A Low-Cost and Low-Power Multi-Standard Video Encoder

R. Peset Llopis, R. Sethuraman, C. Alba Pinto, H. Peters, S. Maul and M. Oosterhuis
Philips Research Laboratories
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
+31 40 2742422
[rafael.peset.llopis, ramanathan.sethuraman, carlos.alba.pinto, harm.peters, steffen.maul]@philips.com

ABSTRACT

Video encoders are an important IP block in mobile multimedia systems. In this paper, we describe a low-cost low-power multi-standard (MPEG4, JPEG, and H.263) video/image encoder. The low-cost and low-power aspects are achieved by the right choice of algorithms and architectures. In the algorithm front, an embedded compression technique for reducing the size of loop memory has enabled a single-chip low-cost realization of the encoder. In the architectural front, an efficient hardware-software partitioning has contributed to the design of a low-power encoder. Further, the hardware components that accelerate the kernels of encoding are implemented as application specific instruction-set processors (ASIPs) thereby providing flexibility to address multi-standard encoding. The power and area estimates for the encoder for QCIF@15fps in 0.18µm CMOS technology are 30mW and 20mm^2 respectively including the loop memory.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-based Systems – *real-time and embedded systems, signal processing systems.*

General Terms

Algorithms, Performance, Design, Reliability, Verification.

Keywords

Video Encoder, Multi-Standard, Low-Power, Low-Cost, ASIPs, Hardware/Software Partitioning.

1. INTRODUCTION

Video compression [1] plays a pivotal role in enabling video processing on mobile multimedia systems (e.g. mobile videophone). We, in this paper, present a low-cost, low-power multi-standard video encoder for mobile applications. The low-power and low-cost constraints on the encoder design are met by using innovative algorithms and efficient architectures. Further, the multi-standard aspect (MPEG4 SP@L3, JPEG, and H.263) demonstrates the flexibility of this encoder. Furthermore, this design is more efficient and flexible compared to other solutions like the ones offered by Hitachi [2] or Fujitsu [3].

Traditional implementations of encoders have always resulted in expensive multi-chip solutions due to the presence of a large loop memory [3,4]. The encoder presented in this paper, which is intended to support up to VGA@30fps (3.5Mbits of loop memory), achieves a single chip implementation by compressing the picture stored in the loop memory. Further, in order to support VGA the use of compressed loop memory is a must to achieve a single-chip solution.

The hardware-software partitioning of the encoder design has been along the traditional path of mapping compute-intensive tasks onto hardware (minimal flexibility) and control-intensive tasks onto software (maximal flexibility). The encoder uses an architecture template C-HEAP (CPU-controlled Heterogeneous Embedded Architectures for signal Processing [5]), wherein the processors with different levels of flexibility run concurrent tasks. The communication between these heterogeneous processors is through the use of C-HEAP communication protocol. The encoder uses ASIPs based on a very large instruction word (VLIW) template [12] to accelerate compute-intensive parts of video encoding.

The paper is organized as follows. Section 2 describes the algorithmic space of the video compression domain. In Section 3, the hardware-software partitioning employed in the encoder design is described apart from the design flow itself. The implementation figures are presented in Section 4. Finally, we conclude in Section 5.

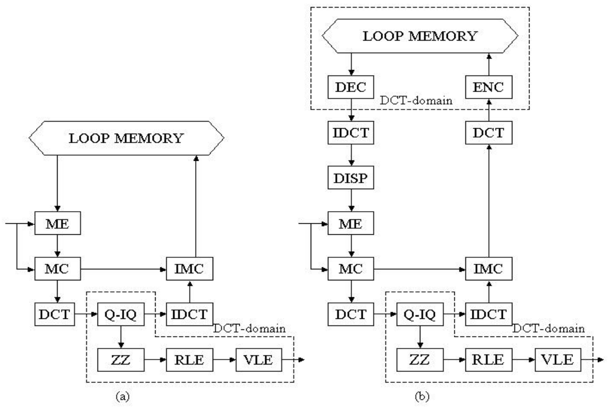

Figure 1. Video Encoders: (a) Conventional, (b) with Embedded Compression

2. ALGORITHM

Compression yields a compact representation of a signal by exploiting spatial and temporal correlation. Figure 1(a) shows a conventional encoder. It contains the following blocks: motion estimator (ME), motion compensator (MC), inverse motion compensator (IMC), (inverse) discrete cosine transform ((I)DCT), (inverse) quantizer ((I)Q), zigzag transform (ZZ), run length encoder (RLE), variable length encoder (VLE), and a loop memory.

