Chapter 2
Multicore SoCs Design Methods

The strong demand for complex and high performance multicore systems-on-chip (MCSoCs) requires quick turn around design methodology. Thus, there is a clear need for efficient methodology for the design of these systems on platforms implementing both hardware and software modules.

This chapter describes conventional multiore SoC design methods in details. It also describes a so called scalable core-based methodology for systematic design environment of application specific heterogeneous multicore SoC architectures. Although the methodology presented here is general and not limited to special architecture, we will consider a real synthesizable core as a case study to make the discussion easy.

2.1 Introduction

Systems-on-chip designs have evolved from fairly simple uni-core, single memory designs to complex multicore SoCs consisting of tens or hundreds of cores in a single chip. As more and more cores are integrated into these chips to share the ever increasing processing load, the main challenges lie in how to efficiently and quickly integrate these cores together into a single system capable of leveraging their individual flexibility. Moreover, for better inter-core communication, the multicore system requires high performance communication architectures and efficient communication protocols, such as hierarchical bus (Diefendorff 1997; Liu 2005), point-to-point connection (Loghi 2004), Time Division Multiplexed Access (TDMA) based bus (Kulkarani 2002), or packet-switching networks (Ben-Abdallah 2006).

Recently, SoC design methods tend toward mixed hardware/software co-designs targeting multicore SoCs for specific applications (Ernst 1993; Jerraya 2005; Lennard 2000). To decide on the lowest cost mix of cores, designers must iteratively map the device's functionality to a particular hardware/software partition and target architecture (platform). When a designer wants to explore different system architectures, the interfaces must be redesigned. This method may lead to a narrow application domain. In addition, managing all these details is 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. Most existing automation algorithms implement the system based on a standard bus protocol (input/output interface) or based on a standard component (processing) protocol. Recent works have used a more generalize model consisting of heterogeneous multicore with arbitrary communication links. The SOS algorithm (Prakash 1992) uses an integer linear programming approach. The co-synthesis algorithm, developed in (Dave 1997), can handle multiple objectives such as cost, performance, power, and fault tolerance. Such design methods allow only limited automation and designers resort to manual architecture design which is time consuming and error-prone.

There are two fundamental steps needed for MCSoC design: (1) selection and construction of a target multicore platform, known as design space exploration phase, and (2) development of the parallel software for exploiting the application parallelism on the selected platform, known as parallel software development phase. We will describe these hardware and software design phases in the following two sections.

2.2 Design Space Exploration

There are various design axes that define the design space of multicore platforms, which include processor architectures and numbers, memory configuration, communication architectures, hardware accelerators, and so on. To determine the target platform, we need a technique that quickly evaluates the expected performance of each candidate and explores the wide design space without actual hardware implementation. Further, the gate densities achieved in current ASIC and FPGA devices give designers enough logic elements to implement all functionalities on the same chip by mixing self-design modules with third party ones (Kulkarani 2002; Sheliga 1996; Jerraya 2005). This possibility opens new horizons especially for embedded systems where space constraints are as important as performance. The most fundamental characteristic of a 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 provide the necessary requirements and performance. In addition, a GPP machine can't provide reliable real-time performance.

In Fig. 2.1, a typical multicore SoC architecture block diagram is shown. This typical model is made of a set of cores communicating through an AMBA communication architecture (Diefendorff 1997). 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

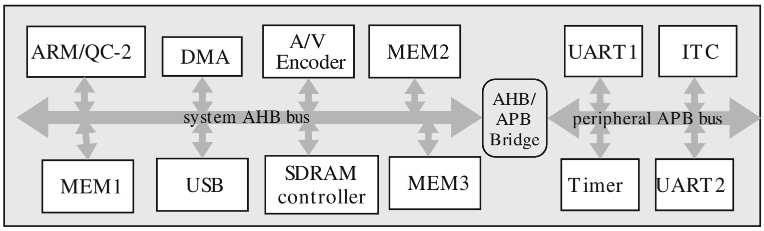


Fig. 2.1 SoC typical architecture

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 (Pasricha 2006). Thus, it becomes imperative for system designers to focus on exploring the communication design space.

