FLEXBLE PROCESSOR ARCHITECTURE OPTIMIZED FOR ADVANCED CODING ALGORITHMS

Adam ŁUCZAK and Olgierd STANKIEWICZ

Chair of Multimedia Telecommunications and Microelectronics
Poznan University of Technology
Polanka 3, 60-965 Poznan, Poland
Email: [aluczak,ostank]@multimedia.edu.pl

ABSTRACT

Many modern multimedia applications exploiting complex coding standards like AVC/H.264 or AAC-HE, require processing that implies synergy of both software and hardware solutions for real-time operation. This paper presents a new architecture of processor designed to achieve efficiency and convenience in such cases. The paper proposes an efficient processing architecture based on the use of fast and lightweight "hardware function calls". This approach reduces hardware-to-software bottlenecks, allowing a smooth transition from software to hardware solution. The presented concepts were implemented on Xilinx Virtex4-SX35 FPGA. It is worth noticing that the core is small enough to be treated as a robust soft-core. In this paper, specificities on the developed instruction set are provided as well as on the module inter-connection. Results of tests conducted using standard implementations of H.264/AVC and AAC algorithms are also presented.

Index Terms— processor architecture, CISC, hardware function calls, AVC, H.264

1. INTRODUCTION

Diversity of modern multimedia leads to a fact that there is no single universal solution that would outperform other competitive solutions in all fields of application. Advanced coding standards like AVC/H.264 [1] require strict cooperation of control and processing modules as well as efficiency in irregular algorithms, while less advanced standards like JPEG coding, cope with extraordinary resolutions up to 3880x2592 (e.g. cam CCD resolution 10Mpix), which results in high processing bandwidth needs. A standard like AAC (Advanced Audio Coding) [2] requires specialized capabilities like floating-point unit.

There are many existing solutions for multimedia applications, referring both to software and hardware. All of these have some drawbacks [3,9]. Some of those solutions are presented in Table 1.

Use of ASIC (Application Specific Integrated Circuit) is reasonable only in case of mass production of already closed design.

Table 1. Comparison of some real-time multimedia solutions

SOLUTION	POWER CONSUMPTION	DEVELOPMENT TIME	DESIGN EXTENSIBILITY	PERFORMANCE	MARKET PRICE
PC	HIGH	LOW	GOOD	HIGH	HIGH
RISC	LOW	LOW	GOOD	LOW	LOW
DSP	HIGH	MEDIUM	MODERATE	HIGH	MEDIUM
ASIC	LOW	HIGH	POOR	HIGH	LOW
RISC with dedicated modules	LOW	MEDIUM	MODERATE	MEDIUM	MEDIUM
CISC with hardware function calls	LOW	MEDIUM	MODERATE	HIGH	MEDIUM

RISC microcontroller connected with dedicated modules would be a good choice, but unfortunately suffers from communication bottlenecks and compatibility matters. Existent standards for module communication, such as PowerPC "Device Control Register Bus" or "OnChip Peripheral Bus" bring several deficiencies as the number of connected modules grows. That has proved to be inefficient regarding scalability, mostly due to large inter-module transaction stalls.

2. ARCHITECTURE

The paper proposes an alternative solution which tries to take the best of existing ones. Three major improvements are introduced: efficient and **scalable** interface between the processor and surrounding modules, based on chain topology with Device Endpoints (DEP) as chain elements, CISC instruction set optimizations, and finally the so called "hardware function calls" which take advantage of CISC architecture and allow faster communication and grant lower command latencies. Such a latency gain from instruction set is desirable, because it balances the latency losses coming from DEP chain length.

Reduced instruction sets (in RISCs) cannot efficiently fit "hardware function calls" and thus use of CISC architecture is beneficial. As the available CISC architectures are obsolete and are not competitive in case of performance, proposal of some novelties was crucial. It turned out to be the slightest way to achieve coherency of assumed concepts in single project.