The ME block determines the best match of the current macro-block (16 by 16 pixels) in the reconstructed image stored in the loop memory. The difference between the current macro-block and the best match in the reconstructed image is computed (MC), transformed (DCT), quantized (Q) and entropy encoded (ZZ, RLE, and VLE). The quantized data is inverse quantized (IQ), inverse transformed (IDCT), reconstructed (IMC) by adding the prediction, and stored in the loop memory to encode the next picture.

Implementations of this encoder have always resulted in two-chip solutions [3,4]. This is primarily due to large size of the loop memory (e.g. 3.5Mbits for a VGA). However, an efficient single-chip solution is possible if the picture stored in the loop memory can be compressed. This is achieved using the technique of embedded compression [7], wherein the loop memory is in the DCT domain and the DCT coefficients are stored using a scalable bit-plane based zonal coding approach. Figure 1(b) depicts a video encoder with embedded compression for the loop memory. The compression factor is fixed to 2.7 in order to avoid any quality degradation. Since the loop memory is organized in blocks (8x8 pixels), access to a macro-block from memory at a displaced position by the motion-estimator (ME) requires at most nine block-reads from the memory. This translates to few scalable decodes (DEC) and inverse-DCT (IDCT) operations. The displacement block (DISP) computes the displaced video block in spatial domain based on the blocks accessed from the loop memory. Note that without embedded compression, data read from e.g. SDRAM memories could be more than just the macro-blocks needed, this is because the addressing mechanism of the SDRAM which does not allow the access to single pixels but groups of them.

The motion estimator is responsible for determining the best match for the current macro-block with the picture stored in the loop memory. The motion estimator, based on the modified 3-dimensional recursive search (3DRS) algorithm [8], uses seven candidate vectors and performs full-, half- and quarter-pixel refinements. Since motion estimation is application dependent, flexibility is very important. This has been achieved by implementing the control parts in software and the compute intensive parts in hardware.

Loop filtering is employed to reduce noise content in the video data. This loop filter is based on a motion-compensated temporal filtering algorithm [4], and is implemented as part of the MC block.

The bit-rate control algorithm TMN8 [9] is used for setting the quantizer value in order to meet the bit-rate requirements. Since the bit-rate control algorithm is application dependent, it is completely done in software.

The above choice of algorithms, namely embedded compression to realize compress frame memory, 3DRS-based motion estimation and loop filtering, provide a low cost realization of video encoding.

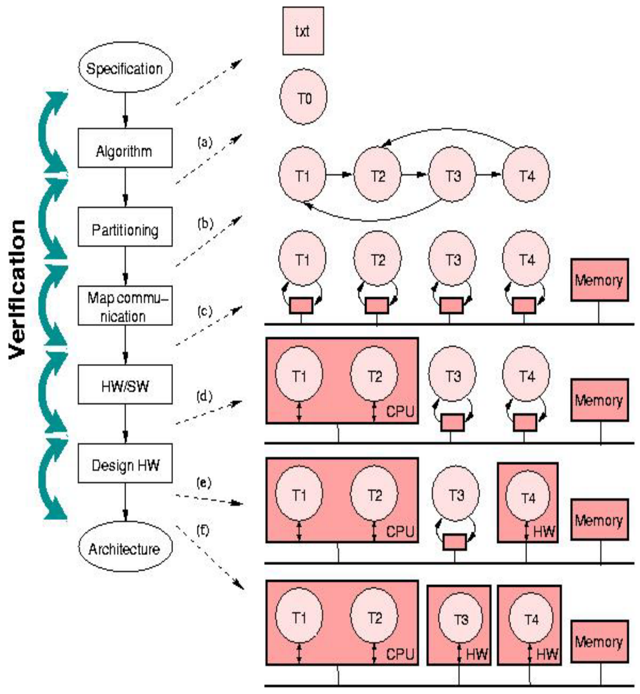

Figure 2. Levels of Design Abstraction

3. ARCHITECTURE

3.1 Design Flow

One of the important premises in the design of systems-on-a-chip (SoC) is to trade off efficiency for flexibility and time-to-market for cost. The design flow that abides by the premise above is to start from a high-level executable specification and converge towards a silicon implementation in several steps and iterations. One of the major tasks in this design flow is to ensure that hardware and software tasks communicate with each other correctly.

