Scalable Core-Based Methodology and Synthesizable Core for Systematic Design Environment in Multicore SoC (MCSoC)

Ben A. Abderazek, Tsutomu Yoshinaga and Masahiro Sowa
The University of Electro-communications
Graduate School of Information Systems
1-5-1 Chofu-gaoka, Chofu-shi, Tokyo 1828585
E-mail: [ben,yosinaga,sowa]@is.uec.ac.jp

Abstract

The strong demand for complex and high performance embedded system-on-chip requires quick turn around design methodology and high performance cores. Thus, there is a clear need for new methodologies supporting efficient and fast design of these systems on complex platforms implementing both hardware and software modules.
In this paper, we describe a novel scalable core-based methodology for systematic design environment of application specific heterogeneous multicore systems-on-chip (MCSoC). We also developed a high performance 32-bit Synthesizable QueueCore (QC-2) with single precision floating point support. The core is targeted for special purpose applications within our target MCSoC system. We present the architecture description and design results in a fair amount of details.

1. Introduction

System on chips designs have evolved from fairly simple uni-core, single memory designs to complex multicore systems on-chip consisting of a large number of IP blocks on the same silicon. As more and more cores (macros) are integrated into these designs to share the ever increasing processing load, the main challenge lies in efficiently and quickly integrating them into a single system capable of leveraging their individual flexibility. Moreover, to connect the heterogeneous cores, the multi-core architecture requires high performance complex communication architectures and efficient communication protocols architecture, such as hierarchical bus [1, 2], point-to-point connection [15], or Time Division Multiplexed Access based bus [3].
Current design methods tend toward mixed HD/SW co-designs targeting multicore system-on-chip for specific applications [16, 18, 21]. To decide on the lowest cost mix of cores, designers must iteratively map the device's functionality to a particular HW/SW partition and target architecture. Every time the designers explore a different system architecture, the interfaces must be redesigned.
Unfortunately, the specific target applications generally lead to a narrow application domain and also managing all of these details is so time consuming that designers typically cannot afford to evaluate several different implementations. Automating the interface generation is an alternative solution and a critical part of the development of embedded system synthesis tools. Currently most automation algorithms implement the system based on a standard bus protocol (input/output interface) or based on a standard component (processing) protocol. Recent work has used a more generalize model consisting of heterogeneous multicore with arbitrary communication links. The SOS algorithm [19] uses an integer linear programming approach. The co-synthesis algorithm, developed in [20], can handle multiple objectives such as costs, performance, power and fault tolerance. Unfortunately, such design practices allow only limited automation and designers resort to manual architecture design, which is time consuming and error-prone especially in such complex SoCs.
Our design automation algorithm generates generic-architecture-template (GAT), where both processing and input/output interface may be customized to fit the specific needs of the application. Therefore, the utilization of the GAT enables a designer to make a basic architecture design without detailed knowledge of the architecture.
High performance processor cores are also needed for high performance heterogeneous multicore SoCs. Thus, we also describe a high performance synthesizable soft-core architecture, which will be used as a task-distributor-core (TDC) in the MCSoC system. The system may consist, then, of multiple processing cores of various types (i.e., QueueCore(s), general purpose processor(s), domain spe-

cific DSPs, and custom hardware), and communication links. The ultimate goal of our systematic design automation and architecture generation is the to improve performance and the design efficiency of large scale heterogeneous multicore SoC. The rest of the this work is organized as follow: Section 2 give conventional SoC design methodology. Section three gives our multicore architecutre platform description. Section four gives our core-based method for a systematic environment in a heterogeneous MCSoC. Section five gives the synthesizable QC-2 core architecture. Section six describes the QC-2 core evaluation. In the last section we give the conclusion.

2. Problem identification and background

The gate densities achieved in current ASIC and FPGA devices give the designers enough logic elements to implement all the different functionalities on the same chip (SoC) by mixing self-design modules with third party one [3, 8, 18]. This possibility opens new horizons especially for embedded systems where space constraints are as important as performance. The most fundamental characteristic of an SoC is complexity. The SoC is generally tailored to the application rather than general-purpose chip, and may contain memory, one or several specialized cores, buses, and several other digital functions. Therefore, embedded applications cannot use general-purpose computers (GPPs) either because a GPP machine is not cost effective or because it cannot provides the necessary requirements and performance. In addition, a GPP machine can't provide reliable real-time performance.
In Fig. 1, a typical multicore architecture is shown. The typical model is made of a set of cores communicating through an AMBA communication architecture [1]. The communication architecture constitutes the hardware links that support the communication between cores. It also provides the system with the required support for the general data transfer with external devices common to most applications. Inter-component link is often in the critical path of such a system and is a very common source of performance bottlenecks [23]. It thus becomes imperative for system designers to focus on exploring the communication design space.

