Tutorial : Easy development of motion estimation algorithms and processors
Authors : Jose Luis Nunez-Yanez, George Vafiadis, Trevor Spiteri
Version : 2009.1 (April 2009)

1. Overview

The Motility motion estimation processor is a reconfigurable ASIP (Application Specific Instruction Set Processor) designed to execute user-defined block-matching motion estimation algorithms optimized for hybrid video codecs such as MPEG-2, MPEG-4, H.264 AVC and Microsoft VC-1. The core offers scalable performance dependent on the features of the chosen algorithm and the number and type of execution units implemented. The ability to program the search algorithm to be used, and to reconfigure the underlying hardware that it will execute on, combines to give an extremely flexible motion estimation processing platform.

A base configuration consisting of a single 64-bit integer pipeline, capable of processing 1080p HD video at 30 frames per second using a hexagonal motion estimation search followed by a square refine (as used the x264) with 1 reference frame and 16x16 block size can be implemented in 2,300 FPGA logic cells. In contrast, a complex configuration including support for motion vector candidates, sub-blocks, motion vector costing using Lagrangian optimization, four integer-pel execution units and one fractional-pel execution unit plus interpolation will need around 14,000 logic cells. A simplified diagram illustrating these two configurations is shown in Fig. 1. At least one integer-pel execution unit must always be present to generate a valid processor configuration but the others units are optional, and are configured at synthesis time.

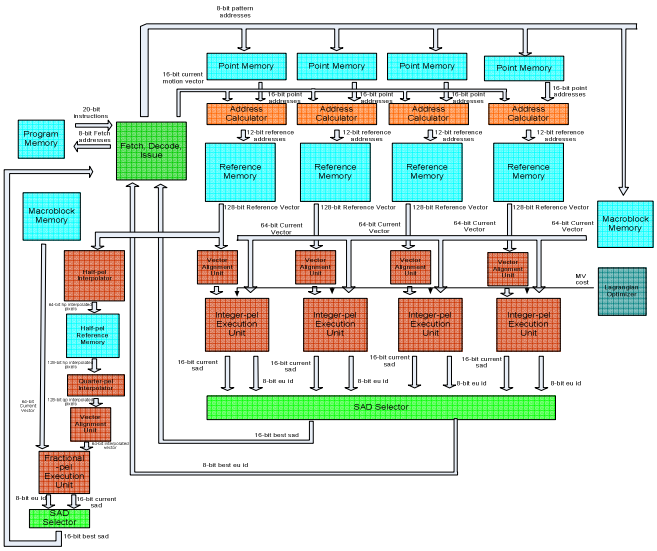

Fig.1.a Complex processor configuration

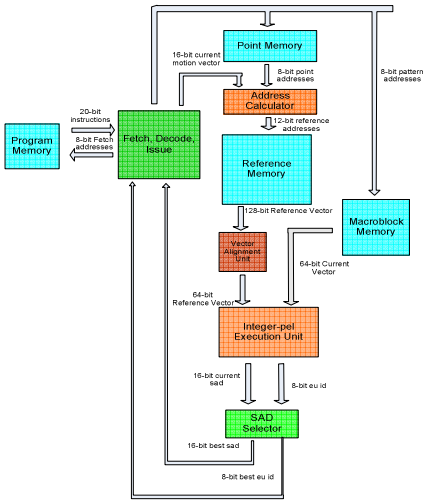

Fig.1.b base processor configuration

1.1 Features

- Intuitive and easy programming using a c-like syntax of user-defined block matching motion estimation algorithms.
- Highly configurable architecture enables the designer to optimize the hardware for the selected algorithm.
- Binary compatibility so that once an algorithm has been compiled it can be executed by any hardware configuration.
- Support of advance features such as rate distortion optimization using Lagrangian techniques, sub-partitions and fractional pel searches according to the codec standard.
- Efficient evaluation of multiple user-defined motion vector candidates transparently to the rest of the algorithm.
- Toolset available to enable the efficient exploration of the large design space and the generation of the RTL configuration file for the hardware processor library.

