

# Design of Application-Specific Instruction-Set Processors for Multi-Media, using a Retargetable Compilation Flow

Werner Geurts

Gert Goossens

Dirk Lanneer

Johan Van Praet

Target Compiler Technologies  
 Technologielaan 11-0002  
 B-3001 Leuven, Belgium  
 Tel.: +32 16 40 81 14

Email: {geurts, goossens, lanneer, vanpraet}@retarget.com

**Abstract** – This paper describes Target’s tool flow for the design and optimisation of application-specific instruction-set processors (ASIPs). Thanks to a user-friendly processor description language, an efficient retargetable C compiler, and accurate simulation and profiling tools, engineers can design the best ASIP for their application in only days of time. The application of the tool flow to design video and image coding ASIPs for multi-media is discussed.

## 1. Introduction

In System-on-Chip design, one can witness an increasing use of programmable processor cores of which the architecture and instruction set are optimised to a specific application domain. These cores are referred to as application-specific instruction-set processor or ASIP cores. ASIPs fill the architectural spectrum between general-purpose programmable processors and dedicated hardware or ASICs. They allow to effectively combine the “best of both worlds”, i.e. high flexibility through software programmability and high performance (high throughput and low energy consumption).

In the application domains of video and image processing, the ideal ASIP may use a mix of very long instruction word computing (VLIW), single-instruction multiple data (SIMD) computing, and custom data paths. Efficient design tools are needed to make optimal architectural tradeoffs, and to produce both the processor hardware and a set of programming tools for the new ASIP in a minimum time.

In this paper a tool flow is described for the design, optimisation, and verification of ASIP cores. Although the tool flow is not restricted to this domain, this paper focuses on mobile multi-media systems, and more particularly on video and image coding applications. As a demonstrator, an ASIP is designed for motion estimation in MPEG4 using a multi-step search algorithm.

At the heart of this tool flow is an optimising retargetable C compiler, complemented with retargetable instruction-set simulation and profiling tools, and a retargetable hardware (VHDL or Verilog) generator. Designers can perform true architectural exploration to match the ASIP architecture to the application. At the same time, excellent software

development tools for the new ASIP become available, as well as an efficient implementation of the ASIP core in hardware.

## 2. Target’s retargetable tool suite for ASIP design

Chess/Checkers is Target’s retargetable tool-suite for the design, programming, and verification of ASIP cores. An outline of the Chess/Checkers tool suite is shown in Figure 1. Chess/Checkers consists of the following retargetable tools:

- *Chess*: a *retargetable C compiler* that translates C source code into machine code for the target processor. Different from conventional compilers such as GCC [1], Chess uses graph-based modelling and optimisation techniques [2], to deliver highly optimised code for specialised architectures exhibiting peculiarities such as complex instruction pipelines, heterogeneous register structures, specialised functional units and in-level parallelism. The compiler comes with a retargetable assembler and disassembler called Darts, and a retargetable linker called Bridge.
- *Checkers*: a *retargetable instruction-set simulator (ISS) generator* that produces a cycle and bit accurate ISS for the target processor. The ISS can be run in a stand-alone mode or be embedded in a co-simulation environment through an application programming interface (API), optionally including SystemC wrappers. Checkers includes a graphical debugger that can connect both to the ISS, and to the available processor hardware for on-chip debugging. The connection is made via a JTAG interface. Source-level debugging and code profiling are supported.
- *Go*: a *hardware description language (HDL) generator* that produces a synthesisable register-transfer level HDL model of the target processor core. Through APIs, users can plug in their own HDL implementations of functional units and of the memory architecture.
- *Risk*: a *retargetable test-program generator* that can generate assembly-level test-programs for the target processor with a high fault coverage. These test programs can then be executed both in the ISS and in the HDL model of the processor, to check for the consistency of both models.