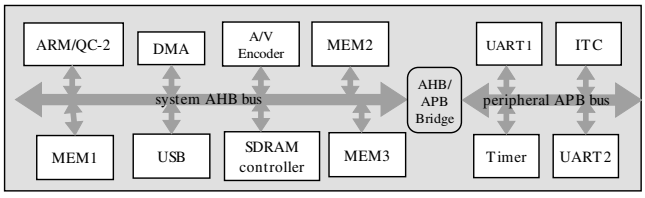

Figure 1. SoC typical architecture

Conventional SoC architectures are classified into tow types: single-processor and multiprocessor architectures. A single-processor architecture consists of a single CPU and one or several ASICs. A master-slave synchronization pattern is adopted in this type. The single-processor SoC type can only offer a restricted performance capability in many applications because of the lack of true parallelism. A multiprocessor SoC architecture is a system that contains multiple instruction set processors (CPUs) and also one or several ASICs. In term of performance, multiprocessor SoCs perform better for several embedded applications. However, they (multiprocessor SoCs) introduce new challenges: first, the inter-processor communication may require more sophisticated networks than a simple shared bus; and second, the architecture may include more than one master processor. In either type, high processing performance is required because most of the applications for which SoCs are used have precise performance requirements deadlines. This is different from conventional computing, where care is generally about processing speed. We will discuss the performance issue in the following section when we introduce the QC-2 core.
In general, the architectures used in conventional methods of multiprocessor SoC design and custom multiprocessor architectures are not flexible enough to meet the requirements of different application domains (e.g. only point-to-point or shared bus communication is supported.) and not scalable enough to meet different computation needs and different complexity of various applications. A promising approach was proposed in [20]. This method is a core-based solution, which enables integration of heterogeneous processors and communications protocols by using abstract interconnections. Behaviour and communication must be separated in the system specification. Hence, system communication can be described at a higher-level and refined independently of the behaviour of the system. There are two component-based design approaches: usage of a standard bus (i.e., IBM CoreConnect) protocol and usage of a standard component (i.e., VSIA) protocol [16, 18, 21].
For the first approach, a wrapper is designed to adapt the protocol of each component to CoreConnect protocols. For the second case, the designer can choose a bus protocol and then design wrappers to interconnect using this protocol. This paper presents a new concepts, called virtual architecture, to cover both methods listed above. The virtual system represents an architecture as an abstract netlist of virtual cores, which should use wrappers to get adapt accesses from the internal component to the external port.

3. ESPOIR multicore architecture platform

The target model of our architecture consists of CPUs (i.e., QueueCore (QC-2), GPPs), hardware blocks, memo-

ries, and communication interfaces. The addition of new CPUs will not change the main principle of the proposed methodology. The processors are connected to the shared communication architecture via communication network, which maybe of whatever complexity from a single bus to a network with complex protocols. However, to ensure modularity, standard and specific interfaces to link cores to the communication architecture should be used. This gives the possibility to design separately each part of the application. For this purpose, we proposed in [22] a modular design methodology. One important feature of the proposed method is that the generic assembling scheme largely increases the architecture modularity. Figure 2 show a typical instance of the platform made of 4 processors (2*QC-2 cores and 2*SH cores -Hitachi SuperH core). The QC-2 core is a special purpose synthesizable core (described in details in section 5).

