.
The proposed program control schemes have been integrated into the ME ASIP [14] and the whole design simulated using Synopsys Processor Designer, based on the LISA architecture description language, and modeled in Verilog HDL.
Figure 4.5 shows the required cycles per macro block when the Search Window is 16x16 and the full search in performed.



(a) ME ASIP without program control schemes



(b) ME ASIP with program control schemes

Figure 4.5: Required cycles per Macro Block. SIMD - Case of study 1.

4.4.2 Case of study 2

Title 'An ASIP Approach for ME Reusing Resources for H.264 Intra Prediction' [13]

Summary

To improve the performance of Motion Estimation, various approaches have been suggested. Among them, ASIP approach became popular because it can narrow the gap between ASICs and General Purpose programmable Processors (GPP) in terms of performance, power, cost and flexibility.
This paper proposes an ASIP for Motion Estimation. Its main features are:

- `baseline architecture`: 8 stage pipeline, 16-bit single issue RISC processor.
- `Instruction memory`: 8 KB instruction memory.
- `Data memory`: four 4.5 KB data memory for frame, 4 KB general data memory, 2 KB data memory for output.
- `Application specific features`: 2D memory access, banked memory structure; 2-mode memory switching; 9 SAD calculation units that process 4x2 pixels matrix; minimum SAD and corresponding Motion Vector calculation in 2 cycles.
- `Miscellaneous`: no cache; maximum operation frequency is 300 MHz.

The architecture is based on a very simple RISC processor extended by adding application specific units and logic (see figure 4.6).
The SAD calculation unit is able to compute the SAD between the current 4x2 Macro Block and nine 4x2 reference blocks of the Search Window in parallel; this specific size has been selected because it is the common divisor of all Macro Block size types used in the H.264 standard, e.g. 16x16, 8x8 or 4x4. After the SAD calculation, the position vector of each reference block is propagated toward the Min Unit, together with the corresponding obtained SAD value. Within two clock cycles, the unit decides the minimum SAD value and the corresponding vector is taken as the final Motion Vector.
The frame data memory, used to store the current and reference frame is located inside the ASIP processor and is a 2D memory. This means that, logically, the memory is not viewed as a linear array but as a 2D matrix. To read or write data, an (x,y) pair of addresses are needed, instead of a single value, and this reduces the coding stress of the programmer because 2D addresses are more intuitive working with images.

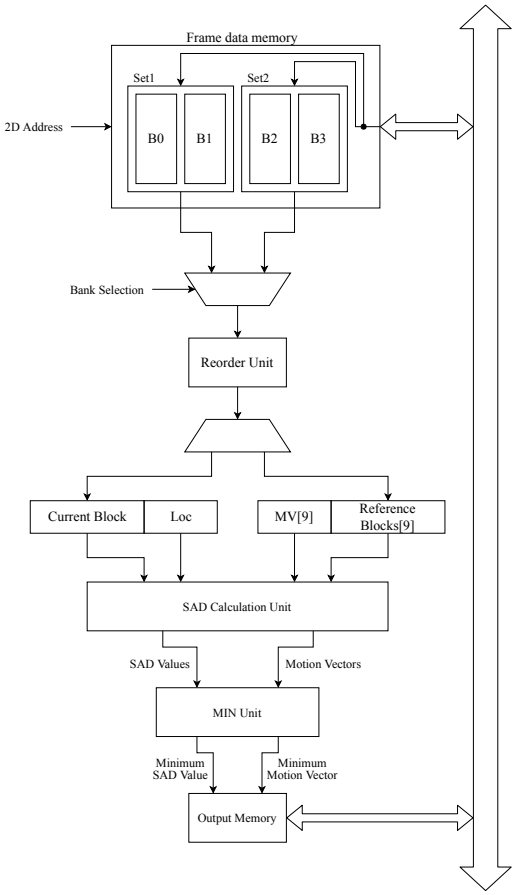


Figure 4.6: Architecture of the proposed ASIP. SIMD - Case of study 2.

The ASIP has been implemented in Verilog HDL and synthesized using Synopsys Design Compiler with a 130 nm technology. The obtained frequency and area are here reported:

```
Frequency  :  300 MHz
Area       :  563093 um^2
```

4.4.3 Case of study 3

Title 'Integer-pel ME Specific Instructions and their Hardware Architecture for ASIP' [14]

Summary

Existing digital signal processing processors have various application specific instructions for multimedia algorithms. However, conventional instructions are not efficient to support the newly adopted features of the H.264 standard. To efficiently implement the new features, new instructions and their hardware architectures have been proposed in this paper.
Two main instructions have been implemented for ASIP: a Variable Block SAD (VBSAD) instruction and a SAD specific Mode Decision (SADMD) instruction.
The VBSAD instruction can compute the SAD operation of a maximum of 256 points, i.e. locations in the Search Window, in parallel. With this instruction, the ME operations can be started at any

point in the Search Window, can be of any size (maximum 16x16 blocks) and can follow any search pattern (see figure 4.7).

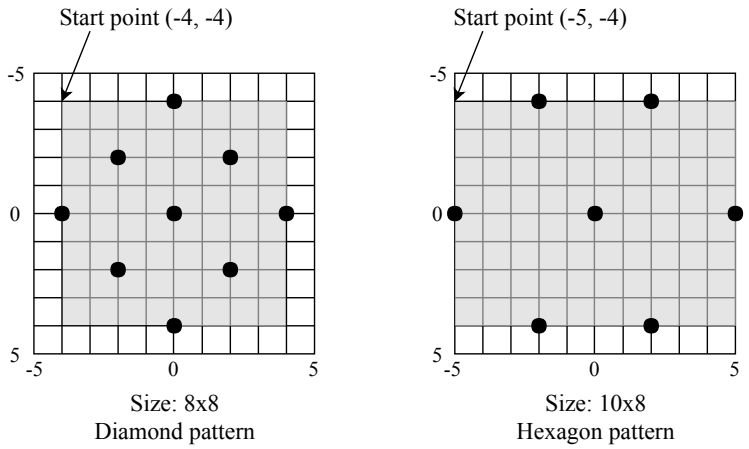


Figure 4.7: Examples of proposed VBSAD instruction. SIMD - Case of study 3.

However some search algorithms use several search patterns to improve the performance. To support this feature, the proposed ASIP has 4 special registers for storing sizes and patters. These special registers can be uploaded before starting the search operations.
Once the operations are concluded, the VBSAD instruction outputs the optimal SAD result of each block mode and the corresponding Motion Vector; the SADMD instruction is finally executed to select the best mode.
The proposed ASIP processor has been simulated using Synopsys Processor Designer, based on the LISA description language, and synthesized using an IBM 90 nm library of cells.