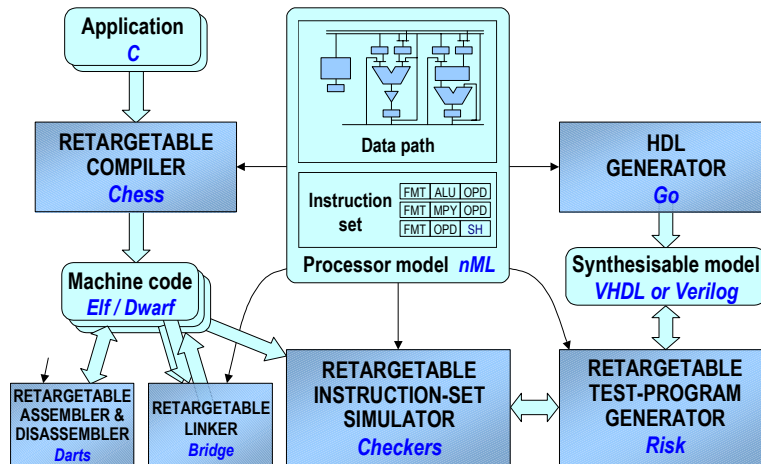


Figure 1 - Outline of Target's Chess/Checkers tool suite.

A unique feature of the Chess/Checkers tool suite is its architectural retargetability, based on the nML processor description language. nML is a high-level language that captures a programmer's model of the target processor [3] [4]. This is the abstraction level commonly found in a programmer's manual of a processor. Using nML, an architecture designer can quickly define the instruction-set architecture of a processor. After reading the nML description, the different Chess/Checkers tools are automatically targeted to the specified architecture.

### 3. Motion estimation example

Our demonstrator is the design of an ASIP for motion estimation. This is a common function in many video coding systems, such as H.263, H.264, and MPEG4.

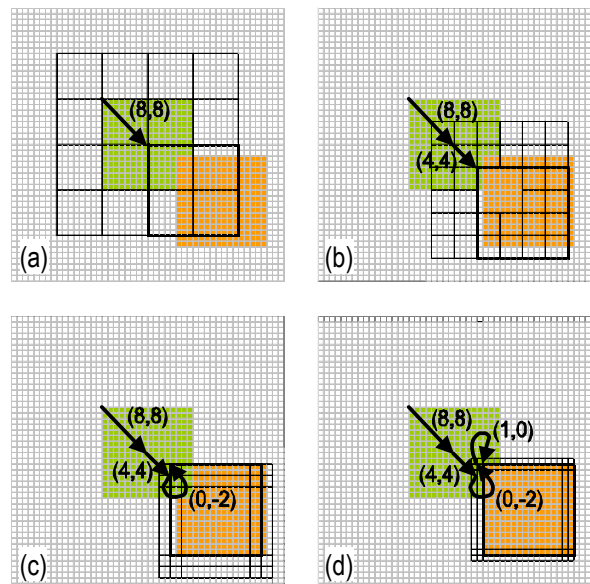
Motion estimation aims at finding the best translation vector between the previous and current video frame. The underlying assumption is that objects in the scene have translational motion. The current frame is divided into macro blocks, typically 16x16 pixels in size. Each block in the current frame is compared to a number of search blocks in the previous frame, and the best matching search block is selected. The purpose of the algorithm is to calculate a motion vector for each block in the current frame, which denotes its displacement with respect to the best matching search block.

The motion estimation algorithm used in this paper is a multi-step iterative search algorithm [5]. While this algorithm is sufficient to illustrate our ASIP design methodology, it should be noted that practical motion estimation algorithms may be more complex. The Chess/Checkers tool flow has been used successfully to design of ASIPs for industrial multi-standard video coding systems.

The multi-step search algorithm works as follows. A search area with a size of 48x48 pixels is used. In every iteration, the best match is determined between a reference block of 16x16 pixels from the current frame, and 9 possible search blocks of the same size located in the previous frame. These search blocks are located at the position of the reference block, and at 8 surrounding positions at a horizontal and/or

vertical distance equal to a given step size. The step size has a value of 8 in the first iteration, and is halved in each consecutive iteration. The total number of iterations therefore equals 4. Each iteration, the displacement vector between the previous and the current best matching blocks is recorded. The eventual motion vector is the sum of these displacement vectors.

