POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Application Specific Instruction-set Processor for HSG-based Pedestrian Detection Algorithm

Relatori:
Prof. Guido Masera
Prof. Maurizio Martina

Candidato:
Beatrice Giannetta

Aprile 2019

Acknowledgments

Table of contents

List of tables

List of figures

Chapter 1

Introduction

The growth in the number of automobiles in circulation in the last century has led to a rise in road accidents that represent an important and common cause of fatalities. Each year, in fact, about 10 million persons are involved with road injuries around the world and accordly with the data reported in the "Global status report on road safety" [6] it can be asserted that 1.2 million of world's inhabitants die each year.

The scientific community and the automotive world have focused their attention on the development of protection systems such as Advanced driver assistance systems also called ADASs. In particular pedestrian protection systems or PPSs, have been an important research area in recent years with the aim of improving traffic safety. The purpose of these systems is to detect stationary but also moving pedestrians inside a ROI (Region of interest), and to perform specific actions in order to avoid the collision.
Moreover object detection is becoming an important part of smart systems for video surveillance in the modern era.

To design reliable pedestrian detection systems complex algorithms and implementation schemes are needed to meet the challenge of detecting human faces against backgrounds. Three main aspects represent a difficult problem in the operations that feature the detection of pedestrians :

1. Continuous changing of scenarios : pedestrian localization represents one very difficult challenge due to very different appearance of humans, in fact many factors like different clothes, different dimensions and proportions, the variety of positions articulate exhibited by human make this process very complicated. [7].

2. Surroundings conditions : the unstructured and various environment that

characterize the urban area contexts, changing of light conditions and the possible overlap of the pedestrian with other objects make very difficult to develope systems with the level of robustness required.

3. Real-time requirement : speed of reaction and accuracy, so the ratio between miss detections and false positive is a fundamental point because this systems could determine the live of people.

Chapter 2

Pedestrian Detection Algorithm

A pedestrian detection framework is composed of some steps.

- Preprocessing: performed in order to reduce noise and to improve lighting conditions. An image could have any size, but there is a constraint to be considered: patches analyzed must have a fixed aspect ratio. In the cases considered in this work the aspect ratio is 1:2, so input images are cropped to this scope[8]. Moreover *Dalal* and *Triggs* also evaluate gamma correction as a step of preprocessing process but accordly to [7] normalization doesn't have a significant effect on performances.

- Feature Vector Extraction: the traditional image encryption is not suitable for classification. With the aim of underlining significant features in the image, a lot of features extraction techniques are exploited to make able the classifier to recognize the presence of a target like a possible pedestrian. The output of feature extraction operation is a set of constant length vectors that are then given as input to the classifier.

- Feature Vector Classification: given a set of objects represented as feature vectors associated to, in general, N dimensional spaces, and a related class label for each object that represents the category which it belongs, the classifier is a model that can predict the class given the features. The model itself can take on many different forms: linear classifiers, decision trees, neural networks and support vector machines are a few popular examples. In particular, the binary classifier is considered when the space of categories allowed is only of two elements. It is the case of the considered algorithm where the two predefined classes are pedestrian or non-pedestrian.Two steps characterize the classification:

  - Training phase: a classifier is learned from a training set of measurements that are representative of the data. In this way the classifier is assisted to build, itself, a classification rule.
  - Testing phase, the learned classifier is used to discriminate between measurements in a test dataset that contains information about realistic pedestrian situation. In this step the performances of classification have to be validate.

  Linear Support vector machines SVM and Ada Boost are the methods most used thanks to their theoretical results, good performance and extensibility. They are also particularly well appropriate because of their speed. Nonlinear kernel isn't common, with the exception of HIK(histogram intersection kernel).

  In order to perform classification, training and and testing phases, and determine efficiently the performance of pedestrian recognition, different datasets have been created:

  1. INRIA
  2. Caltech;
  3. ETH;

- Postprocessing: performed in order to enhance results of detection trought non maxima suppression[9]

2.1 Histogram of Oriented Gradient

The HOG descriptors has been proposed in 2005 by *N.Dalal* and *B.Triggs* [7] and has characterized by an high accuracy on the INRIA data set. This method is based on Sliding Windows Approach. The basic idea is that human shape can be characterized looking at the distribution of pixel intensity gradient and edge directions. The reference image is represented by Feature Descriptors that collect useful shape information. An image of size width x height x 3 (3 is the number of channels if an RGB image is considered) to a feature array/vector of length n.

2.1.1 Feature Extraction

Two computation units are considered in order to obtain HOG descriptors : *cell* and *block*. According to [1] the size of the cell is tipically 8x8 pixels and a block contains 4 cells, so its dimension is 16x16 pixels.
Features extraction can be executed following three steps :

- Gradient Computation
- Histogram Generation (Gradient Vote)
- Histogram Normalization

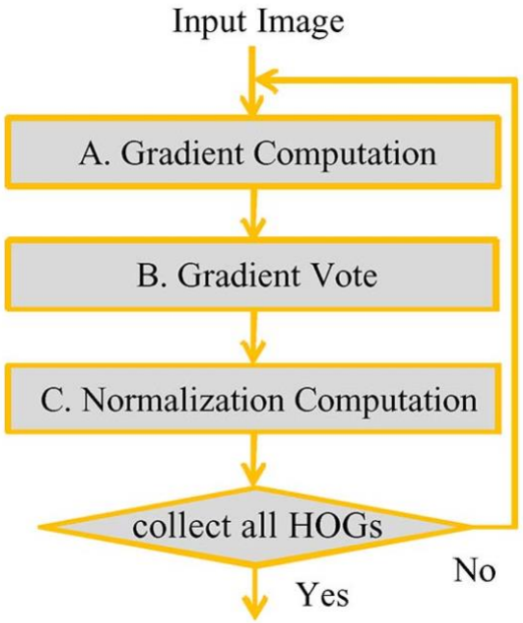

Figure 2.1: Flow of Feautures Extraxion Operation[1].

Gradient Computation The Gradients Gx and Gy are directional changes in the intensity or color of an image. Their magnitude is enough high around edges and corners (that are defined "abrupt intensity changes") so these parameters allow

to obtain precise informations about the shape of the object.
The horizontal and vertical Gradients can be defined as:

$$f_x(x,y) = \frac{\partial f(x,y)}{\partial x} \tag{2.1}$$

$$f_y(x,y) = \frac{\partial f(x,y)}{\partial y} \tag{2.2}$$

These values are computed by utilizing filter kernels for the input image, the first for G_x and the second for G_y :

$$\begin{cases} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} & and \quad \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \end{cases} \tag{2.3}$$

Magnitude and orientation are then computed using the following formulas:

$$\begin{cases} m(x,y) = \sqrt{f_x(x,y)^2 + f_y(x,y)^2} \\ \theta(x,y) = arctan\frac{f_y(x,y)}{f_x(x,y)} \end{cases} \tag{2.4}$$

Histogram Generation After the computation of magnitude and orientation, for each image patch, and for each pixel of a block it is obtained a set of 2 values. Each pixel computes one weighted vote for an histogram channel according to the direction of the element of the gradient centred on it, these votes are accumulated into orientation bins over each cell, the local spatial area considered .The vote is related to the gradient magnitude of the pixel, the histogram is composed by 9-bins if it is considered the so-called "unsigned gradients representation". In this case, the bins are spaced over 0, π. In order to reduce the effect of the aliasing, votes are bilinearly interpolated between the neighbouring bins for both position and orientation.

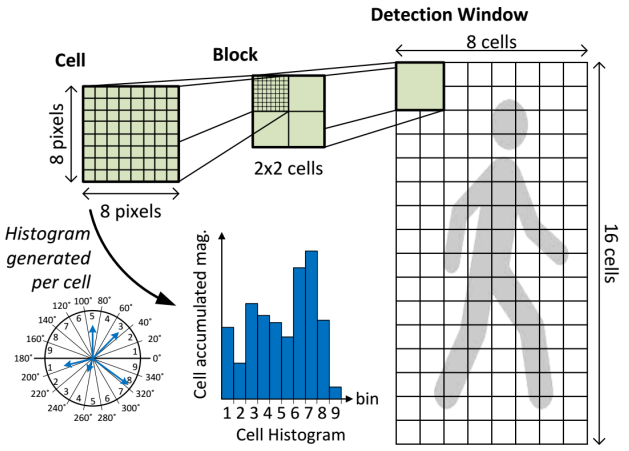

Figure 2.2: Detection Windows divided into block and cells, cell histogram generated[2].

The steps needed to fill an histogram are presented in the following picture. In order to make more direct the example a simplest situation is reported [3], the cell has a dimension of 4x4 pixels and the histogram presents only 3 bins.

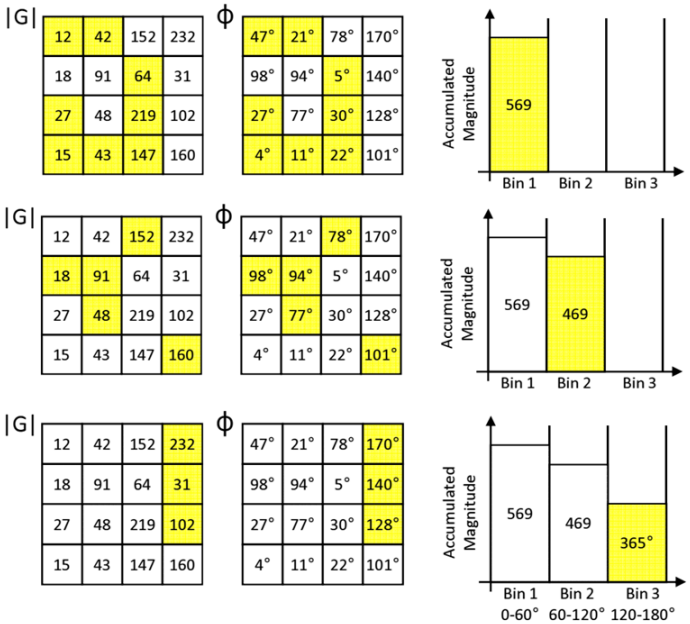

Figure 2.3: Simplified Example of how histogram are filled[3].

At this point, all the pixels in a cell have contribuited to the 9-elements vector

and the HOG for this cell is obtained.

Histogram Normalization As last step a large histogram is generated by combining all histograms which belonging to one block that consist of four cells. Finally the big histogram obtained has to be normalized, in fact the obtained values are clearly influenced by light changing and normalization allows to overcame this problem and makes better the invariance to illumination and shallowing. Moreover, once the computation of the current block is finished, the next computation is performed and the next block considered overlaps of one cell (8x8 pixels) the previous.

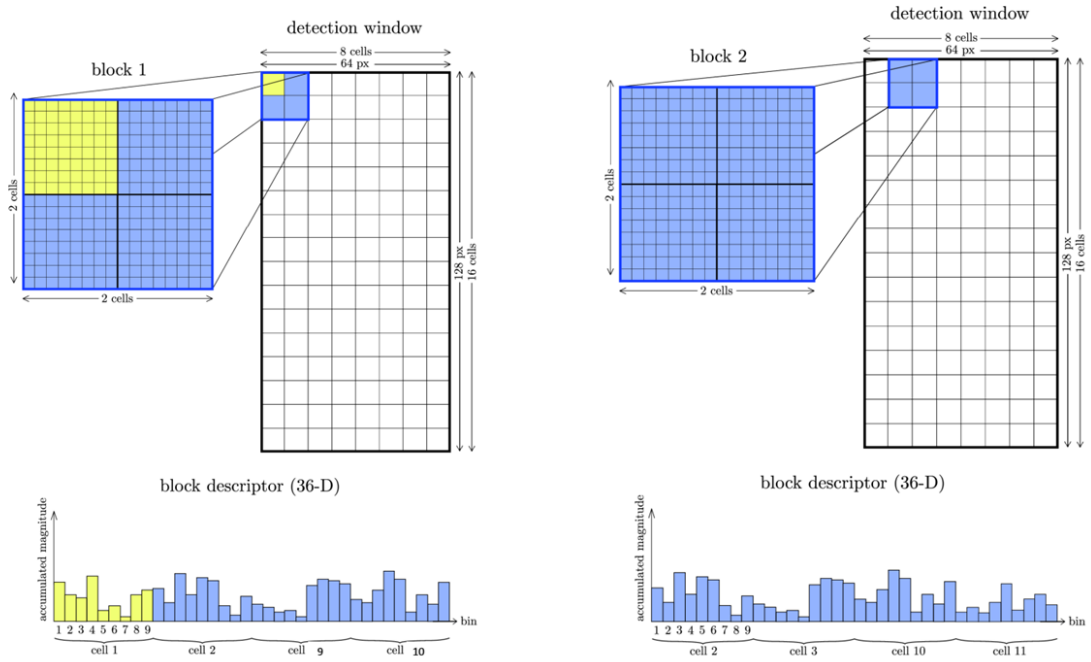

Figure 2.4: Normalized histogram related to overlapping cells[3].

The overlapping blocks make possible that the descriptors at the end contains different contributions from one single cell which is normalized with respect to a different block. The normalized result can be realized by appling the so called "L2-norm":

$$V_i = \frac{V_i}{\|v\|^2 + \epsilon^2} \tag{2.5}$$

where V_i is the vector which correspond to the combined histogram for the whole block, i is a number from 1 to 36 ([9 bins]x[4 cells]), $\|v\|^2 = v_1^2 + v_2^2 + ... + V_{36}^2$ e ϵ is a constant very small to prevent dividing by zero.
Finally the output vector is obtained, its dimension can be computed tacking into account some considerations:

- The detection window has a size of 64x128 and is divided into blocks of dimension 16x16. The blocks are overlapped each other and share one cell. So sliding the window 7 blocks are considered for the horizontal dimension and 15 for vertical one. In total $7 * 15 = 105$ overlapping blocks are handled.
- For each block a vector of features of size 36x1 is obtained. The final concatenated vector have dimension of $105 * 36 = 3780$.

