Tradeoffs in the Design Space Exploration of Application-Specific Processors

N. Kavvadias and S. Nikolaidis
Section of Electronics and Computers, Department of Physics,
Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece
E-mail: nkavv@skiathos.physics.auth.gr

Abstract

An application-specific instruction set processor (ASIP) design methodology can exploit special characteristics of applications to meet the performance and time-to-market requirements. In this paper, tradeoffs encountered in the design of application-specific processors targeting embedded applications are discussed. The exploration of the architecture design space is a crucial step to effective instruction set generation as well as micro-architecture design. In this context, we identify operation pattern matching and hardware module allocation possibilities that should be accounted in hardware generation frameworks. For this reason, such modifications are applied on a pipelined processor model and their effect on execution performance and energy dissipation is measured via cycle-accurate simulations. As proof-of-concept, a real application, namely the full-search motion estimation algorithm used in video coding, is investigated.

1. Introduction

Embedded processors for consumer applications, present interesting architectural refinements, in order to support power-hungry algorithms e.g. for video compression and decompression [1]. Portability is a key issue for a large portion of the embedded market, which makes energy consumption a critical design concern. For successfully implementing domain-specific processors, the mapping of the application code on a suitable target machine is an important design consideration. In the development of such processors, closely matched design of instruction sets and micro-architectures is required, in respect to the application benchmarks, while being subject to several constraints.

Prior to developing a customized architecture, the potential of using an existing general-purpose processor is examined. Such a solution is unlikely to be viable, due to the fact that conventional RISC/DSP approaches pose limitations in tuning the architecture towards narrow application domains or they may be prohibitively expensive in respect to energy consumption. Thus, the embedded systems industry has shown an increasing interest in ASIPs, which are processors tailored to the needs of the targeted applications [1].

An important issue in ASIP design is the identification of the tradeoffs involved in instruction set and micro-architecture design, which requires efficient architecture design space exploration. This paper explores tradeoffs in micro-architecture design of application-specific processors under specified hardware constraints. The exploration space is restricted to single instruction issue and process width for an integer unit accounting for the power consumption limitations of embedded media applications. During design space exploration, frequent operation pattern matching for the generation of instructions as well as micro-architecture decisions have to be considered. The architectural model assumed is a four-stage pipelined machine with data stationary control, found in many ASIPs as the ARC [2] and Xtensa [3] configurable processors. The target application resides in the multimedia field motivated by the requirement of high performance. Hardware resources can be chosen from a parameterized model database [4], pre-characterized in terms of area, delay and power consumption. The impact of different architectural choices is evaluated by cycle-accurate simulation.

A contribution of this paper is the identification of beneficial architecture tradeoffs that affect instruction set and micro-architecture design. Examples of such decisions are: the incorporation of special storage units as split register files and the evaluation of candidate instructions as comprehended by analysis of the application code in basic operations. In our opinion, intermediate steps in automated design flows for instruction set and/or hardware generation [5] can be augmented with such information to improve the derived results.

The rest of this paper is organized as follows. Previous research efforts are discussed in Section 2. In Section 3 a simulation model for pipelined processors is analyzed. In Section 4 the proposed approach for ASIP design space exploration is discussed. Section 5 refers to different architectural configurations explored for a multimedia application, and evaluated for execution performance. Finally, Section 6 summarizes the paper.

2. Related work

Research efforts on architecture design space exploration focus on operation pattern matching under resource constraints. A mapping procedure for the primitive operations (hereafter termed as micro-operations or MOPs) is involved to produce special

single or multi-cycle instructions. Most of the produced instructions cannot be directly supported by the processor core and they are implemented on dedicated functional units. In [6] operations are extracted from a high-level description of the application to run on specialized hardware units. An identification algorithm is used that can also handle multiple-output instructions which demand increased level of parallelism. Similar issues are discussed in [7] where the computational capabilities of the processor are extended in the form of custom hardware and new instructions. The maximum number of MOPs that constitute a single instruction is not restricted. Most of the resulting instructions could only be implemented on customized accelerators. For operation patterns of moderate size this approach can exhibit lower than expected performance gains due to significant communication overheads between the main processor, or memory system and the dedicated units. In [8] the operation matching procedure results in architectures with a VLIW datapath. Some performance benefits are observed however wide datapaths are not usually used in power or energy critical embedded applications. Our approach avoids the introduction of specialized coprocessor units and attempts to introduce performance enhancements solely due to the main processor even in the case of multi-cycle instructions. Generally, multi-cycle candidates have not been extensively studied, however the use of such instructions as block loads and stores reduces the number of instruction fetches which tend to consume significant power in embedded systems.