1.2 Applications

- Video coding (H.264, MPEG-4, MPEG-2, VC-1, AVS)
- Video enhancement applications such as frame rate conversion, de-interlacing, super-resolution and video stabilization.

1.3 Tools

A toolset has been developed that enables the algorithm designer access to the hardware features without any knowledge of the processor microarchitecture. The toolset IDE is a fully integrated environment composed of a compiler, assembler, cycle accurate model and RTL export. The algorithm designer can create a new algorithm for the required application using typical C constructs such as for, while loops and if-else constructs. The compiler automatically recognises the search points that correspond to fractional-pel searches and generates the correct instructions.

Parallelism is extracted by the compiler by coding search patterns composed of a variable number of search points in a single instruction. The hardware analyses the instruction and distributes the load to the available execution units. Using the cycle accurate model the designer can quickly explore the performance of many configurations in terms of frame per second throughput, compressed video bit-rate and PSNR, hardware complexity and power/energy consumption.

The impact of changes in the original search algorithm can be evaluated before exporting the selected configuration hardware file and program binary. The final implementation can then be generated by processing the configuration file and the rest of the RTL processor description with standard tools such as Synplicity and/or Xilinx ISE.

1.4 Deliverables

Toolset (compiler, cycle accurate model, documentation and analysis tools), VHDL configurable processor description, VHDL testbench, FPGA prototype implementation using the PCI bus also available together with the original design team.

PROCESSOR CONFIGURATION	CYCLES PER MB (1080P FRAMES PER SECOND)	FPGA COMPLEXITY (LUTS)	MEMORY (BRAMS)
Intel P4 C code (Integer search)	~ 30,000	N/A	N/A
Intel P4 SSE2 assembly (Integer search)	~ 6,000	N/A	N/A
Intel P4 SSE2 assembly (Fractional search)	~ 14,000 (+ ~6000 half pel interpolation)	N/A	N/A
One integer execution unit (Integer search)	785 (30 FPS at 200 MHz)	~2,300 (~9% of Virtex-4 SX35)	21 BRAMs (~10% of Virtex-4 SX35/2 reference windows of 112x128 pixels each)
One integer and one fractional execution units (Fractional search)	966 (25 FPS at 200 MHz)	~9,100 (~30% of Virtex-4 SX35)	33 BRAMS (~17% of Virtex-4 SX35/2 reference windows of 112x128 pixels each)
Three integer and two fractional execution units (Fractional search)	566 (43 FPS at 200 MHz)	~14,000(~47% of Virtex-4 SX35)	79 BRAMS (~41% of Virtex-4 SX35/2 reference windows of 112x128 each)

Table.1 Evaluation of fast motion estimation search using 1080P high definition sequences. H.264 integer hexagon search algorithm with square refine, up to 8 MV candidates, Lagrangian MV cost optimization, 112x128 search reference area. Fractional refinement adds 2 diamond half-pel interactions followed by 2 diamond quarter-pel interactions. Intel P4 results included as a computational complexity reference.

2. Programming model

2.1 EstimoC Language

EstimoC is a high level language, powerful enough to express a broad range of block matching motion estimation algorithms in a natural way. The Estimo C code is written in the SharpEye Studio or any other compatible editor and is processed by the EstimoC compiler.

The EstimoC has a natural syntax with elements from C and with special structures for the development of motion estimation algorithms. Part of the language is dedicated to the preprocessor and other parts are for the core decode control unit. The preprocessor is a crucial part of the compiler because it provides macro facilities for the development of the algorithms. The preprocessor is behind the flexibility of Estimo C.

In Estimo C, there is no variable data type definition. The data type is obvious from the usage. The variable names are used as an alias of expressions ex. a = 1232+ (12/2)*3. Then you can use the name a instead of the expression. A single search point is defined anonymously without a name. It is possible to be part of a pattern (pattern scope), block scope or global scope. A search point is defined as “check (1, 5)”.