Figure 2 illustrates the multi-step search algorithm. (a) through (d) show the different iterations or steps. The frame size is 48x48. Shaded areas represent the reference block (middle) and its best matching search block (lower right). With each step, the best matching block among 9 candidates is shown (bold rectangle) as well as the displacement vectors. The eventual motion vector is the sum of all four displacement vectors.



$$\begin{aligned} \text{Motion vector} &= (8,8) + (4,4) + (0,-2) + (1,0) \\ &= (13,10) \end{aligned}$$

Figure 2 - Illustration of multi-step search algorithm for motion estimation.

The computational kernel of the algorithm is a function for calculating the similarity between the

reference block and a search block. The similarity measure is the “sum of absolute differences” (*SAD*) function:

$$SAD(x,y) = \sum_{i=0}^{15} \sum_{j=0}^{15} |Search(i+x, j+y) - Ref(i, j)|$$

which is implemented by the C code of Figure 3.

```
#define ABS(x) ((x) < 0) ? -(x) : (x)
const int LINE_STEP = 48; // advance 1 line
                          // in search frame
int sad_16x16(unsigned char* search, unsigned
              char* ref)
{
    int sad = 0;
    for (int i = 0; i < 16; i++)
    {
        for (int j = 0; j < 16; j++)
        {
            sad += ABS(search[j] - ref[j]);
        }
        search += LINE_STEP; // advance to next
                             // row
        ref += 16;
    }
    return sad;
}
```

Figure 3 - C code of *SAD* function.

#### 4. Design of a motion estimation ASIP

This section describes the process of designing an ASIP, optimised for the motion estimation algorithm introduced in Section 3. The Chess/Checkers retargetable tool suite is used for architectural exploration.

The design process is iterative and interactive. A simple baseline architecture, described in the nML processor description language, is used as the starting point. The C description of the motion estimation application is compiled on this architecture, using the Chess compiler. The resulting machine code is simulated in the Checkers ISS. Based on profiling tables produced by the ISS, critical sections are identified in the machine code. An analysis of these sections reveals potential improvements of the instruction-set architecture. The designer then quickly adds these improvements to the nML description, and repeats the whole process to verify the gain achieved, and to explore additional architectural optimisations.

The main optimisation criterion in this process is the number of instruction cycles needed to run the application. Since a low cycle count allows to run the ASIP with a lower clock frequency and a lower supply voltage, the energy consumption is optimised indirectly.

When desired, an RTL model can be generated quickly for each architecture, by running the Go HDL generator. Logic synthesis allows to verify the achievable clock frequency. If necessary, the designer can modify the pipeline structure of the ASIP in nML, in order to get timing closure.

#### 4.1. Design 1: a 16-bit microprocessor

As a baseline architecture, a 16-bit microprocessor is selected, called Base. Base is included in a library of example nML descriptions that comes with the Chess/Checkers tools. Base offers full support for compiling integer C code.

The data path of Base is shown in Figure 4, and contains the following units: an arithmetic and logic unit (ALU), a multiplier-accumulator, a shifter, an address generation unit, a central register file with 8 registers, and a 64K byte-addressed data memory. Note that a pixel is stored as one byte. Base has a separate program memory with 64K words of 16 bits. Instructions are coded in 16, 32 or 48 bits.

On Base, the *SAD* function is fully implemented in software. For example, absolute values will be calculated using conditional branching. Table 1 (Design 1) shows the results obtained by running the Chess/Checkers tools.

	Cycles Total	Cycles <i>SAD</i>	Instr. Total	Instr. <i>SAD</i>	Gates (10 ns)	Gates (5 ns)
Design 1	87,730	86,040	134	20	11,290	13,053
Design 2	14,114	12,420	132	16	12,396	14,393
Design 3	5,906	4,212	128	13	26,370	29,585
Design 4	3,782	2,088	129	14	26,623	29,502
Design 5	3,350	1,656	132	17	26,536	29,641

Table 1 - Results of different iterations in the design of a motion estimation ASIP.