This can be accomplished by using a modular, flexible and scalable heterogeneous multi-processor architecture template based on distributed shared memory and an efficient and transparent protocol for communication. The protocol implementations have been incorporated in libraries that allow quick traversal of the various abstraction levels, hence enabling incremental design. Further, for ease of use, a well-defined set of primitives is required to hide the implementation of this protocol. The design decisions to be taken at each abstraction level are evaluated by either (co-) simulation or prototyping.

Following are the levels of abstraction used (Figure 2):

a) Algorithmic level (single-threaded C code), which is derived from the specification.

b) Partitioned algorithmic level (concurrent tasks).

c) Cycle-true communication level; bit- and cycle-true models for the communication and abstract functional models for the processing tasks.

d) Cycle-true embedded software level; bit- and cycle-true models for the communication, bit- and cycle-true functional models for software processing tasks, behavioral models for hardware processing tasks.

e) Partially implemented cycle-true hardware level; bit- and cycle-true models for the communication, bit- and cycle-true functional models for software and hardware processing tasks, combined with behavioral models for tasks still to be implemented in hardware.

f) Bit- and cycle-true level; bit- and cycle-true models for the communication, bit- and cycle-true functional models for hardware/software processing tasks.

The approach used for controlling the data-flow in heterogeneous multi-processor architectures is to have the processing devices synchronized autonomously [10]. This implies that each processing device should be able to initiate communication with any other device; hence the communication protocol must have a distributed implementation. With this approach, we can go to a smaller grain of synchronization, which allows smaller buffer sizes, and hence on-chip communication. For example, in a video context, we can synchronize on a block-line or block basis.

Our application specification is based on Kahn process Networks [6]. In this article, we refer to these processes as tasks. In this model, when a task wants to read from a channel and no data is available, the task will block. However, write actions are non-blocking. In our model, FIFOs are bounded in order to achieve an efficient implementation of the function. This means that a task will also block when it wants to write to a channel if the associated FIFO is full. In the remainder of this article we will refer to this model as a process network or task graph. The synchronization takes place on a per-token basis. While a token is the unit of synchronization, the amount of data associated with a token can vary. However, the size of a token is statically fixed per channel.

The communication protocol we describe involves the realization of the FIFO-based communication between the tasks, and does not refer to low-level protocols, e.g. bus protocols. Communication in our case is divided into 1) synchronization, and 2) data transportation. For efficiency reasons, all communication buffer memory is allocated at set-up and is reused during operation. Therefore, we need primitives to get and put buffer memory space. On the data producing side we want to get emptied token buffers and put full buffers. On the consuming side we want to get filled token buffers and put empty ones. Getting a token buffer is blocking, i.e. when no buffer is available the task blocks. Releasing (put) is non-blocking. The synchronization primitives are listed in Table 1.

Table 3.1. Synchronization Primitives

Primitive	Description
get_space	Claims empty token buffer (blocking)
put_data	Releases full token buffer (non-blocking)
get_data	Claims full token buffer (blocking)
put_space	Releases empty token buffer (non-blocking)

Two of the synchronization primitives, the "get_data" and the "get_space", return a pointer to the token on the input or output channel. Two additional primitives “load_data” and “store_data” are provided by C-HEAP to do the reading and writing from these pointers.

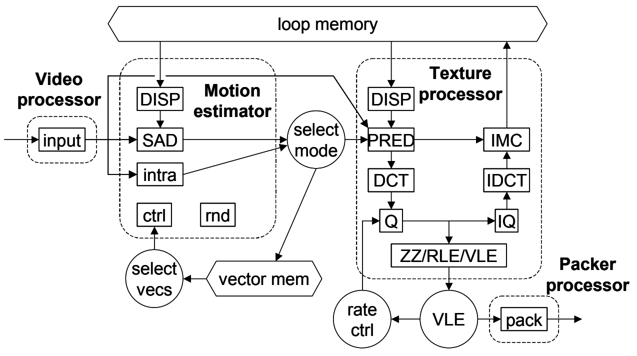

Figure 3. Hardware-Software Partitioning

3.2 Hardware-Software Co-Design

The first step consists of partitioning the video encoder application into hardware and software tasks. This step translates a single application task (task T0 in level ‘a’ of Figure 2) into multiple parallel tasks (T1 to T4 in level ‘b’). This step can be divided into the following actions:

- A behavioral description in the C-language for the video encoder was used as the starting point. In order to obtain an estimate of the computational load, the C-code was profiled on an ARM processor. This provided the required clock frequency for a software-only solution and a breakdown of the computational load for different functional modules of the encoder. The latter was used as a starting point for determining the hardware-software partitioning. The main strategy was to implement the compute-intensive parts in hardware, while keeping the encoding standard and control related parts in software. This is illustrated in Figure 3, wherein the small rectangles represent hardware blocks, whereas circles represent software.
- The VLIW-based ASIPs used for implementing the hardware parts have the benefit that they offer the flexibility needed for the application domain (e.g. video coding) while achieving performance, area, and power numbers close to that of hard-wired solution. The hardware parts were clustered into processors, thus resulting in four processors for the video encoder: video input processor, motion estimation processor, texture processor and packer processor. The objective of this clustering is twofold. First, to reduce the synchronization overhead between hardware and software. Second, to hide all local communication inside the processors from global bus activity. The latter is crucial for low power. Figure 3 depicts the four hardware processors by dashed blocks. The communication between the processors is based on the previously described C-HEAP protocol. A behavioral version of the protocol was implemented on top of a multithreading package [11].

The four hardware processors perform the functions given below. The video input processor is responsible for doing a stripe to macro-block conversion. The motion estimation processor evaluates candidate motion vectors, and does motion vector refinements. The texture processor is responsible for encoding a macro-block. The packer processor interleaves the Huffman codes of the coefficients with those of the headers, in order to generate a compressed output video stream.

The partitioning resulted in a single software task. This partitioning maintains the necessary flexibility. Consider the motion estimation task as an example. The candidate motion vector selection is done in software. The motion estimation processor then determines the best motion vector, does a refinement, and transmits the results back to software. Further, in software, the encoding mode is selected, and the quantizer value is computed. The quantizer value and the choice of encoding mode are passed to the texture processor, which then encodes the macro-block. Furthermore, since the bit-rate control algorithm is application dependent, it is completely done in software.

The second step consists of mapping the communication backbone onto hardware, which corresponds to translation from level 'b' to 'c' in Figure 2. This is done as follows:

- The input variables of the hardware processors consist of two parts: initialization variables (e.g. number of pixels in horizontal and vertical direction of a picture), and run variables (e.g. the coordinates of the current macro-block being encoded). In the behavioral simulations, pointers to all the variables are passed to the processors, which use these pointers to access the variables. Once the communication backbone is mapped onto hardware, the processors will have to read the initialization variables for each macro-block from a shared memory. However, it is much more efficient to store the initialization variables locally during initialization. Therefore, an initialization mode was added to each processor, during which the initialization variables are read and stored locally. Only the run variables, which vary across macro-blocks, are communicated. This reduces the bandwidth on the bus considerably.
- The event-based C simulation framework (TSS simulator – Philips Internal Tool) was used for system simulations, wherein (bit- and cycle-true) TSS models of the AMBA bus and the memory were used. UNIX processes modeled the software and hardware tasks. A C-HEAP library using UNIX sockets implements the synchronization and communication of these processes with TSS. These simulations were used to obtain bus utilization figures, and to optimize the communication structure.

The third step focuses on improving the performance of the implementation. This can be seen as moving from level 'c' to 'd' in Figure 2. The result of the previous step consists of a system without concurrency. A software task starts a hardware task and awaits the completion of the hardware task (and vice versa), thus using the resources inefficiently. Concurrency can be implemented by pipelining the software and hardware tasks. However, some dependencies between software and hardware have to be broken. Since the breaking of the dependencies leads to a different behavior, extensive simulations were carried out in order to verify that the compression ratio and SNR are comparable to that of the original reference C description. Also the synchronization overhead has been optimized, by choosing the synchronization granularity as large as possible (given the flexibility constraints). This has resulted in two hardware processors (motion estimation and texture) synchronizing at macro-block granularity, and two other hardware processors (video input and packer) synchronizing at stripe granularity. Again the TSS simulator was used for the system simulations. TSS models for the ARM, the AMBA bus and the memory were used. UNIX processes, as in the previous step, modeled the hardware tasks. These simulations showed that the required number of clock cycles for the software tasks was smaller than the available cycle budget.