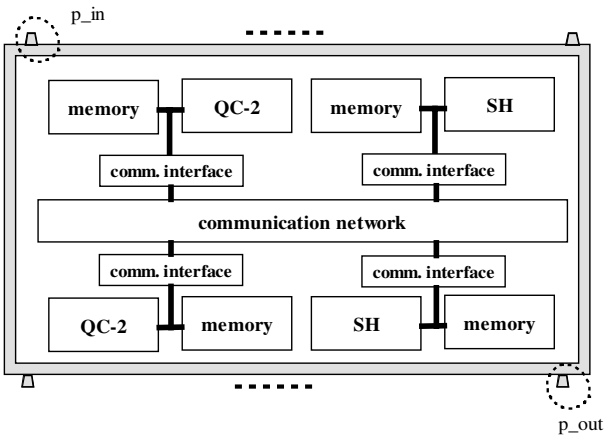

Figure 2. MCSoC system platform. This is a typical instance of the architecture. In this system, the addition of a new core will not change the principle of the methodology.

The designer can configure: the number of CPUs, I/O ports for each processor and interconnections between processors, the communication protocols and the external connections (peripherals). The communication interface depends on the processor attributes and on the application-specific parameters. The communication interface that we intend to use to connect a given processor to the communication architecture, consists of two parts: one part specific to the processor's bus; the second part is generic and depends on communication protocols and on the number of communication channels used. This structure allows the "isolation" of the CPU core from the communication network.

Each interface module acts as a co-processor for the corresponding CPU. The application dependent part may include several communication channels. The arbitration is done by the CPU-dependent part and the overhead induced by this communication co-processor depends on the design of the basic components and may be very low. The use of this architecture for interfaces, provides huge flexibility and allows for modularity and scalability.

4. Application specific MCSoC design method

In our design methodology, the application-specific parameters should be used to configure the architecture platform and an application-specific architecture is produced. These parameters result from an analysis of the application to be designed. The design flow graph (DFG) is divided into 14 "$linked - tasks$" as shown in Fig. 3 (a)-(b) and summarized in Table 1. The first task (node T1) defines the architecture platform using all fixed architectural parameters: (1) Network type, (2) Memory architecture, (3) CPU types, and (4) other HW modules. Using the application system level description (second task) and the architectural fixed parameters, the selection of the actual design parameters (number of CPUs, the memory sizes for each core, I/O ports for each core and interconnections, between cores, the communication protocols and the external peripherals) is performed in task 3 (node T3). The outputs of task 3 are: an abstract architecture description (node T7) and a map-

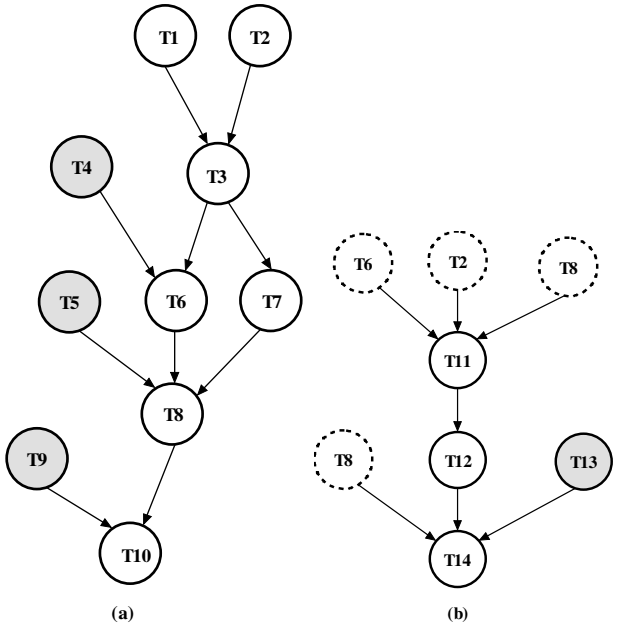

Figure 3. Linked-task design flow graph (DFG).(a) Hardware related tasks,(b) Application related tasks.

Table 1. Linked-task description.

Task	Description
T1	Define architecture platform
T2	Describe application system level
T3	select design parameters
T4	Instantiate Pr. att.
T5	Instantiate communication
T6	mapping table
T7	Describe abstract architecture
T8	Design architecture
T9	Inst.IP cores (Pr.and Mem)
T10	H-SoC synthesis
T11	Software adaptation
T12	Binary code
T13	Pr. and Mem. emulators
T14	H-SoC validation

ping table (node T6). Node T7 is the internal structure of the target system architecture. It contains all the application specific parameters. The mapping table (T7) contains the addresses allocation and memory map for each core. The complete architecture design task (T8) is linked to the abstract architecture and the mapping table nodes (tasks). Finally, binary programs that will run on the target processors are produced in task 11 (node T11). For validation, cycle accurate simulation for CPUs and HDL (Verilog or VHDL) modeling for other cores/modules can be used for the whole architecture.