Table entries include:

- Cycle count for the entire application and for the *SAD* function only (obtained from the ISS);
- Instruction count for the entire application and for the *SAD* function only (obtained after compilation);
- Gate count for the architecture, excluding memories, for different clock speeds (obtained after HDL generation and logic synthesis). In this design a 130 nm technology is used.

When simulating the generated code, the ISS generates profiling information in different forms. Profiles include statistics on the use of instructions and hardware resources (registers, busses, etc). Table 2 shows an excerpt of an instruction profile table generated by the ISS. For each C function, the generated machine code is displayed, with the number of times each instruction is executed and the number of cycles spent in each instruction.

Using these tables, the designer can immediately locate time-critical sections in the code. An analysis of the assembly code for these sections quickly reveals potential improvements of the instruction-set architecture.

In Table 2, the critical section is from program counter (PC) value 24 to 31, which is the inner loop of the *SAD* function. The following observations can be made:

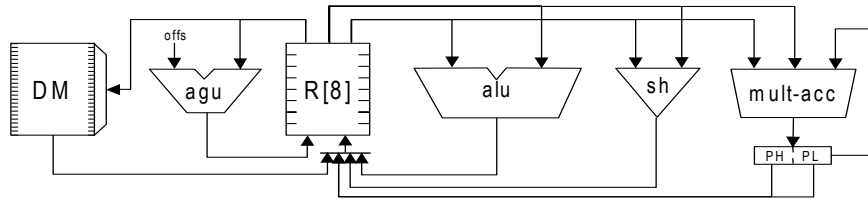


Figure 4 - Data path of Base.

- Absolute value computations in the *SAD* function require 4 instructions: compare (*lt*), conditional jump (*jcr*), unconditional jump (*jr*), and subtract (*sub*).
- Except for address calculations (e.g. *r2++*) in parallel with loads from data memory (*lbu*), no instruction-level parallelism is available in the Base processor.

Based on these observations, the designer identified several possible architectural optimisations, described below.

#### 4.2. Design 2: microprocessor with scalar accelerator

A significant speed-up may be expected from adding a dedicated functional unit to compute the absolute difference calculation followed by an accumulation in one cycle. This requires only a small amount of extra hardware. The additional unit will be referred to as the scalar accelerator.

Furthermore, additional instruction-level parallelism would be desirable. A second load path to the data memory will allow for simultaneous loads of a reference block pixel and a search block pixel, in parallel with the absolute difference calculation and accumulation. The second load path requires a second address generation unit.

Figure 5 shows the modified data path. Updating the nML description requires little time. The scalar accelerator unit is shown on the right hand side.

After running the Chess/Checkers tools for the new processor, the results of Table 1 (Design 2) are obtained. The total cycle count has reduced by as much as 84%, while the instruction count is almost unchanged and the gate count increases by only 10%.