Figure 4 shows the pipelining of hardware and software tasks. The video input processor is one stripe ahead of motion estimation and texture processors. The motion estimation processor is two macro-blocks ahead of the texture processor, as required by the OBMC option of H.263. The header generation of a macro-block (VLE SW) is done one macro-block later than the encoding of the same macro-block. Last but not the least, the packer processor is one stripe delayed with respect to the texture processor. As stated before, only a single software task is required. Therefore, the scheduling of the hardware and software tasks is fixed (known at compile time).

For more details on the hardware-software co-design methodology, the reader is referred to [10].

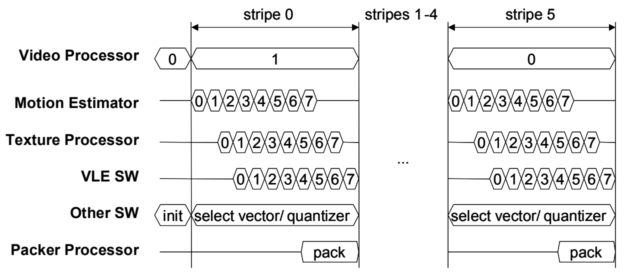

Figure 4. Pipelining of HW/SW Tasks

3.3 Hardware Processor Design

All hardware tasks are implemented as very-long- instruction-word (VLIW) processors. A|RT Designer [12] enables design of these VLIW processors. All application specific functional units (ASUs) used in the VLIW processors are designed using A|RT Builder [12], which translates a C-based functional specification of an algorithm into an RTL VHDL description. ASUs are used for performing dedicated compute-intensive kernels and hence are crucial for an efficient implementation of HW tasks. The choice and flexibility of those ASUs enable multi-standard encoder realization.

The following sections describe the design of hardware processors apart from the ASUs used in these designs (moving from level 'd' to 'e' and 'f' in Figure 2). The motion estimator and the texture processor have been implemented in two different configurations, namely, with and without embedded compression (EC).

3.3.1 Video Processor

The video processor is used for re-organizing the pixel format. This processor uses standard resources apart from an ASU for re-organizing pixels more efficiently.

The input format of pixels (namely, YYUYYV) is converted into four luminance and two chrominance 8x8 blocks in order to generate a macro-block output.

3.3.2 Motion Estimator

In the motion estimation hardware task, the current macro-block is motion estimated for all the candidate motion vectors in order to determine the best matching motion vector. Further, optionally, this best vector can be refined to half/quarter pixel accuracy. The 3DRS motion estimation uses sum-of-absolute-differences (SAD) as the matching criterion. The architecture of the motion estimator is shown in Figure 5. The granularity of the motion estimator processing is a macro-block.

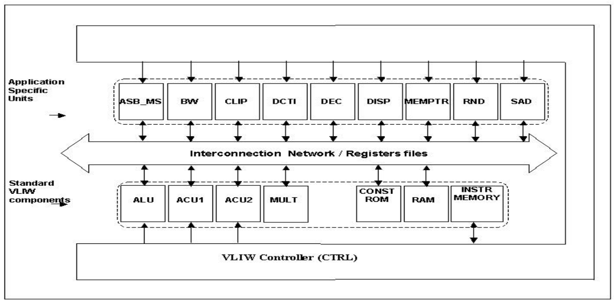

Figure 5. Motion Estimator Architecture

Table 3.2. Application Specific Units of ME

ASU	Purpose
BW	Byte/word conversions
CLIP	Clip engine (pixels, coordinates, vectors)
DCTI	(Inverse) discrete cosine transform (EC)
DEC	Embedded decompression (EC)
DISP	Displacer (cache/interpolator)
MEMPTR	Memory pointer address calculation
RND	Random number generator
SAD	Sum of absolute difference engine

```
For (every macro-block)
•   Fetch a block from current frame
•   Determine actions to be performed
•   For every action:
        → Initialization
        → For every vector:
              ▪ Initialization
              ▪ For every component:
                    • Displace DCT block
                    • Calculate SAD
              ▪ Update results
        → Update and write results
```

Figure 6. Pseudo Code for Motion Estimator

The VLIW architecture of the motion estimator consists of standard resources like arithmetic-logic-units (ALUs), address computation units (ACUs), multipliers, constant-ROM, RAM, instruction memory, VLIW controller, apart from ASUs (refer to Table 2).