4.4.4 Case of study 4

Title 'A Configurable and Programmable ME Processor for the H.264 Video Codec' [15]

Summary

This paper proposes a programmable and configurable ASIP processor for Motion Estimation applications, based on the H.264 standard. The processor is able to handle the processing requirements of high definition video and it is suitable for FPGA implementations.
The instruction set is based on few specialized instructions targeted to accelerate Block Matching Motion Estimation algorithms. The configurability comes from the ability of optimizing the micro architecture for the selected algorithm and performance requirements, through varying the number and type of Execution Units (EUs) at compile time, before implementing the design on FPGA.
Three are the main accelerating units: the Integer Pixel Execution Unit (IPEU), the Fractional Pixel Execution Unit (FPEU) and the Interpolation Execution Unit (IEU).
Each functional unit is pipelined at 64 bit word level. All the accesses to the reference and the current Macro Block memory are performed through a 64 bit wide data bus and the SAD engine also operates with 64 bits in parallel. The memory is organized in 64 bit words and typically all accesses are unaligned, since they refer to Macro Blocks that can start at any position inside a word. By

performing two aligned 64 bit read operations simultaneously, the desired 64 bits inside two words can be selected; this is performed by the vector alignment unit.
The engine also supports half and quarter pixel Motion Estimation, thanks to a half-pixel interpolator execution unit and a specifically designed fractional-pixel unit. These two EUs can be instantiated if required by the algorithm.
The processor has been implemented on the Virtex-4 SX35 FPGA, fabricated using 65 nm CMOS technology. The results of implementing the processor with one to eight execution units are illustrated in table 4.2.

Number of EUs	Used LUTs	Embedded RAMs	Critical Path
1 IPEU	2083 (6%)	7 (3.6%)	7.253
2 IPEU	3660 (11%	13 (6.7%)	7.325
3 IPEU	5242 (17%)	20 (14%)	7.335
4 IPEU	6898 (22%)	27 (14%)	7.383
5 IPEU	8478 (28%)	34 (17.7%)	7.363
6 IPEU	10060 (33%)	41 (21.3%)	7.307
8 IPEU	13342 (43%)	48 (25%)	7.392
+ 1 FPEU	+6619 (21%)	+13 (7%)	+ 0

Table 4.2: Processor complexity. SIMD - Case of study 4.

4.4.5 Case of study 5

Title 'An ASIP Approach for Adaptive AVC Motion Estimation' [16]

Summary

The Motion Estimation algorithms developed for the H.264/AVC coding standard are mainly based on two approaches. The first approach consists in evaluating the Best Matching Block by guiding the search procedure using predefined search patters; an example is the Hibrid Unsymmetrical-cross Multi-Hexagon-grid Search (UMHexagonS), that makes use of a complex pattern structure based on four different search patters. This algorithms generally require a significant amount of computations. As a consequence, they are often combined with adaptive schemes, making use of prediction and early termination techniques; an example is the Enhanced Predictive Zonal Search (EPZS), adopted by the H.264/AVC standard.
However, the memory and computational requirements of these algorithms are generally quite high, and this make them difficult to be implemented in real time.
This paper proposes a new adaptive ME algorithm and its hardware implementation on ASIP, to speed up the ME procedure and decrease the memory requirements. This algorithm not only exploits the spatial and temporal redundancies through prediction techniques, but it also applies adaptive search patterns and early termination thresholding.

The algorithm is based on three different techniques:

- `prediction`: to select the best starting search point, in order to accelerate the search procedure for the best matching MB;
- `adaptive search patterns`: to further optimize the search procedure;

- `threshold`: to stop the search as soon as the obtained result is good enough and it is not useful to continue the search.

The ASIP implementation of the algorithm has been specially optimized for portable and mobile platforms, where power consumption and area are mandatory constraints. The instruction set is composed of only eight instructions, including the SAD16 instruction that allows to compute the SAD between two sets of 16 pixel values in parallel and accumulate the result.
The ASIP architecture can be analyzed in figure 4.8.

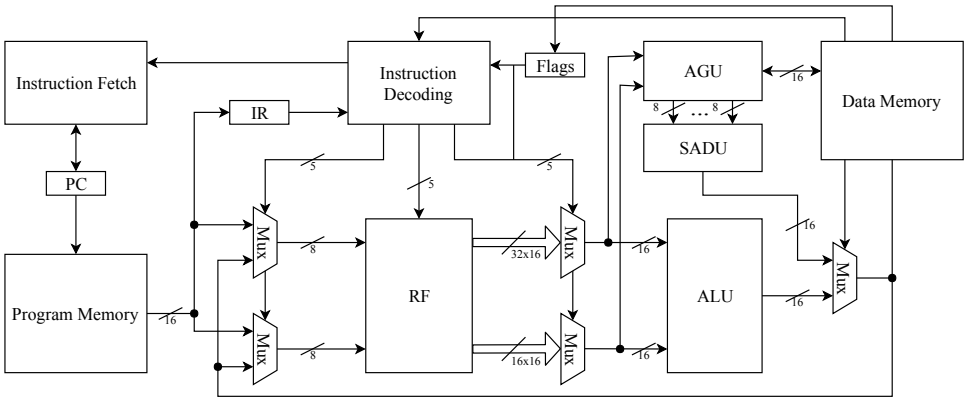


Figure 4.8: Architecture of the proposed ASIP. SIMD - Case of study 5.

4.5 MIMD approach

The MIMD approach consists of an architecture that allows the execution of multiple instructions and the process of several data in parallel.

4.5.1 Case of study 1

Title 'Multi-ASIP Based Parallel and Scalable Implementation of ME Kernel for HD Videos' [17]

Summary

Parallel implementations of Motion Estimation for high definition videos typically exploit various forms of parallelism to deliver real-time throughput. However, they often suffer from limited flexibility and scalability due to the form of parallelism and architecture used.
This article proposes a multi-ASIP architecture to provide a flexible and scalable platform for Motion Estimation of high definition videos. Two forms of parallelism are combined: Group Of Macro Blocks (GOMBs) and Intra Macro Blocks (IMBs).
In GOMB, a frame is partitioned into groups of Macro Blocks and the Motion Estimation is performed on those groups in parallel. At the implementation level, multiple ASIP processors are used to process the GOMBs of a frame: each ASIP is associated to a GOMB where the MBs are processed in a sequential manner.
The IMB-level parallelism is realized through the use of custom instructions, implemented for each of the multiple ASIPs, for the SAD calculations. Here the scalability comes from the fact that different

implementations of the SAD accelerator can be used, such as HCCI (Hand Coded Custom Instruction), XCI (Tensilica's XPRES generated custom instruction) and NoCI (No Custom Instruction).
The proposed parallel and scalable Motion Estimation approach was implemented using a commercial design environment from Tensilica.
The graph in figure 4.9 shows the throughput (in frames per second), on the Y axis, of the execution of three different video sequences (Pedestrian, Rushhour and Tractor) implemented with the HCCI, XCI and NoCI approaches. The number of Motion Estimation ASIP processors is shown on the X axis, increasing from 1 to 9 to highlight the performance improvement.