Table 3 shows an excerpt of the instruction profile for this solution. Thanks to the software pipelining capabilities of the compiler, the body of the inner loop of the *SAD* function is now implemented in a single, highly parallel instruction, requiring only one cycle per pixel.

```

Profile information for ::base generated by Checkers 5.19.8 on Tue Aug 09 10:49:31 2005
Program being simulated: /ChessCheckers/designs/Target/mbase/design1/motion/motion

Cycle-count      :      87730
Instruction count :      68435

Function summary:

Cycles   % of total Instruction % of total Function      Relative cycle use in simulation
-----
 80538   91.80%      67032   97.95% sad_16x16      *****
  1559    1.78%      1393    2.04% motion_estimation
    15    0.02%         10    0.01% main_main

Function detail: sad_16x16

Low PC      :      14
High PC     :      33
Cycle-count :      80538 (91.80%)
Instruction-count:      67032 (97.95%)

PC      Instruction Assembly      Exe-count Cycles      Relative cycle use within function
-----
 14      3203 mvib r3,32                36      36
 15      3105 mvib r5,16              36      36
 16      3004 mvib r4,0                36      36
 17      2edd0020 do r5,32             36      72
 19      2504 mv r0,r4                36      36
 20      3105 mvib r5,16              576     576 *
 21      2edd001f do r5,31             576     1152 ***
 23      2e00 nop                      576     576 *
 24      4a95 lbu r5,dm(r2++)           9216    9216 *****
 25      4a56 lbu r6,dm(r1++)           9216    9216 *****
 26      0575 sub r5,r6,r5             9216    9216 *****
 27      0e2c lt r5,r4                 9216    9216 *****
 28      2f01 jcr 1                     9216    18432 *****
 29      2d01 jr 1                       3642    7284 *****
 30      0565 sub r5,r4,r5             5574    5574 *****
 31      0028 add r0,r5,r0             9216    9216 *****
 32      0059 add r1,r3,r1             576     576 *
 33      2eb8 rt                        36      72

Function detail: motion_estimation
...
    
```

Table 2 – Excerpt from instruction profile for motion estimation code on Base processor.

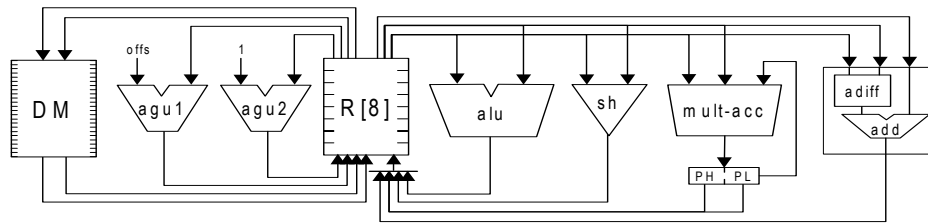


Figure 5 - Data path of Base extended with scalar accelerator.

```
Function detail: sad_16x16
Low PC      : 14
High PC     : 30
Cycle-count : 12492 (88.53%)
Instruction-count: 11808 (89.38%)
```

PC	Instruction Assembly	Exe-count	Cycles
14	2532 mv r3,r2	36	36
15	3205 mvib r5,32	36	36
16	30f6 mvib r6,15	36	36
17	3002 mvib r2,0	36	36
18	2541 mv r4,r1	36	36
19	2ee001c0010 doi 16,28	36	108
22	4b11 lbu r1,dm(r4++)	576	576
23	2ede001a do r6,26	576	1152
25	4ad0 lbu r0,dm(r3++)	576	576
26	9e00 aad r2,r0,r1   ld r0,dm(r3++)   ld r1,dm(r4++)	8640	8640
27	012c add r4,r5,r4	576	576
28	8081 aad r2,r0,r1	576	576
29	2502 mv r0,r2	36	36
30	2eb8 rt	36	72

Table 3 - Excerpt from instruction profile for motion estimation on Base with scalar accelerator.

If a further speed-up is required, an alternative architecture is needed that can process multiple pixels in a single cycle. This leads to the introduction of vector processing, discussed in Section 4.3.

### 4.3. Design 3: a vector ASIP

To speed up the execution of motion estimation beyond the performance of the scalar accelerator, an architecture is needed that can process multiple pixels in parallel. This is referred to as data-level parallelism. The Base microprocessor is therefore extended with a vector accelerator, using the concept of single-instruction multiple-data (SIMD) processing. The data path of the new ASIP is shown in Figure 6.

The ASIP supports vectors with of 16 elements of 1 byte each. This corresponds to one line of the reference and search blocks. The data memory can now load and store complete vectors at once. An extra register file is added, consisting of 4 vector registers.

The vector accelerator can compute the absolute differences for complete vectors in one cycle. The resulting values are produced as a vector. Also a separate vector sum instruction is provided, to add up all elements of a vector containing absolute differences. Together, both instructions can implement the complete inner loop of the C code of Figure 3.