2.2 Preprocessor Statements

Estimo C preprocessor has all the traditional conditional and looping structures:

- if (expression) {…} if the expression is evaluated to 1 is considered to be true
- if(expression) {…} else {…} if-then-else

- while(expression) {…}
- do {} while(expression)
- for(variable = value to upto-value step value) {…} in the for structure you specify a variable and the range to take (the starting value the goal value and the step).

2.3 Motion Estimation Language Features

2.3.1 Patterns

The pattern is the most important concept for a motion estimation algorithm. The efficiency of the algorithm is based on the underline pattern used. In EstimoC there are three ways to define a pattern:

1. Using the graphical pattern generator, you specify the coordinates of the point offsets.
2. Using a static pattern specification (preferred), an example is given below:

```
Pattern(hexagon)
{
 check(1,2)
 check(2,0)
 check(1,-2)
 check(-1,-2)
 check(-2,0)
 check(-1,2)
}
```

Using this syntax we define a new pattern, named hexagon. The statement "check(hexagon);" after the pattern's definition, instructs the motion estimation control unit to calculate the sum of absolute differences (SAD) for each motion vector specified in the pattern and the one with the smallest SAD is the winning motion vector, the ID of this vector is stored in a register which can be referenced from EstimoC programs using the WINID identifier. The sum of absolute differences can also be referenced using the SAD identifier used for example in the UMH (Uneven Multi-Hexagon Cross search) implementation available in the samples directory. For two sequential check commands "check(hexagon); check(hexagon)" The processor will calculate the best match from the first pattern and the next check is applied to a new origin based on the winning vector of the first pattern.

2.3.2 Dynamic Pattern Generation.

The dynamic pattern generation is the most advanced method to specify a pattern. This functionality is performed in the preprocessor level hence there is no overhead in execution to the hardware unit. The central idea is to generate all the pattern points through code. The algorithm designer writes a sequence of simple check instructions in the form check(x,y) followed by the "update;" instruction. All simple point checks are collected from the compiler and a new pattern is generated, this pattern is checked by the hardware unit. Using the program statements supported by the preprocessor it is possible to generate a great variety of motion estimation algorithms expressed in a natural way.

3. Getting started with the tools

You can get the latest version of the compiler tools installer at http://sharpeye.borelspace.com/.
Once you have installed the tool and after starting the designer is presented with window shown in Fig.2. At this point the designer can either do *file -> new* and create a new *name.est* file defining a

new motion estimation algorithm or open an already developed motion estimation algorithm. In this tutorial we will open the *hex.est* hexagonal search file available in the samples subdirectory. The source code for *hex.est* is illustrated in Fig.3. It corresponds to an integer hexagonal search with up to 8 interactions followed by a square refinement and then followed by up to 2 half-pel diamonds and 2 quarter-pel diamonds. The point centered at (0,0) is search first and then the rest of the algorithm executes. The WINID checks take care of breaking the loop once the winning position in an interaction is centered in the middle of the pattern. The compiler button indicated in Fig.4 is used to compile the source code into the binary code ready to be executed by the processor.

```
/// Copyright (C) Jose Nunez-Yanez
/// Source D:/projects/me_interpolation_16m/estimo_sources/hex.est

qpi = 2;
hpi = 2;
fpi = 8;

Pattern(square)
{
 check(0,1)
 check(0,-1)
 check(1,0)
 check(-1,0)
 check(-1,-1)
 check(-1,1)
 check(1,-1)
 check(1,1)
}

Pattern(hexbs)
{
 check(2,0)
 check(-2,0)
 check(1,2)
 check(-1,2)
 check(1,-2)
 check(-1,-2)
}

Pattern(diamondhp)
{
 check(0,0.5)
 check(0,-0.5)
 check(0.5,0)
 check(-0.5,0)
}
Pattern(diamondqp)
{
 check(0,0.25)
 check(0,-0.25)
 check(0.25,0)
 check(-0.25,0)
}

check(0,0);
update;

//hexagon search
for(loop = 1 to fpi step 1)
{
 check(hexbs);
 #if( WINID == 0 )
    #break;
}

//square refinement
check(square);

//half pel diamonds
for(loop = 1 to hpi step 1)
{
 check(diamondhp);
 #if( WINID == 0 )
    #break;
}

//quarter pel diamonds
for(loop = 1 to qpi step 1)
{
 check(diamondqp);
 #if( WINID == 0 )
    #break;
}
```