5. QC-2 core architecture

We proposed in [5, 6] a produced order parallel Queue processor (QueueCore) architecture. The key ideas of the produced order queue computation model of our architecture are the operands and results manipulation schemes. The Queue computing scheme stores intermediate results into a circular queue-register (QREG). A given instruction implicitly reads its first operand from the head of the QREG, its second operand from a location explicitly addressed with an offset from the first operand location. The computed result is finally written into the QREG at a position pointed by a queue-tail pointer (QT). An important feature of this scheme is that, write after read false data dependency does not occur [5]. Furthermore, since there is no explicit referencing to the QREG, it is easy to add extra storage locations to the QREG when needed. The other feature of the POC computing model is its important affect on the instruction issue hardware. The QC-1 core [6] exploits instruction-level parallelism without considerable effort and need for heavy run time data dependence analysis, resulting in a simple hardware organization when compared with conventional Superscalar processors. This also allows the inclusion of a large number of functional units into a single chip, increasing parallelism exploitation. Since the operands and result addresses of a given static-instruction (compiler generated) are implicitly $computed$ during run-time, an efficient and fast hardware mechanism is needed for parallel execution of instructions. The queue processor implements a so named queue computation mechanism that calculates operands and result addresses for each instruction (discussed later). The QC-2 core, presented in this work, implements all hardware features found in QC-1 core and also supports single precision floating point accelerator.
In this section we describe the QC-2 (extended and optimised version of the QueueCore processor) architecture and prototyping results. As we explained in earlier section, the QC-2 core will be integrated into our H-SoC system.

5.1 Hardware pipeline structure

The QC-2 supports a subset of the produced order queue processor instruction set architecture [6]. All instructions are 16-bit wide, allowing simple instructions fetch and decode stages and facilitate instructions pipelining. The pipeline's regular structure allows instructions fetching, data memory references, and instruction execution to proceed in parallel. Data dependencies between instructions are automatically handled by hardware interlocks. Bellow we describe the salient characteristics of the QueueCore architecture.
(1) *Fetch (FU)*: The instruction pipeline begins with the fetch stage, which delivers four instructions to the decode unit each cycle. This is the same bandwidth as the maximum execution rate of the functional units. At the beginning of each cycle, assuming no pipeline stalls or memory wait states occur, the address pointer hardware of the fetched instructions issues a new address to the Data/Instruction memory system. This address is either the previous address plus 8 bytes or the target address of the currently executing flow-control instruction.
(2) *Decode (DU)*: The QC-2 decodes four instructions in parallel during the second phase and writes them into the decode buffer. This stage also calculates the number of consumed (CNBR) and produced (PNBR) data for each instruction [5]. The CNBR and PNBR are used by the next pipeline stage to calculate source and destination locations for each instruction. Decoding stops if a queue becomes full.
(3) *Queue computation (QCU)*: The QCU calculates the first operand ($source1$) and destination addresses for each instruction. The mechanism used for calculating the $source1$ address in given in Fig. 4. The QCU unit keeps track on the current value of the queue-head and queue-tail

pointers. Four instructions arrive to the QCU unit each cycle.
(4) *Barrier:* The major goal of this unit/stage is to insert barrier flags for all barrier type instructions.
(5) *Issue:* Four instructions are issued for execution each cycle. In this stage, the second operand ($source2$) of a given instruction is first calculated by adding the address $source1$ to the displacement that comes with the instruction. The second operand's address calculation could be earlier calculated in the QCU stage. However, for a balanced pipeline consideration, the $source2$ is calculated in this stage. The hardware mechanism used for calculating the second operand ($source2$) address is shown in the left part of Fig. 4 (discussed later).
An instruction is ready to be issued if its data operands and its corresponding functional unit are available. The processor reads the operands from the QREG in the second half of stage 5 and execution begins in stage 6.
6) *Execution (EXE)*: The macrodataflow execution core consists of 1 integer ALU unit, 1 floating-point accelerator unit, 1 branch unit, 1 multiply unit, 4 set-units, and 2 load/store units.
The load and store units share a 16-entry address window (AW), while the integer unit and the branch unit share a 16-entry integer window (IW). The FPA has its own 16-entries floating point window (FW). The load/store units have their own address generation logic. Stores are executed to memory in-order.