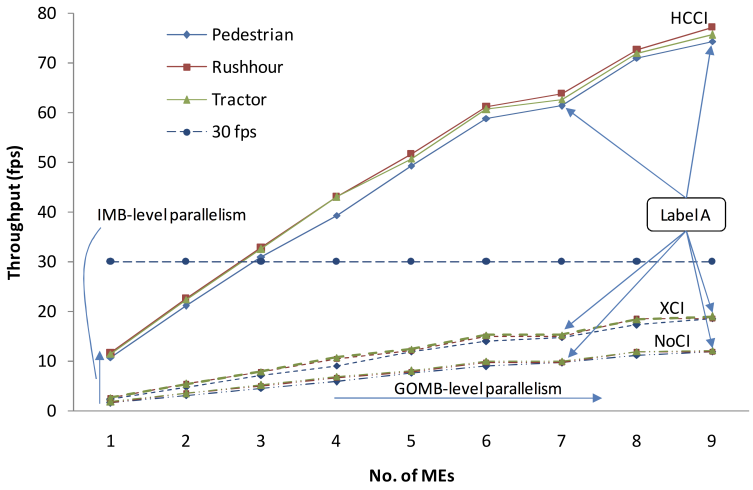


Figure 4.9: Throughput of ME tasks. MIMD - Case of study 1.

4.5.2 Case of study 2

Title 'Memory-centric Motion Estimator' [18]

Summary

In the streaming of video processing domain, the only way to meet strict performance and quality requirements is to apply buffering of the pixel data. So, the importance of carefully design efficient memory subsystems is significant. This paper proposes a new specialized memory subsystem integrated in a Very Large Instruction Word (VLIW) ASIP processor optimized for Motion Estimation.
The ASIP architecture can be analyzed in figure 4.10.
Two Application Specific Units (ASUs) have been design in order to speed up the ME calculations: a Sum of Absolute Differences unit (SAD) and a Bi-linear Interpolation unit (BI). In the ASUs, data level parallelism is heavily exploited to allow the process of multiple pixel values at the same time. Instruction level parallelism is also exploited through a VLIW architecture.
The Search Area buffer (SA buffer), which is part of the memory subsystem, has been designed to process high definition (1920*1080 pixels) input data at 25 fps. This performance has been enabled by a reorganization of the pixels within the buffer to support single cycle read of a block of 8*6 pixels. To process one Motion Vector candidate, three clock cycles are needed. In the first cycle, the controller initializes its pointers according to the Motion Vector candidate. Since the design is pipelined, two cycles are needed for the read action to be performed, meaning that after the third clock cycle, the complete block of 8*6 pixels is available. The SAD and BI functional units exploit a 48 bit paral-

lelism and can be software pipelined with the reading action from the buffer, resulting in a very low overhead: the total cycle count to process 5 Motion Vector candidates is of the order of 20 clock cycles.

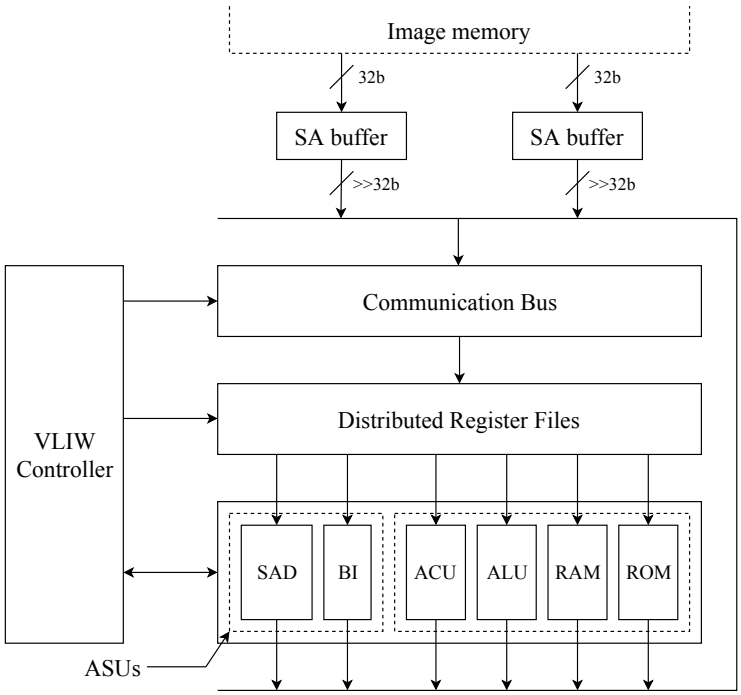


Figure 4.10: Architecture of the proposed ASIP. MIMD - Case of study 2.

Finally, the VHDL files of the processor has been automatically generated from the high level C description, using the A|RT software tools, and then, the design synthesized using a 0.18 um CMOS technology, obtaining a maximum operating frequency of 100 MHz.

4.5.3 Case of study 3

Title 'Segment-based Motion Estimation Using Block-based Engine' [19]

Summary

The Motion Estimation is a key function in video processing. Most Motion Estimation implementations are block-based, this means that each frame is divided into a number of blocks and a Motion Vector is evaluated for each of them. However, an object boundaries commonly do not coincide with the block boundaries and artifacts may be visible at object boundaries using this approach.
Current encoding standards, such as H.264, use variable block sizes and shapes to reduce block-related artifacts; this can be considered as an intermediate solution between the block-based approach and the full segment-based approach.
Segment-Based Motion Estimation (SBME) replaces the fixed blocks of traditional Motion Estimation algorithms with segments of arbitrary shapes and sizes. Motion Vectors are now assigned to segments instead of blocks and this allows to obtain motion boundaries with pixel accuracy (see figure 4.11).

Figure 4.11: Left: Block Partitioning. Right: Segmentation.

However, from a hardware implementation point of view, SBME has a significant disadvantage: since segments can be of arbitrary shapes and sizes, a straightforward implementation of an SBME algorithm will either suffer from an inefficient use of data memory bandwidth or irregular data addressing. This paper proposes a SBME algorithm that applies block-based processing to calculate Segment-Based Motion Vectors.
The algorithm has been implemented designing a custom VLIW ASIP processor (Very Large Instruction Word Application Specific Instruction-set Processor).
Apart from several general purpose functional units like an arithmetic and logic unit, a multiplier and an address generation unit, the ASIP processor contains some Application Specific Units (ASUs) used to accelerate the execution of video signal processing algorithms in general and of the modified SBME algorithm in particular.
The global architecture is shown in figure 4.12.