Architecture design space exploration often regards decisions at a more fine-grain level. In [9] the effect of changing register file size on the performance, power and energy consumption is observed via instruction-level statistics. However, the exploration procedure is limited to varying the number of registers for a specific architecture (ARM7TDMI), which narrows the design space into homogeneous register file topologies.

3. Hardware architecture model for ASIP design

In our work, the architecture model assumed is a single-issue machine with a four-stage pipeline, with separate instruction and data buses (Harvard architecture), consistent to the RISC paradigm. The control model is data stationary and execution is performed in-order. The corresponding block diagram is shown in Fig. 1.

Functional and storage units are characterized by the specific type (register file, ALU, multiplier), number and additional attributes employed. Interconnect configurations can be defined within the simulation model by adding a few lines of high-level code. As attributes to a functional unit, the latency (determined by registering or bypassing inputs/outputs) and bandwidth (number of I/O ports) can be considered [10]. The number of read/write ports for register files and memories is also specified. Moreover, low-level parameters as the instruction and data word width are defined.

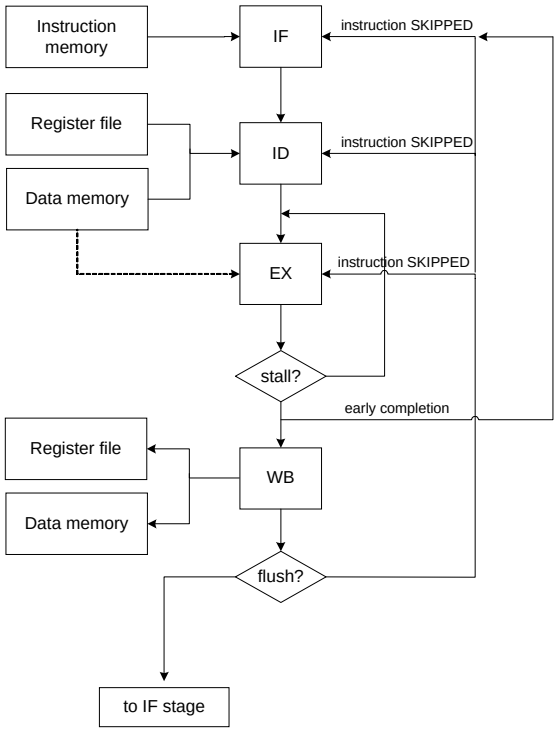

Figure 1. Architecture template for a 4-stage pipelined processor

As illustrated in Fig. 1, the pipelined operation is divided into instruction fetch (IF), instruction decode and operand fetch (ID), execute and memory read (EX), and write-back (WB) stages. The pipeline can be stalled at the EX stage in any of the following cases: to allow for data memory accesses, when operating on multi-cycle functional resources or when the execution criterion for a conditionally encoded instruction is not satisfied. If a non-sequential value is transferred to the Program Counter (flush), the subsequent instructions in the pipeline are marked as invalid (SKIPPED) and the corresponding work is aborted. An early completion path at the end of EX stage is provided for instructions that do not perform register write-back.

Moderate depth pipelines fit into the description of modern ASIPs as the ARC processor [2] that can be configured to suite an application domain. A simple yet modular design allows for the seamless integration of instruction set extensions, customization of functional units and rapid retargeting of software development tools to the derived architecture.

4. The followed approach for ASIP design space exploration

Initially, coarse-level requirements for the ASIP are extracted. The application, documented in a high-level language (ANSI C) is parsed to identify operators that can be directly mapped to hardware modules. Also,

application parameters as the proportion of overhead computations to the useful computations in the algorithm should be evaluated [11]. Overhead computations are inferred by data requirements and correspond to address calculations and data memory accesses. For this reason, the original application code is instrumented with performance counters and profiling is performed dynamically. The corresponding results can reveal the need for dedicated address generators. This would result in generating instructions with special addressing modes, which is not yet accounted.

In the following step of the design space exploration procedure, the application has to be analyzed in the form of MOPs. The LCC compiler [12] is retargeted to a new machine definition, in which a pre-selected set of basic operations is determined. Then, the C source code can be translated into a listing of generic instructions that contains the application description at the MOP level.