Vector load and store instructions work on blocks of data aligned at 16 byte boundaries in memory.

However, a block of the search frame may not always be stored with this particular alignment. A vector alignment unit is therefore added that can extract a complete line of a block after two consecutive vectors have been loaded.

The Chess compiler supports vector or SIMD processing, provided that vector data types and functions are introduced in the C source code. Standard C operators such as +, \*, &, etc. can be overloaded on these vector data types. Also, intrinsic functions such as `vadiff()` (vector absolute difference) can be introduced. The C code of the *SAD* function is therefore rewritten as in Figure 7.

Table 1 (Design 3) shows the results obtained with the Chess/Checkers tools for this ASIP. An excerpt from the instruction profile is shown in Table 4.

As can be seen, a single vector instruction (`vadiff`) processes a complete row of pixels in one cycle, which effectively exploits the data-level parallelism offered by the ASIP. This reduces the cycle count by 60% compared to the previous architecture. However, except for address calculations in parallel with memory loads, no instruction-level parallelism is available in this vector ASIP.

The total gate count has doubled, as a result of the additional vector register file and vector functional units.

To further speed up the execution, one can look for opportunities to introduce additional instruction-level parallelism, i.e. to execute some of the instructions of Table 4 in parallel.

### 4.4. Designs 4 and 5: vector ASIPs with instruction-level parallelism

Two additional design iterations are made. Design 4 supports the execution of one vector load instruction (`ld`) in parallel with vector alignment (`valign`) and the vector arithmetic instructions (`vadiff`, `vsum`).

In Design 5, a second parallel vector load instruction is added. This requires separate memories for the reference and search blocks. The Chess compiler user directives to allocate a variable to a specific memory.

The results are listed in Table 1 (Design 4, Design 5). As can be seen, another 43% speedup is obtained, at virtually no extra hardware cost. Table 5 shows an excerpt from the eventual instruction profile, showing that the compiler is able to exploit both the data level and the instruction level parallelism.

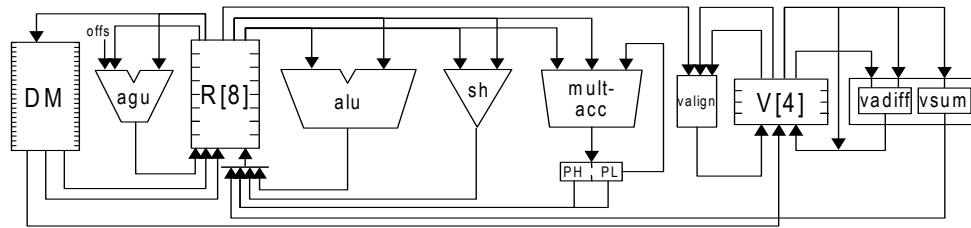


Figure 6 - Data path of vector ASIP.

```
static int sad_16x16(unsigned char* search,
                   unsigned char* ref)
{
    int sad = 0;
    vector* pvr = (vector*)ref;
    vector* pvs = (vector*)search;
    for (int i = 0; i < 16; i++) {
        vector vr = *pvr++;
        vector vs = valign(pvs+LINE_STEP/16,
                          pvs, *(pvs+1));
        vector adiff = vdiff(vr, vs);
        sad += vsum(adiff);
        pvs += LINE_STEP/16;
    }
    return sad;
}
```

Figure 7 - Vectorised C code of SAD function.

Function detail: sad_16x16				
...				
PC	Instruction Assembly	Exe-count	Cycles	
14	3204 mvib r4,32	36	36	
15	3000 mvib r0,0	36	36	
16	2ee000190010 doi 16,25	36	108	
19	c051 ld v0,dm(r1++)	576	576	
20	c065 ld v1,dm(r1+=r4)	576	576	
21	c091 valign v0,r1,v0,v1	576	576	
22	c056 ld v1,dm(r2++)	576	576	
23	c004 vdiff v1,v0,v1	576	576	
24	c02d vsum r3,v1	576	576	
25	0018 add r0,r3,r0	576	576	
26	2eb8 rt	36	72	

Table 4 - Excerpt from instruction profile for motion estimation on vector processing ASIP.

