SIxD: A CONFIGURABLE AND CUSTOMIZABLE SISD/SIMD MICROPROCESSOR SOFT CORE

by

Nehir Sönmez

B.S., Computer Engineering, Syracuse University, 2003

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2006

SIxD: A CONFIGURABLE AND CUSTOMIZABLE SISD/SIMD MICROPROCESSOR SOFT CORE

APPROVED BY:

Assoc. Prof. Arda Yurdakul . . . . . .
(Thesis Supervisor)

Assist. Prof. Şenol Mutlu . . . . . .

Prof. Dr. Oğuz Tosun . . . . . .

DATE OF APPROVAL: 11.9.2006

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Arda Yurdakul for her guidance and help throughout my research for this thesis and for providing me with all the prerequisites for a research mind and environment. I am also grateful for the helpful advices of Prof Dr. Oğuz Tosun and Assist. Prof. Şenol Mutlu. I would also like to thank my parents for their endless support, my brother Rüzgar, the Pokemon master, for sharing the computer with me, my friends for always being there when I needed a break and Zeyno for being so close and so patient.

This thesis has been supported by TUBITAK, the Turkish Foundation of Science and Technology, under Kariyer Research Project Program, under project number 104E038. It has also been supported by Boğaziçi University Scientific Research Project, under project number 06M105.

ABSTRACT

SIxD: A CONFIGURABLE AND CUSTOMIZABLE SISD/SIMD PROCESSOR SOFT CORE

The demand for FPGA-based processor cores increases as more embedded systems are built on FPGA platforms. The flexible choice is the “soft” processor IP core, a processor implemented in the reconfigurable logic of the FPGA. Commercial and academic soft processors have been widely deployed, but most are synthesized implementations of legacy instruction sets that fill up large and costly FPGAs. With high performance media processing applications dominating the embedded scene, and many modern microprocessors adopting the SIMD technology, it is a fact that soft cores could also make use of array and vector processing functionality.

This thesis presents the SIxD, a configurable CPU soft core designed to combine computer architecture basics to exploit instruction level parallelism with the flexibility and customizability advantages of soft cores realized on reconfigurable fabric. With run-time configuration options such as variable data space, customizable instruction set, and array processing capabilities, the SIxD is a novel soft core that can be configured to fit in as low as a forty thousand system gate FPGA, or offer higher performance array processing on bigger FPGAs.

ÖZET

SIxD: YAPILANDIRILABİLİR VE UYARLANABİLİR BİR TKTV/TKÇV YUMUŞAK MİKROİŞLEMCİ ÇEKİRDEĞİ

Gömülü sistemlerin Alan Programlamalı Kapı Dizileri (APKD) içinde tasarlanması arttıkça, bu ortamlar için özel olarak tasarlanmış mikroişlemci ihtiyacı da artmaktadır. Mevcut seçeneklerden esnek olanı, "yumuşak" Entelektüel Mülk mikroişlemci çekirdeği, yani mikroişlemcinin APKD'nin yeniden programlanılabilir mantık kapıları içerisinde gerçekleştirilmiş olanıdır. Günümüzde ticari ve akademik yumuşak işlemciler yaygın olarak kullanılsa da, bunların çoğu eski ve halihazırdaki komut kümelerinin sentezlenmesiyle oluşmuş, pahalı ve büyük APKD'lere sığan uygulamalardır. Yüksek başarımlı görüntü ve ortam işleme uygulamalarının gömülü sistemler piyasasına hakim olması ve günümüz işlemcilerinin TKÇV (tek-komut, çoğul-veri yolu) özelliğini benimsemesi ile birlikte, yumuşak mikroişlemci çekirdekleri de dizi ve yöney işleme kabiliyetinden yararlanabilmelidirler.