At this point, we recognize the basic blocks in the code. A basic block consists of MOPs situated between control-flow boundaries and is represented by a data flow graph (DFG). The collection of these DFGs accompanied by the processor model parameters as discussed in the previous section, provide an initial set-up for the target micro-architecture.

The application MOPs have to be scheduled in time steps, and instructions are generated with each time step corresponding to one instruction, taking into account resource allocation and timing constraints. Generally, micro-operations can be determined to execute in the same control step, which could extent to several cycles. If micro-operations are permitted to occupy more than one time steps, multi-cycle instructions can be produced.

Finally, the generated code for the target application is simulated on the modified architecture model. Candidate instruction sets and architectures are compared on the following metrics: static and dynamic instruction count, execution cycles, average power and total energy consumption. Instanced processor resources are annotated with power consumption figures for zero-valued input operands [4]. The contribution to power consumption of each functional and storage resource including pipeline registers is regarded except from the centralized control unit.

5. Case study: Full-search motion estimation algorithm

To evaluate the proposed approach, an application-specific processor has been designed for the full-search motion estimation (FSME) algorithm. Motion estimation algorithms are used in MPEG video compression systems [13,14] to remove the temporal redundancy in video sequences which is determined by the similarities amongst consecutive pictures. Instead of transmitting the whole picture, only the displacements of pixel blocks (motion vectors) between neighboring pictures (frames) and the difference values for these blocks have to be encoded.

The calculation of the motion vector is performed by means of the matching criterion, a cost function to be minimized [14]. Usually the Sum of Absolute Differences (SAD) computation is used. In equation (1), the SAD is calculated as:

$$SAD(i,j)=\sum_{k=0}^{M-1}\sum_{l=0}^{N-1}\left|C(x+k,y+l)-R(x+i+k,y+j+l)\right| \quad (1)$$

where C is the current picture, R the reference picture, (x,y) defines a block in the current picture, (i,j) are the coordinates for the corresponding block in the reference window, (k,l) the position of a pixel in the current block and M,N the block dimensions. Thus, a unique SAD value is calculated for each candidate block. If the new SAD value is smaller than the current minimum, a new minimum and motion vector (MVx, MVy) are declared.

In Figure 2, the pseudocode of the FSME algorithm is shown. It consists of three double nested *for* loops. The outer (x,y) loops address the current block and by iterating the (i,j) couple, a reference block is selected. For each position in the search region, the distance metric is computed for all (k,l) pixels in the current picture block.

The FSME algorithm parameters are: the size of the current and reference picture ($H\times W$), block size ($B\times B$) and p which determines the search region $[-p, p]$ around the location of the specific block in the reference picture.

```
for x=0 to H/B-1, step 1
 for y=0 to W/B-1, step 1
  min = 255*B*B;
  for i=-p to p, step 1
   for j=-p to p, step 1
    dist = 0;
    for k=0 to B-1, step 1
     for l=0 to B-1, step 1
       1: p1 = current[B*x+k,B*y+l];
       2: if  (p2 out of picture borders) then
            p2 = 0;
          else
            p2 = reference[B*x+i+k,B*y+j+l];
          endif;
       3: dist = dist + absolute_value(p1-p2);
     endfor;
    endfor;
    if (dist < min) then
      min = dist;
      MVx = i;
      MVy = j;
    endif;
   endfor;
  endfor;
 endfor;
endfor;
```

Figure 2. Pseudocode for the Full-Search Motion Estimation algorithm

As shown in Fig. 2, six index values are needed to address data structures accessed from the inner loop which comprises of four basic blocks, named *loop6*, *p2_case1*, *p2_case2* and *sad_calc*. The *loop6* basic block consists of two C-level operations: loading pixel p1 (operation 1), and evaluating a memory access check operation and loading p2 (operation 2). In Figure 3, the DFG for the MOPs of this basic block is given. Edges indicate data dependencies and all MOPs are control dependent to MOP36 (the final MOP).