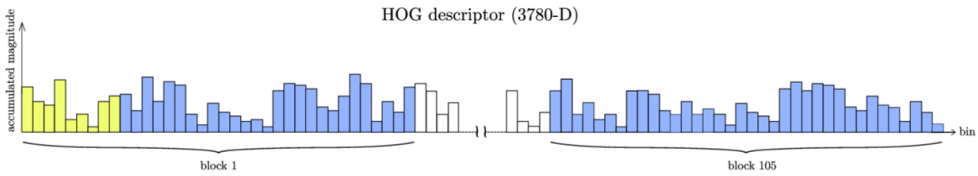

Figure 2.5: Finel vector of descriptors of dimensions 3780x1[3]

2.2 Histogram of Significant Gradient

HOG combined with SVM classifier is by now the most performant feature detector. With the aim to have better detection rate, the world of research has combined HOG with more complicated features world in cascade. But since HOG feature

extraction requests floating point computation and a lot of memory accesses, the obtained solutions in hardware and software result to be very power expensive and complex. In [4] it is proposed a pedestrian detection framework which is efficient and fast, characterized by less expensive computation. In fact in the proposed solution, the orientation histograms are built without utilizing floating point operations but exploiting the capability of discrimination of locally significant gradients in addiction to a fast Look-up-table based Support Vector Machine classifier. According to [4] HSG presents better performance with respect to HOG on INRIA and ETH Data sets.

HSG Computation Starting from detection windows of 64x128, blocks of dimension 8x8 are extracted. For each pixel belonging to the block gradient magnitude and direction are computed. Now, while in HOG method histograms were filled exploiting bilinear interpolation, in HSG solution the average value of gradient magnitudes related to the block is computed. This parameter is used as a threshold value: only at the edges that have a higher value of magnitude is allowed to binary vote the corresponding bin. The final features vector is obtained by concatenating the histograms coming from overlapping blocks (stride of 4). In this way, it is sure that the consequences of the variation in local illumination are neglected. This solution is simpler than HOG not only because the algorithm has been reduced in complexity, but also because only binary voting is allowed so, only integer operands are involved in the computation. In figure a result of HSG is illustrated. Also the EOH method is shown, a difference can be noticeable: HSG captures the contours of the figure in a more efficient way, it captures only dominant edges while EOH allows all gradients to vote and histograms appear distributed uniformly, the shape is not clearly recognizable.

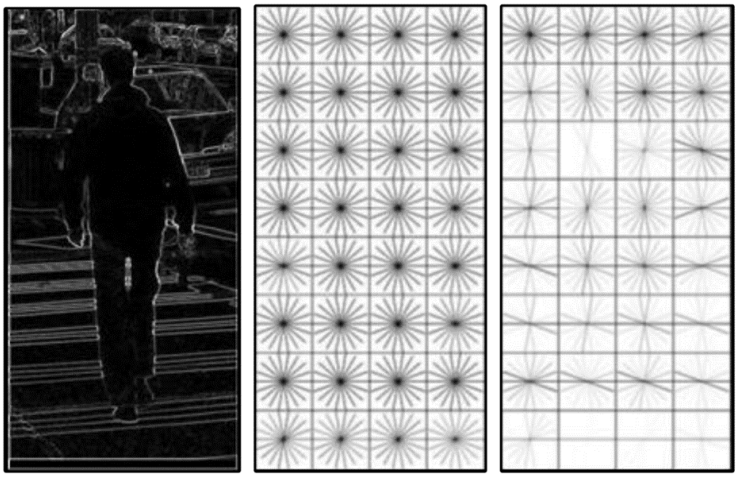

Figure 2.6: Pictorial illustration of histograms corresponding to EOH and HSG in nonoverlapped cells [4].

HSG Classification An additional advantage is that final feature vector is classified with SVM method using a LUT approach. This also allows reducing the computational complexity avoiding floating point value. The following formula shows the h SVM classification function:

$$h(x) = \sum_{i=1}^{n} \left(\sum_{j=1}^{m} \alpha_j K(x(i), SV_j(i)) \right) + b \tag{2.6}$$

where:

- $\mathbf{x}$ is equal to the input feature vector composed by n elements
- $\mathbf{SV_j}$ is the element in position jth of support vector
- $\mathbf{i}$ is the access index for SV_j and x
- $\alpha_\mathbf{j}$ represents support vector learned coefficient
- $\mathbf{b}$ is the learned bias
- $\mathbf{K}$ is the kernel function [10]

k, the kernel function considering the linear case is a multiplication, also $x(i)$ is a linear term so it can be put out of the first summation.

$$h_{linear}(x) = \sum_{i=1}^{n}(x(i)\left(\sum_{j=1}^{m}\alpha_j SV_j(i)\right) + b \tag{2.7}$$

In this way the term contained in the brackets can be calculated previously, so the equation above can be rewritten as:

$$h_{linear}(x) = \sum_{i=1}^{n}(x(i) * C(i)) + b \tag{2.8}$$

where C_i is the precomputed term that contains the support vector and the learned coefficient.

It can be observed that SupportVectorMachine classification, for the linear case is simply a dot product among input feature vector and coefficient vector summed to the bias b.
In general, the kernel function is not linear but finds the minimum of the elements corresponded both input feature vector and the current support vector.

$$h(x) = \sum_{i=1}^{n}\left(\sum_{j=1}^{m}\alpha_j min(x(i),SV_j(i))\right) + b \tag{2.9}$$

The solution useful to reduce the run-time computation can be to utilize a LUT considering an input vector that only allows integer value in a reduced dynamic range. This result can be reached by previously computing the inner summation in (2.9) considering all possible value assumed by x_i and storing them in a 2D LUT $T(i,k)$[10] that has the following form:

$$T(i,k) = \sum_{j=1}^{m}(\alpha_j min(k,SV_j(i))) \tag{2.10}$$

$$h_{HIK}(x) = \sum_{i=1}^{n}(T(i,x(i))) + b \tag{2.11}$$

Looking at the relation (2.10), it can be asserted that $T(i,k)$ allows only values of x_i greater than zero and k can vary from 0 to the maximum of x_i. $T(i,k)$ presents

floating point values because of $alpha_j$ is a real value and it is multiplied with an integer value.

The greater advantage utilizing the LUT is that in this case only n floating point addictions are exploited to perform classification, n floating point multiplications are prevented. This can be traduced in terms of performance in a evident speedup and better discrimination Power.

Chapter 3

ASIP

3.1 ASIP Design Approach

ASIP stands for Application Specific Instruction set Processor. It can be seen as an intermediate approach between two opposite method to obtain processors: Asic and General Purpose uP.

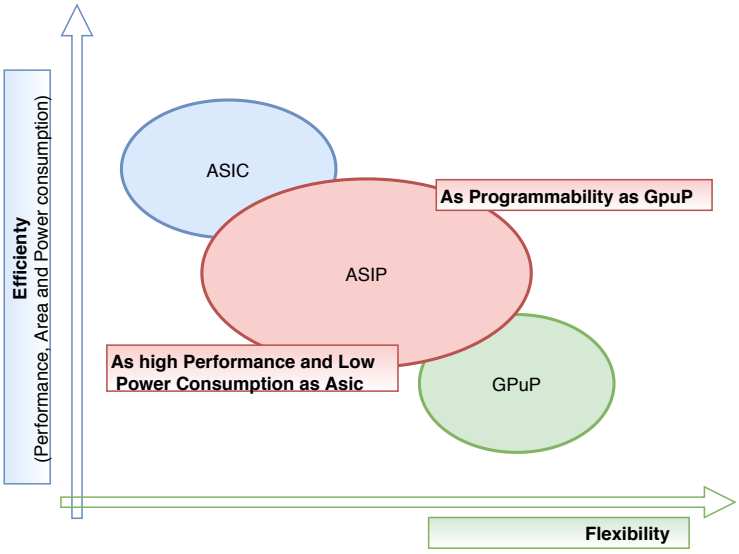

Figure 3.1: ASIP vs ASIC and GPuP

ASIC

Application Specific Integrated Circuits or ASICs are non-standard integrated circuits that have been custom designed for a specific use or application. ASIC is a completely hardwired hardware design, where algorithms are completely described

at RTL level and then realized in silicon. They are incredibly expensive, time-consuming, and resource-intensive to develop, but they offer extremely high performance, smallest area and low power consumption. The main disadvantage of this method is the total lack of flexibility of the project: when a function is implemented in hw it is impossible to modify the algorithm; the only solution is to re-design the whole circuit. This implies very high additional cost. For this reason, ASIC design is only used for applications with very strict constrains and is a dominant solution when the level of integration is limited.

GP*u*P

General purpose instruction processors are programmable devices used in a variety of applications that have dominated computing for a long time. In this approach, design is handled at high level, algorithms are implemented at software level, by using C or Assembly languages. It is a pure software design and it does not allow any optimization at hardware level. They are characterized by low design costs and high flexibility but present big area, high power consumption and tend to lose performance when dealing with non-standard operations and non-standard data not supported by the instruction set format.

ASIP

ASIPs, instead, represent a third implementation option for the design of a lot of application in which the power, area and performance of a standard general purpose processor is not enought and hardware accelerators of ASIC are not sufficient flexible. Starting from a general purpose processor, this approach is implemented to optimize it. In fact its assembly instruction set is designed to accelerate the most appearing function and critical functions, its architecture is modified in order to accelerate them to make ASIP fit to the specific application.

One of the key part of the design of an ASIP is the choice of the general purpose processor which is used as a starting point for the project. It is important to consider two main aspects:

1. Wasted area and power have to be avoided, so it has to be simple.

2. It has to be powerful enough in order to allow to perform all the operations needed to execute the application with a low number of instructions.

3.2 ASIP DESIGNER Tool

ASIP DESIGNER by Synopsys is a tool which allows to design, program and verify ASIPs. It provides all aspects of design of ASIP, its main capabilities concern the fast exploration of architectural possibilities, development of a C C ++ compiler based SDK (software development kit) that adapts automatically to architectural changes, simulation of instruction set and automatic generation of synthesizable RTL optimized in terms of area and power.

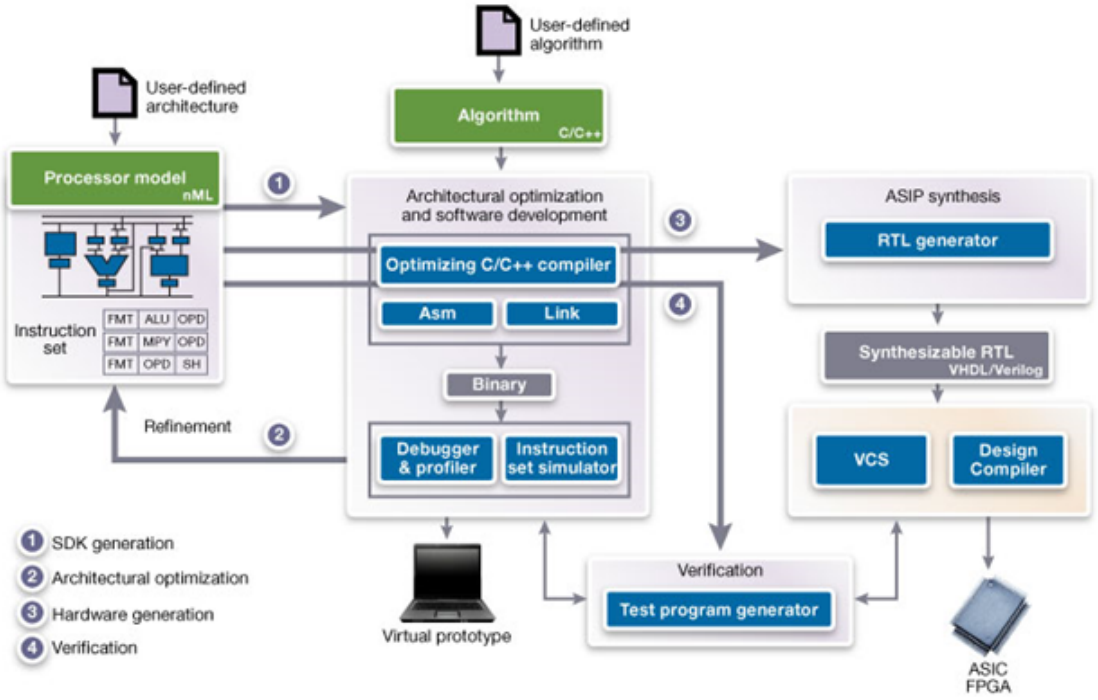

Figure 3.2: ASIP Designer Tool Flow [5]

The technology of Asip Dsigner supports different features:

- Modeling the ISA of the ASIP using the processor description language nML, where nML is a high level language used to define the instruction set and processor architecture. This description includes the instruction pipeline, the instruction set , the primitive operations of processors and the resurces that it exploit. In addition nmL language allows to model the Input Output interfaces.

- Compiler in the loop technology that enables the generation of SDK (software development kit) for each ASIP.
  The software is made of different components:

  - An optimizing compiler, that generates automatically the code. The compiler uses C, C++, OpenCL C. The compiler supports the data-level parallelism, the instruction level parallelism, the pipelined instructions, specialized functions arithmetic, data-type custom.
  - A linker that builds an executable file from separately compiled Elf/Dwarf object files for different C functions.
  - An assembler and disassembler that allows the translation of the machine code written in assembly into binary format and viceversa.
  - A fast instruction-set simulator that provides cycle and instruction accurate abstraction level.
  - A multicore debugger that is used to connect the instruction set of simulators and on-chip debug hardware.