Bu tez, komut düzeyi paralelliği ile yapılandırılabilir yumuşak mikroişlemci çekirdeklerinin esneklik ve uyarlanabilirliğinden yararlanarak tasarlanmış SIxD'i sunuyor. Değişken veri alanı, özelleştirilebilir komut kümesi ve veri dizileri işleme kabiliyetiyle SIxD kırk bin dizilik APKD'lere sığabilen, veya veri dizileri işleme seçeneğiyle daha büyük APKD'lerde daha yüksek performans gösterebilen yeni bir yumuşak çekirdekli işlemcidir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTS OF SYMBOLS/ABBREVIATIONS

mW	Milliwatt
ns	Nanosecond
μs	Microsecond
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
CAD	Computer-Aided Design
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
CPU	Central Processing Unit
DIT	Decimation in Time
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
I/O	Input/Output
IP	Intellectual Property
LUT	Look-up Table
MAD	Motion Activity Descriptors
MPEG	Motion Pictures Experts Group
NoC	Network-on-Chip
PE	Processing Element
RAM	Read Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read-only Memory
SIMD	Single Instruction – Multiple Data
SISD	Single Instruction – Single Data
SIxD	Single Instruction – Variable (Single/Multiple) Data
SoC	System-on-Chip
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word

1. INTRODUCTION

Each day, our lives become more affected by embedded systems, the digital information technology embedded in our environment. This does not only include consumer electronics such as cellular phones, digital cameras and printers, home appliances like microwave ovens and washing machines, business equipment, but also safety-critical applications such as automotive devices and controls, aircraft and medical devices, enough to say that nearly any device that runs on electricity either already has, or will soon have, a computing system embedded within it.

More than 98% of processors applied today are in embedded systems, and are no longer visible to the customer as computers in the ordinary sense. New processors and methods of processing, sensing, actuating, communications and infrastructures enable this pervasive computing. As a result, billions of embedded systems units are produced yearly, versus millions of desktop units: there are perhaps 50 per household and per automobile. All of these have wide-ranging impacts on society, including security, privacy and modes of working and living, forming the basis for a significant economic trend.

The evident design goal for embedded systems is to construct an implementation with desired functionality, simultaneously optimizing the measurable features of a system's implementation, called design metrics. Common design metrics of embedded systems are [1]: size, performance, power, the non-recurring engineering cost, the unit cost, flexibility, the time-to-prototype (until a working version of the system), the time-to-market, maintainability, correctness and safety. Optimizing design metrics is a key challenge, as they typically compete with one another. Improving one metric often leads to degradation in another; i.e. reducing the size may cause the performance to suffer. embedded systems are usually tightly-constrained for low cost, low power, small size and high performance; they often must cost just a few Euros, must be sized to fit on a single chip, must perform fast enough to process data in real-time, and must consume minimum power to extend battery life.

Field-Programmable Gate Arrays (FPGA) are the first configurable fabrics that have been initially designed for rapid prototyping of digital complex systems. Later on, reconfigurable devices have been used as co-processors to speed up computations. In these implementations, either computation-intensive tasks or user-defined instructions are mapped on fabric in compile-time, [2-5] or frequently used loops and tasks are mapped on reconfigurable fabric during run-time [6].

Although programmable logic saves development cost and time over increasingly complex ASIC designs, FPGAs have started becoming popular over the past decade as the unit price has dropped and gate count per chip has reached numbers that allow for the implementation of more complex applications and soft cores [7-9]. A soft processor is one that is implemented using the FPGA logic, rather than being a fixed part of the chip circuitry, providing the flexibility to implement and potentially customize processors as required. The register-rich nature of FPGA chips do not solely consist of an array of look-up tables and flip-flops, but now include on-chip RAMs and hardware multipliers making them perfect candidates for processor design. Soft cores can also support custom instructions and functional units, and can be reconfigured to enhance system-on-chip (SoC) development. FPGA systems offer high integration and easy field updates of entire systems. Several companies sell FPGA CPU cores, but most are synthesized implementations of existing instruction sets, filling huge, expensive FPGAs, and are too slow and too costly for production use.

The architecture of these processors is usually SISD (Single Instruction-Single Data). Xtensa [5] is a SISD architecture whose instruction set can be enhanced by including user-defined application-specific instructions, as well as by including a SIMD (Single Instruction-Multiple Data) engine for DSP applications. Many modern commercial microprocessors have the SIMD capability [10-14] as an integral part of the Instruction Set Architecture.