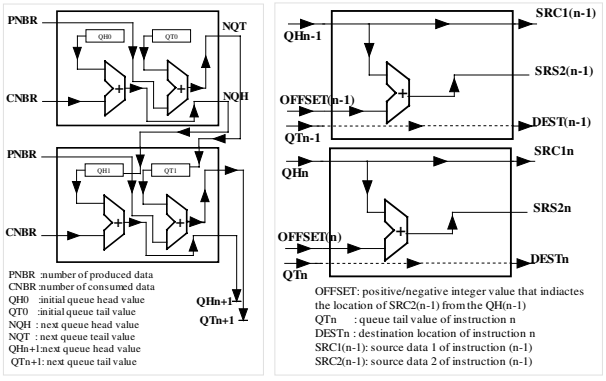

Figure 4. QC-2's operands addresses calculation hardware.

5.2 Dynamic operands calculation

To execute instructions in parallel, the QC-2 core must calculate the operands addresses ($source1$, $source2$ and $destination$) for each instruction. Fig. 4 illustrates QC-2's dynamic operands computation hardware. To calculate the $source1$ address, the consumed operands (CNBR) field (port field) is added to the current QH value (QH0). To find the address of the first operand and the number of produced results (PNBR- 8-bit filed) is added to the current QT value (QT0) to calculate the result address (QT1) of the first instruction. Similar mechanism is used for the other three instructions. Because the next QH and QT values are dependent on the current QH and QT values, the calculation is performed sequentially. Each QREG entry is written exactly once and it is busy until it is written. If a subsequent instruction needs its value, that instructions must wait until it is written. After QREG entry is written, it is ready.

5.3 Floating point organization

The QC-2 floating-point accelerator (FPA) is a pipelined structure and implements a subset of the IEEE-754 single precision floating-point standard [13, 14]. The FPA consists of a floating-point ALU (FALU), floating-point multiplier (FMUL), and floating point divider (FDIV). The FALU, FMUL, FDIV and the floating-point queue-register (FQREG) employ 32-wide data paths. Most FPA operations are completed within three execution cycles. The FPA's execution pipelines are simple in design for high speeds that the QC-2 core requires. All frequently used operations are directly implemented in the hardware. The FPA unit supports the four rounding modes specified in the IEEE 754 floating point standard: round toward-to-nearest-even, round toward positive infinity, round toward negative infinity, and round toward zero.

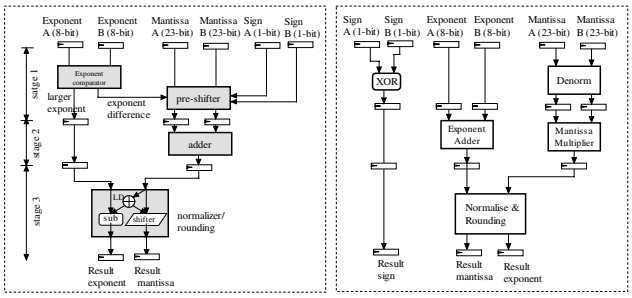

Figure 5. QC-2's FPA hardware.

5.3.1 Floating point ALU implementation

The FALU does floating-point addition, subtraction, compare and conversion operations. Its first stage subtracts the operands exponents (for comparison), selects the larger operand, and aligns the smaller mantissa. The second stage adds or subtracts the mantissas depending on the operation and the signs of the operands. The result of this operation

may overflow by a maximum of 1-bit position. Logic embedded in the mantissa adder is used to detect this case, allowing 1-bit normalization of the result on the fly. The exponent data path computes (E+1). If the 1-bit overflow occurred, (E+1) is chosen as the exponent of stage 3; otherwise, E is chosen. The third stage performs either rounding or normalization because these operations are not required at the same time. This may also result in a 1-bit overflow. Mantissa and exponent corrections, if needed, are implemented exactly in this stage, using instantiations of the mantissa adder and exponent blocks.