- The generation of an hardware implementation optimized in terms of area and power, using synthesizable VHDL or Verilog. In order to support the debug on chip a debug controller and Jtag can be generated

- Multifaceted capabilities of verification with an automatically generation of specific test program written in C or in assembly code.

Chapter 4

ASIP Implementation

In this chapter is presented the microprocessor core in different versions optimized for pedestrian detection algorithm. The starting point of the obtained microprocessor is the Tmotion processor provided by ASIP Designer. It has the same instruction set and structures and additional special features have been added to improve performances.

Tmotion

Tmotion[11] is based on the basic processor Tmicro[12] presents in ASIP Designer library, it is a 16 bit microcontroller general purpose characterized by the following architectural features:

- 16 bit ALU able to perform 16 bit instructions for integer arithmetic operations,compare and bitwise logical operations. They operate on a register file with eight fields.
- Instructions for integer multiplication among 16 bit operands producing a result on 32 bit, stored on 2 separated registers of 16 bit.
- Instruction for 16 bit shift .
- instructions for 16 bit division .
- Instructions for load and store operations from and to a data memory (address space=64k words), characterized by indirect addressing and by little endian format.
- Instruction fetch from the program memory(address space=64k words) characterized by little endian format.

- Control instructions like subroutine call, jumps and return.
- Architecture characterized by three stages of pipeline that are IF, ID, E1.
- Zero overhead loop.

In addition, it provides instructions to store and load bytes in memory. Byte access implies that the smallest possible value (the value that occupies one address location in the memory) is a byte. In fact, as unsigned bytes are used to represent pixel values in application of video processing (they have a range from 0 to 255), these additional features have been inserted to improve memory usage. The extensions added to the processor model are:

- A byte memory DMb.
- Byte load and store instructions.
- Representations for the C built-in character types were added to the compiler header file.

Byte load and store instructions added to the instruction set are shown in figure.

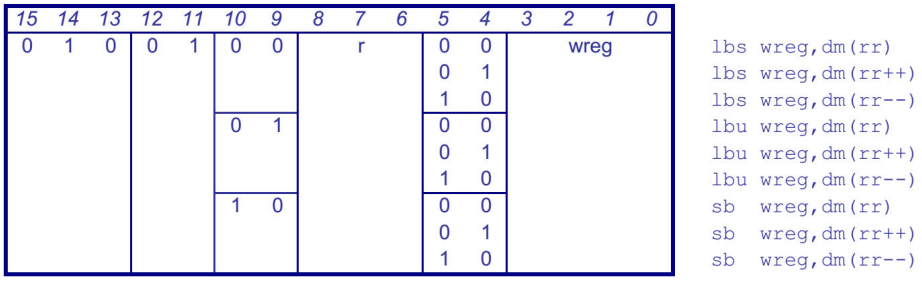

Figure 4.1: Byte Instructions

The above table represents the binary encoding and the assembler sintax of the nine instructions added.

The instructions *lbs* executes an operation of signed byte load. Before the byte value is written in the destination register that is of 16 bits, it need to be sign

extended at the Most Significant Bit side thanks to the *extend_sign* function.

$$lbs : wreg \leftarrow extend\_sign(DMb[R[r]])$$

The instruction *lbu* executes an operation of unsigned byte load. Before the byte value is written in the destination register, it need to be zero extended at the Most Significant Bit side thanks to the *extend_zero* function.

$$lbs : wreg \leftarrow extend\_zero(DMb[R[r]])$$

The instruction *sb* executes an operation of byte store. The memory location of destination have a size of 8 bits so only the eight least significant bits of the source register have to be stored. The function *extract_low* is used to this aim.

$$sb : DMb[R[r]] \leftarrow extract\_low(wreg)$$

These instructions utilize indirect addressing, the address is taken from the R register file and its field can be adeguately postincremented or postdecremented.

4.1 SW Implementation

This first Version is an absolute software implementation where the HSG algorithm has been compiled on the TMotion core without any modification, no accelerating instructions or hardware support has been added.

4.1.1 C_Code

The Source code is composed of:

1. Y.inc: this file contains a 134x70 pixel windows, obtained by the reference frame. The YCoCg (luminance + offset orange + offset green) color space is considered, it ensures improved coding gain relative to both RGB and YCbCr[13].The YCoCg color space transform is defined by:

$$\begin{bmatrix} Y \\ Co \\ Cg \end{bmatrix} = \begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 0 & -1/2 \\ -1/4 & 1/2 & -1/4 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{4.1}$$

2. *HSG_ASIP.c*: that contains the C source code able to implement the HSG pedestrian detection algorithm.

The C code is based on same functions as *Compute_Gradient*, *Compute_Mag_flag*, and *Fill_Histogram*.

Compute_Gradient function performs the exploration of the windows of 128x64 pixel (3 pixels on each side have been discarded to avoid boundary conditions), separating each block of 8x8 pixel and computing for each the x and y gradient. Data related to a block are organized in a bidimensional array format.

Compute_Mag function receives as input the x and y gradients related to a block and computes the gradient magnitude by taking into account the simplification presented in [4].

$$Mag = |D_x| + |D_y|$$

This function also produces the flag values. Starting from gradients D_x and D_y values, it gives an information about the sign of the ratio $\frac{D_x}{D_y}$. This parameter is useful to assign the vote to the correct bin histogram.

Compute_Threshold function computes the average value of all magnitudes on a block. This is the threshold value with that magnitudes are compared to decide if the corresponding pixel votes to bin histogram.

Fill_Histogram: The calculation of the arctangent function represents an operation very complex in terms of complexity. Accordly to the algorithm, the direction values, are computed to fill bins of histograms in the correct way. Looking at the instructions presented in [14] arctangent computation can be avoided by considering the following expression:

$$f_x(x,y) * tan(\theta_i) <= f_y(x,y) <= f_x(x,y) * tan(\theta_{i+1})$$

By restricting the range of angles from 0 to π it can be obtained the representation in figure :

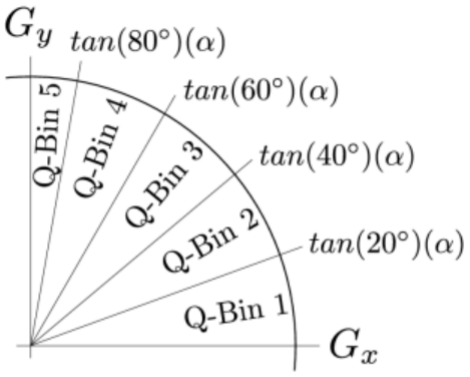

Figure 4.2: Angolar binning, from 0 to $(\pi)/4$ of the gradient vector

. .

Moreover, the computation of flag values allows to reduce the number of calculations to perform.

So *Fill_Histogram* function allows to fill in the correct way the nine bins of the histogram accordly to some factors as gradient magnitude and direction, threshold of magnitude values and flag like described above. Block is divided into cells and 9 bins for each cell are computed. The final output is a vector of 36 elements for a block.

4.1.2 Compilation

Using ASIP DESIGNER the processor model of tmotion and the C code presented above are compiled and simulated. The file obtained is compared with the correct one in order to ensure the consistency of results. As can be seen in the Instruction Report provided by the tool, the program is simulated requiring a total number of cycles equal to 723510 while the number of instructions that are executed is 660237. The total size occupied by instructions in program memory is 439 bytes. The values mentioned are divided among the various functions of the program as listed in table 4.1.

Function	*Instr_Count*	*%*	*Cycle_Count*	*%*
Fill_hist	*289865*	*43.90%*	*323935*	*44.77%*
Compute_Mag_Flag	*238793*	*36.17%*	*264887*	*36.61%*
Compute_gradient	*110502*	*16.74%*	*112313*	*15.52%*
Compute_Th	*20992*	*3.18%*	*22272*	*3.08%*

Table 4.1: Instructions count among functions. Pure SW Implementation.

It is noticeble that the two functions most expensive are $Fill\_hist$ and $Compute\_Mag\_flag$, the 81.02% of total clock cycles is used to perform them. With the aim to optimize this functions it is important to focus on them looking at their assembly instructions. The following image presents the assembly related to $Compute\_Mag\_flag$.

Pc	Instruction	Assembly	Exe-count	Cycles	User Count	Wait states	Relative cycle use within function
4	5040	addb sp, 4	128	128	0	0	
5	7feb	st lr,dm(sp-2)	128	128	0	0	
6	7fc3	st r3,dm(sp-4)	128	128	0	0	
7	3086	mvib r6,8	128	128	0	0	
8	301b	mvib lr,1	128	128	0	0	
9	2ede 0030	do r6,48	128	256	0	0	
11	3003	mvib r3,0	128	128	0	0	
12	3086	mvib r6,8	1024	1024	0	0	**
13	2ede 002f	do r6,47	1024	2048	0	0	*****
15	2e00	nop	1024	1024	0	0	**
16	4006	ld r6,dm(r0)	8192	8192	0	0	**********
17	0fb3	ge r6,r3	8192	8192	0	0	**********
18	2f00	jcr 0	8192	17706	0	0	**********
19	059e	sub r6,r3,r6	3435	3435	0	0	*********
20	4286	st r6,dm(r2)	8192	8192	0	0	**********
21	2e00	nop	8192	8192	0	0	**********
22	4046	ld r6,dm(r1)	8192	8192	0	0	**********
23	0fb3	ge r6,r3	8192	8192	0	0	**********
24	6fc7	ld r7,dm(sp-4)	8192	8192	0	0	**********
25	2f00	jcr 0	8192	16770	0	0	**********
26	059e	sub r6,r3,r6	3903	3903	0	0	**********
27	43d6	st r6,dm(r7++)	8192	8192	0	0	**********
28	7fc7	st r7,dm(sp-4)	8192	8192	0	0	**********
29	2e00	nop	8192	8192	0	0	**********
30	4097	ld r7,dm(r2++)	8192	8192	0	0	**********
31	01b7	add r6,r6,r7	8192	8192	0	0	**********
32	4316	st r6,dm(r4++)	8192	8192	0	0	**********
33	2e00	nop	8192	8192	0	0	**********
34	4016	ld r6,dm(r0++)	8192	8192	0	0	**********
35	4057	ld r7,dm(r1++)	8192	8192	0	0	**********
36	1837	mulss r6,r7	8192	8192	0	0	**********
37	2568	mv r6,pl	8192	8192	0	0	**********
38	0e33	lt r6,r3	8192	8192	0	0	**********
39	3023	mvib r3,2	8192	8192	0	0	**********
40	2f02	jcr 2	8192	15042	0	0	**********
41	2e00	nop	4767	4767	0	0	**********
42	2c03	jrd 3	4767	4767	0	0	**********
43	434b	st lr,dm(r5)	4767	4767	0	0	**********
44	3006	mvib r6,0	3425	3425	0	0	*********
45	4346	st r6,dm(r5)	3425	3425	0	0	*********
46	015d	add r5,r3,r5	8192	8192	0	0	**********
47	3003	mvib r3,0	8192	8192	0	0	**********
48	2e00	nop	1024	1024	0	0	**
49	6feb	ld lr,dm(sp-2)	128	128	0	0	
50	2ec0	rtd	128	128	0	0	
51	5fc0	addb sp, -4	128	128	0	0	
52	2e00	nop	128	128	0	0	

Figure 4.3: Assembly instr. of magflag function.

The first for loop starts at instruction 12 and terminates at 48. There is one

delay space before that *do* is performed and the compiler utilizes it to perform a *mvib* instruction. The second *for* loop starts at instruction 16 and finishes at 47. After it the compiler inserts a *nop* because no operations can be scheduled.
The computation of absolute value of Dx starts at instruction 16, where a load is performed. The instruction 17 compares its value against zero and if it is negative the conditional jump is taken at instruction 22 where the absolute value of Dy is computed in the same way. In case of negative value, the jump is not performed and the instruction 19 negates the value.
At instruction 31 the addition is performed and the Magnitude value is computed. At instruction 36 the multiplication between Dx and Dy is computed, and flag value calculation is performed from instruction 38 to 45.

It can be observed that the assembly instruction which are executed a lot of times are those internal to the loops with address from 16 to 45, related to computation of absolute values of Dx and Dy, magnitude and flag.

Same considerations can be done for the function *Fill_hist* looking at the following listing of instructions related to a portion of function.

```
131 7ea6          st r6,dm(sp-22)          512       512       0       0 *
132 7ec5          st r5,dm(sp-20)          512       512       0       0 *
133 2edf 00ed     do r7,237                512      1024       0       0 **
135 6f45          ld r5,dm(sp-12)          512       512       0       0 *
136 7e80          st r0,dm(sp-24)         2048      2048       0       0 *****
137 7e62          st r2,dm(sp-26)         2048      2048       0       0 *****
138 0126          add r4,r4,r6            2048      2048       0       0 *****
139 0096          add r2,r2,r6            2048      2048       0       0 *****
140 004e          add r1,r1,r6            2048      2048       0       0 *****
141 0006          add r0,r0,r6            2048      2048       0       0 *****
142 3046          mvib r6,4               2048      2048       0       0 *****
143 2ede 00e1     do r6,225               2048      4096       0       0 ***********
145 2e00          nop                     2048      2048       0       0 *****
146 4116          ld r6,dm(r4++)          8192      8192       0       0 **********************
147 0eb5          le r6,r5                8192      8192       0       0 **********************
148 2f49          jcr 73                  8192     18132       0       0 ****************************************
149 3035          mvib r5,3               3222      3222       0       0 ********
150 4083          ld r3,dm(r2)            3222      3222       0       0 ********
151 14dd          lsl r3,r3,r5            3222      3222       0       0 ********
152 4046          ld r6,dm(r1)            3222      3222       0       0 ********
153 3007          mvib r7,0               3222      3222       0       0 ********
154 0e1f          lt r3,r7                3222      3222       0       0 ********
155 182e          mulss r5,r6             3222      3222       0       0 ********
156 400b          ld lr,dm(r0)            3222      3222       0       0 ********
157 2558          mv r5,pl                3222      3222       0       0 ********
158 3077          mvib r7,7               3222      3222       0       0 ********
159 2f0b          jcr 11                  3222      3222       0       0 ********
160 0f9d          ge r3,r5                3222      3222       0       0 ********
161 2f09          jcr 9                   3222      7990       0       0 **********************
162 7e46          st r6,dm(sp-28)          838       838       0       0 **
163 256b          mv r6,lr                 838       838       0       0 **
164 3017          mvib r7.1                838       838       0       0 **
```

Figure 4.4: Assembly instr. of fillhist function.