Conventional SoC architectures are generally classified into tow types: single-core based and multicore based systems. Single-core architecture consists of a single CPU core and one or several ASICs. A master-slave synchronization pattern is adopted in this type. The single-core SoC type can only offer a restricted performance capability in many applications because of the lack of true parallelism.

A multicore SoC architecture is a system that contains multiple CPU cores and also one or several ASICs. In term of performance, multicore SoCs perform better for several embedded applications. However, these systems generally 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 both types, 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 general purpose computing.

In general, the architectures used in conventional methods of multicore SoC design and custom multicore 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 Dave (1997). This method is a core-based solution, which enables integration of heterogeneous processors and communications protocols by using abstract interconnections. Behavior and communication must be separated in the system specification. Hence, system communication can be described at a higher-level and refined independently of the behavior of the system. There are two known component-based design approaches: (1) usage of a standard bus (i.e., IBM CoreConnect) protocol, and (2) usage of a standard component protocol (Ernst 1993; Jerraya 2005; Lennard 2000). For the first approach, a wrapper is designed to adapt the protocol of each component to CoreConnect protocol. For the second case, the designer can choose a bus protocol and then design wrappers to interconnect components using the above protocol.

2.3 Parallel Software Development Phase

Embedded parallel software development for multicore platforms involves parallel programming for homogeneous and heterogeneous multicore SoC architectures under several design constraints such as power, area, cost, and timeliness.

The sequential Von Neumann programming model is not a good option for the multicore based systems because it simply cannot exploit the huge parallelism which is available in different forms in multicore platforms. Thus, it is clear that we now need new programming models and corresponding software development tools that are capable of exploiting all forms of available parallelism. Recently, big efforts have been made to develop methods and tools that solve the design problems of multicore SoCs targeted for various applications and under several design constraints. Bellow, we will describe these methods in details.

2.3.1 Compiler-Based Schemes

In compiler-based schemes, the sequential Von Neumann program is used as input, where all specifications are defined (Phase 1). Then, a parallelizing compiler automatically parallelizes (Phase 2) the source code (or binary) as illustrated in Fig. 2.2. Using several parallelizing techniques, this phase (Phase 2) analyses the input code and finds parallel regions. More specifically, Phase 2 parallelizes the serial code. A well known technique is to identify all loops and examine their dependencies by analyzing indexes. The mapper, then, transforms each parallel region into a set of concurrent tasks and maps them onto multiple cores (Phase 3).

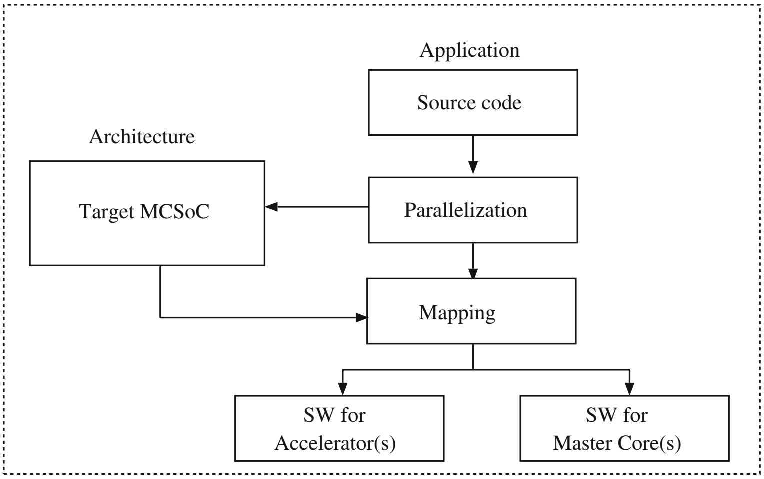


Fig. 2.2 Compiler based scheme

2.3.2 Language Extensions Schemes

The *Language extension schemes* require that application programmer provides all parallelism information as well as where and how to parallelize the code with language extension that has annotations and/or additional application programming interfaces (APIs). As a result, compilers in *language-extension schemes* can focus on exploiting the specified parallelism according to the the target platform.

2.3.2.1 Language Extension with Annotations

The main merit of the language extension with annotations approach is simplicity. That is, it simplifies the compiler's job by relieving the burden of extracting parallelism while it gives only a little overhead of annotations to the software developer.