Fig.3. hex.est source code

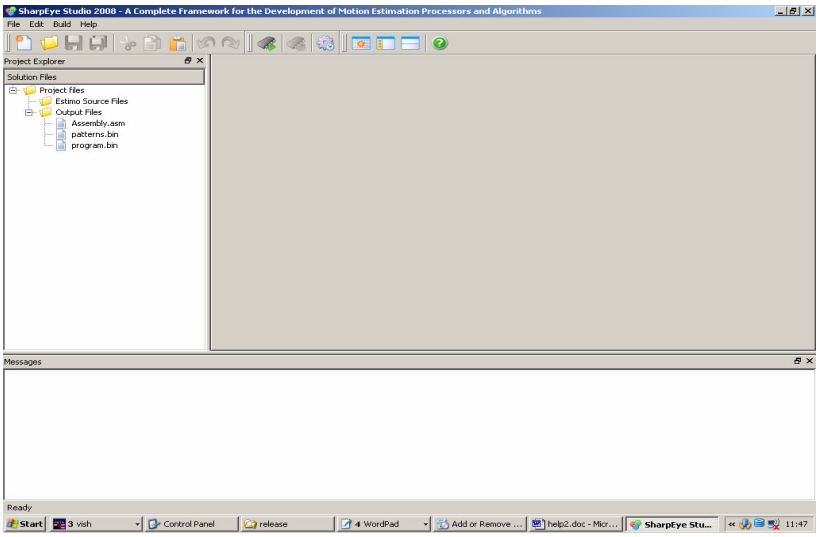

Fig.2. Starting SharpEye

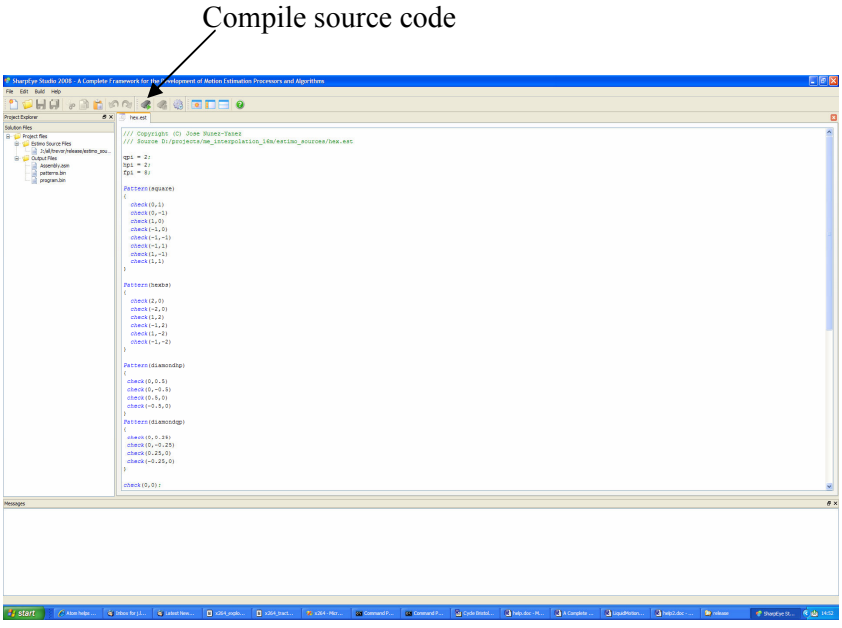

Fig.4. Launching the compiler

Any problems during compilation are reported in the IDE and can be corrected by the designer. Once compilation is successful the results of the process are displayed in a new window. The assembly view in this window is visible in Fig.5. The binary itself for the program and pattern memory can also be checked together with the configuration of the compiler. The results of the compilation are written in the estimo.output directory created in the same directory in which the original source code is available. Notice that there are two binaries produced in the output the *program memory* that defines the search algorithm and the *patterns memory* that defines all the patterns used during the search process.