The area efficient FALU hardware is shown in Fig. 5 (left block). The exponents of the two inputs (Exponent A and Exponent B) are fed into the exponent comparator, which is implemented with a subtracter and a multiplexer. In the pre-shifter, a new mantissa in created by right shifting the mantissa corresponding to the smaller exponent by the difference of the exponents so that the resulting two mantissas are aligned and can be added. The size of the preshifter is about $m * log(m)LUTs$, where m is the bit-width of the mantissa. If the mantissa adder generates a carry output, the resulting mantissa is shifted one bit to the right and the exponent is increased by one. The normalizer transforms the mantissa and exponent into normalized format. It first uses a leading-one detector (LD) circuit to locate the position of the most significant one in the mantissa. Based on the position of the LD, the resulting mantissa is left shifted by an amount subsequently deducted from the exponent. If there is an exponent overflow (during normalization), the result is saturated in the direction of overflow and the overflow flag is set. Underflows are handled by setting the result to zero and setting an underflow flag.
We have to notice that the LD anticipator can be also predicted directly from the input to the adder. This determination of the leading digit position is performed in parallel with the addition step so as to enable the normalization shift to start as soon as the addition completes. This scheme requires more area than a standard adder, but exhibits reduced latency. For hardware simplicity and logic limitation, our FPA hardware does not support earlier LD prediction.

5.3.2 Floating point multiplier implementation

The right part of Fig. 5 shows the data path of the FMUL unit. As with other conventional architectures, QC-2's FMUL operation is much like integer multiplication. Because floating point numbers are stored in sign-magnitude form, the multiplier needs only to deal with unsigned integer numbers and normalization. Similar to the FALU, the FMUL unit is a three stages pipeline that produces a result on every clock cycle. The bottleneck of this unit was the $24 * 24$ integer multiplications.
The first stage of the floating-point multiplier is the same denormalization module used in addition to insert the implied 1 to the mantissa of the operands. In the second stage, the mantissas are multiplied and the exponents are added. The output of the module are registered. In the third stage, the result is normalized or rounded.
The multiplication hardware implements the radix-8 modified Booth [17] algorithm. Recoding in a higher radix was necessary to speed up the standard Booth multiplications algorithm since greater numbers of bits are inspected and eliminated during each cycle, effectively reduces the total number of cycles necessary to obtain the product. In addition, the radix-8 version was implemented instead of the radix-4 version because it reduces the multiply array in stage 2.

6. QC-2 synthesis and evaluation results

6.1 Methodology

In order to estimate the impact of the description style on the target FPGAs efficiency, we have explored logic synthesis for FPGAs. The idea of this experiment was to optimise critical design parts for speed or resource optimisations.
Optimising the HDL description to exploit the strengths of the target technology is of paramount importance to achieve an efficient implementation. This is particularly true for FPGAs targets, where a fixed amount of each resource is available and choosing the appropriate description style can have a high impact on the final resources efficiently [4, 7]. For typical FPGAs features, choosing the right implementation style can cause a difference in resource utilization of more than an order of magnitude [9, 10]. Synthesis efficiency is influenced significantly by the match of resource implied by the HDL and resources present in a particular FPGAs architecture. When an HDL description implies resources not found in a given FPGAs architecture, those elements have to be emulated using other resources at significant cost. Such emulation can be performed automatically by EDA tools in some cases, but may require changes in the HDL description in the worst case, counteracting aim of a common HDL source code base. In this work, our experiments and the results described are based on the Altera Stratix architecture [12]. We selected Stratix FPGAs device because it has a good tradeoffs between routability and logic capacity. In addition it has an internal embedded memory that eliminates the need for external memory module and offers up to 10 Mbits of embedded memory through the TriMatrix TM memory feature. We also used Altera Quartus II professional edition for simulation, placement and routing. Simulations were also performed with Cadence Verilog-XL tool [11].

Table 2. QC-2 processor design results: modules complexity as LE (logic elements) and TCF (total combinational functions) when synthesised for FPGA (with Stratix device) and Structured ASIC (HardCopy II) families.