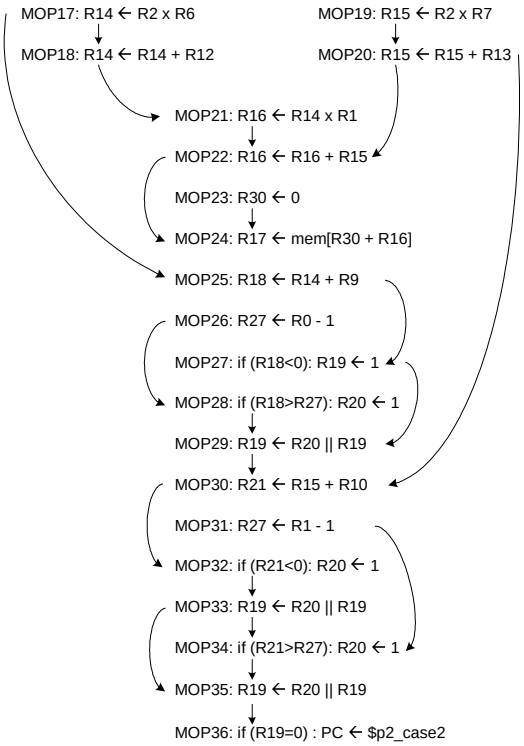

Figure 3. Data flow graph for an inner loop basic block (loop6) of the FSME algorithm

In the first step, we derive a baseline four-stage pipeline architecture for the ASIP. The instanced architectural resources are: single ALU for arithmetic/ logic operations and memory address calculation, a single-cycle array multiplier, a shifter with left-right shift capability, dedicated adder for PC update, a single multi-port (three read ports and one write port) register file, local instruction and data memory. It is assumed that the entire program code can be stored on-chip and no wait states should be inserted. The number and width of the instruction fields determine that the minimum instruction length is 28 bits and a 24-bit datapath is required for covering the data memory address space. Most instructions present a CPI of one, with the exception of load register from memory that requires two cycles at the EX stage.

The extracted instructions for the baseline configuration are shown in Table 1. In column "MOP description" the supported micro-operations are given. In column "Instruction definition" the instruction counterparts to whom these micro-operations are mapped, are provided.

Table 1. Baseline instruction set definition

MOP description	Instruction definition
Rd ← Rs1 + Rs2	ADD Rd, Rs1, Rs2
Rd ← Rs1 * Rs2	MUL Rd, Rs1, Rs2
Rd ← Rs1 ‖ Rs2	ORR Rd, Rs1, Rs2
flags: Rs1 - Rs2	CMP Rs1, Rs2
if (Rs1>Rs2) : Rd ← 1	SGT Rd, Rs1, Rs2
Rd ← m[Rb + Ro]	LD Rd, Rb, Ro
m[Rb + Ro] ← Rs	ST Rs, Rb, Ro
Rd ← Rs + imm	ADDI Rd, Rs, #imm
Rd ← Rs - imm	SUBI Rd, Rs, #imm
if (Rs<imm): Rd ← 1	SLTI Rd, Rs, #imm
Rd ← Rs >> imm	SHL Rd, Rs, #imm
Rd ← imm	MOVI Rd, #imm
Rd ← Rs	MOV Rd, Rs
Rd ← -Rs	MVN Rd, Rs
PC ← target	B $target
if (N=1): PC ← target	BLT $target
if (N=0): PC ← target	BGT $target
if (Rd=0): PC ← target	BRC Rd, $target
if (Rd=1): PC ← target	BRS Rd, $target

In the following subsections, modifications are applied on the initial specification of the ASIP (original) and 3 different configurations are produced which are denoted as versions 1, 2, 3. Version 3 is subdivided into the 3a and 3b alternatives. These configurations incorporate differences in the hardware allocation (1, 2) or result due to manipulation of the MOPs (3a, 3b). In version 1, additional register files are introduced to store the index values. Such modification cannot be easily extracted by studying the application code and has to be applied by the designer via heuristics. In version 2, interconnections are altered between the ALU and hardware multiplier to support a multiply-add instruction. In versions 3a and 3b the packing of MOPs produces alternative instructions, however a constraint on the maximum number of consecutive MOPs producing a single instruction is applied only in the first case. In version 3b, this constraint is removed to permit for multi-cycle instructions to be generated. A multi-cycle candidate is accepted if the aggregate number of cycles is reduced for an operation sequence that is utilized frequently during execution. Introducing instruction extensions increases the instruction decoder complexity. However, the processor cycle time is determined by the latency of the execute stage which remains the critical path in all versions of the ASIP.

5.1. Version 1: Effect of register file topology

For the basic architecture, a single register file is used. While a data word width of at least 24 bits is demanded for accessing the data memory modules, 12 storage bits are sufficient for handling the loop indices.