The Open Multiprocessing Standard [OpenMP] is an example of language extension with annotations. OpenMP is a widely used API for parallel programming and is attractive because programmers can continue using their familiar programming model while re-using their existing codes.

As an example, suppose a programmer is writing a ray tracing program, which goes through each pixel of the screen, and using lighting, texture, and geometry information, the color of that pixel is determined. The program goes on to the next pixel and repeats (loops) the process. The calculation for each pixel is completely separate from the calculation of any other pixel, therefore making this program highly suitable for OpenMP. The code for the above example is shown in Fig. 2.3. This piece of code simply goes through each pixel of the screen, and calls a function, RenderPixel, to determine the final color of that pixel. Note that the results are simply stored in an array. Because each pixel is independent of all other pixels, and because RenderPixel is expected to take a noticeable amount of time, this small snippet of code is a prime candidate for parallelization and can be simple annotated with OpenMP directive: *#pragma omp parallel for*.

We have to note here that OpenMP standard was originally developed for symmetric multiprocessor (SMP) computers with shared memory. Recently, it was ported to heterogeneous multicore platforms, such as in IBM Cell processor (Obrien 2008). GNU GCC also adopted the GOMP OpenMP implementation. Thus, many GCC-enabled multicore processors now support OpenMP [GOMP] The Cell processor is a heterogeneous multicore processor with one Power Processing Engine (PPE) core and eight Synergistic Processing Engine (SPE) cores. Each SPE has a directly accessible small local memory (256K), and it can access the system memory through DMA operations. Programming Cell system is difficult since an SPE core has a small local memory and accesses the system memory only through DMA operations. The other difficulty comes from the availability of several layers of parallelism in the architecture, including heterogeneous cores, multiple SPE cores, multi-threading. Cell compiler is built upon an IBM XL compiler therefore

Fig. 2.3 Parallel for loop with OpenMP

```
for(int x=0; x < width; x++)
{
 for(int y=0; y < height; y++)
 {
  finalImage[x][y] = RenderPixel(x,y, &sceneData);
 }
}
```

(a) Before parallelization

```
#pragma omp parallel for
for(int x=0; x < width; x++)
{
 for(int y=0; y < height; y++)
 {
  finalImage[x][y] = RenderPixel(x,y, &sceneData);
 }
}
```

(b) After parallelization

translates the parallel region into a set of concurrent tasks that run on the SPE cores with a control task that schedules the SPE tasks (Obrien 2008).

2.3.3 Language Extensions with APIs

In this scheme, a software developer writes a parallel program with specifically defined APIs for parallel execution. Compared with the annotation scheme, the APIs based approach allows more low-level control of parallelism by the software developer. Although this scheme has better performance, it requires that the programmer manually discovers the parallel regions, distributes the code and data to the processors, and restructures the code using the APIs.

Message passing interface (MPI) is an example of the language extension with APIs since it started to find its use in embedded heterogeneous multicore SoCs.

2.3.4 Model-Based Schemes

Model-based schemes are advocated for multicore and MCSoC design since they simplify the application behavior and reveals the top-level structure of the behavior; this eliminates the complex low-level implementation details. In this scheme, the software developer determines which model of computation is used to capture application algorithms. For example, the actor based models are used to specify the computation-oriented applications and the FSM (finite state machine) model for control-oriented applications.

2.4 Generic Architecture Template for Real Multicore SoC Design

In this section we will describe a design design method based on a so called generic-architecture-template (GAT), where both processing and input/output interface may be customized to fit the specific needs of the application. GAT design method enables a designer to make a basic architecture design without detailed knowledge of the architecture.

A high performance synthesizable soft-core architecture, called QueueCore, is also presented here and is used as a task-distributor-core (TDC) in the a multicore SoC system design. The system may consist, then, of multiple processing cores of various types (i.e., QueueCore(s), general purpose processor(s), domain specific DSPs, and custom hardware), and communication links. The ultimate goal of the above systematic design automation and architecture generation is the to improve performance and the design efficiency of large scale heterogeneous multicore SoC.