Descriptions	Modules	LE	TCF
instruction fetch unit	IF	633	414
instruction decode unit	ID	2573	1564
queue compute unit	QCU	1949	1304
barrier queue unit	BQU	9450	4348
issue unit	IS	15476	7065
execution unit	EXE	7868	3241
queue-registers unit	QREG	35541	21190
memory access	MEM	4158	3436
control unit	CTR	171	152
Queue processor core	QC-2	77819	42714

6.2 Design results

Figure 6 compares two different target implantations for 256x33 QREG for various optimisations. Depending on the target implementations device, either logic elements (LEs) or total combinational functions (TCF) are generated as storage elements. Implementations based on HardCopy device, which generates TCF functions give almost similar complexity for the three used optimisations − area (ARA), speed (SPD) and balanced (BLD). For FPGA implementation, the complexity for SPD optimisation is about 17% and 18% higher than that for ARA and BLD optimisations respectively. Table 2 summarizes the synthesis results of

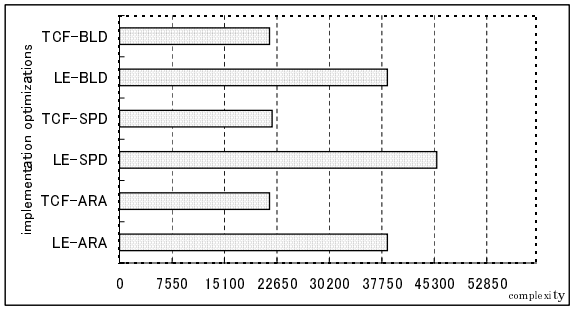

Figure 6. Resource usage and timing for 256*33 bit QREG unit for different coding and optimization strategies.

the QC-2 for the Stratix FPGA and HardCopy targets. The complexity of each core module as well as the whole QC-2 core are given as the number of logic elements (LEs) for the Stratix FPGA device and as the TCF cell count for the HardCopy device (Structured ASIC). The design was optimised for BLD optimisation guided by a properly implemented constraint table. We also found that the processor consumes about 80.4% of the total logical elements of the target device.

The achievable throughput of the 32-bit QC-2 core on different execution platforms is shown in Fig. 7. For the hardware platforms, we show the processor frequency. For comparison purposes, the Verilog HDL simulator performance has been converted to an artificial frequency rating by dividing the simulator throughput by a cycle count of 1 CPI. This chart shows the benefits which can be derived from direct hardware execution using a prototype when compared to processor simulation. The data used for this simulation are based on event-driven functional Verilog HDL simulation [5].

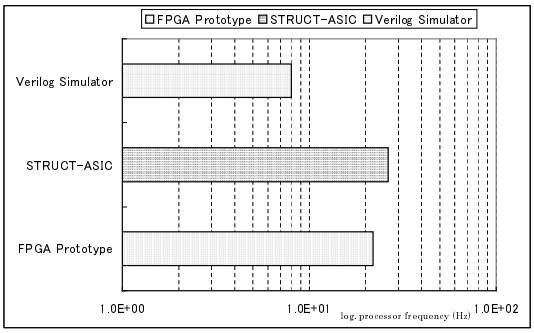

Figure 7. Achievable frequency is the instruction throughput for hardware implementations of the QC-2 processor. Simulation speeds have been converted to a nominal frequency rating to facilitate comparison.

7. Conclusion

In this research work we described two main contributions: (1) a scalable core based methodology for generic architecture model and (2) a Synthesizable 32-bit QC-2 core with floating point support targeted for high performance heterogeneous multicore SoC (MCSoC). The proposed design methodology, although it was not tested, is expected to have a big effect of system scalability, modularity and design time. The method also should permit a systematic generation of multicore architecture for multicore embedded system-on-chip MCSoCs.

The second contribution is the implementation and optimisation of a QC-2 core - an improved version of our earlier designed QC-1 core. The 32-bit synthesizable QC-2 core

supports single precision floating point support. It was correctly synthesesized and tested with several Testbenches. The QC-2 core was, then, optimised for speed guided by a properly implemented constraint table. We found that the processor consumes about 80.4% of the total logical elements of the target FPGA device. It achieves about 22.5 and 25.5 MHz for 16 and 264 QREG entries respectively.

References

[1] D. Flynn, "AMBA: enabling reusable on-chip designs", IEEE Micro, vol.17, n.4,July 1997, pp.20-27.

[2] IBM CoreConnect Bus Architecture, www-03.ibm.com/chips/products/coreconnect/

[3] D. Wingard and A. Kurosawa, Integration Architecture for System-on-a-Chip Design, Proceedings of IEEE 1998 Custom integrated Circuits Conference, May 1998, pp. 85-88.