2.1. Processor

The processor core exploits well known pipeline architecture, yet some original enhancements have been introduced. Four major stages in the pipe can be distinguished:

a) Instruction pre-fetch – stage of instruction fetching with *conditional flags* checking (see 2.2.4 for details). If flag is set (is not locked for writing) the instruction is passed by, otherwise instruction is skipped. Such mechanism can eliminate conditional instructions at the entry of processor pipe.
b) Register access – at this stage register data and conditional flag (from second set) are read. Further a control word for next stages is generated. Only data validated by conditional flag (registers, flags and control) is passed by. If any of registers is locked for writing, the whole pipe up to this stage is held up.
c) Memory access – at this stage all the registers data and constants enclosed in instruction word are combined in order to obtain memory pointer or are passed to next stage. In case of memory access the data from local memory is read in two clock cycles and appear at the output synchronously to the rest of data and control. It means that all flags and registers data which are not used at this stage are delayed for two clock cycles.
d) Execution units – consist of ALU and SMU units, and perform all standard arithmetic and logic operations. Additionally device chain entry point is connected in parallel to computational units. It is important because it means that devices in chain are seen as local units of processor. Therefore the same arguments like for ALU and SMU (Rx, [Rx], constant, etc.) can be passed to the chain and devices. Each device has assigned dedicated instructions in processors *OPCODE* space (Table 2).

PC - program counter,
REG FILE - register file (64 registers),
FLG FILE - conditional flags file,
AGU - address generator unit,
LSU - load/store unit,
REG STACK - dedicated stack for registers,
ARG STACK - dedicated stack for function arguments
ALU - arithmetic/logic unit,
SMU - shift/multiply unit,
DEP - device endpoint,

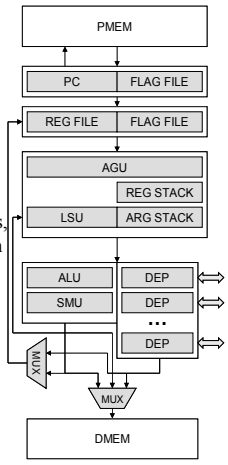

Fig.1. Processor core block diagram

Devices connected to the chain realize hardware functions (HF). Hardware functions are designed by the user and can help to accelerate some of the most time consuming tasks. Moreover, because arguments are passed to hardware functions in that same way as the software functions (by *PUSH* instruction), testing of HF is very easy and efficient. Also switching between software and hardware version of function does not introduce any need for changes in program code.

Proposed architecture is very flexible, new instructions and new hardware functions (devices) can be easy included. Number of devices is limited to 32, and 5 bits operation code for each device is provided. It means that device can perform 32 hardware functions.

2.2. Processor instruction set

The proposed processor is a CISC class (Complex Instruction Set Computer). The reason behind it is that performance gain from proposed "hardware function calls" depends on flexibility of argument passing mechanism [7]. Comparing to competitive RISC architecture, statistical number of cycles required for whole "call" operation is smaller. The very specific problem with CISC architecture is instruction decoding process, which depends on syntax of instructions. Irregular syntax leads to complex decoding logic and to overcome that effect, we propose 64-bits long instructions that provide room for redundancy. This allows for efficient fetching stage, as in case of Digital Signal Processors.

The whole instruction set is divided into four classes, basing on *OPCODE* residing in highest 8-bits of 64-bit instruction word: Arithmetic/logical, Branch, Flag Manipulation and Hardware Function Calls. General form of instruction is presented in Table 2.

Table 2. General form of instruction word.

8 bits	6 bits	8 bits	6 b.	24 bits	6 bits	6 b.
OPCODE	CONDITION	MODE	RA	OFFSETS/ CONSTANTS	RB/ CONST-EXT	RC

- **OPCODE (8 bits)** – this field identifies instruction as belonging to one of the instruction classes mentioned below
- **CONDITION (6 bits)** – field that controls conditional execution of the instruction, basing on conditional registers
- **MODE (8 bits)** – field that controls argument extraction, size and signed/unsigned extension
- **RA (6 bits)** – register index of argument A
- **OFFSETS/CONSTANTS (24 bits)** – set of address offsets or constants depending on MODE field
- **RB/CONST-EXT (6 bits)** – register index of argument B or constant extension, depending on MODE field
- **RC (6 bits)** – register index of output register C

2.2.1. Arithmetic/logical instruction class

These instructions perform arithmetic and logical instructions of general form C = A operation B. Possible operations are gathered in table 3.