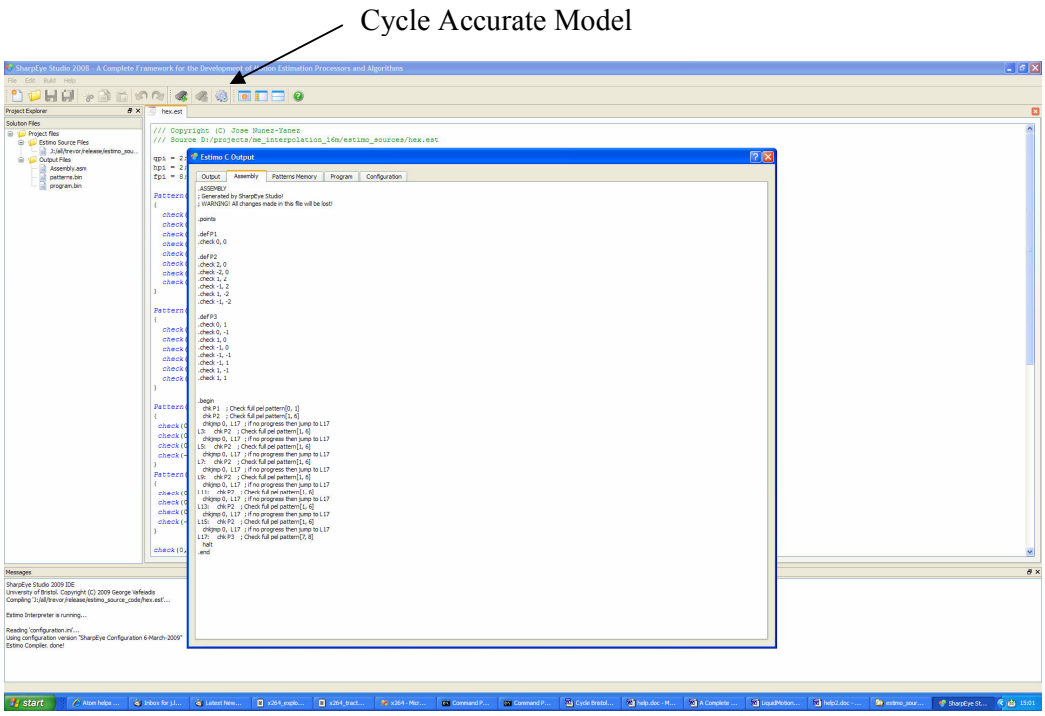

Fig.5. Launching the cycle accurate simulator

The cycle accurate model window enables the exploration of multiple processor configurations for the candidate algorithm just compiled. The options available enable using different number of integer and/or fractional pel execution units to evaluate the effects on performance and energy comsumption, options such as using motion vector candidates and/or Lagrangian optimization are also available both of which have positive effects in reducing the bit rate. H.264 options such as using more than one reference frames and/or sub-partitions are also available. Notice that if you intend to use motion vector candidates your source code should start checking the (0,0) point as shown in the example of Fig.3. This is a good idea since the processor will evaluate all the motion vector candidates using the first instruction of the program. Then the (0,0) point will be offset by each motion vector candidate and evaluated. The winner of this initial search will then be used as the starting point for the rest of the algorithm. Fig. 6 shows the options available under the cycle accurate model view using a different sample algorithm consisting of a number of diamonds searches followed by small full search fractional refinement. Once the algorithm binaries, processor configuration and input video data have been set the designer will click on *run* to execute the cycle accurate simulation. The results of the simulation are also visible in the same window and can be plotted using the *new plot* button. The plots enable quick exploration of different configuration comparing the results according different requirements: power, throughput, bit rate, PSNR, hardware resources, etc. The *table* button launches a different view that enables the designer to