Thus, we can potentially benefit from a split register file topology with a general-purpose register file for storing arbitrary data values, and others of lesser bit-width to store the loop index values. In the baseline architecture, a single register file of 32 24-bit registers is instantiated, with 3 read and 1 write ports. For this architectural configuration, the general-purpose registers are reduced to 16 and 2 additional register files are used to store the running index and final values (the step value is always unary). The index register files are identical and consist of 8 12-bit registers with 2 read and 1 write ports.

A drawback of the heterogeneous register file topology is the introduction of an additional address decoder for the index register files. Also the incorporation of size-extension circuitry for the different data sizes is proven to have negligible effect on propagation delay.

Performance results for all modified architectures against the original are given in Table 4. These measures have been obtained when applying the FSME algorithm with the following parameter values: H=144, W=176, B=16 and p=7. Columns "% diff" indicate the percentage difference against the corresponding metric for the original version.

As can be seen, while the number of execution cycles remains the same, the decision on register file configuration drastically affects the energy consumption where a 35% reduction is observed.

5.2. Version 2: Effect of implementing the multiply-add operation

The resulting architecture from subsection 5.1 will be used as input to subsequent design space exploration steps. By introducing minimal changes to the interconnections of the ALU and multiplier, a new instruction, namely the multiply-add is supported. In terms of MOP description, the MUL and ADD micro-operations are executed in the same control step. Note that a frequent pattern consisting of the MUL and ADD micro-operations would have also been identified at the subsequent stage of MOP pattern matching.

The corresponding performance results are shown in Table 4. A reduction of 10% in the clock cycles and energy consumption are observed, compared to version 1 of the ASIP. However, the power consumption remains nearly the same since there have been no changes in hardware resource selection.

5.3. Version 3: Effect of frequent operation pattern matching

In the MOP description of the case study application, there exist interesting candidates for replacement by single instructions. By studying the MOP specification of the application the following operation patterns have been identified for version 3a: a MOVI-SHL couple is implemented by a move immediate and shift left (MVISL) instruction. A loop increment and conditional branch instruction, namely the single-cycle FOR, is inferred from an ADDI-CMP-BLT operation pattern. An absolute difference instruction (ABS) is also implemented. The number of MOPs permitted for a match is up to four to avoid generated instructions that would map onto hardware modules of high latency. A higher number of MOPs could be applied in case of multi-cycle instructions. All operation patterns and generated instructions are given in Table 2.

For calculating loop indices, a dedicated adder and additional multiplexing are required. In the FOR instruction definition, two register fields are saved, since only a single field is used for specifying the running index register, with the final index register address and step value implicitly encoded.

The incorporation of additional hardware has slightly increased average power consumption (7%), however the number of machine cycles is decreased to improve execution performance by 25-30% against version 2. Moreover, due to the reduced amount of hardware cycles, an energy reduction factor of 2.2 is estimated compared to the initial configuration for the ASIP.

Table 2. Operation pattern matching for version 3a of the ASIP

Operation pattern	Produced instruction
movi Ro, #imm	LDI Rd, Rb, #imm
ld Rd, Rb, Ro	
movi Rd, #imm	MVISL Rd, #shamt, #imm (Rd=Rs)
shl Rd, Rs, #shamt	
addi Rix, Rix, #1	FOR Rix, $loop_start
cmp Rix, Rfin	
blt $loop_start	
sub Rd, Rs1, Rs2	ABS Rd, Rs1, Rs2
cmp Rd, (Rtemp=0)	
bgt $abs_oper2	
mvn Rd, Rd	
abs_oper2: …	

In version 3b, a special multi-cycle instruction is generated in addition to version 3a instructions. Referring to the pseudocode in Fig. 2, it is observed that a memory access check operation (the conditional in operation 2) is required for the p2 pixel fetch. Usually, a memory controller/sizer unit undertakes this task in order to avoid an invalid state for the processor due to a data abort. In our approach, the equivalent to the memory control operation is the DMC (data memory check) instruction. A new flag (M_flag), is also introduced that holds the status for a valid memory access. A branch instruction (BMC) is used for conditional jump if the memory flag is clear. Specifically, the operation patterns that lead to introducing the DMC instruction are shown in Table 3.

The effect of the four-cycle DMC instruction is an even higher reduction of clock cycles by 25% against version 3a. A more profound impact on the dynamic

instruction count is also observed. This is due to the elimination of a significant percentage of instruction memory accesses.

To summarize our results, the final configuration for the ASIP presents a reduction of 40% on code size, about 50% on the dynamic instruction count and number of cycles, 20% on power consumption and nearly 65% on the energy consumption compared to the baseline architecture.