[4] G. De Micheli, R. Ernst and W. Wolf, *"Readings in Hardware/Software co-design"*, Morka Kaufmann Publishers, ISBN 1-55860-702-1.

[5] M. Sowa, B. A. Abderazek and T. Yoshinaga, *"Parallel Queue Processor Architecture Based on Produced Order Computation Model"*, Int. Journal of Supercomputing, Vol.32, No.3, June 2005, pp.217-229.

[6] B. A. Abderazek, M. Arsenji, S. Shigeta, T. Yoshinaga and M. Sowa, *"Queue Processor for Novel Queue Computing Paradigm Based on Produced Order Scheme"*, Proc. of HPC, IEEE CS, Jul. 2004, pp. 169-177.

[7] S. Aditya, B. R. Rau and V. Kathail, *"Automatic Architectural Synthesis of VLIW and EPIC Processors"*, Proc. 12th Int. Symposium of System Synthesis, IEEE CS Press, Los Alamitos, Calif., 1999, pp.107-113.

[8] M. Sheliga and E. H. Sha, *"Hardware/Software Co-design With the HMS Framework"*, Journal of VLSI Signal Processing Systems, Vol. 13, No.1, 1996, pp. 37-56.

[9] S. Chaudhuiri, S. A. Btlythe and R. A. Walker, *"A solution methodology for exact design space exploration in a three dimensional design space"*, IEEE Transactions on VLSI Systems, Vol. 5,1997, pp.69-81.

[10] D. Lewis, V. Betz, D. Jefferson, et al., *"The Stratix Routing and Logic Architecture"*, in Proc. IEEE FPGAs, Monterey, CA, 2003, pp. 1220.

[11] Cadence Design Systems:http://www.cadence.com/.

[12] Altera Design Software: http://www.altera.com/.

[13] IEEE Standard for Binary Floating-point Arithmetic, ANSI/IEEE Standard 754, 1985.

[14] IEEE task P754, "A proposed standard fr binary floating-ponit arithmetic", IEEE Computer, vol. 14, no. 12, pp.51-62, March 1981.

[15] Mirko Loghi, Federico Angiolini, Davide Bertozzi, Luca Benini, Roberto Zafalon, "Analyzing On-Chip Communication in a MPSoC Environment,Proceedings of the conference on Design", Design Automation and test in Europe, Vol.2, Feb.16-20, 2004.

[16] R. Ernst, J. Henkel, T. Benner, "Hardware-software cosynthesis for microcontrollers", IEEE Design and Test,Dec. 1993, pp. 64-75.

[17] A. D. Booth, A signed binary multiplication technique, Quart. J. Mech. Appl. Math., vol. 4, 1951,pp. 236240.

[18] Multiprocessor System-on-Chip, Morgan Kaufman Pblishers,ISBN:0-12385-251-X, 2005.

[19] S. Prakash and A. Parker, "SoS: Synthesis of application-specific heterogeneous multiprocessor systesm", J. Parallel Distributed Computing, vol. 16, 1992, pp. 338-351.

[20] B. Dave, G. Lakshminarayama, and N. Jha, "COSFA: Hardware-software co-synthesis of heterogeneous distributed embedded system architecures for low overhead fault tolereance", In Proc. IEEE fault-Tolerant computing symp., 1997.pp. 339-348.

[21] C. K. Lennard, P. Schaumont, G. de Jong, A. Haverinen, and P. Hardee, "Standards for System-Level Design: Practical Reality or Solution in Search of a Question?", Proc. Design Automation and Test in Europe, Mar. 2000, pp. 576-585.

[22] B. A. Abderazek, Sotaro Kawata, Tsutomu Yoshinaga, and Masahiro Sowa, "Modular Design Structure and High-Level Prototyping for Novel Embedded Processor Core", Proceedings of the 2005 IFIP International Conference on Embedded And Ubiquitous Computing (EUC'2005), Nagasaki, Japan, Dec. 6 - 9, 2005, pp. 340-349.

[23] S. Pasricha, N. Dutt, M. Ben-Romdhane, "Constraint-Driven Bus Matrix Synthesis for MPSoC", Asia and South Pacific Design Automation Conference (ASPDAC 2006), Yokohama, Japan, January 2006,pp.30-35.