select the configuration that meets his or her application requirements. The *table* view also enables renaming the plot points to names more representative of the configuration details. Once the designer is happy with the quality of results and performance of his software algorithm and hardware configuration he/she can generate a VHDL configuration file from the *table* window. This configuration file can then be added to the VHDL hardware library that contains the sources for the configurable processor description. This configuration.vhd together with the rest of the hardware library can then be synthesized using standard tools such as Simplicity and/or Xilinx ISE to obtain the final bistream implementation.

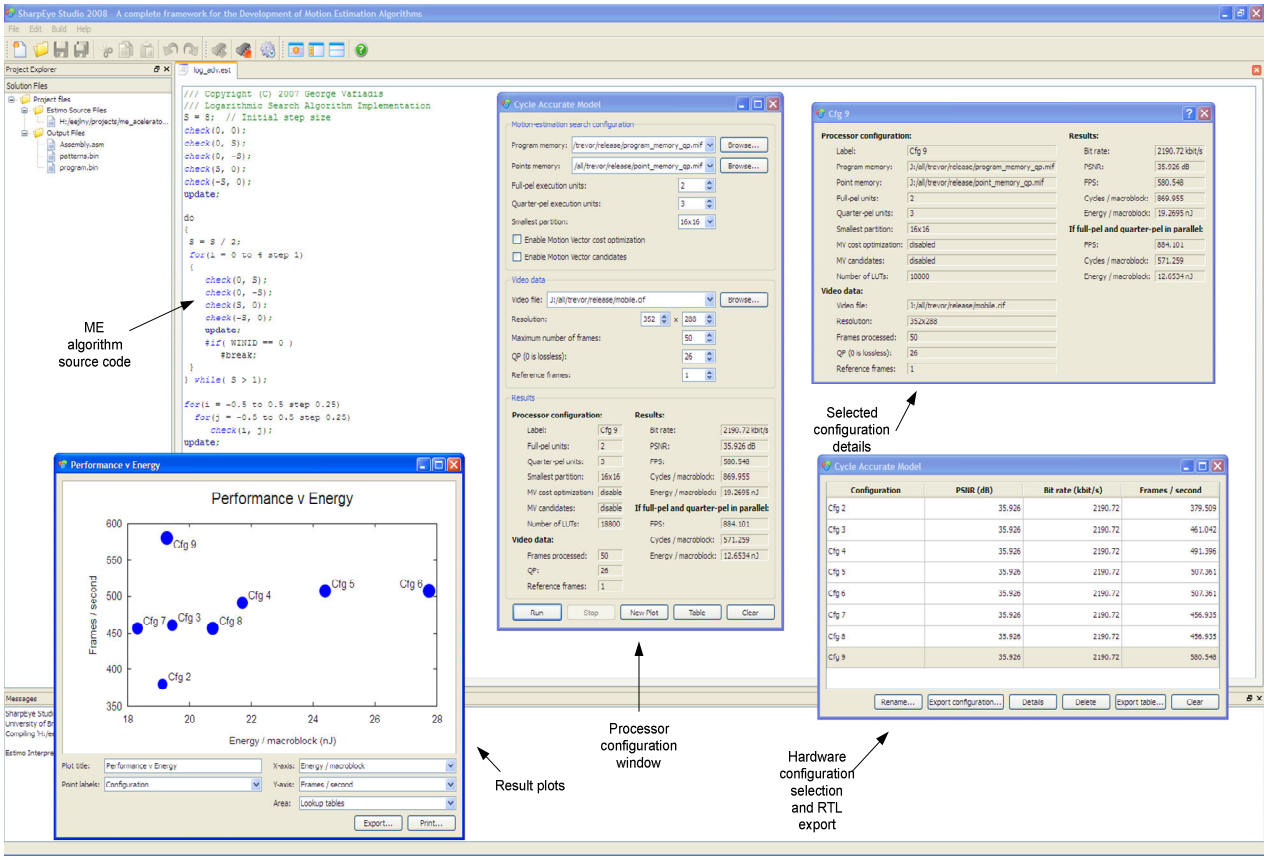

Fig.6. Design space exploration view

4. Using the open-source processor.

4.1 Simulation

A modelsim project file called me_top.mpf has been provided. After opening you should do a compile all. Notice that you need access to the XilinxCoreLib library that contains the memory models. The XilinxCoreLib library should be available as part of your Xilinx ISE release.If you get a message like ** Error: (vsim-13) Recompile work.me_top because work.config has changed. Simply recompile everything again to eliminate compilation order dependencies. Once the project has been compiled you can simulate the provided testbench tb_me_top.vhd. If you run the simulation for 1000 us you should see messages from the testbench like :

```
# 329011 ns MV/SAD OK  Expected: FF000444   Received: FF000444
#  382811 ns MV/SAD OK  Expected: FF000669   Received: FF000669
#  436511 ns MV/SAD OK  Expected: 0000095E   Received: 0000095E
#  532411 ns MV/SAD OK  Expected: 02FC03BB   Received: 02FC03BB
#  586211 ns MV/SAD OK  Expected: FF000869   Received: FF000869
#  640011 ns MV/SAD OK  Expected: 00000C46   Received: 00000C46
#  693911 ns MV/SAD OK  Expected: FF000CEE   Received: FF000CEE
```

This indicates that obtained MV/SAD combinations are OK and the core works correctly. The current program that the core is executing can be found in the point_memory_qp.mif and program_memory_qp.mif. Other example programs are available in the sample programs directory. If you want to create your own motion estimation programs and simulate them in the core the process is simple. Using the toolset you will obtain the bin files and then simply change the names to point_memory_qp.mif and program_memory_qp.mif and replace the original ones. Notice that now a new simulation will execute your own programs and the MV/SAD values will change and the testbench will complaint since the testbench is hardwired to the values of one algorithm only.