The first step in the motion estimator process is to determine the flow of actions, by analyzing software-controlled options. Subsequently all actions are executed as depicted in the pseudo code shown in Figure 6. The inner loop iterates over all luminance components of the macro-block. Within this loop the candidate vector block is displaced, delivered to SAD engine and SAD calculation is done. The middle loop iterates over the number of candidate vectors for this particular action.

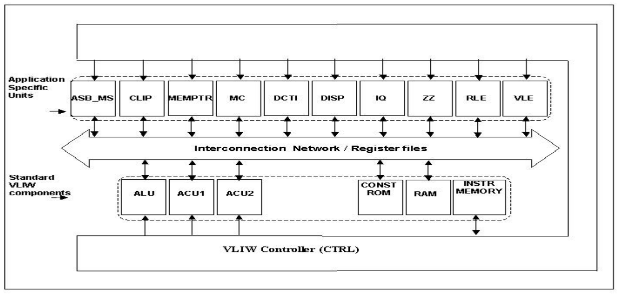

Figure 7. Texture Processor Architecture

Table 3.3. Application Specific Units of TC

ASU	Purpose
BW	Byte/word conversions
CLIP	Clip engine (pixels, coordinates, vectors)
MEMPTR	Memory pointer address calculation
MC	Motion compensator (predictor)
DCTI	(Inverse) discrete cosine transform (EC)
DISP	Displacer (cache/interpolator)
IQ	(Inverse) quantizor
ZZ	Zigzag transform
RLE	Run length encoder
VLE	Variable length encoder (Huffman table)

```
For (all blocks in macro-block)
•   Fetch a block from current frame
•   Fetch a block from the previous reconstructed frame (if not
    available in the cache)
        → Perform displacement to get sub-pixel accurate
          block
•   Motion compensate current block from the block fetched from
    the previous reconstructed frame
        → Perform motion compensated loop filtering
        → Perform overlapped block motion compensation
          (OBMC)
•   Perform DCT and Quantization on the motion compensated
    block
•   Perform IQ/IDCT/IMC and store the reconstructed block in the
    loop memory (loop memory holds the reconstructed frame)
•   Perform loss less coding (ZZ/RLE/VLE) and store the Huffman
    encoded coefficients
```

Figure 8. Pseudo Code for Texture Processor

3.3.3 Texture Processor

The tasks of the texture processor are to calculate the difference between the current macro-block and the best match in the reconstructed image (prediction), transform, quantize, and entropy encode. The quantized data is inverse quantized, inverse transformed, reconstructed by adding the prediction, and stored in the loop memory to encode the next picture. The texture processor also has macro-block granularity. The architecture of the texture processor is shown in Figure 7. Again, the architecture contains standard resources apart from ASUs. An overview of ASUs is shown in Table 3 and the pseudo code is provided in Figure 8.

3.3.4 Packer Processor

The packer processor combines the Huffman coded video stream together with header information in order to generate compressed video stream of a given format. The packer processor uses one ASU for packing bit streams apart from standard resources. The ASU packs the Huffman codes into multiples of 32-bit words.

4. RESULTS

The C-based design flow and VLIW-based ASIPs used in the encoder realization enable faster design time (36 man-months) while addressing cost and flexibility. Table 4 presents the area and power numbers of a video encoder implementation described in this work with and without embedded compression. The implementation supports up to CIF@30fps. The motion estimator, the texture, and the packer processors where synthesized as hardware components for typical case in 0.18µm CMOS technology. The functionality of the video processor was implemented in software run by the CPU. The multi-standard video encoder was verified (bit-true and cycle-true) until the netlist.

Table 4.1. Area/Power Numbers of Encoder

Instance	Area (mm^2)	Power (mW)	Area with EC (mm^2)	Power with EC (mW)
ARM7TDMI-S	0.62	3.93	0.62	5.15
CPU memory	3.25	1.56	3.25	1.88
Loop memory	off-chip	off-chip	4.39	0.77
Motion estimator	1.62	6.03	2.74	8.70
Texture codec	4.24	4.30	5.39	6.21
Packer processor	0.50	0.08	0.50	0.08
Shells & busses	3.41 (*)	5.50 (*)	2.98	7.55
Total	13.64 (*)	21.40 (*)	19.86	30.34