2.4.1 Target Multicore SoC Platform

The target model of the architecture consists of CPUs (i.e., QueueCore (QC-2), GPPs), hardware blocks, memories, and communication interfaces. The addition of new core will not change the main principle of the proposed methodology. The core 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. Reader can refer to Ben-Abdallah (2005) for more details about a modular design methodology. One important feature of the above method is that the generic assembling scheme largely increases the architecture modularity. Figure 2.4 shows a typical instance of the platform made of 4 cores (2*QC-2 cores and 2*SH cores). The QC-2 core is a special purpose synthesizable core (described in details in Sect. 2.4.3).

The designer can configure: the number of CPUs, I/O ports for each processor and interconnections between cores, the communication protocol and the external peripherals. The communication interface depends on the core attributes and on the application-specific parameters. The communication interface connects a given core to the communication architecture and consists of two parts: the first part specific to the core's bus and 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 cores from the communication network.

Each interface module acts as a co-processor for the corresponding core. The application dependent part may include several communication channels. The arbitration is done by the CPU-dependent part and the overhead induced by

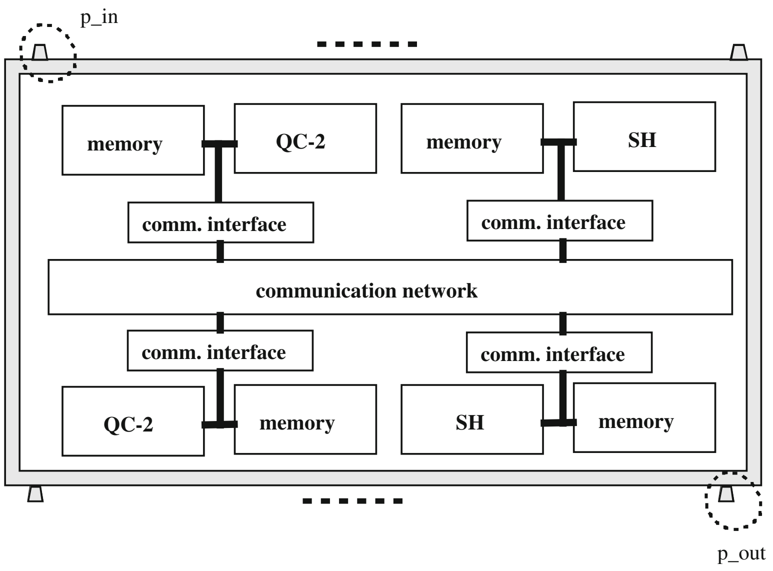


Fig. 2.4 Multicore SoC system platform. This is a typical instance of the architecture, where the addition of a new core will not change the principle of the methodology

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.

2.4.2 Design Method

In this methodology, the application-specific parameters should be used to configure the architecture platform and an application-specific architecture is produced. These parameters are determined from an analysis of the application to be designed. The design-flow-graph (DFG) is divided into 14 *linked-tasks* as shown in Fig. 2.5a, b and summarized in Table 2.1. 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 mapping table (node T6). Node T7 is the internal structure of the

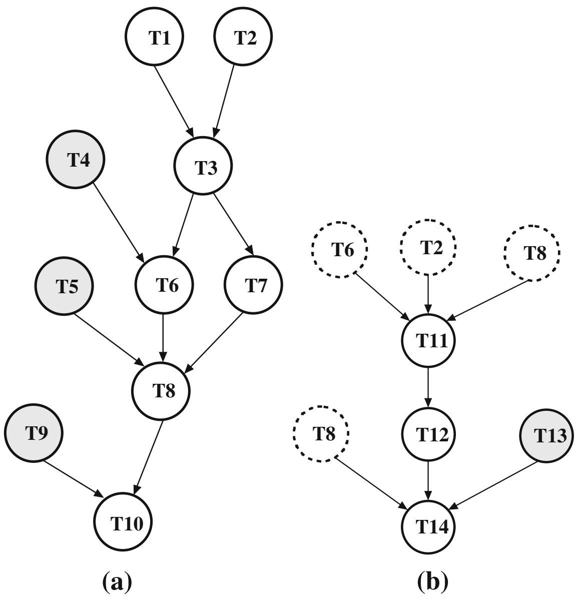


Fig. 2.5 Linked-task design flow graph (DFG). **a** Hardware related tasks, **b** Application related tasks

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.