Function detail: sad_16x16				
...				
PC	Instruction Assembly	Exe-count	Cycles	
14	3204 mvib r4,32	36	36	
15	c045 ld v0,dm(r1++)	36	36	
16	2532 mv r3,r2	36	36	
17	c04b ld v1,dm(r1+=r4)	36	36	
18	c0c1 ld v3,rm(r3++)	36	36	
19	c085 valign v2,r1,v0,v1	36	36	
20	c045 ld v0,dm(r1++)	36	36	
21	c04b ld v1,dm(r1+=r4)	36	36	
22	c00e vdiff v3,v2,v3	36	36	
23	3006 mvib r6,0	36	36	
24	2ee0001c0010 doi 16,28	36	108	
27	b5b7 ld v0,dm(r1++)	576	576	
	ld v3,rm(r3++)			
	valign v2,r1,v0			
	vsum r5,v3			
28	80be ld v1,dm(r1+=r4)	576	576	
	vdiff v3,v2,v3			
	add r6,r5,r6			
29	2506 mv r0,r6	36	36	
30	2eb8 rt	36	72	

Table 5 - Excerpt from instruction profile for motion estimation on vector processing ASIP with dual load.

The motion estimation ASIP discussed in this paper is a single-processor solution, in which the scalar and vector data paths are controlled from a single instruction set. Alternatively, Chess/Checkers can also be used to design multi-processor architectures, with scalar and vector engines running as separate processors. An example implementation is described in [8].

### 5. Conclusions

A methodology has been presented for the design and programming of ASIPs, based on the Chess/Checkers retargetable tool suite. The application is described in C code. Using the nML processor description language and feedback from the retargetable compiler and instruction-set simulator, designers can perform true architectural exploration to determine the best possible ASIP for their application.

Previously the use of the Chess/Checkers tool suite has been demonstrated for application domains like audio, speech processing, and telecom modems [6] [7]. In this paper it is shown how the same tools can scale to the video and image processing domain. In this case it is important to find the best mix of instruction level and data level parallelism, for the application at hand. The Chess compiler is able to exploit these different forms of parallelism, so that the designer can easily determine the best mix.

As a demonstrator, an ASIP has been designed for motion estimation in video coding. The whole architectural exploration cycle has taken less than a person-week of effort. This includes the hardware implementation and verification.

### 6. References

- [1] R.M. Stallman, "Gnu Compiler Collection Internals", <http://gcc.gnu.org>, Dec. 2002.
- [2] J. Van Praet et al., "Processor modelling and code selection for retargetable compilation", ACM Tr. Design Automation of Electronic Systems, Vol. 6, no. 3, pp. 277-307, July 2001.
- [3] A. Fauth et al., "Describing instructions set processors using nML", Proc. European Design and Test Conf., pp. 503-507, March 1995.
- [4] "Tutorial: The nML processor description language", Target Compiler Technologies, <http://www.retarget.com/nml/>, June 2002.
- [5] J. Koga et al., "Motion Compensated Interframe Coding for Video Conferencing", Proc. Natl. Telecommunications Conf., pp. G5.3.1-5.3.3, 1981.
- [6] L. Dawance, "To a more powerful DSP based on TCT technology for ADSL+", DSP Valley Ann. Research and Technology Symp., Oct. 2003.
- [7] H. Roeven et al., "CoolFlux DSP - The embedded ultra low power C-programmable DSP core", Proc. GSPx 2004, Sept. 2004.
- [8] "Hands-on tutorial: Design of multi-core systems with SystemC and retargetable processor design tools", Target Compiler Technologies and Mentor Graphics, 42<sup>nd</sup> Design Automation Conf., Anaheim, July 2005.