(*) Off-chip loop memory data are not included

The CPU processor is an ARM7TDMI-S built in 0.18µm CMOS technology, with 0.39mW/MHz of power consumption. The frequency of the CPU clock is scaled down to just meet the performance requirements of the application. The memory instance includes only the CPU memory in the case without compression and the CPU- and loop memories in the case with embedded compression. While the loop memory has been embedded on-chip in the second case, the size of the motion estimation processor and texture processor have been increased due to the extra functional units needed for compression. The shells around the processors and busses support the communication among the components. Table 4 presents also the power dissipation numbers that are obtained under a typical simulation scenario (25ºC, 1.8V) for QCIF@15fps. QCIF@15fps has been chosen to facilitate benchmarking with existing solutions. The hardware processors run at 100MHz and are clock-gated when not in use.

Table 4.2. Benchmarking

Benchmark	Standard	Area (mm^2)	Power (mW)
Hitachi [2]	MPEG4	43	170
Our design	MPEG4, H.263, JPEG	19.9	30.3
Toshiba [13]	MPEG4	86	40
Fujitsu [3]	MPEG4	28	9

Table 5 shows the benchmarking of our design with three different encoder solutions [2,3,13]. The implementation in [2] is a complete SW solution using a fused RISC/DSP CPU and hence has higher area and power numbers. The implementation in [3] is an ASIC solution with off-chip SDRAM for loop memory, and the power indicated in the table does not consider the off-chip memory consumption. Compared to the ASIC solution, our design is more flexible and multi-standard. The implementation of [13] addresses flexibility similarly to our design while consuming more power for the same functionality.

5. CONCLUSIONS

In this work we presented the design of a low-cost, low-power and multi-standard video encoder. The right choice of algorithms and architectures had enabled an efficient solution for the video encoder. The use of embedded compression technique for reducing the size of the loop memory had enabled a single-chip low-cost solution. Further, the choice of HW/SW partitioning had contributed to the design of a low-power encoder. The power and area estimates for the encoder in 0.18µm CMOS technology are 30mW (for QCIF@15fps) and 20mm^2 respectively including the loop memory, and it is more competitive compared to state-of-the-art designs.

6. REFERENCES

[1] V. Bhaskaran, and K. Konstantinides. Image and Video Compression Standards. Algorithms and Architectures. 2nd Edition. Kluwer Academic Publishers. 1997.

[2] T. Yamada. A 133MHz, 170mW, 10µA Standby Application Processor for 3G Cellular Phones. In: *IEEE Solid-State Circuits Conference*. 2002.

[3] H. Nakayama. An MPEG-4 Video LSI with a 9mW Error-Resilient Codec based on a Fast Motion Estimation Algorithm. In: *IEEE Solid-State Circuits Conference*. 2002.

[4] A. van der Werf, et al. I.McIC: A Single-Chip MPEG-2 Video Encoder for Storage. In: *IEEE Journal of Solid-State Circuits*, vol. 32, no. 11. Nov. 1997.

[5] A. Nieuwland, et al. C-HEAP: A Heterogeneous Multi-Processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems. In: *Kluwer Design Automation of Embedded Systems*, Oct. 2002.

[6] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In: *Information Processing*. J. Rosenfeld, Ed. North-Holland Publishing Co. 1974.

[7] R. Kleihorst, et al. DCT-Domain Embedded Memory Compression for Hybrid Video Coders. In: *Journal of VLSI Signal Processing Systems*, vol. 24, pp. 31-41, 2000.

[8] G. de Haan, et al. True-Motion Estimation with 3-D Recursive Search Block Matching. In: *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 3, no. 5. Oct. 1993.

[9] ITU-Telecommunication Standardization Sector. H.263 Version 2 (H.263+). Video Coding for Low Bit-Rate Communication. 1998.

[10] A. Nieuwland and P. Lippens. A Heterogeneous HW/SW Architecture for Hand-held Multi-media Terminals. In: *IEEE Workshop on Signal Processing Systems*. 1998.

[11] PAMELA. A Performance Modeling Language, ce-serv.et.tudelft.nl

[12] Adelante Technologies. A|RT Builder and A|RT Designer Manuals. www.adelante.com

[13] H. Arakida, et al. A 160mW, 80nA Standby, MPEG-4 Audiovisual LSI with 16Mb Embedded DRAM and a 5GOPS Adaptive Post Filter. In: *IEEE Solid-State Circuits Conference*. 2003.