Table 2.1 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 Memory emulators
T14	H-SoC validation

2.4.3 QueueCore Architecture

The key idea of the produced order queue computation model is the operands and results manipulation schemes (Ben-Abdallah 2004). 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 (Ben-Abdallah 2005). 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 this computing model is its important affect on the instruction issue hardware.

The QC-1 core (Ben-Abdallah 2004) exploits ILP without considerable effort for heavy run time data dependence analysis, resulting in a simple hardware organization when compared with conventional Super-scalar 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 implements all hardware features found in QC-1 core and also supports single precision floating point accelerator.

2.4.3.1 Hardware Pipeline Structure

The QC-2 supports a subset of the produced order queue processor instruction set architecture (Ben-Abdallah 2004). 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. 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 (*source*1) and destination addresses for each instruction. The QCU unit keeps track on the current value of the QH and QT pointers. Four instructions arrive to the QCU unit each cycle. To execute instructions in parallel, the QC-2 core must calculate the operands addresses (*source*1, *source*2 and *destination*) for each instruction. Figure 2.6 illustrates QC-2's next QH and QT pointers calculation mechanism. To calculate the *source*1 address, the consumed operands (CNBR) field (port field) is added to the current QH value (QH0). The second operand address in calculated as shown in Fig. 2.7. 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.

(4) *Barrier*: The major goal of this unit/stage is to insert barrier flags for all barrier type instructions.

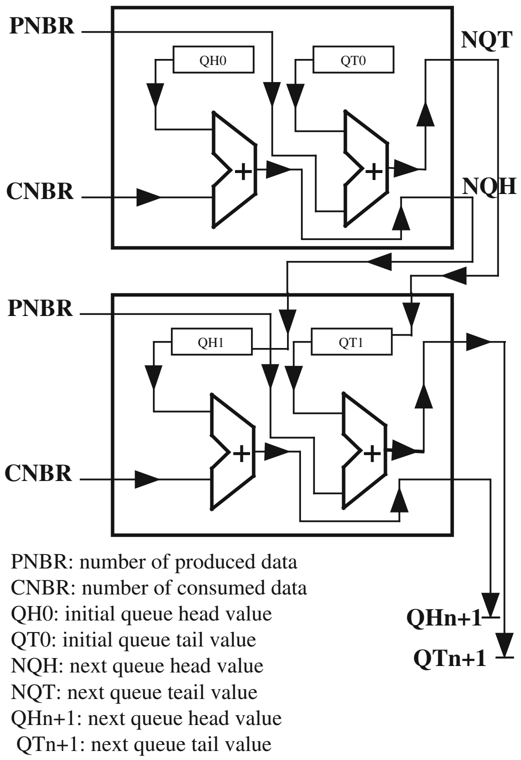
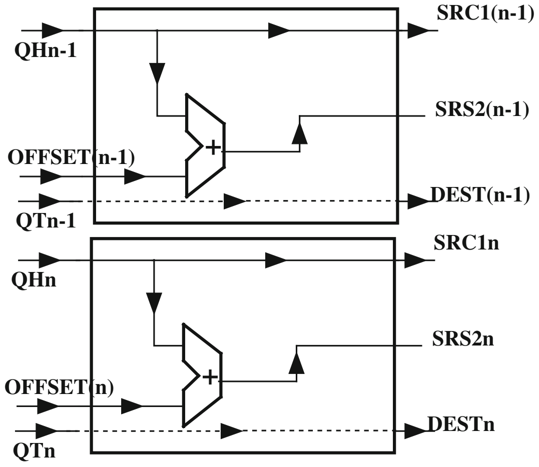

PNBR
NQT
QH0
QT0
CNBR
NQH
PNBR
QH1
QT1
CNBR
PNBR: number of produced data
CNBR: number of consumed data
QH0: initial queue head value
QT0: initial queue tail value
NQH: next queue head value
NQT: next queue teail value
QHn+1: next queue head value
QTn+1: next queue tail value
QHn+1
QTn+1

Fig. 2.6 Next QH and QT pointers calculation mechanism



OFFSET: positive/negative integer value that indiactes the location of SRC2(n-1) from the QH(n-1)
QTn: queue tail value of instruction n
DESTn: destination location of instruction n
SRC1(n-1): source data 1 of instruction (n-1)
SRC2(n-1): source data 2 of instruction (n-1)

Fig. 2.7 QC-2's source 2 address calculation

(5) *Issue*: Four instructions are issued for execution each cycle. In this stage, the second operand (*source*2) of a given instruction is first calculated by adding the address *source*1 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 *source*2 is calculated in this stage.