Depending on MODE field, there are many possible configurations of how A, B and OFFSET/CONSTANTS fields are interpreted by the processor. In particular, it is possible to embed complex memory read/write operation into single instruction (Table 4).

For indexing arrays of data, it is possible to use Scale-Index-Base concept known from PC x86 architecture. Effective address consists of a linear combination of form:

BaseReg + IndexReg*Scale + Offset

Only *Scale* values that are consist ent with memory operand size existing in MODE field are allowed.
Such a instruction allows to execute most of high-level language addressing constructs in single operation.

Additionally, some *OPCODEs* allow concatenation of fields to enlarge the range of constant argument - in such case 24 bits of OFFSETS/CONSTANTS field, 6 bits of RB/CONST-EXT field, and 2 bits of MODE field are concatenated to give single 32bit CONSTANT argument.
All arithmetic/logical instructions can execute conditionally, basing on simple logical test performed on single specified flag register.

Table 3. Summary of arithmetic/logical instructions

Mnemonic name	Description of operation
OR	Logical sum
AND	Logical multiplication
XOR	Logical exclusive sum
SELA	Select argument A
SELB	Select argument B
ADD	Arithmetic sum
SUB	Arithmetic subtraction
ADDC	Arithmetic sum with carry
SUBB	Arithmetic subtraction with borrow
MIN	Minimum of A and B
MAX	Maximum of A and B
ABSA	Absolute value of A
NEGA	Negative value to A
ROR	Rotate right
RORC	Rotate right with carry
SHR	Unsigned shift right
SHRS	Signed shift right
ROL	Rotate left
ROLC	Rotate left with carry
SHL	Shift left
SHLS	Shift left, with least significant bit replication
MUL	Arithmetic multiplication
PUSH	Pushes argument A on one of stacks
POP	Pops value from one of stacks

Table 4. Possible argument configurations

Argument configuration
RC = RA @ C32
RC = [RA] @ C32
[RC] = RA @ C32
[RC] = [RA] @ C32
RC = [RA+O12] @ C18
[RC] = [RA+O12] @ C18
[RC+O12] = RA @ C18
[RC+O12] = [RA] @ C18
[RC+O12] = [RA+O12] @ C18
RC = RA @ RB
RC = [RA] @ RB
[RC] = RA @ RB
[RC] = [RA] @ RB
RC = [RA+O24] @ RB
[RC] = [RA+O24] @ RB
[RC+O24] = RA @ RB
[RC+O24] = [RA] @ RB
[RC+OC12] = [RA+OA12] @ RB
RC = RA+RB<<SS
RC = [RA+O24+RB<<SS]
[RC] = [RA+O24+RB<<SS]
[RC+O24+RB<<SS] = RA
[RC+O24+RB<<SS] = [RA]
[RC+OC12+RB<<SS] = [RA+OA12+RB<<SS]

- SS – size selector
- [ ] - memory access operator
- @ - instruction operation
- C18 – 18 bit constant
- C32 – 32 bit constant
- OA12, OC12–separate 12 bit offsets
- O12 – single 12 bit offset

2.2.2. Flow control instruction class
These instructions control execution of the program and include: branches, software function calls, and program returns. Flow control instructions can execute conditionally, basing on logical test performed on single specified flag register or hardware flag.

Unconditional branches execute transparently and at zero-cost because of small instruction cache between Program Counter module and entry of processor pipeline.