4.2 Second Design

A second design of the processor has been implemented. It has been extended adding two special purpose instructions. The first is able to compute the absolute value of the gradients, make the sum of them obtaining the magnitude, and the flag value. Moreover, this computation is combined with parallel load operations over the needed operands. The second instruction is able to perform a number of if statements in order to determine the position of bin that has to be implemented. Also this second instruction is combined with parallel load operand operations. The aim is to accelerate the calculation in a way that the one iteration of the loop employs one clock cycle.

4.2.1 Magflagcomp Instruction

In order to define an instruction that computes *magnitude* and $flag$, some items are been put into the processor model.

Magflagcomp primitive function The $magflagcomp$ primitive function has to be added to the header file *tmotion.h*.

$$void \quad magcompflag(word,word,word\&,word\&,word\&);$$

The function receives as input two parameters of length 16 bit, word type, and returns three values of type word.

The file *tmotion.p*, is modified adding the function implementation.

```
void magflagcomp(word x,word y, word& adx, word& ady, word& mag,word& flag){
word adx_t = x<0? -x:x;
word ady_t = y<0? -y:y;
adx=adx_t;
ady=ady_t;
mag = adx_t + ady_t;

word flag_t = x*y;
if (flag_t < 0) flag=0;
else flag=1;
}
```

Finally, it is necessary to define a promotion and add it to the header file *tmotion_int.h* with the aim to use directly the $magflagcomp$ function in the C code and call it as an intrinsic function.

$$promotion \quad void \quad magflagcomp(int,int,int\&,int\&,int\&) =$$

$$void \quad magflagcomp(word,word,word\&,word\&,word\&);$$

In this way the function can be directly call using *int* arguments type.

Magflagcomp added instruction The first part of the new instruction defines the $magflagcomp\_instr$, it is an Or-rules between two instructions as can be seen in the following portion of code:

```
opn magflagcomp_instr(magflagcomp_operation_instr| magflagcomp_parr_instr)
{
image
: "100"::magflagcomp_operation_instr
| "100"::magflagcomp_parr_instr

;
}
```

The most significant part of the binary encoding, for these instructios is defined in the *image* attribute: “100”.

Functional unit The functional unit that implements the primitive function is defined as shown below. Also the transistories which are connections like buses and nets (wires) utilized to connect the unit to the rest of the datapath without introducing delay, are defined. Input and output are declared by the keyword *trn*.

```
fu magflagcomp;  // compute magnitude_flag unit

trn mfc_r<word>; // input
trn mfc_s<word>; // input

trn mc_u<word>; // output
trn mc_v<word>; // output
trn mc_t<word>; // output

trn fc_u<word>; // output
```

In the following portion of code is described the instruction core, using the nML code:

```
opn magflagcomp_operation_instr( r: c3u, s: c3u, u:c3u,t:c3u )
{
        action {
                stage E1:

                magflagcomp(mfc_r=rre1=R[r],mfc_s=rse1=R[s],mc_u,mc_v, mc_t,fc_u)
                      @magflagcomp;

                R[u]=rue1=mc_u;
                BN0=mc_v;
                R[t]=rte1=mc_t;
                PL=fc_u;

}
        syntax :  "mfc("r "," s "," u "," v ","t")";
        image  : "0"::r::s::u::t;
}
```

r,s,u and t are the four parameters used by this instruction. They correspond to the addresses of Register File, r and s are the addresses of the location in which input values are stored, u and t are the addresses of the location in which output values are stored. Also two additional registers are used to memorize outputs. It is important that the registers BN0 and PL are involved in the move and load and store base instructions of the processor in order to guarantee the transfer of these values in memory.

The sequence of actions that the instruction performs are described in the *action* attribute and depicted in the image 4.5.

The values in the register R[r] and R[s] are elaborated to compute the absolute values and then added. The results are stored in the register R[u], R[t] and BNO. The input values are also multiplied, and their product is elaborated to compute flag, that is stored in the register PL. These operations are executed by the $magflag$ functional unit thanks to the command $@magflag$.

The third attribute of the instruction, the *image* attribute specifies the sequence of bits that compose the second part of thirteen bits of the opcode regarding this instruction:

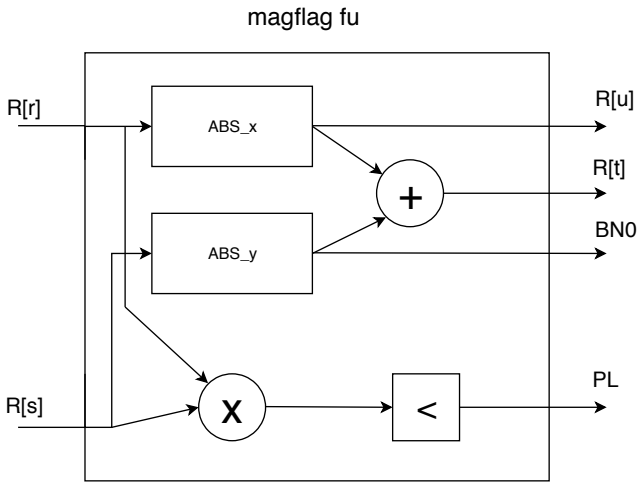

Figure 4.5: Magflag Functional Unit

```
"0"::r::s::u::t;
```

The processor structure has to be modified to perform this instruction. In particular, three write ports are added to the Register file in order to execute two reading and four writing operations in a single clock cycle.

4.2.2 Fillhist

In order to define an instruction that compute the position of the bin that has to be incremented , the following items are been put into the processor model in a similar way as described above.

Fillhist primitive function The *fillhist* primitive function has to be added to the header file *tmotion.h*.

$$word \quad fillhist(word,word,word);$$

The function receives as input three parameters of length 16 bit, and returns one values of type word.

The file *tmotion.p*, is modified adding the function implementation.

```
word fillhist(word x,word y,word flag){

word x_t= x;
word y_t= y<<3;

word lim1=TAN0*x_t;
word lim2=TAN20*x_t;
word lim3=TAN40*x_t;
word lim4=TAN60*x_t;
word lim5=TAN80*x_t;

word pos;

        if((y_t >= lim1)){ if((y_t < lim2))   {if(flag==1) pos=0; if(flag==0)
              pos=8;}}

        if((y_t >= lim2)) {if((y_t < lim3))  {if(flag==1) pos=1; if(flag==0)
              pos=7;}}

        if((y_t >= lim3)) {if((y_t < lim4))  {if(flag==1) pos=2; if(flag==0)
              pos=6;}}

        if((y_t >= lim4)) {if((y_t < lim5))  {if(flag==1) pos=3; if(flag==0)
              pos=5;}}

        if(y_t >= lim5) pos=4;

return pos;

}
```

A promotion is defined and add it to the header file *tmotion_int.h*.

$$promotion \quad int \quad fillhist(int,int,int) =$$

$$word \quad fillhist(word,word,word);$$

Fillhist added instruction The *Fillhist_instr* instruction is an Or-rule between the operation instruction and the parallel operand load instruction:

```
opn fillhist_instr( fillhist1_operation_instr | fillhist1_parr_instr)
{
image
: "11"::fillhist1_operation_instr
| "110"::fillhist1_parr_instr
```

```
;
}
```

Functional unit The functional unit that implement the primitive function is the following:

```
fu fillhist1;

trn fh_r<word>; // input
trn fh_s<word>; // input
trn fh_d<word>; // input

trn fh_pos<word>; // input
```

The operation performed by the instruction, listed below, show that four field of Register File are used to store operand values, three for inputs and one for the output, so four parameter are requested by $fillhist\_operation\_inst$.

```
opn fillhist1_operation_instr( r: c3u, s: c3u, d:c3u , t:c3u )
{
action {
stage E1:

R[t]=rte1=fh_pos=fillhist1(fh_r=rre1=R[r],fh_s=rse1=R[s],fh_d=rde1=R[d])
      @fillhist1;

}
syntax :  "fh("r "," s "," d ","t ;
image  : "11"::r::s::d::t;
}
```

The sequence of actions that the instruction performs are described in the *action* attribute and depicted in the image 4.5.

The value stored in register R[r] is multiplied with the constant values that that tangent can assume and the limits that determine each bin are obtained. The value in register R[s] is shifted of three (the whole program acts on integer value, for this reason tangent values are been approximate and y need to be multiplied by 8). The value in register R[d] is read and then a series of conditions, exploited by a series of if statements, are verified and the the correct value is stored in register

R[t]. These operations are executed by the fillhist1 functional unit thanks to the command @$fillhist1$.

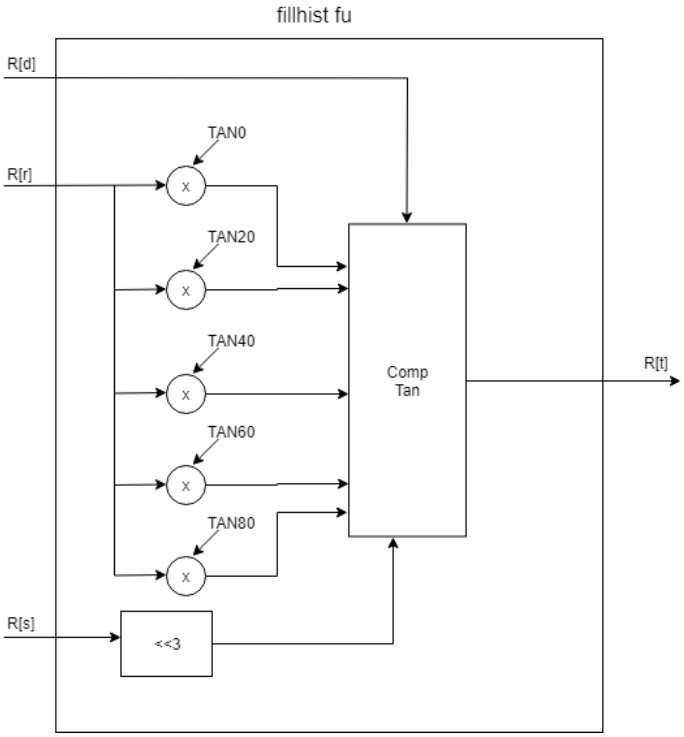

Figure 4.6: Fillhist Functional Unit

4.2.3 Parallel loads of operands

With the aim to perform these two operations just described in parallel with load operations from memory some additional functional units are added named $ag2$, $ag3$, $xs2$ and $xs3$. Functional units ag and xs are already present in processor model, defined in *load_store* operation.

```
fu ag2;
trn agp2<word>;
trn agm2<word>;
trn agq2<word>;

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

```
fu ag3;
trn agp3<word>;
trn agm3<word>;
trn agq3<word>;

fu xs3;
trn wbus3<word>;
```

AG2 and AG3 fus are two additional address generation units, each of them has the task to generate the address needed to read the data memory by incrementing the value stored in the source register.

XS2 and XS3 allow to zero extend the two bytes fetched from memory to 16 bits before to be put in the register of destination.

Moreover, some modifications have to be done on the processor model. Original data memory has one read port and one write port. In order to perform parallel loads, so enter in memory two times simultaneously, two additional read port has to be provided. The data memory Dmb is declared in this way in *tmotion.n* file:

```
def  dm_size = 2**16;           // data memory
mem  DMb[dm_size,  1]<w8,   addr> access {
dmb_load‘ID‘: dmb_read‘E1‘ = DMb[dm_addr‘ID‘]‘ID‘;
dmb_load2‘ID‘: dmb_read2‘E1‘ = DMb[dm_addr2‘ID‘]‘ID‘; // additional read port
dmb_load3‘ID‘: dmb_read3‘E1‘ = DMb[dm_addr3‘ID‘]‘ID‘; // additional read port
dmb_st‘E1‘: DMb[dm_addr‘E1‘]‘E1‘ = dmb_write‘E1‘;
};
```

*dmb_load*2 and *dmb_load*3 are the two extra ports. *dmb_addr*2 and *dmb_addr*3 are the signals in input of memory, address signals, on the bus at the stage ID, while *dmb_read*2 and *dmb_read*3 are the output signals and carry the values taken by memory available on the data bus in E1 stage.

The instructions are so composed:

```
opn magflagcomp_parr_instr(mc: magcomp_opn,
ld0 : load_R0,
ld1 : load_R1)
{
        action { mc; ld0; ld1; }
        syntax : mc " | " ld0 " | " ld1;
        image  : "1"::"000"::"000"::"000"::"000"::mc::ld0::ld1;
}


opn fillhist1_parr_instr(fh: fillhist1_opn,
ld0 : load_R0fh,
ld1 : load_R1fh,
ld2 : load_R2fh )
{
```