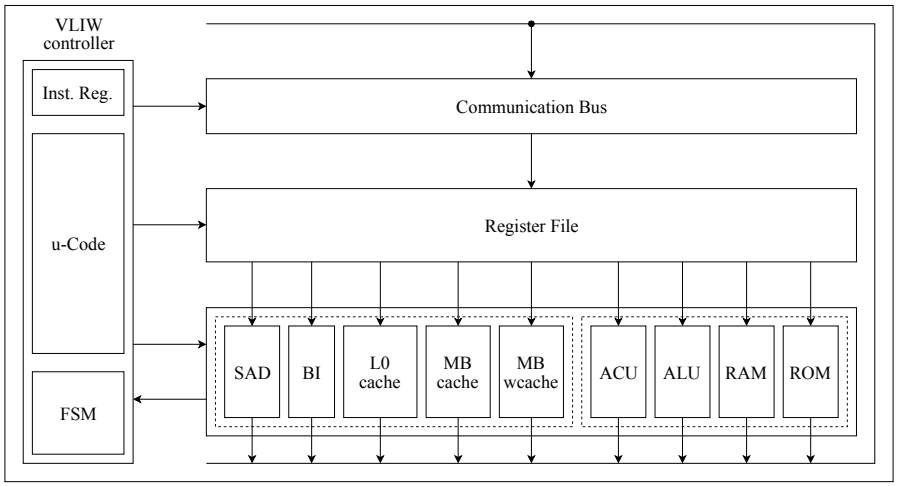


Figure 4.12: VLIW ASIP processor architecture. MIMD - Case of study 3.

The main functional unit used to speed up the Motion Estimation is the SAD unit: this unit calculates the sum of absolute differences of 16 pairs of pixels in parallel; pixels are read from the MB cache, that stores the selected Macro Block of the current frame, and from the L0 cache, that stores the entire Search Window for that Macro Block. Macro Blocks of size 16x16 pixels are used and each is processed in 16 clock cycles.
The modified SBME ASIP was designed using the A|RT tool-set. Table 4.3 summarizes the synthesis results of the design.

IC technology	0.18 um
Conditions	25 °C, 1.8 V
Area	2.48 mm^2
Clock frequency	100 MHz
Power	70.5 mW

Table 4.3: VLIW ASIP processor synthesis results, Case of study 1.

Chapter 5

Solutions comparison

This chapter compares the MECORE versions, described in chapter 3, to the design solutions found in the literature and analyzed in chapter 4.
Each section examines and compares the solutions from a different point of view:

- software used for the design;
- design approach;
- hardware features;
- algorithm used to compute the ME.

Software used for the design

The solutions found in the literature have been designed and implemented with the following software tools: LISATek, Synopsys Processor Designer, Xtensa Xpress, and A|RT.

LISATek and Synopsys Processor Designer are software that allow you to design ASIP processors through the LISA description language. LISA stands for Language for Instruction Set Architecture and is used to describe the instruction set and architecture of your own processor. After the design phase, the software generates a complete set of tools, e.g. a compiler, an assembler, a linker and a simulator, as well as the RTL description of the designed processor in VHDL or Verilog.
Similarly, the Xtensa Xpress software allows you to customize Tensilica's Xtensa processor cores, where Xtensa is the type of architecture, using a proprietary description language called TIE (Tensilica Instruction Extension).
The A|RT software is instead a high level synthesizer. The behavior of the processor is described using the C language and then the code is synthesized to obtain the RTL description in VHDL or Verilog. Differently from the LISATek design tool, the A|RT software does not allow a deep control of the final architecture, and is generally used to speed up the design process when your attention is more focused on the algorithmic behavior with respect to the processor final implementation.
On the other side, the three versions of MECORE have been implemented using the ASIP Designer software, described in detail in chapter 2. This software provides a set of tools, very similar to that offered by the LISATek and Synopsys Processor Designer, that assists the user across all phases of the design process.

In table 5.1, you can find the software used to design each ASIP processor coming from the literature.

Note that, the *custom* keyword means that the architecture has not been designed using a software tool, but has been directly described at the RTL level in VHDL or Verilog.

Case of study	Adopted software
SISD 1 [8]	LISATek
SISD 2 [9]	Custom
SISD 3 [10]	Custom
MISD 1 [11]	A|RT
SIMD 1 [12]	Synopsys Processor Designer
SIMD 2 [13]	Custom
SIMD 3 [14]	Synopsys Processor Designer
SIMD 4 [15]	Custom
SIMD 5 [16]	Custom
MIMD 1 [17]	Xtensa Xpress
MIMD 2 [18]	A|RT
MIMD 3 [19]	A|RT

Table 5.1: Software adopted by each case of study.

Design approach

As you can read in chapter 4, four main architectural design approaches have been defined, and each *case of study* classified accordingly.
For what regards the three versions of MECORE, the first and the second versions adopt a Single Instruction Single Data (SISD) approach, where neither instruction nor data parallelism is exploited, while the third version can be classified as a Single Instruction Multiple Data (SIMD) processor, due to the use of an architecture able to execute the same instruction on vectors of data in parallel.

The SISD approach has been exploited in two main ways:

- using the existing processor instructions to implement the required algorithms;
- designing an hardware accelerator to speed up the execution of the most complex functions.

Case of study [8] and the first version of MECORE implement the Motion Estimation algorithms using the instructions present in the processor model instruction set only, while case of studies [9] and [10], and the second version of MECORE implement hardware accelerators to speed up, mainly, the Sum of Absolute Differences (SAD), that is the most time expensive function of the Motion Estimation.

The SIMD approach, instead, has been exploited using different sizes of data parallelism.
Considering the literature: case of studies [12] and [14] propose a processor able to compute the SAD of a maximum of 256 points, i.e. locations in the Search Window, in parallel; case of study [13] implements an architecture able to perform the SAD between the current 4x2 Macro Block and nine 4x2 reference blocks of the Search Window in parallel; case of study [15] adopts from one to nine Integer Pixel Execution Units (IPEU), whose SAD engines operate on 64 bits; finally, case of study [16] implements the SAD16 instruction that allows to compute the SAD between two sets of 16 pixel values in parallel and accumulate the result.
Similarly to this final example, the third version of MECORE exploits a 128 bit level parallelism and computes the SAD between 16 pairs of pixel values in parallel, accumulating the result.

Hardware features

Table 5.2 shows the hardware results (whenever available) obtained after having synthesized the ASIP processors coming from the literature and the three versions of MECORE.

ASIP processor	IC technology	Maximum frequency	Gates	Area	Dynamic power	Leakage power
SISD 1 [8]	CMOS 0.13 um	160 MHz	42500	-	-	-
SISD 3 [10]	CMOS 0.18 um	80 MHz	15000	1.41 mm^2	-	-
SIMD 2 [13]	CMOS 130 nm	300 MHz	-	0.563 mm^2	-	-
SIMD 3 [14]	CMOS 90 nm	-	-	-	-	-
SIMD 4 [15]	CMOS 65 nm	135.28 MHz	-	-	-	-
MIMD 2 [18]	CMOS 0.18 um	100 MHz	-	-	-	-
MIMD 3 [19]	CMOS 0.18 um	100 MHz	-	2.48 mm^2	-	-
MECORE VER 1	CMOS 65 nm	699.30 MHz	-	0.022 um^2	51.03 uW	2.06 uW
MECORE VER 2	CMOS 65 nm	694.44 MHz	-	0.024 um^2	52.92 uW	2.14 uW
MECORE VER 3	CMOS 65 nm	558.66 MHz	-	0.048 um^2	56.28 uW	3.69 uW

Table 5.2: Synthesis results of each ASIP processor.