2.2.3. Hardware function calls
Hardware function call is a set of instructions dedicated for servicing communication with light-weight external modules. These modules are connected to DEP chain which is connected directly to the execution unit. There are three instructions in that class:

- PUSH(argument)
- Result = HF_CALL(argument)
- Result = POP()

Because arguments to DEP are passed via FIFO, it is possible to call functions with arbitrary number of arguments and return arbitrary number of results. In case of more than one result value, device designates special POP *OPCODE* which returns additional results.

Because HF_CALL instruction passes single argument implicitly, it is possible to execute sequence of single-argument calls without losses related to PUSH overhead.

2.2.4. Flag manipulation instruction class
These instructions perform operations on flag register bank and are crucial for efficient use of conditional execution available in remaining instruction classes.

Flag manipulation instructions set single output flag register basing on one of following conditional expressions:

- (!FA) &| (!FB)
- (RA?CA) &| (RB?CB)
- (!RA) &| (!RB)
- (RA ? RB)

where:
&| - logical OR or AND
! - optional negation
? - arithmetic comparison or logical constant

In particular, output flag register can be:

- set to resultant conditional expression
- set to 1 if resultant conditional expression is true
- set to 0 if resultant conditional expression is false

This mechanism allows efficient execution of majority of conditional constructs generated by high-level language compilers.

2.4. Device endpoint (DEP) & Memory Bus (MBUS)

Also new device interface has been proposed. Existing solutions like IBM PowerPC OPB (On-Chip Peripheral Bus) exploits fast buses for processor-to-device or device-to-device interconnection. Unfortunately communication protocol like *handshake* requires feedback from external device. Such acknowledgement feedback introduces transaction latency and decreases bus throughput [6].

Therefore the chain topology has been introduced. In that solution required feedback from device is eliminated, but time of response is increased and is equal N * clock cycles, where N is a number of devices in chain (chain length). Proposed chain topology has been used for processor devices connections and as memory interface. Chain interfaces are shown on figure 2.

Chain element used for devices is named DEP (Device endpoint). Each DEP includes a short instruction queue (FIFO) avoiding blocking of processors pipeline under busy of device condition. Destination address is kept in another short queue. Thanks to that, response latency from device/hardware function can vary and can be delayed even several clock cycles.

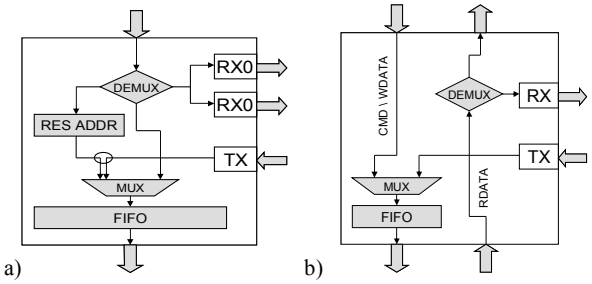

Fig. 2 Device Endpoint (DEP) (a) and Memory Bus (MBUS) (b) block diagram

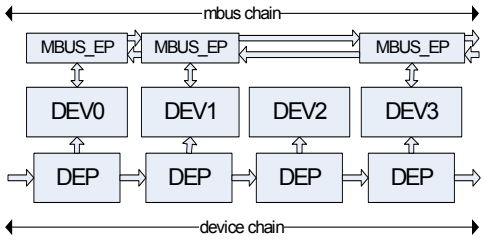

Fig. 3 Exemplary device chain with selected devices connected to the MBUS chain

Moreover, in the same time instructions to other devices which are further in chain, can be passing through. It means that instruction running in DEP chain can be swapped.

Similar chain element is used to form memory access chain topology. All devices that require memory access are connected to MBUS_EP (Memory Bus EndPoint). All endpoints form a chain, whose end is connected to memory controller or memory hub. When memory hub is used many chains can be used. Such chain architecture leads to growth of maximal operating frequency and bus throughput. It is worthily to notice that this solution exploits many short connections between chain elements instead of long bus. That structure is better for ASIC and especially for FPGA implementation where long distant connections are scarce resource.