```
action { fh; ld0; ld1; ld2; }
syntax : fh " | " ld0 " | " ld1 " | " ld2;
image  : "0111"::"000"::"000"::"000"::fh::ld0::ld1::ld2;
}
```

In particular the *load_R*0 rules is characterized by the code:

```
opn load_R0()
{
        action {
                stage ID:
                R2 = rtid = ag1q = add(ag1p=rrid=R2,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‘) @xs1;
        }
        syntax : "ld r0,dm(r3++)";
}
```

The *action* attribute describes the operations acted:

- In the stage ID the value in the register R2 is put on the address bus in order to access memory, and in the same cycle this address is incremented by one, thanks to $ag1$ and restored in R2.
- In the stage E1 the value of eight bit taken in the memory is extended by $xs1$ and written in R0.

In order to fit the opcode related to the parallel instructions in one single string of 16 bit, no parameters are used in this instruction but specific registers are considered: R0 and R2.

The same is done for *load_R*1 and *load_R*2.
The operations described are represented in a graphical way in figure:

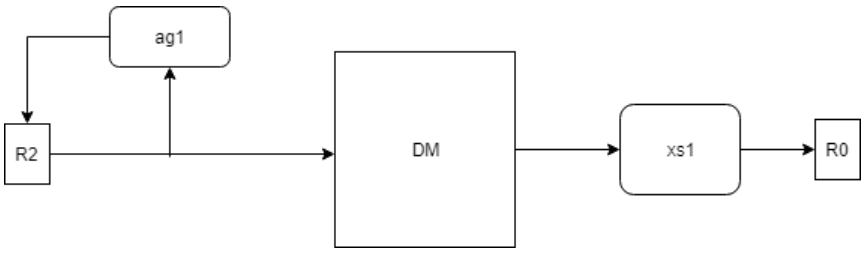

Figure 4.7: Load R0 behavior

Also instructions $magflagcomp$ Fig.4.8 and $fillhist$ Fig.4.9 are reworked in order to reduce the opcode bits by using specific registers.

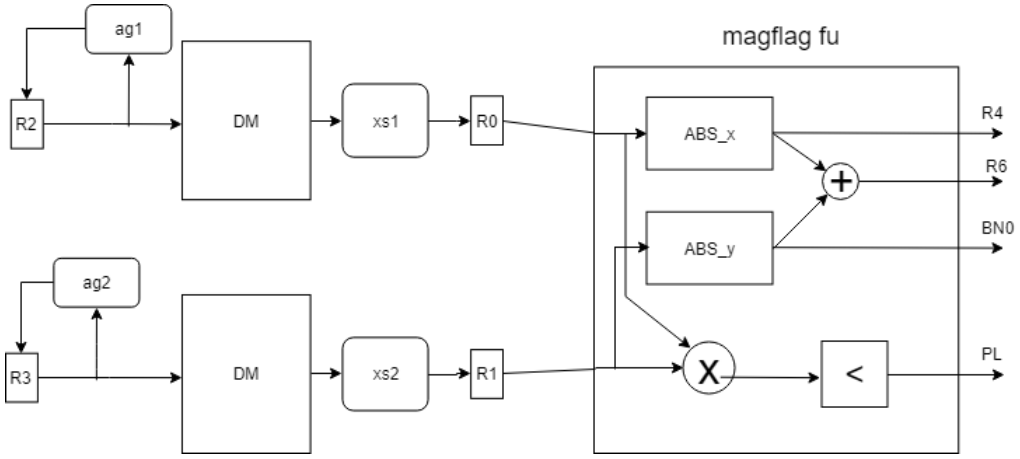

Figure 4.8: Magflag and Parallel load

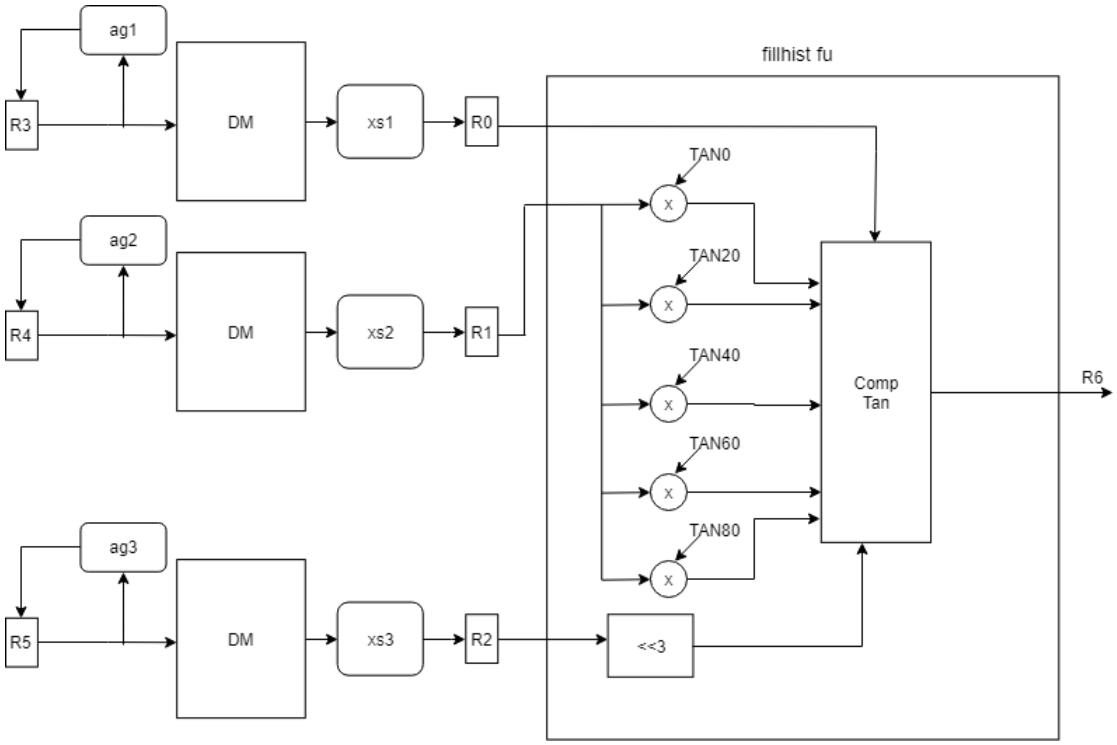

Figure 4.9: Fillhist and Parallel load

To be executed, this instruction required some modifications to the Register File,

in particular some ports have been added:

- two read ports in the ID stage in order to allow three reading operations
- two read ports in the E1 stage in order to allow three reading operations
- two write ports in the E1 stage in order to allow three writing operations
- four read port in the E1 stage in order to allow five writing operations.

The nml desctription of RF in file *tmotion.n* is modified in this way:

```
// register declarations
reg R[8]<word,uint3>            // general purpose registers
syntax ("R")
read( rrid rr2id  rr3id        // ID stage read port  // added rr2id rr3id
          rre1 rse1 rde1)          // E1 stage read ports
write(rtid rt2id rt3id         // ID stage write port // added rt2id rt3id
rue1  rte1 rle1 rke1 rme1);    // E1 stage write port // added rl and rk rm
```

C_Code

The C code is modified in order to use these two instructions with proper line that call the primitive functions:

```
void compute_magflag(int Dx[][8],int Dy[][8],int aDx[][8],int aDy[][8], int Mag[][8], int flag[][8]) {
int l,h;
        for (l=0; l < 8; l++){
                for (h=0; h < 8; h++){

                        magflagcomp(Dx[l][h],Dy[l][h],aDx[l][h],aDy[l][h],Mag[l][h],flag[l][h]);
|
                }
        }
}
```

Figure 4.10: Magflag function

```
// fill histogram
for (k=0; k < 8; k=k+4){

        for (h=0; h < 8; h=h+4){

                for (l=0; l < 4; l++){
                        for (m=0; m < 4; m++){


                                idx=cell*9;


                        if(Mag[k+l][h+m] > Mag_Th){

                                pos=fillhist1(aDx[k+l][h+m],aDy[k+l][h+m],flag[k+l][h+m]);
                                bin[pos+idx]+=1;

                        }

                        }

                }
                        cell=cell+1;
        }
}
```

Figure 4.11: Fillhist Function

4.2.4 Compilation

The processor model of tmotion and the C code are compiled and simulated. The file obtained is compared with the correct one in order to ensure the consistency of results and the correctness of obtained design.
The program is simulated exploiting a total number of cycle equal to 388370 while the number of instructions that are executed is 369117. The total size occupied by instructions in program memory is 331 bytes. The values mentioned are divided among the various function of the program as listed in table:

Function	***Instr_Count***	***%***	***Cycle_Count***	***%***
Fill_hist	***159519***	***43.21%***	***120505***	***44.82%***
Compute_Mag_Flag	***69888***	***18.93%***	***71424***	***18.39%***
Compute_gradient	***118694***	***32.15%***	***120505***	***31.03%***
Compute_Th	***20992***	***5.69%***	***22272***	***5.73%***

Table 4.2: Instructions count among functions. Design 2.

4.3 Second Sw implementation

In order to exploit data parallelism it is necessary to obtain a data organization suitable for this kind of approach.

4.3.1 C_code

The source code has been modified to this aim. Data related to a block are arranged in a vectorial way, not in matrix way, like in the precedent case. So a vector of 64 elements is obtained for each block and it is organized in a way that first them related to the first cell are listed, then the ones related to the second cell and so on.

The C code is based on functions: *Compute_Gradient* and *Fill_Histogram*.

Compute_Gradient function performs the exploration of the windows of 128x64 pixel, separating each block of 8x8 pixel and computing for each the x and y gradient.

Fill_Histogram function receives as input the x and y gradients related to a block and computes the magnitude, the flag and threshold values. Then it produces the final output of 36 elements for a block.

4.3.2 Compilation

The processor model of tmotion and this new version of C code presented above are compiled and simulated. The file obtained is compared with the correct one in order to ensure the consistency of results. As can be seen in the Instruction Report provided by the tool, the program is simulated exploiting a total number of cycle equal to 926330 while the number of instructions that are executed is 852976. The total size occupied by instructions in program memory is 545 bytes. The values mentioned are divided among the various function of the program as listed in table:

Function	***Instr_Count***	***%***	***Cycle_Count***	***%***
Fill_hist	***658151***	***77.16%***	***726731***	***78.45%***
Compute_gradient	***194740***	***22.83%***	***199496***	***21.54%***

Table 4.3: Instructions count among functions. Second pure sw implementation.

4.4 Third Design: parallel approach

A third design of the processor has been implemented in order to exploit SIMD (Single Instruction Multiple Data) instructions. This kind of instructions are able to elaborate in an equal way and simultaneously all the elements of a vectorial datum.

4.4.1 Additional Features of processor model

New primitive data type A new data type vword is added to processor model in order to shape a SIMD vector. This type has been defined in a way that all the values related to a cell, 16 elements, can be fit in a vector. The type is declared in the header file *tmotion.h*:

$$class \quad v8word \quad property \quad (vector \quad word[VSIZE]);$$

The costant value VSIZE, instead, is defined in the file $tmotion\_config.h$ as follow:

$$\#define \quad VSIZE = 16; \quad //number \quad of \quad words$$

The SIMD instructions identify this type as a vector of 16 word, 256 bit wide, as shown in the following figure:

Figure 4.12: The vword type

Alias Vector Data Memory In order to store and load one whole vector in the data memory, a record alias DMv must be defined[15].

```
mem  DMb[dm_size,  1]<w8,   addr> access {
```

```
        dmb_ld‘ID‘: dmb_read‘E1‘ = DMb[dm_addr‘ID‘]‘ID‘;
        dmb_st‘E1‘: DMb[dm_addr‘E1‘]‘E1‘ = dmb_write‘E1‘;
5       };

        mem  DMv [dm_size,32]<vword, 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‘;
10      };
```

In this way Dmv has been declared like an alias of memory DMb.

It can be noticed that bytes in Dmb have an alignment of 1 and vwords in DMv have an alignment of 2*VSIZE because it deals with vectors of word type. An alignment of 32 means that vword elements have to be allocated at addresses that must be multiple of 32. Moreover, Tmotion has a memory with an organization of type little endian, so least significant part of the vector data are stored at the smaller storage address, that is the address of the whole vector.

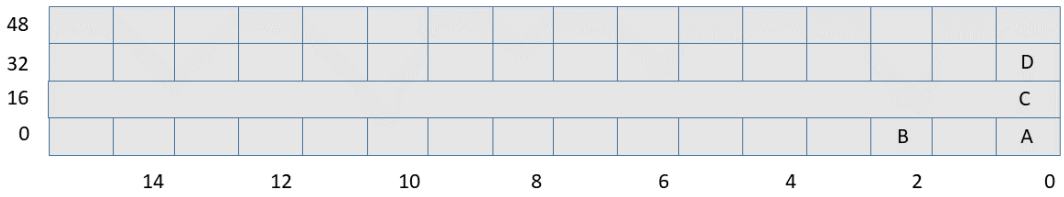

Figure 4.13: Alignment in DM

The figure 4.13 shows how the objects are allocated in data memory. A and B are word type elements and are stored at addresses 0 and 2 respectively. C is of vword type and is stored at the aligned address 16. The last object D is stored at address 32.

Memory has two ports: *dmv_ld* and *dmv_st*. The first is taken to load vectors in memory: in the stage ID the address is on the bus and an access of memory is performed, in the E1 stage, instead, the data vector is available on data bus. The second port is taken to store vectors in memory, in E1 stage the whole operation is performed. The vector that has to be stored is on data bus in E1 and it is put at the location specified in address bus. Also two signals are defined: *dmv_read* and *dmv_write*, they correspond respectevely to vector data read signal and vector data write signal.

Vector Register File A vector Register File is defined in order to put in vectors when they are taken from memory. In file *tmotion.n* the following portion of code is added:

```
reg V[6]<vword,uint3>  // general purpose Vector registers
syntax ("V")
read( vecr vecs )      // read ports
write(vect);           //  write port
```

V Register file has 6 fields, two read ports and one write port. Three bits are necessary to address it.

Vector Primitive functions The new primitives that allow to manipulate vectors are declared in the file *tmotion.h*

```
void vmagflagcomp(vword,vword,vword&,vword&,vword&,vword&);
vword fillhist1(vword , vword ,vword , vword , word );
vword fillhist2(vword);
addr  force_align(addr);
```

The *vmagflagcomp* primitive function receives in input two vectors containing x and y gradient values related to a whole cell, compute their absolute values, sum them to calculate the magnitude vector and compare their sign to obtain the flag vector. These vectorial operations are done by performing scalar operations on the elements that have the same position in the input vectors and storing the results in an output vector that has the same dimension.

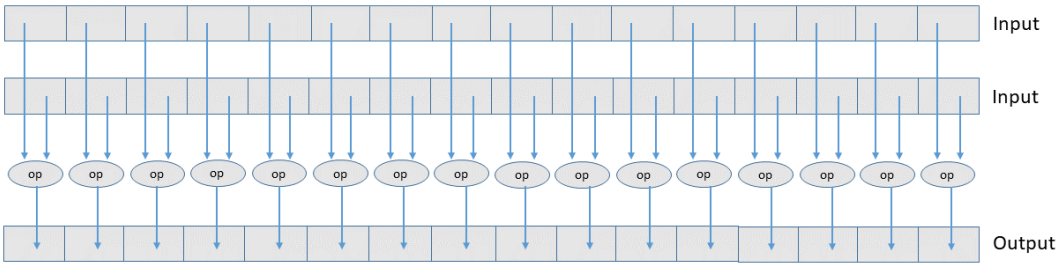

Figure 4.14: Vector Operation

The file *tmotion.p*, is modified adding the function impementation.