As you can notice, the three versions of the MECORE processor are able to reach a much higher maximum operating frequency with respect to the other case of studies. The reasons for that are mainly two. First, the use of a simple processor model (TMICRO) as a starting point for the design, guarantees a short critical path and, by consequence, a very high frequency; moreover, the added features, implemented to design the three versions, did not affect much the critical path of the circuit, allowing it to keep high performance. Second, the use of a 65 nm library of cells leads to a faster architecture with respect to the implementations with higher transistor gate lengths.

Algorithm used to compute the ME

In table 5.3, you can find the algorithm (whenever available) adopted by each case of study and by each version of MECORE to implement the Motion Estimation.

Case of study	Adopted algorithm
SISD 2 [9]	Modified Normalized Partial Distortion Search (MNPDS)
SISD 3 [10]	Ultra Low Complexity Motion Estimation Algorithm (ULCMEA)
MISD 1 [11]	Object-based Motion Estimation (OME)
SIMD 1 [12]	Full search
SIMD 5 [16]	New adaptive Motion Estimation algorithm
MIMD 3 [19]	Segment Based Motion Estimation (SBME)
MECORE VER 1	Four step search
MECORE VER 2	Four step search
MECORE VER 3	Four step search

Table 5.3: ME Algorithm adopted by each case of study.

Chapter 6

Conclusion

The Motion Estimation is one of the most complex and time consuming functions of any video compression standard. Its algorithm is mainly based on the computation of a single Motion Vector for each Macro Block of the current frame. To perform that, each Macro Block is compared to a number of candidate blocks of the reference frame, by evaluating the Sum of Absolute Differences (SAD), i.e. the most complex and time expensive function of the Motion Estimation, between them.
The first part of this thesis gave an introduction on the Motion Estimation algorithm, followed by a detailed description and analysis of the possible design approaches, i.e. ASIC, GP-uP and ASIP. The ASIP approach has been identified as the design solution that better guarantees high performance, still providing high design flexibility. Moreover, it allows the designer to explore the complete design space, from the pure software solution up to the pure hardware solution, and find the best performance and flexibility trade off, that satisfies the design constraints.
In the second part of this thesis, three versions of the MECORE microprocessor core have been implemented, following the design steps provided by the Synopsys training manual [1]. The obtained results showed a considerable increase in performance, moving from the first version to the third, due to the additional hardware features introduced to speed up the Sum of Absolute Differences (SAD). In particular, the implementation of a scalar accelerator allowed to perform the SAD between two pixel values in a single clock cycle, while the introduction of a SIMD architecture enabled the computation of the SAD between 16 pairs of pixel values (128 bits) in parallel. The maximum frequency reached by the final version of MECORE was 558.66 MHz, using a CMOS 65 nm technology. Moreover, the time required to compute the Motion Vector of a single Macro Block of size 16x16 pixels over a Search Window of 48x48 pixels was only 9.46 us; this at the cost of a considerable increase in area, of about 111%, with respect to the first version of MECORE.
In the last part of this thesis, several case of studies, found in the literature, have been classified into 4 main categories, i.e. SISD, MISD, SIMD and MIMD, depending on the adopted design approach, and then analyzed to give to the reader an overview of the content of each paper. Finally, the major features and results have been compared to the three versions of MECORE, previously implemented: the use of a newer technology, in combination with the implementation of faster architectures, allowed the MECORE versions to reach higher maximum operating frequencies, compared to any other design found in the literature.

6.0.1 Future works

Further works can be performed in order to improve the architecture of MECORE. This subsection proposes two possible future developments that may improve the performance of the processor:

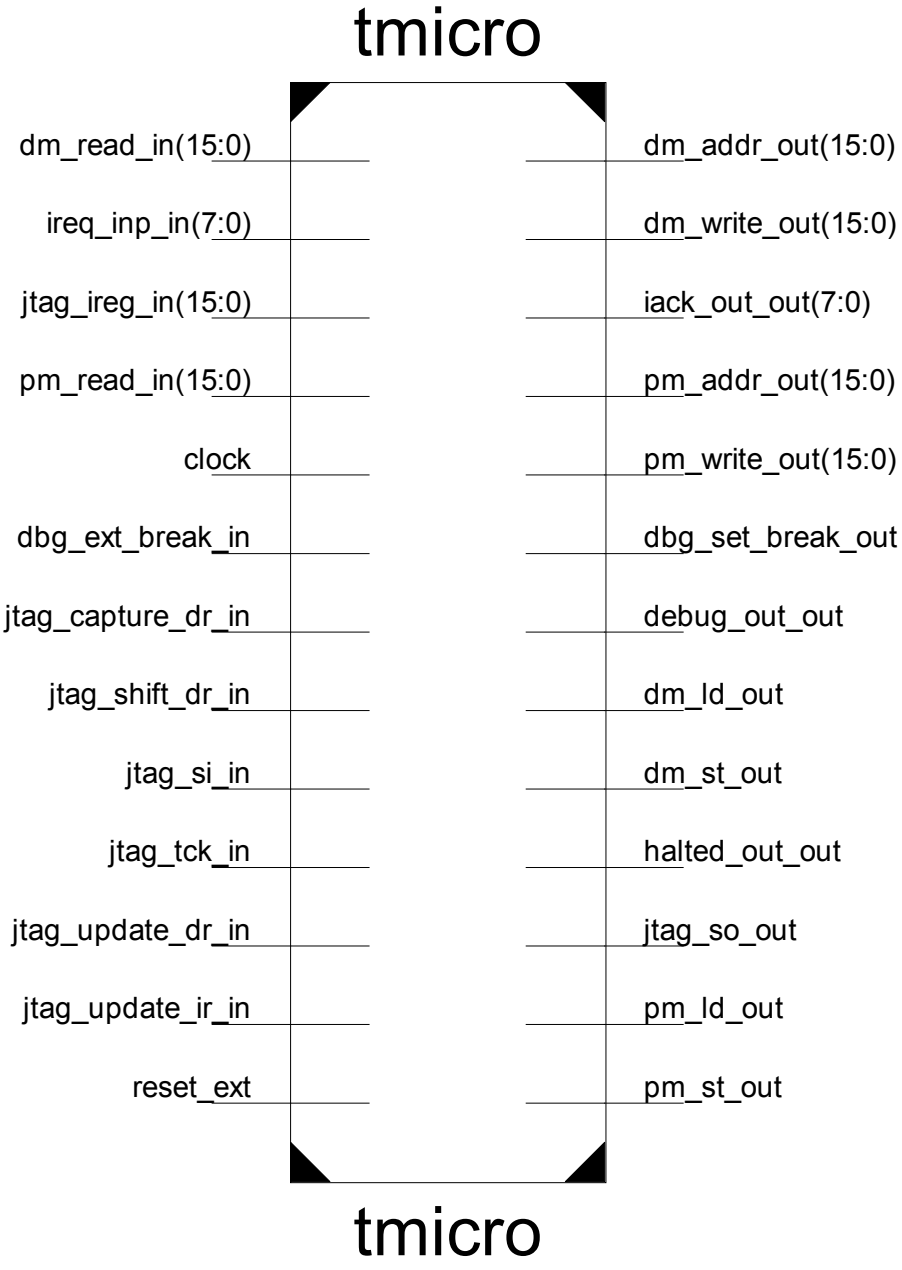
- the design of a SIMD hardware accelerator at the RTL level;

- the use of a MIMD architecture.