3. PERFORMANCE TESTS

The proposed architecture has been tested during design of AVC (baseline profile) and AAC-HE (decoding) [1,2]. The projects were implemented in VERILOG language and verified on XILINX VIRTEX4-SX35 FPGA. Also compiler with IDE has been created to allow debugging of both processor code and hardware. Achieved results are shown in table 5.

Table 5. Performance of exemplary applications

APPLICATION	INPUT LIMITATIONS	CLOCK FREQUENCY	OUTPUT PERFORMANCE
AVC decoding	20 Mbit/s	66MHz	SDTV @ 30FPS
AAC decoding	1 Mbit/s	8 MHz	5.1CH @ 48kHz
AVC encoding	SDTV@ 25FPS	100MHz	I, P with CAVLC

The most time consuming functions [8] were transited to hardware and connected as hardware functions (devices on DEP chain) e.g. entropy codec, spatial and temporal prediction, transforms.

4. CONCLUSIONS

Presented results show that proposed architecture is competitive to existing solutions like ARM, PowerPC or Altera's NIOS II processors [4]. For example NIOS II processor on Altera FPGA running at 90MHz, with included specially designed blocks (binary decoding, inverse transform, intra prediction, etc), achieves only 12 frames per second for QCIF video sequence (252x288). In comparison our architecture enables to achieve more than 30 frames per second for SDTV video sequences (720x576) with processor running at 66MHz. The presented architecture gives much better performance then competitors which results in lower power consumption for same task. It turned out that introduced ideas have big impact on processors performance in case of multimedia content processing, and help to achieve much better efficiency per MHz and per Watt [5].

Exemplary applications suggest that proposed CISC architecture let to achieve high efficiency with little effort spent on designing additional hardware.

High complexity of tested algorithms leads to a conclusion that proposed architecture might be suitable also for other advanced multimedia solutions.

5. ACKNOWLEDGEMENTS

This paper was supported by public funds of "Fundacja na rzecz Nauki Polskiej" ("Foundation for Polish Science") in year 2007.

6. REFERENCES

[1] ISO/IEC International Standard 14496-10, "Advanced Video Coding", 3rd Edition, Redmond WA, July 2004.

[2] ISO/IEC International Standard 14496-3:2001/Amd.1:2003, "Coding of Audio-Visual Objects: Audio. Amendment 1: Bandwidth extensions", 2003.

[3] H. Yue, M. Lai, K. Dai, Z. Wang, "Design of a Configurable Embedded Processor Architecture for DSP Functions", Proceedings of 11th International Conference on Parallel and Distributed Systems, 2005, vol. 2, 20-22 July 2005, pp.: 27-31.

[4] Im Yong Lee, Il-Hyun Park, and Dong-Wook Lee, "Implementation of the H.264/AVC Decoder Using the Nios II Processor", Seoul National University, Nios® II Embedded Processor Design Contest, Asia/Pacific, 2005.

[5] Verbauwhede, P. Schaumont, C. Piguet, B. Kienhuis. "Architectures and Design Techniques for Energy Efficient Embedded DSP and Multimedia Processing", date, Design, Automation and Test in Europe Conference and Exhibition Volume II (DATE'04), 2004.

[6] Wieferink, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, A. Nohl, "A System Level Processor/Communication Co-Exploration Methodology for Multi-Processor System-on-Chip Platforms," date, p. 21256, Design, Automation and Test in Europe Conference and Exhibition Volume II (DATE'04), 2004.

[7] J. van de Waerdt, S. Vassiliadis, "Instruction Set Architecture Enhancements for Video Processing", 16th IEEE International Conference on Application-Specific Systems, Architecture Processors, ASAP 2005, 23-25 July 2005, pp.: 146 – 153.

[8] Łuczak, P. Garstecki, "A Flexible Architecture for Image Reconstruction in H.264/AVC Decoders", European Conference on Circuit Theory and Design ECCTD 2005, Cork, Ireland, 2005.

[9] H. Peter Hofstee, "Power Efficient Processor Architecture and The Cell Processor;, 11th International Symposium on High-Performance Computer Architecture", HPCA 2005 pp.: 258-262.