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 macro-data flow 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.

2.4.3.2 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 (IEEE 1981, 1985). 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.

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 FADD hardware is shown in Fig. 2.8. The exponents of the two inputs (Exponent A and Exponent B) are fed into the exponent comparator,

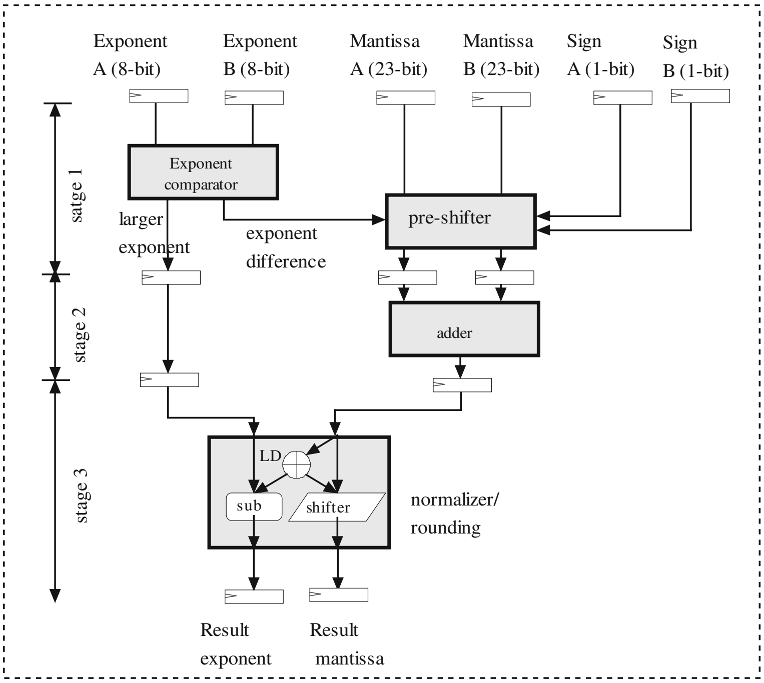


Fig. 2.8 QC-2's FADD hardware

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 pre-shifter 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.

Floating point multiplier implementation: The data path of the FMUL hardware is shown in Fig. 2.9. 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 (Booth 1951) 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.

2.4.4 Performance Analysis

In order to estimate the impact of the description style on the target FPGAs efficiency, logic synthesis for FPGAs are explored. The idea of this experiment was to optimize critical design parts for speed or resource optimizations.