4.2 ISE & Synplify

The RTL synthesis can be completed using Synplify or ISE. We recommend Synplify since the performance and quality of results will be higher. A sample prj file for synplify synthesis is available in the SYN directory. Once you have obtained the EDF netlist is time to place&route using ISE to obtain the bitstream file. You can create the project in the ISE directory and add the netlist from Synplify as source. Notice that you will need to tell ISE about the netlist directory that contains the netlist files for the core memories. The point and program memory edn implement the same search algorithm as used in the simulation.
If you want to change the program that the core is going to execute you must either obtain new netlists with different initialization data or load the new programs at run time. To load a new program at run-time in a real FPGA you need to know how the the program and point memory are mapped . You can obtain this information from the accompanying data sheet.

5. Conclusions

New video coding standards such as h.264 include a plethora of options for performing motion estimation that can be efficiently exploited using the programmable and configurable motion estimation processor technology described in this paper. The compiler tool chain enables a designer to quickly optimize the core for a particular application in a very short time. The current processor implementation has been prototyped in Xilinx FPGAs clocking at 200 MHz in a Virtex-4 SX35 and 100 MHz in a Spartan 3 device which is sufficient to support high definition video formats and obtain high quality motion estimation results. Porting to another FPGA or ASIC technologies should not represent a problem since the RTL is fully synchronous and specific building blocks such as the dual-port RAM memories are generally available.

The open-source processor is hardwired to a single integer execution unit, one reference frame, 16x16 block size, no adaptive thresholds or fractional pel support. In the open-source RTL version you should not change anything in the config.vhd file since you need access to the full RTL library to enable the extra features.

Contact us at eejlny@bristol.ac.uk or eejlny@byacom.co.uk if you would like to know more and get access to the full RTL library which supports these features.