```
void vmagflagcomp(vword x,vword y, vword& adx, vword& ady, vword& mag, vword& flag
     )
{

vword result0, result1,result2, result3;
int32_t i;
int32_t k;
int32_t l;
int32_t m;

for (int32_t m = 0; m < VSIZE; m++)
result3[m]=((x[m]*y[m])<0)? 0:1;


for (int32_t k = 0; k < VSIZE; k++)
result1[k]=(x[k]<0)? -x[k]:x[k];
for (int32_t l = 0; l < VSIZE; l++)
result2[l]=(y[l]<0)? -y[l]:y[l];
for (int32_t i = 0; i < VSIZE; i++)
result0[i]=result1[i]+result2[i];


mag=result0;
adx=result1;
ady=result2;
flag=result3;
}
```

The $fillhist1$ primitive function receives in input four vectors, and returns an output vector containing the values of positions of bin vector to be incremented of one.

```
vword fillhist1(vword x, vword y ,vword Mag, vword flag, word Mag_th){

int32_t k;
vword pos;

for (int32_t k = 0; k < VSIZE; k++){pos[k]=14;}

for (int32_t k = 0; k < VSIZE; k++){

word x_t= x[k];
word y_t= y[k]<<3;
word lim1=TAN0*x_t;
word lim2=TAN20*x_t;
word lim3=TAN40*x_t;
```

```
word lim4=TAN60*x_t;
word lim5=TAN80*x_t;

if(Mag[k]>Mag_th){

if((y_t >= lim1)){ if((y_t< lim2))   {if(flag[k]==1) pos[k]=0; if(flag[k]==0)
      pos[k]=8;}}

if((y_t >= lim2)) {if((y_t < lim3))  {if(flag[k]==1) pos[k]=1; if(flag[k]==0)
      pos[k]=7;}}

if((y_t >= lim3)) {if((y_t< lim4))  {if(flag[k]==1) pos[k]=2; if(flag[k]==0)
      pos[k]=6;}}

if((y_t>= lim4)) {if((y_t< lim5))  {if(flag[k]==1) pos[k]=3; if(flag[k]==0)
      pos[k]=5;}}

if(y_t >= lim5) pos[k]=4;
                }
       }
return pos;
}
```

The $fillhist2$ primitive function receives in input one vector of position, and returns as output the bin vector, its elements has been incremented according to input vector.

```
v8word fillhist2(v8word pos){

int32_t k;
vword bin;

for (int32_t k = 0; k < VSIZE; k++){bin[k]=0;}

for (int32_t k = 0; k < VSIZE; k++){

if(pos[k]==0){ bin[0]+=1;}
if(pos[k]==1){ bin[1]+=1;}
if(pos[k]==2){ bin[2]+=1;}
if(pos[k]==3){ bin[3]+=1;}
if(pos[k]==4){ bin[4]+=1;}
if(pos[k]==5){ bin[5]+=1;}
if(pos[k]==6){ bin[6]+=1;}
if(pos[k]==7){ bin[7]+=1;}
if(pos[k]==8){ bin[8]+=1;}
       }
return bin;
}
```

The *force_align* primitive function receives as argument an address, force the 5 least significant bit to zero and return it. This operation allows to perform aligned store and load operations to and from memory.

```
addr   force_align(addr a) { return a[15:5]::"00000"; }
```

4.4.2 New vector instruction

A new instruction defined as *vector_instr* is added, it is an Or-rules between two instructions as can be seen in the following portion of code.

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

SP-indexed addressing The instructions for SP-indexed are requested by the C compiler. They are utilized in order to store registers of vector types on the stack frame like a portion of the context switch, or in order to perform spilling when the vector register file has a capacity that is insufficient. For the SP indexed load and store instructions, the generated address is calculated by adding to the stack pointer an offset that is negative. In order to have an offset to an aligned address, it is necessary that the offset change with a step of 16. For this reason, a new primitive type is defined:

$$class \quad nint16s16 \quad property(16 \quad bit \quad signed \quad min = -32768 \quad max = -16 \quad step = 16);$$

this type is used to model the $offs$ parameter of *load_store_vreg_sp_indexed*.

```
opn load_store_vreg_sp_indexed(ls: load_store_op, u: c3u, offs: c16n) //,
{
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[u]‘E1‘ =vect ‘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‘ = vecr‘E1‘= V[u]‘E1‘;
}
  }
  syntax : ls ","u ",dm(sp" offs ")" ;
  image  : ls::offs[one 10..4 zero]::"00"::u;
  }
```

In ID stage the ag1 functional unit computes the address by adding the offset to the Stack Pointer and the stack is accessed. In E1 stage, if a load operation (ls=ld) is performed the vector read from memory is put on the data bus and stored in V at the address specified by u parameter. If a store operation (ls=st) is executed, instead, the vector present in V[u] is read and written at the address computed in ID stage.

Because the address has to be aligned, the four least significant bits of offset are always equal to zero and aren't encoded in the opcode.

Vector Single instruction This second instruction contains different items, an instruction to load and store in aligned way vectors from and to memory, one to execute move operation between vector registers, and three instructions to perform the computations before described. Additional functional unit and transistories are added:

```
// Vector unit
fu vmagflag;
trn tvecr<v8word>; //in
trn tvecs<v8word>;
trn tvect<v8word>; //out
trn tveca<v8word>;
trn tvecb<v8word>;
trn tvecc<v8word>;

fu vfill;
trn tvecfr<v8word>; //in
trn tvecfs<v8word>;
trn tvecft<v8word>;
trn tvecfb<v8word>;
trn tvecd<word>;
trn tvecfa<v8word>; //out
```

```
fu vfill2;

trn tvecf2r<v8word>; //out
trn tvecf2s<v8word>; //in
```

```
trn ag1_addr<addr>;
fu fa;
```

The following nML code describes the *vec_vmagflagcom_opn* instruction rule:

```
opn vec_vmagflagcomp_opn(r: c1u, s: c1u, t: c1u, u: c1u, v: c1u, z: c1u)
{
action {
stage E1:
vmagflagcomp(tvecr=VA[r],tvecs=VA[s],VB[t]=tvect,VB[u]=tvecb,VC[v]=tveca,VC[z]=tvecc)
      @vmagflag;
}
syntax : "vmagcomp v"r ",v"s ",v"t",v"u",v"v",v"z;
image  : r::s::t::u::v::z::"00";
}
```

It can be noticed that alias of vector register file V are used in order to reduce the number of bit that encoding the instruction.

```
// Register aliases
reg VA[2]<v8word,uint1> alias V[0];
reg VB[2]<v8word,uint1> alias V[2];
reg VC[2]<v8word,uint1> alias V[4];
property unconnected: VA, VB, VC;
```

In this way the parameter r,s,t,c,u,v,z are all addresses of one bit. In the *action* section, the behaviour of instruction is described: input vectors are taken from the alias VA of V , their absolute values, magnitude and flag are computed and written into four output vectors. The functional unit @$vmagflag$ performs these operations.

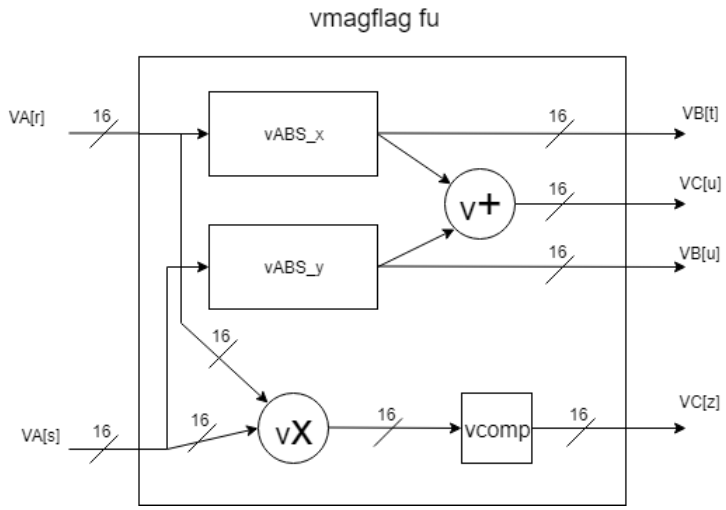

Figure 4.15: Vmagflag functional unit

The following nML code describes the *vec_fillhist1_opn* instruction rule:

```
opn vec_fillhist1_opn(r: c1u, s: c1u, t: c1u, u: c1u, z: c3u,v: c1u )
{
action {
stage E1:
VA[v] =tvecfa =
      fillhist1(tvecfr=VC[r],tvecfs=VC[s],tvecft=VB[t],tvecfb=VB[u],tvecd=rfe1=R[z])
      @vfill;
}
syntax : "fillhist usc"v ",ing"r","s","t","u","z;
image  : r::s::t::u::v::z::"0";
}
```

In the *action* section, the behaviour of instruction is described: four input vectors are taken from the alias VC and VB registers of V and from register R at the address z, these values are elaborated by $@vfill$ functional unit as shown in the figure below and the results are stored in vector VA.

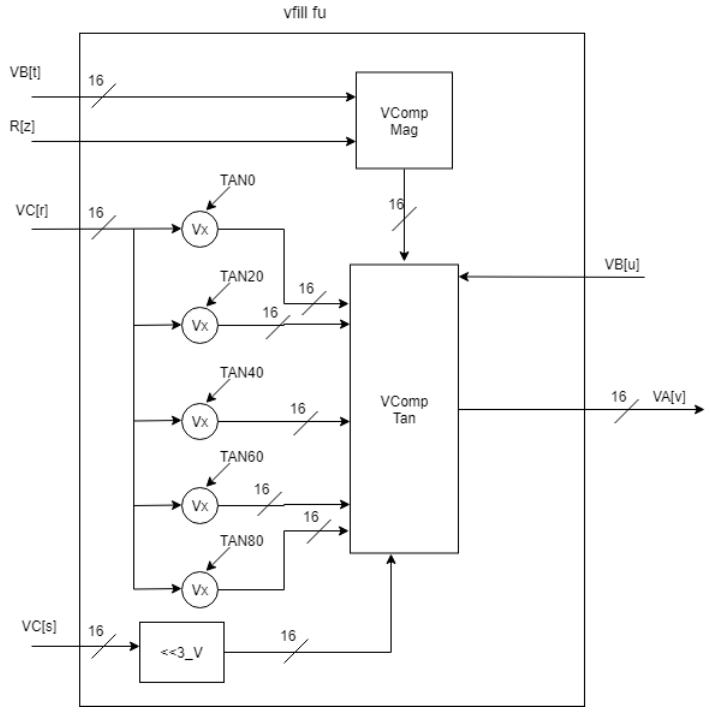

Figure 4.16: Vfill functional unit

```
opn vec_fillhist2_opn(r: c1u,v: c1u )
{
```

```
action {
stage E1:
VA[r] =tvecf2r = fillhist2(tvecs=VC[v]) @vfill2;
}
syntax : "fillhist2 usc"r ",ing"v;
image  : r::v;
}
```

In the *action* section, the behaviour of instruction is described: the input vector is taken from the alias register VC of V, according to the position written in the input register the elements of output vector VA, at address r, are incremented. The functional unit @$vfill2$ performs these operations.

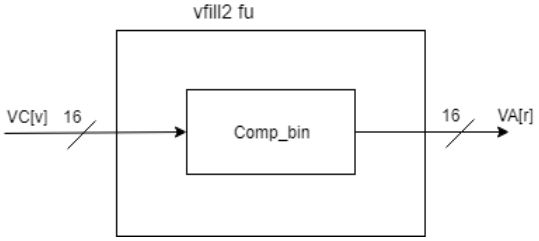

Figure 4.17: Vfill2 functional unit

The instruction $fillhist1$ and $fillhist2$ are been separated in order to avoid a significant increment of the critical path.

The following nML code describes the *vec_vmagflagcom_opn* instruction rule:

```
// Indirect vector load_store with post modification
trn ag2_addr<addr>;
trn ag1_addr_algn<addr>;

//enum vec_ld_op { in "", pp "++", pm "--" };

enum vec_ld_op { in "", pp "++", pm "--" };

opn vec_load_store_opn(op: vec_ld_op, ls: load_store_op, v: c3u, 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=-16) @ag1;
}
ag2_addr = ag1p;
```

```
  dm_addr =ag1_addr_algn = force_align(ag2_addr) @fa;
20stage ID..E1:
  switch (ls) {
  case ld:
  V[v]‘E1‘ = vect‘E1‘ = dmv_read‘E1‘ = DMv[dm_addr‘ID‘]‘ID‘;
  case st:
25dm_addr_pipe‘ID‘ = dm_addr‘ID‘;
  dm_addr‘E1‘ = dm_addr_pipe‘E1‘;
  DMv[dm_addr‘E1‘]‘E1‘ = dmv_write‘E1‘ = vecr‘E1‘= V[v]‘E1‘;
  }
  }
```

This instruction is able to execute load and store operation from and to the data memory of aligned vectors thanks to the utilization of the primitive function $force\_align$.

In the *action* section, the behaviour of instruction is described: in stage ID The value read from RO3[r] is elaborated by the @fu functional unit, so its least significant bits are forced to zero. It is made available on address bus in order to access memory DMv. Moreover, this value is elaborated by @$ag1$ functional unit that performs a post-incrementation on it. The value in the register RO3 is incremented or decremented of the dimension of a vector(16). In the E1 stage:

- If a *load* operation is performed, the vector fetched from memory is on data bus and written in V.
- If a *store* operation is performed, the vector stored in V, is put in memory. A pipe register is utilized.

4.4.3 Promotion

Finally, it is necessary to define the promotions and add them to the header file *tmotion_vector.h* with the aim to use directly the primitive function added in the c code as intrinsic functions.