The SIMD hardware accelerator, used in the MECORE processor, is an automatically generated module implemented by ASIP Designer reading the nML specifications. However, describing the accelerator directly at the RTL level (in VHDL or Verilog) would allow a better control on the final implementation and may provide higher performance and lower area and power consumption.
At the same time, the MECORE processor does not exploit instruction parallelism. So, the implementation of a MIMD architecture would allow the processor to execute more then one instruction in parallel, with a considerable increase in speed. For example, the introduction of a Very Long Instruction Word (VLIW) architecture would lead to much higher performance, at the cost of a more sophisticated compiler and a reasonably more complex hardware.

Appendix A

TMICRO top-level block schematic



PORT DEFINITIONS:

clock → Clock Signal Port

reset_ext → Reset Signal Port

pm_read_in(15:0) → Program Memory Input Read Port

pm_addr_out(15:0) → Program Memory Output Address Port

pm_write_out(15:0) → Program Memory Output Write Port

pm_st_out → Program Memory Output Store Signal Port

pm_ld_out → Program Memory Output Load Signal Port

dm_read_in(15:0) → Data Memory Input Read Port

dm_addr_out(15:0) → Data Memory Output Address Port

dm_write_out(15:0) → Data Memory Output Write Port

dm_st_out → Data Memory Output Store Signal Port

dm_ld_out → Data Memory Output Load Signal Port

ireq_inp_in(7:0) → Interrupt Request Input Lines Port

halted_out_out → Halt Output Signal Port

debug_out_out → Debug Output Signal Port

iack_out_out(7:0) → Interrupt Ack Output Lines Port

jtag_tck_in → JTAG Clock Input Signal Port

jtag_ireg_in(15:0) → JTAG Interrupt Request Input Port

jtag_si_in → JTAG Scan In Input Signal Port

jtag_so_out → JTAG Scan Out Output Signal Port

jtag_capture_dr_in → JTAG Capture Input Signal Port

jtag_update_dr_in → JTAG Update DR Input Signal Port

jtag_shift_dr_in → JTAG Shift Input Signal Port

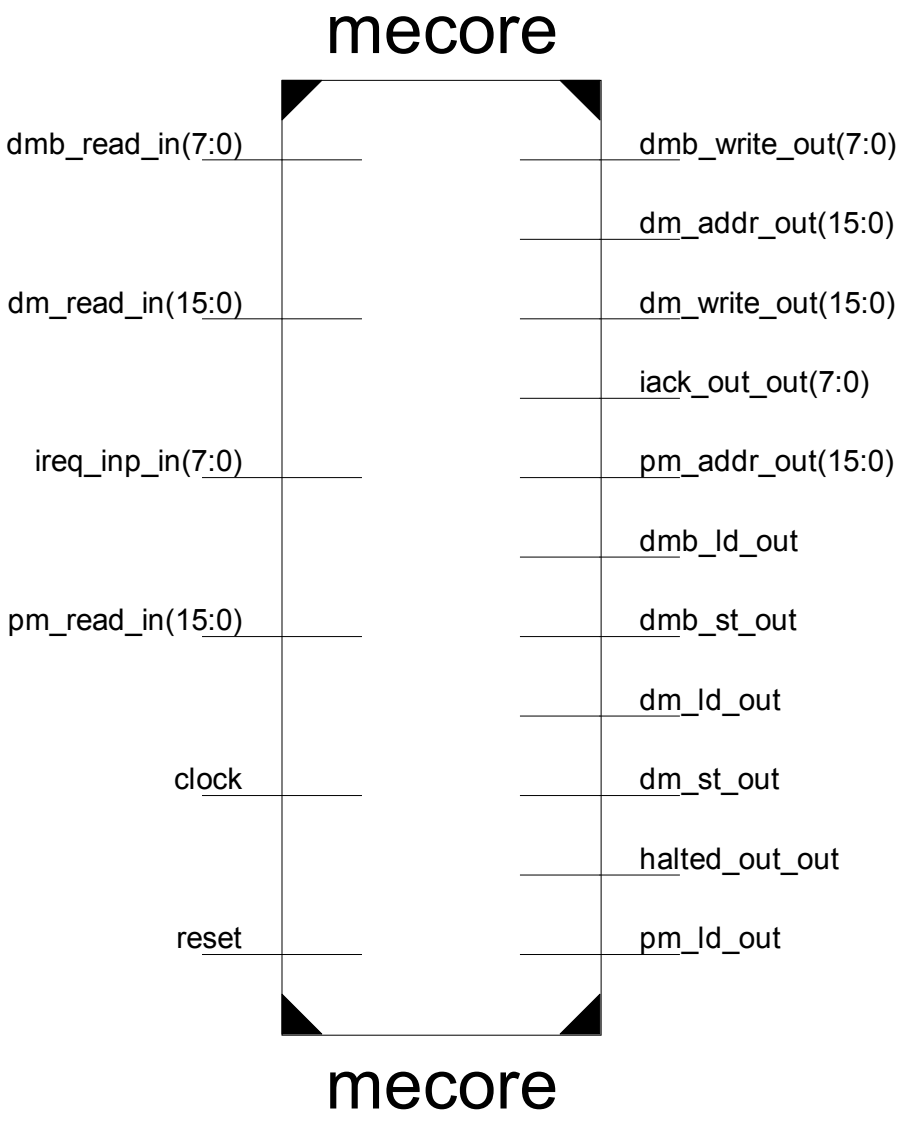
jtag_update_ir_in → JTAG Update IR Input Signal Port

dbg_ext_break_in → JTAG Break Input Signal Port

dbg_set_break_out → JTAG Break Output Signal Port

Appendix B

MECORE VER 1 top-level schematic



PORT DEFINITIONS:

clock → Clock Signal Port

reset → Reset Signal Port

pm_read_in(15:0) → Program Memory Input Read Port

pm_addr_out(15:0) → Program Memory Output Address Port

pm_ld_out → Program Memory Output Load Signal Port

dm_read_in(15:0) → Data Memory Input Read Port

dm_addr_out(15:0) → Data Memory Output Address Port

dm_write_out(15:0) → Data Memory Output Write Port

dm_st_out → Data Memory Output Store Signal Port

dm_ld_out → Data Memory Output Load Signal Port

dmb_read_in(7:0) → Byte Data Memory Input Read Port

dmb_write_out(7:0) → Byte Data Memory Output Write Port

dmb_st_out → Byte Data Memory Output Store Signal Port

dmb_ld_out → Byte Data Memory Output Load Signal Port

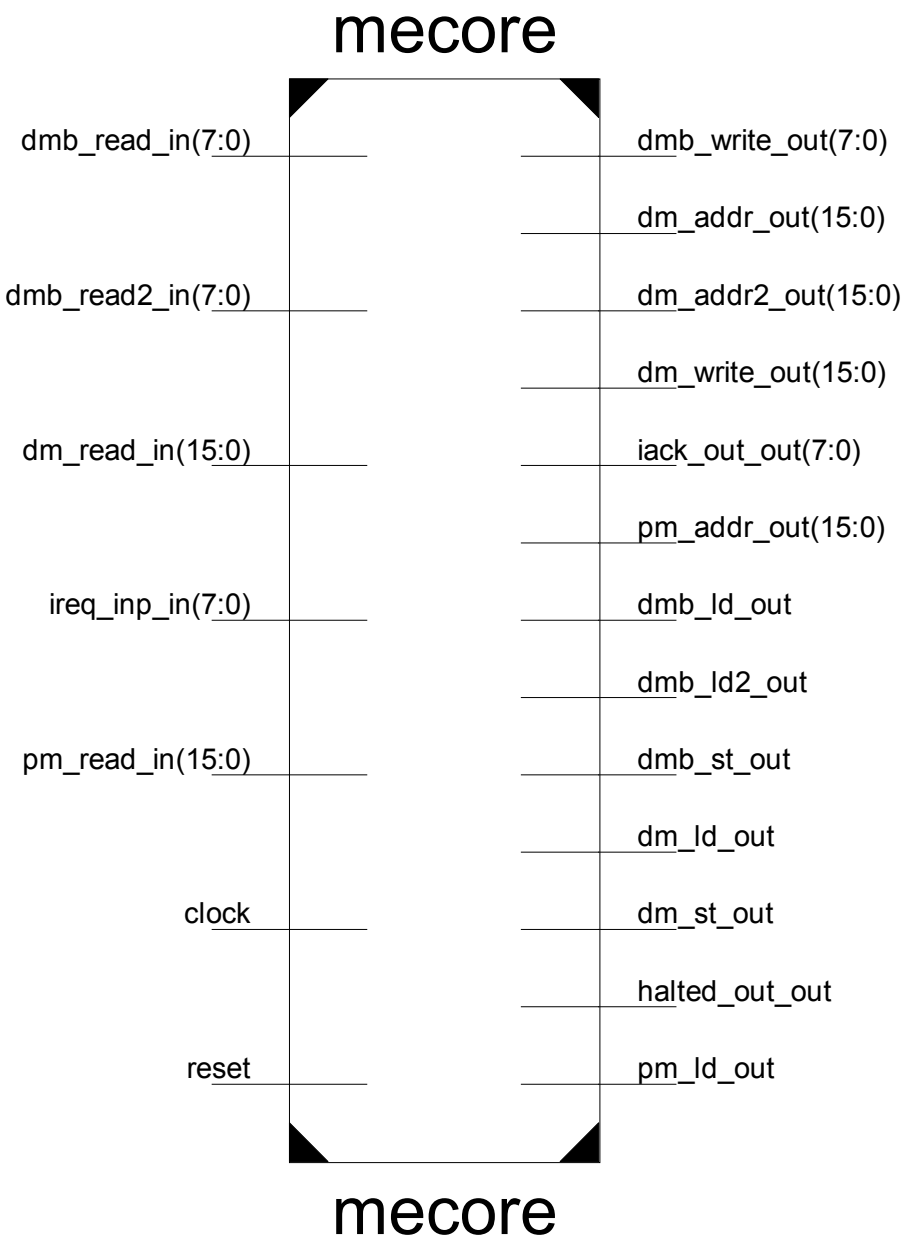
ireq_inp_in(7:0) → Interrupt Request Input Lines Port

iack_out_out(7:0) → Interrupt Ack Output Lines Port

halted_out_out → Halt Output Signal Port

Appendix C

MECORE VER 2 top-level schematic



PORT DEFINITIONS:

clock → Clock Signal Port

reset → Reset Signal Port

pm_read_in(15:0) → Program Memory Input Read Port

pm_addr_out(15:0) → Program Memory Output Address Port

pm_ld_out → Program Memory Output Load Signal Port

dm_read_in(15:0) → Data Memory Input Read Port

dm_addr_out(15:0) → Data Memory Output Address Port

dm_addr2_out(15:0) → Data Memory Output Address Port 2

dm_write_out(15:0) → Data Memory Output Write Port

dm_st_out → Data Memory Output Store Signal Port

dm_ld_out → Data Memory Output Load Signal Port

dmb_read_in(7:0) → Byte Data Memory Input Read Port

dmb_read2_in(7:0) → Byte Data Memory Input Read Port 2

dmb_write_out(7:0) → Byte Data Memory Output Write Port

dmb_st_out → Byte Data Memory Output Store Signal Port

dmb_ld_out → Byte Data Memory Output Load Signal Port

dmb_ld2_out → Byte Data Memory Output Load Signal Port 2

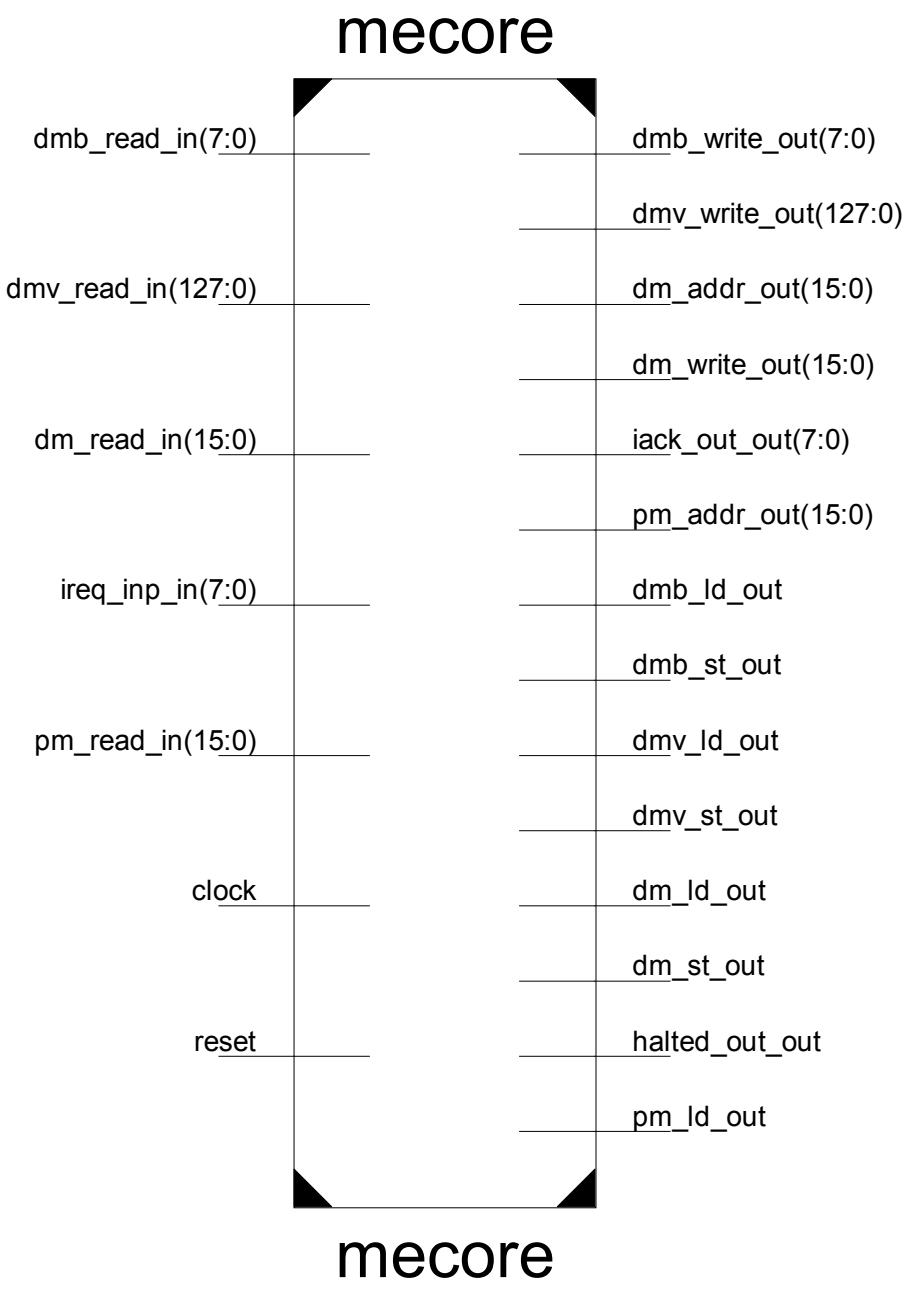
ireq_inp_in(7:0) → Interrupt Request Input Lines Port