Table 3. Additional operation pattern matches for version 3b of the ASIP

Operation pattern	Produced instruction
SLTI Rtmp1, Rs1, #imm	DMC Rs1, Rs2, Rs3, Rs4
SGT Rtmp2, Rs1, Rs3	
ORR Rtmp1, Rtmp2, Rtmp1	
SLTI Rtmp2, Rs2, #imm	
ORR Rtmp1, Rtmp2, Rtmp1	
SGT Rtmp2, Rs2, Rs4	
ORR Rtmp1, Rtmp2, Rtmp1	

6. Conclusions

In this paper, we identified tradeoffs in micro-operation description analysis, and micro-architecture design space exploration in order to direct efficient instruction set generation for ASIPs. Particularly, issues which should be captured in design flows that target hardware generation from applications have been analyzed. For this reason, an architecture model is presented and then used as a template to explore the effect of different architectural modifications on execution performance and energy consumption behavior. To evaluate possible configurations, a cycle-based energy simulator has been developed. To illustrate our approach, tradeoffs inferred by different register file topologies and instruction candidates have been studied and compared to a baseline configuration for a case study application.

7. References

[1] M.F. Jacome and G. de Veciana, "Design Challenges for New Application-Specific Processors", *IEEE Design and Test of Computers*, 2000, Vol. 17, No. 2, pp. 40-50.

[2] ARC Cores, http://www.arccores.com

[3] Tensilica, http://www.tensilica.com

[4] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai, "PEAS-III: An ASIP design environment," *IEEE Int. Conf. on Computer Design*, 2000, pp. 430-436.

[5] A. Hoffmann et al., "A Novel Methodology for the Design of Application-Specific Instruction Set Processors (ASIPs) Using a Machine Description Language," *IEEE Trans. on Computer-Aided Design*, vol. 20, no. 11, Nov. 2001, pp. 1338-1354.

[6] A.K. Verma, K. Atasu, M. Vuletic, L. Pozzi and P. Ienne, "Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints", *Proc. of the 1st Workshop on Application Specific Processors*, Istanbul, Turkey, November 2002.

[7] N. Clark, W. Tang and S. Mahlke, "Automatically Generating Custom Instruction Set Extensions", *Proc. of the 1st Workshop on Application Specific Processors*, Istanbul, Turkey, November 2002.

[8] M. Arnold and H. Corporaal, "Designing Domain-Specific Processors", *Proc. of the 9th International Symposium on Hardware/Software Codesign* (CODES2001), Copenhagen, Denmark, April 2001.

[9] M.K. Jain, L. Wehmeyer, S. Steinke, P. Marwedel, and M. Balakrishnan, "Evaluating register file size in ASIP design", *Proc. of the 9th Int. Symp. on Hardware/Software Codesign*, Copenhagen Denmark, 2001, pp. 109-114.

[10] I.J. Huang, A.M. Despain, "Synthesis of application specific instruction sets", *in IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 1995, Vol. 14, No. 6, pp. 663-675.

[11] D. Talla and K.L. John, "Cost-effective hardware acceleration of multimedia applications," *IEEE Int. Conf. on Computer Design*, Austin, Texas, 2001, pp. 415-424.

[12] C.W. Fraser, D.R. Hanson, "A Retargetable Compiler for ANSI C," *SIGPLAN Notices 26*, Vol. 10., October 1991, pp. 29-43.

[13] International Organization of Standardization, Working Group on Coding of Moving Pictures and Audio, MPEG-4 Video Verification Model Version 18.0, Pisa, January 2001.

[14] Vasudev Bhaskaran and Konstantinos Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*, Second Edition, Kluwer Academic Publishers, Boston, 1999.

Table 4. Performance estimation for the examined architectural configurations

Architecture config.	Static instruction count	% diff	Dynamic instruction count	% diff	Cycles	% diff	Power (W)	% diff	Energy (J)	% diff
Original	79	0.00	233523086	0.00	246370022	0.00	2.087	0.00	6.016	0.00
version 1	81	2.53	235230088	-0.12	246370024	0.00	1.627	-22.02	3.930	-34.67
version 2	77	-2.53	213040691	-9.55	224179827	-9.01	1.631	-21.86	3.589	-40.43
version 3a	56	-29.11	155616986	-33.93	159406177	-35.30	1.752	-16.05	2.738	-54.49
version 3b	48	-39.24	109596592	-53.47	132406192	-46.26	1.652	-20.82	2.145	-64.35