```
  chess_properties {
  representation vpix : vword;
  }
5
  promotion void vmagflagcomp(vpix, vpix, vpix&, vpix&, vpix&, vpix&) = void
        vmagflagcomp(vword, vword, vword&,vword&,vword&,vword&);
```

```
promotion vpix fillhist1(vpix, vpix ,vpix, vpix, int)=vword fillhist1(vword ,
      vword ,vword , vword , word );
promotion vpix fillhist2(vpix)=vword fillhist2(vword);
```

It can be seen that a new type, *vpix*, is defined, it correspond to a vector of 16 fields and it is related to the vword, primitive type defined in processor model. It is a new data type in C code.

C_Code

The C code is modified in order to use SIMD instructions with proper line that call the primitive functions:

```
vpix *d_x;
vpix *d_y;
vpix absx;
vpix absy;
vpix mag_t;
vpix flag_t;

        for (h=0; h < 64; h=h+16){

                        d_x= (vpix*)(prova1+h);
                        d_y= (vpix*)(prova2+h);

                        vmagflagcomp(*d_x,*d_y,absx,absy,mag_t,flag_t);

                        int* temp1=(int*)&mag_t;
                        int* temp2=(int*)&absx;
                        int* temp3=(int*)&absy;
                        int* temp4=(int*)&flag_t;

                        for(int zz =0; zz<16;zz++)
                        {
                            Mag[h+zz]=temp1[zz];
                            aDx[h+zz]=temp2[zz];
                            aDy[h+zz]=temp3[zz];
                            flag[h+zz]=temp4[zz];
                        }


        }
```

Figure 4.18: Fillhist Function,part 1

```
vpix *mag_v;
vpix *flag_v;
vpix *absx_v;
vpix *absy_v;

vpix pos;
vpix bin_t;

        for (k=0; k < 64; k=k+16){


                        mag_v=  (vpix*)(Mag+k);
                        flag_v= (vpix*)(flag+k);
                        absx_v= (vpix*)(aDx+k);
                        absy_v= (vpix*)(aDy+k);

                        pos=fillhist1(*absx_v, *absy_v ,*mag_v, *flag_v,Mag_Th);
                        bin_t=fillhist2(pos);

                        int* temp1=(int*)&bin_t;
                        if(k==0) idx=0;
                        if(k==16) idx=9;
                        if(k==32) idx=18;
                        if(k==48) idx=27;

                        for(int zz =0; zz<9;zz++)
                        {
                            bin[zz+idx]=temp1[zz];
                        }
```

Figure 4.19: Fillhist Function,part 2

It can be noticed that a *for* loop that implements only four iterations is required, in fact the utilization of vectorial instructions allows to elaborate in a single iteration a quarter of the whole vector. In the first loop the int pointers on the desired element of the arrays prova1 and prova2 are cast to *vpix* pointers d_x and d_y. The *vmagflagcomp* function is called, it receives parameter of type vpix and returns values of the same type. Because of the fact that the parameters returned by the vectorial function have to be processed in a second time by a scalar function that computes the average values of magnitude related to a whole block they need to be stored in scalar variables, otherwise a data dependency problem would be introduced. This implies a waste of cycles of execution and leads to mitigate the improvement of this parallel approach. In the second *for* loop the needed cast operations are performed and the two vectorial functions are called. Also in this case the final output, *bin*, needs to be rewritten in a scalar way leading to ulterior waste of cycles. Moreover a certain number of cycles is wasted because a local copy of gradient values have to be performed in order to load these values from stack pointer instead of memory.

4.4.4 Compilation

The processor model of tmotion and the C code are compiled and simulated. The file obtained is compared with the correct one in order to ensure the consistance of results and the correctness of obtained design. As can be seen in the Instruction Report provided by the tool, the program is simulated exploiting a total number of cycle equal to 420701 while the number of instructions that are executed is 412225. The total size occupied by instructions in program memory is 361 bytes. The values mentioned are divided among the various function of the program as listed in table:

Function	***Instr_Count***	***%***	***Cycle_Count***	***%***
Fill_hist	***217472***	***52.76%***	***22118***	***52.58%***
Compute_gradient	***194740***	***47.24%***	***199496***	***47.42%***

Table 4.4: Instructions count among functions. Design3.

4.5 Fourth Design: a second parallel approach

A fourth design of the processor has been implemented with the aim to reduce the number of cycles wasted in the previous version due to data dependency problems. For this scope the parallelism of the instructions has been incremented in a way that all the elements of a block are elaborated at the same time.

4.5.1 Processor model

To implement this processor the starting point considered is the Design_3 above described. This version shares with the previous the following items:

- Primitiv Data type *vword*
- Alias Vector Data Memory
- Vectorial instruction for SP_indexed addressing
- Vectorial instructions for indirect addressing of memory

Additional Features

Vector Register File Three vector Register Files are added in order to put in vectors when they are taken from memory. the four Register files have 6 fields, two read ports and one write port each. Three bits are necessary to address them.

Vector Prinitive functions The new primitives that allow to manipulate vectors are declared in the file *tmotion.h*

```
void vmagflagcomp(vword,vword,vword,vword,vword,vword,vword,vword&,vword&,
vword&,vword&,vword&,vword&,vword&,vword&,vword&,vword&,vword&,vword&,vword&,
vword&,vword&);

word vmagth(vword ,vword ,vword ,vword );

void fillhist1(vword,vword,vword,vword,vword,vword,vword,vword,vword,vword,vword,
      vword,vword,vword,vword,vword,word,vword&,vword&,vword&,vword&  );

void fillhist2(vword,vword,vword,vword,vword&,vword&,vword&,vword&);

addr  force_align(addr);
```

Primitive functions $vmagflagcomp$ $fillhist1$ and $fillhist2$ are very similar to those defined in design_3 but have a greater number of inputs and outputs because 4 groups of vector inputs are processed at a time and results are produced in a parallel way.

The primitive $vmagth1$, $vmagth2$, $vmagth3$ has been added. The first two calculates the sum of the elements of half vector, for each input vector. $vmagth3$ sums the received values and executes the shift of six bits. In this way the average value of 64 elements of the block is obtained.
The computation of the threshold has been divided into three different instructions in order to avoid the growth of critical path.

```
word vmagth1(v8word mag1,v8word mag2,v8word mag3,v8word mag4){
word result0=0;
word result1=0;
word result2=0;
word result3=0;
word magth1_t;
```

```
  int32_t m;
for (int32_t m = 0; m < 8; m++)
  result0+=mag1[m];
  for (int32_t m = 0; m < 8; m++)
  result1+=mag2[m];
  for (int32_t m = 0; m < 8; m++)
result2+=mag3[m];
  for (int32_t m = 0; m < 8; m++)
  result3+=mag4[m];

  magth1_t=result0+result1+result2+result3;
return magth1_t;
  }

  word vmagth3(word mag1,word mag2){
  word magth;
word magth_t;

  magth_t=mag1+mag2;
  magth="000000"::magth_t[15:6];
  return magth;
}
```

4.5.2 New vector instruction

A new instruction defined as *vector_instr* is added, it is an Or-rules between two instructions as can be seen in the following portion of code.

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

Vector Single instruction The instructions related to the computations of threshold has been added, while the instructions related to the magnitude flag and bis computations are been modified in order to extend the processing at 4 groups of vectors at the same time.

Additional functional unit and transistories are added:

```
// Vector unit
fu vmagth1;                              fu vmagth2;
trn tvecth1r<v8word>;                    trn tvecth2r<v8word>;
trn tvecth1s<v8word>;                    trn tvecth2s<v8word>;
trn tvecth1t<v8word>;                    trn tvecth2t<v8word>;
trn tvecth1b<v8word>;                    trn tvecth2b<v8word>;
trn tvecth1d<word>;                      trn tvecth2d<word>;

fu vmagth3;
trn tvecth3n<word>;
trn tvecth3m<word>;
trn tvecth3d<word>;
```

The following nML code describes the *vec_vmagth*1*_opn*, *vec_vmagth*2*_opn* and *vec_vmagth*3*_opn*, instruction rules:

```
opn vec_vmagth1_opn(r: c1u, s: c1u, t: c1u, u: c1u, z: c3u)
{
action {
stage E1:
R[z]=rte1=tvecth1d=
      vmagth1(tvecth1r=VC[r],tvecth1s=VC[s],tvecth1t=VB[t],tvecth1b=VB[u])
      @vmagth1;
}
syntax : "vmagth1"r ","s","t","u","z;
image  : s::r::t::u::z;
}

opn vec_vmagth2_opn(r: c1u, s: c1u, t: c1u, u: c1u, z: c3u)
{
action {
stage E1:
R[z]=rte1=tvecth2d=
      vmagth2(tvecth2r=VC[r],tvecth2s=VC[s],tvecth2t=VB[t],tvecth2b=VB[u])
      @vmagth2;
}
syntax : "vmagth2"r ","s","t","u","z;
image  : s::r::t::u::z;
}

opn vec_vmagth3_opn(r: c3u, s: c3u,  z: c3u)
{
action {
stage E1:
R[z]=rte1=tvecth3d= vmagth3(tvecth3m=rre1=R[r],tvecth3n=rse1=R[s]) @vmagth3;
}
syntax : "vmagth3"r ","s","z;
image  : s::r::z;
}
```

In the *action* sections, the behaviour of instructions are described: the functional units @$vmagth1$,@$vmagth2$,@$vmagth3$ perform the operations in the following figures:

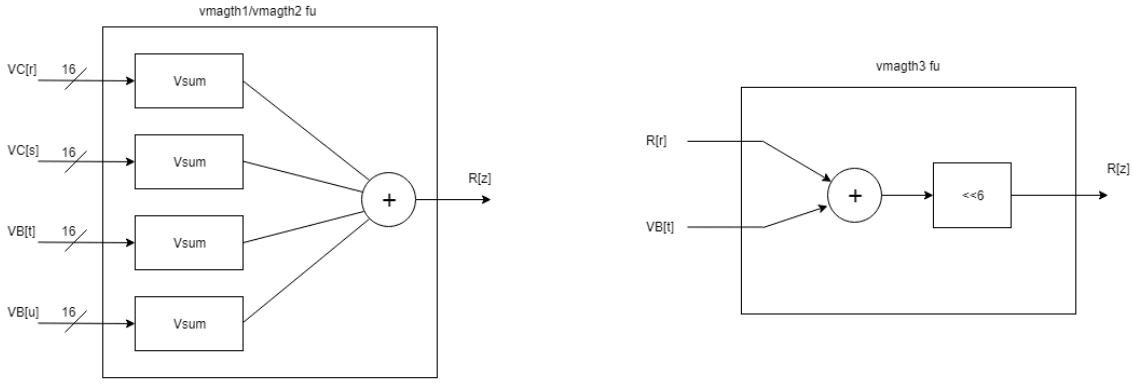

Figure 4.20: Vmagth

C_Code

The C code is modified in order to use SIMD instructions with higher parallelism.It can be noticed that the for loop is not more necessary: the 64 elements of D_x and D_y arrays are put into four *vpix* type variable. In this case it is obtained that only the final result *bin* need to be stored in scalar variables, so a great number of cycles can be saved.