iack_out_out(7:0) → Interrupt Ack Output Lines Port

halted_out_out → Halt Output Signal Port

Appendix D

MECORE VER 3 top-level schematic



PORT DEFINITIONS:

clock → Clock Signal Port

reset → Reset Signal Port

pm_read_in(15:0) → Program Memory Input Read Port

pm_addr_out(15:0) → Program Memory Output Address Port

pm_ld_out → Program Memory Output Load Signal Port

dm_read_in(15:0) → Data Memory Input Read Port

dm_addr_out(15:0) → Data Memory Output Address Port

dm_write_out(15:0) → Data Memory Output Write Port

dm_st_out → Data Memory Output Store Signal Port

dm_ld_out → Data Memory Output Load Signal Port

dmb_read_in(7:0) → Byte Data Memory Input Read Port

dmb_write_out(7:0) → Byte Data Memory Output Write Port

dmb_st_out → Byte Data Memory Output Store Signal Port

dmb_ld_out → Byte Data Memory Output Load Signal Port

dmv_read_in(127:0) → Vector Data Memory Input Read Port

dmv_write_out(127:0) → Vector Data Memory Output Write Port

dmv_st_out → Vector Data Memory Output Store Signal Port

dmv_ld_out → Vector Data Memory Output Load Signal Port

ireq_inp_in(7:0) → Interrupt Request Input Lines Port

iack_out_out(7:0) → Interrupt Ack Output Lines Port

halted_out_out → Halt Output Signal Port

Bibliography

[1] Target Tmotion core, Implementation of a Motion Estimation Algorithm, Synopsys, November 2015, Version K-2015.12.

[2] Overview of the Manuals, Synopsys, September 2017, Version N-2017.09.

[3] Example Processor Models, Synopsys, September 2017, Version N-2017.09.

[4] The nML Processor Description Language, Synopsys, September 2017, Version N-2017.09.

[5] Primitives Definition and Generation manual, Synopsys, September 2017, Version N-2017.09.

[6] Tmicro Core, Processor Manual. Synopsys, March 2017. Version M-2017.03.

[7] Motion Compensated Interframe Coding for Video Conferencing. J. Koga e.a. In Proceedings of the National Telecommunications Conference, pages G5.3.1−5.3.3, 1981.

[8] Muhammad Rashid, Ludovic Apvrille, Renaud Pacalet, *Application Specific Processors for Multimedia Applications*, 2008 11th IEEE International Conference on Computational Science and Engineering, pages 109-116, 2008.

[9] Jin Ho Ha, Jin Soo Kim, Myung H. Sunwoo, *An ASIP Approach for H.264/AVC Implementation Having Novel Coprocessors*, 2007 IEEE Workshop on Signal Processing Systems, pages 499-504, 2007.

[10] Seiichiro Hiratsuka, Satoshi Goto, Takeshi Ikenaga, *A 0.3mW 1.4mm2 Motion Estimation Processor for Mobile Video Application*, 2006 IEEE Asian Solid-State Circuits Conference, pages 103-106, 2006.

[11] J. C. Greiner, R. Sethuraman, J. van Meerbergen, G. de Haan, *A cost-effective implementation of object-based motion estimation*, 2003 IEEE Workshop on Signal Processing Systems, pages 148-153, 2003.

[12] Sung Dae Kim, Myung Hoon Sunwoo, *Efficient program control schemes for Motion Estimation specific processor*, 2011 International SoC Design Conference, pages 270-273, 2011.

[13] Ingoo Heo, Sanghyun Park, Jinyong Lee, Yunheung Paek, *An ASIP approach for motion estimation reusing resources for H.264 intra prediction*, 2010 International SoC Design Conference, pages 186-189, 2010.

[14] Hee Kwan Eun, Sung Jo Hwang, Myung Hoon Sunwoo, Young Hwan Kim, Hi Seok Kim, *Integer-pel Motion Estimation specific instructions and their hardware architecture for ASIP*, 2011 IEEE International Symposium of Circuits and Systems (ISCAS), pages 953-956, 2011.

[15] Jose Luis Nunez-Yanez, Eddie Hung, Vassilios Chouliaras, *A configurable and programmable motion estimation processor for H.264 video codec*, pages 149-154, 2008.

[16] Svetislav Momcilovic, Numo Roma, Leonel Sousa, *An ASIP approach for adaptive AVC Motion Estimation*, 2007 Ph.D Research in Microelectronics and Electronics Conference, pages 165-168, 2007.

[17] Hong Chinh Doan, Haris Javaid, Sri Parameswaran, *Multi-ASIP based parallel and scalable implementation of motion estimation kernel for high definition videos*, 2011 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia, pages 56-65, 2011.

[18] A. Beric, R. Sethuraman, J. van Meerbergen, G. de Haan, *Memory-centric motion estimator*, 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design, pages 816-819, 2005.

[19] Patrick Meuwissen, Ramanathan Sethuraman, Fabian Ernst, Harm Peters, Rafael Peset Llopis, *Segment-based motion estimation using a block-based engine*, 2005 13th European Signal Processing Conference, 2005.