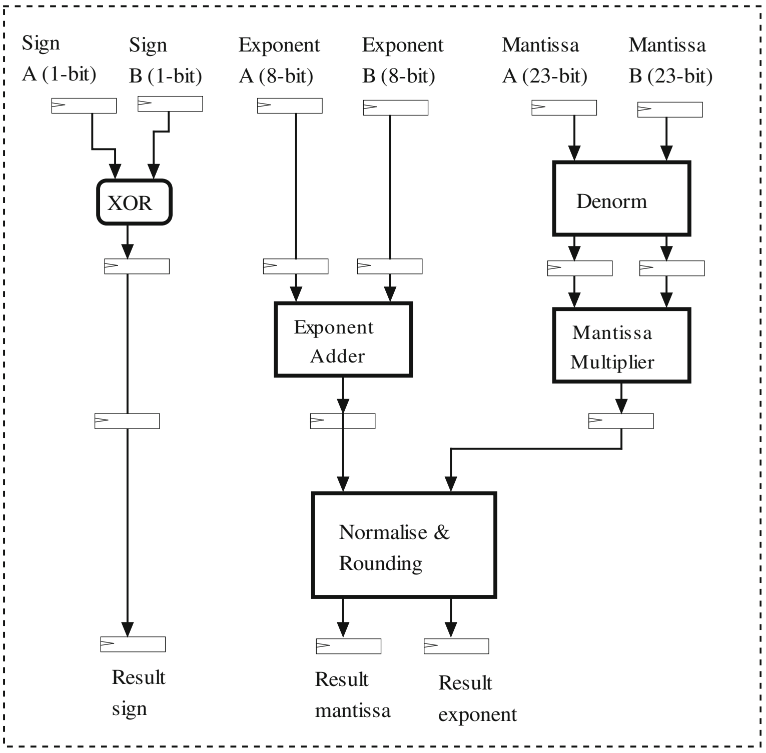


Fig. 2.9 QC-2's FMUL hardware

Optimizing 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 (Micheli 2001; Gohringer 2008). For typical FPGAs features, choosing the right implementation style can cause a difference in resource utilization of more than an order of magnitude (Alsolaim 2000; Xilinx 2009). 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. We selected Stratix FPGAs device because it has a good trade-offs 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

Table 2.2 QC-2 processor design results: modules complexity as LE (logic elements) and TCF (total combinational functions) when synthesized 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	2,573	1,564
Queue compute unit	QCU	1,949	1,304
Barrier queue unit	BQU	9,450	4,348
Issue unit	IS	15,476	7,065
Execution unit	EXE	7,868	3,241
Queue-registers unit	QREG	35,541	21,190
Memory access	MEM	4,158	3,436
Control unit	CTR	171	152
Queue processor core	QC-2	77,819	42,714

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.

Figure 2.10 compares two different target implantations for 256 × 33 QREG for various optimizations. 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 optimizations—area (ARA), speed (SPD) and balanced (BLD). For FPGA implementation, the complexity for SPD optimization is about 17 and 18 % higher than that for ARA and BLD optimizations respectively. Table 2.2 summarizes the synthesis results of 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 optimized for BLD optimization 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. 2.11. 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

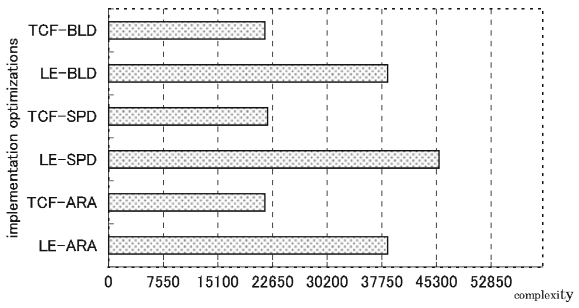


Fig. 2.10 Resource usage and timing for 256 × 33 bit QREG unit for different coding and optimization strategies

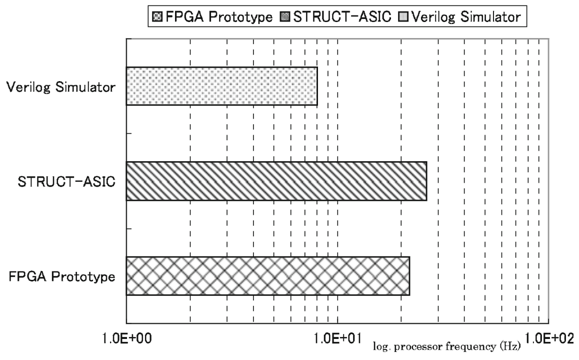


Fig. 2.11 Achievable frequency is the instruction throughput for hardware implementations of the QC-2 processor. Simulation speeds have been converted to a nominal frequency rating to facilitate comparison

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.

2.5 Conclusion

SoC designs have evolved from fairly simple single-core designs to complex multicore SoCs consisting of hundreds of PEs in a single chip. As more and more cores are integrated into these chips, the main challenges lie in how to efficiently and quickly integrate these cores together into a single system capable of leveraging their individual flexibility.

There are two fundamental issues for MCSoC design: (1) design space exploration, and (2) parallel software development. This chapter focused on these two schemes. The chapter also presented a scalable core based methodology for generic architecture model and a synthesizable 32-bit soft core suitable for high performance multicore SoC architectures. The presented GAT method should permit a systematic generation of multicore architecture for embedded multicore SoCs.

Springer

http://www.springer.com/978-94-91216-91-6

Multicore Systems On-Chip: Practical Software/Hardware Design
Ben Abdallah, A.
2013, XXVI, 273 p. 196 illus., 79 illus. in color.,
Hardcover
ISBN: 978-94-91216-91-6
A product of Atlantis Press