```
vmagflagcomp(*d_x,*d_y,*d_x2,*d_y2,*d_x3,*d_y3,*d_x4,*d_y4,absx,absy,mag_t,flag_t,absx2,absy2,mag_t2,
flag_t2,absx3,absy3,mag_t3,flag_t3,absx4,absy4,mag_t4,flag_t4);

magth_t1=vmagth1(mag_t,mag_t2,mag_t3,mag_t4);
magth_t2=vmagth2(mag_t,mag_t2,mag_t3,mag_t4);
Mag_Th=vmagth3(magth_t1,magth_t2);

fillhist1(absx, absy ,mag_t, flag_t,absx2, absy2 ,mag_t2, flag_t2,absx3, absy3 ,mag_t3,
flag_t3,absx4, absy4 ,mag_t4, flag_t4,Mag_Th,pos0,pos1,pos2,pos3);
fillhist2(pos0,pos1,pos2,pos3,bin0_v,bin1_v,bin2_v,bin3_v);

int* temp1=(int*)&bin0_v;//addr1;
int* temp2=(int*)&bin1_v;//addr1;
int* temp3=(int*)&bin2_v;//addr1;
int* temp4=(int*)&bin3_v;//addr1;

                for(int zz =0; zz<9;zz++)
                {
                    bin[zz]=temp1[zz];
                    bin[zz+9]=temp2[zz];
                    bin[zz+18]=temp3[zz];
                    bin[zz+27]=temp4[zz];
                }
```

Figure 4.21: Fillhist Function

4.5.3 Compilation

The processor model of tmotion and the C code are compiled and simulated. The file obtained is compared with the correct one in order to ensure the consistency of results and the correctness of obtained design.
The program is simulated exploiting a total number of cycles equal to 267997 while the number of instructions that are executed is 261441. The total size occupied by instructions in program memory is 269 bytes. The values mentioned are divided among the various function of the program as listed in table:

Function	***Instr_Count***	***%***	***Cycle_Count***	***%***
Fill_hist	***66688***	***26.01%***	***68480***	***26.05.%***
Compute_gradient	***194740***	***47.24%***	***199496***	***47.42%***

Table 4.5: Instructions count among functions. Design 4.

Chapter 5

Simulation and Synthesis

5.1 HDL

GO tools, provided by ASIP DESIGNER allows to obtain a direct translation of the nML description of the processor: for each hardware unit nML (register, functional unit) a corresponding entity in VHDL or module in Verilog HDL is obtained. The following figure shows schematic representation of the datapath generated by GO, each rectangle represents an entity/module [16]. GO also generates a HDL test-bench, that can simulate the external stimulus of the processor.

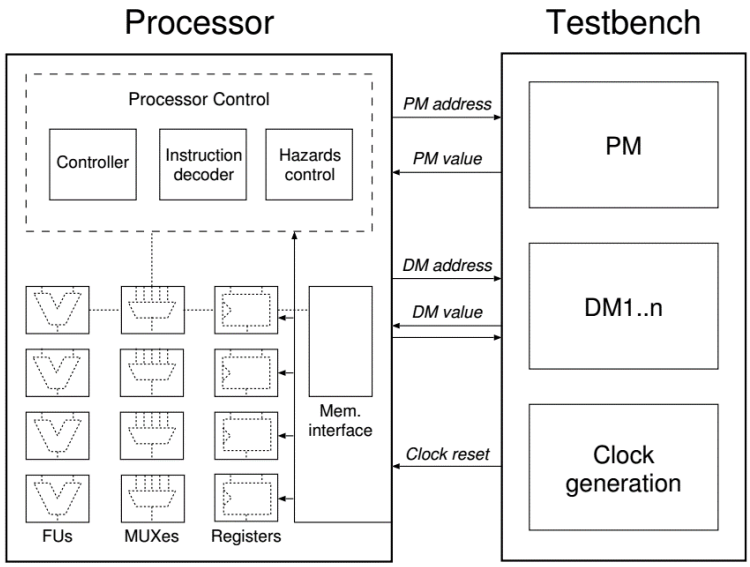

Figure 5.1: Schematic representation of the data path, generated by GO[16].

The files created by GO when Verilog HDL is generated are:

- In the *tmotion* folder, the Verilog file *tmotion.v* can be found: it represents

the top-level module that describes the processor, in which all its modules are interconnected.

- In the *controller* folder, *tmotion*'s subdirectory, there are the different parts of the controller. It contains the files:

  decoder.v: is the module that contains the instruction set decoder.

  controller.v: is the module that contains the implementation of the control behavior of primitive fetch and the computation of the next value assumed by program counter.

- The *prim* folder, *tmotion*'s subdirectory, contains the modules of all the functional units defined in the nML description.

- The *reg* folder contains the modules implementation of the physical registers and register files present in the nML description.

- The *mem* folder, *tmotion*'s subdirectory, contains the implementation of a memory interface for physical memories presented in the nML description: *Data_memory* and *Program_memory*. Memory interface puts in communication the data-path and the memory ports.

- The *pipe* and *mux* folders, *tmotion*'s subdirectories, contain all the implementations of pipeline registers and multiplexers.

- In the *test_bench* folder it can be found the following files: *tb_mem_DM* and *tb_mem_PM* test-bench entities are generated for the two phisical memories, they contain a behavioral model of the memory.

  clock_gen.v that implements the generation of Clock and reset signal.

  test_bench.v Test-bench module without ports, instantiating all components needed for simulation, including the processor unit under test.

Configuration options The GO configuration file is a text files that lists a number of GO configuration options needed to obtain the hdl implementation. In particular it is advisable to set this configurations to 0 in order to simplify synthesys steps:

log_register_writes[: 1|0] Generates extra HDL code to log register writes during simulation. Logging is done whenever a register location is written, changing its value or not.

log_memory_writes[: 1|0] Generates extra HDL code to log memory writes during simulation. Logging is done whenever a memory location is written, changing its value or not.

5.1.1 HDL simulation

In order to simulate the HDL generated by Go with an RTL simulator same steps and settings are needed.

Settings the Go configuration file In addition to test_bench Go generate a Makefile, with this means it is directly possible to simulate the HDL of the processor while it executes a program.

With the default settings for the generation of the Makefile, GO is able to generate commands that allow to analyze, elaborate and simulate the RTL. For Mentor/ModelSim the option needed are:

$modelsim\_makefile$;

$modelsim\_others\_ini : x$;

Where x is the path to a modelsim.ini file that maps the standard and IEEE VHDL libraries to their install directories.

Memory Content File Generation To simulate a program in the ELF binary format it first have to be translated in a format understandable by the test-bench. There are different ways to obtain the memory content: if the model doesn't contain I/O-interfaces, it is possible to use the $read\_elf$ command [17]. The generated Makefile supplies a target for this aim that calls $read\_elf$ with appropriate parameters. It can be used in this way (from the GO output directory):

$make\ test\ TEST = ../../HSG/Code/Release/tmotion$

This provides the memory content files data.PM and data.DM. It is also possible to open the Release folder of the project and digit the following command:

$read\_elf\ -G\ -fhath\ -e\ -PM = 16\ -mDMb = 8\ me\_HSG\ -o\ data$
where me_HSG is the name of executable contained in the Release folder.

5.2 Synthesis

As final step of this work the obtained processor has been synthesized by using Synopsys Design Compiler. The synthesis has been performed in order to find the maximum operating frequency that the design can achieve and the area. The synthesis flow can be divided into the following steps:

- reading Verilog source files;
- applying constraints;
- start the synthesis;
- save the results (timing and area) and the netlist;
- preparing saif files of the technological libraries;
- modifying the verilog test-bench including statements to get the switching activity;

```
initial begin
$read_lib_saif("/home/thesis/beatrice.giannetta/Desktop/Design1/saif/uk65lscllmvbbr_120c25_tc.saif");
$set_gate_level_monitoring("on");
$set_toggle_region(inst_tmotion);
$toggle_start;
end


always @ ( END_SIM_i ) begin
if (END_SIM_i) begin
$toggle_stop;
$toggle_report("../saif/tmotion_back.saif", 1.0e-9, "test_bench.inst_tmotion");
end
end
```

Figure 5.2: Statements included in test_bench to record switching activity.

- launching Modelsim with options to record switching activity;

- power consumption estimation with Synopsys Design Compiler.

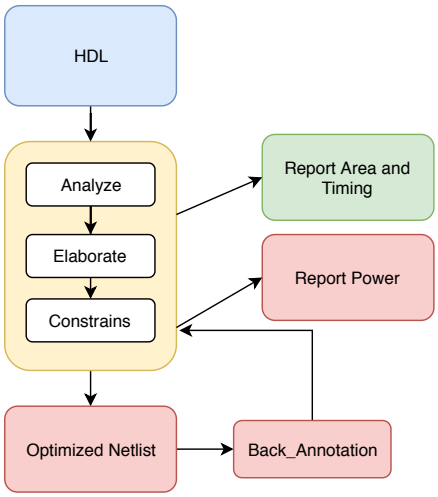

Figure 5.3: Synthesys Flow

The synthesis process generates two important files:

-*tmotion.v* file, the netlist file that is the description of the electronic circuit. It provides a list of the electronic components in the circuit and of the nodes.

-*tmotion.sdf* file describing the delay of the netlist, it provides the timing of each cell present in the netlist .

-*tmotion.sdc* file containing the constraints to the input and output ports in a standard format.

A worst case anaysyis has been done, in fact the technology library used is "$uk65lscllmvbbr\_090c125\_wc$" and the standard cell considered is $BUFM14R$.

HSG:Pure Sw Implementation	
Frequency	***271.7*** MHz
Area	***22417.92*** μm^2
Power	***251.8*** μW

Table 5.1: HSG:Pure Sw Implementation

HSG:Design2	
Frequency	***271.7*** MHz
Area	***27447*** μm^2
Power	***281.05*** μW

Table 5.2: HSG:Design2

HSG:Design3	
Frequency	***271.7*** MHz
Area	***124365*** μm^2
Power	***982.65*** μW

Table 5.3: HSG:Design3

HSG:Design4	
Frequency	***232.4*** MHz
Area	***524820*** μm^2
Power	***2.07*** mW

Table 5.4: HSG:Design4

5.3 Graphical result

A Matlab script is been developed in order to visualize, in a more direct way, the feature results obtained.

Figure 5.4: HSG Final Results

Chapter 6

Conclusions and Future works

The HSG algorithm has been identified as a very good solution to perform feature extraction for pedestrian detection application. It is based on the well-known HOG algorithm but simpler. In fact, HOG feature extraction requests floating point computation and a lot of memory accesses and solutions in hardware and software result to be very power expensive and complex. HSG, instead, allows to obtain a pedestrian detection framework which is efficient and fast, characterized by less expensive computation. In fact in the proposed solution, the orientation histograms are built without utilizing floating point operations but exploiting the capability of discrimination of locally significant gradients in addition to a fast Look-up-table based Support Vector Machine classifier. Moreover HSG presents better performance with respect to HOG on INRIA and ETH Data sets.

The initial part of this work of thesis provides an overview of the HSG Algorithm and a description of different design solutions like ASIP, ASIC and GPμP.

The central part of the thesis instead presents the implementation of designed processors in different versions realized exploiting the ASIP DESIGNER tool provided by Synopsys. Moving among these versions, starting from a pure sw implementation to a SIMD solution, an improvement in performance can be noticed but a significant growth in area is the cost to pay.

In the final part the results of synthesis, performed using a 65 nm CMOS technology have been presented.

6.1 Future works

Further works could be exploited with the aim to obtain improvement in the architecture of realized processors.
A possible solution could be to increase instruction parallelism. In fact implementing

a processor of MIMD (Multiple Instruction Multiple Data) type allows to execute different instructions in parallel, this could lead to a very significant increase in performances .

Bibliography

[1] C.-Y. L. Pei-Yin Chen, Chien-Chuan Huang and Y.-H. Tsai, "An efficient hardware implementation of hog feature extraction for human detection," in *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS*, vol. 15, pp. 656–662, IEEE, 2015.

[2] B. B.-R. W. Colm Kelly, Fahad Manzoor Siddiqui, "Histogram of oriented gradients front end processing: an fpga based processor approach," in *ECIT, Queens University, Belfast*.

[3] S. Bauer, *FPGA Implementation of a HOG-based Pedestrian Recognition System*. MPC Workshop, 2009.

[4] M. Bilal, A. Khan, M. U. K. Khan, and C.-M. Kyung, "A low-complexity pedestrian detection framework for smart video surveillance systems," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 10, pp. 2260–2273, 2017.

[5] "Asip designer." https://www.synopsys.com/dw/ipdir.php?ds=asip-designer/.

[6] W. H. ORGANIZATION, *World Health Organization. Global status report on road safety*. Management of Noncommunicable Diseases, Disability, Violence and Injury Prevention (NVI), 2015.

[7] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1, pp. 886–893, IEEE, 2005.

[8] "Histogram of oriented gradients." https://www.learnopencv.com/histogram-of-oriented-gradients/.

[9] P. Dollár, C. Wojek, B. Schiele, and P. Perona, "Pedestrian detection: An evaluation of the state of the art," *PAMI*, vol. 34, 2012.

[10] J. Wu and J. M. Rehg, "Beyond the euclidean distance: Creating effective visual codebooks using the histogram intersection kernel," in *Computer Vision, 2009 IEEE 12th International Conference on*, pp. 630–637, IEEE, 2009.

[11] *Target Tmotion core-ASIP Designer*. Synopsys, K-2015.12.

[12] *Tmicro Core Processor Manual-ASIP Designer*. Synopsys, Version M-2017.03.

[13] G. S. Henrique Malvar, *YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range*. Microsoft Corp., 2003.

[14] S. S.-M. Sebastian Bauer, Ulrich Brunsmann, "Fpga implementation of a hog-based pedestrian recognition system," in *Faculty of Engineering Aschaffenburg University of Applied Sciences, Aschaffenburg, Germany*.

[15] *Tvec core-ASIP Designer*. Synopsys, Version L-2016.03.

[16] *Go User Manual nML to synthesizable HDL translation-ASIP Designer*. Synopsys, Version N-2017.09.

[17] *ASIP Programmer Read elf User Manual-ASIP Designer*. Synopsys, Version N-2017.09.