In the literature, there are processor architectures that incorporate both SISD and SIMD in a single core using partitioned data words [15,16], or use VLIW techniques to exploit Instruction Level Parallelism [17], however no cases of a completely parameterizable SIMD core have been observed.

This thesis presents a novel processor soft core, the SIxD, a non-pipelined RISC system (load/store architecture) with fixed 16-bit instruction word and variable data space. Depending on the size of the FPGA, the user can select the operation mode (SISD or SIMD) and shrink, extend or modify the instruction set for including application-specific instructions or excluding redundant instructions at compile-time. The SIxD is a quite novel soft core that can fit in as low as a 40 thousand system gate FPGA, or offer high performance array processing on bigger FPGAs. The core is designed with VHDL for Xilinx FPGAs, but modifications to fit FPGAs of other vendors can easily be done.

This thesis is organized as follows. Section 2 provides background information on embedded systems, reconfigurable computing, the FPGA architecture and soft cores on FPGAs. In Section 3, three experimental non-configurable soft cores have been developed, having the Fast Fourier Transform as a custom instruction. Section 4 explains the SIxD system in detail. An illustrative implementation concerning the MPEG-7 Motion Activity Descriptors is explained in Section 5, with conclusions, future work and a method for dynamic runtime reconfigurability of the SIxD presented in Section 6.

2. BACKGROUND

In this section, the relevant background required to understand this work is discussed. Section 2.1 presents the uses of Reconfigurable Computing for application performance and flexibility, while Section 2.2 explains the architecture and the benefits of the Field Programmable Gate Array, a canonical example of a configurable device. Section 2.3 gives an overview of Intellectual Property cores and compares the advantages and trade-offs of soft, firm, and hard cores. Finally, Section 2.4 discusses about the architectures of several academic and commercial soft processor cores.

2.1. Reconfigurable Computing

There are two primary methods in traditional computing for the execution of algorithms. The first is to use an Application Specific Integrated Circuit (ASIC), to perform the operations in hardware. Because these ASICs are designed specifically to perform a given computation, they are very fast and efficient when executing the exact computation for which they were designed. The biggest disadvantage with ASICs is inflexibility: the circuit cannot be altered after fabrication, so modification causes the need for redesign [18]. The second method is by using microprocessors which execute a set of instructions. By changing the software instructions, the functionality of the system is altered without changing the hardware. However, this time the performance suffers, and is worse than of an ASIC due to the fact that the processor must read each instruction from memory, decode it and determine its meaning, and only then execute it. This results in a high execution overhead for each individual operation. Additionally, the set of instructions that may be used by a program is determined at the fabrication time of the processor. Any other operations that are to be implemented must be built out of existing instructions.

Today's general-purpose processors are highly optimized for executing complex sequences of instructions for a popular set of basic operations. Almost by definition, these processors are good at executing average application workloads. Nevertheless, most computationally complex applications spend 90 per cent of their execution time in only 10 per cent of their code. Many algorithms contain such critical kernels whose performance

significantly impacts total application performance, and which are unlikely to be perfectly implemented by any processor, making the idea of a fast, yet general purpose CPU seem inconsistent. Reconfigurable hardware may be better at supporting such kernels for two reasons: at implementing functions that do not map well to the standard set of operations, and hard-coding the control flow within the reconfigurable array logic, avoiding instruction bandwidth bottlenecks and thus providing more potential to exploit parallelism.

Reconfigurable computing is intended to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware, therefore emerging as the new paradigm for satisfying the demand for application performance and flexibility. The ability to customize the architecture to match the computation and the data flow of the application has demonstrated significant performance benefits compared to general purpose architectures. The term configurable is used to refer to architectures where the active circuitry can perform any of a number of different operations, but the function cannot be changed from cycle to cycle. FPGAs (Field-Programmable Gate Arrays) are canonical examples of a configurable device. Once the instruction has been “configured” into the device, it is not changed during an operational epoch. The functional computation on the piece of hardware is defined by a set of configuration bits which tell each gate and wire (interconnect) how to behave. FPGAs can perform any computational task that fits in the machine’s finite state and computational capacity that is set by its operational resources.

Many researchers have used the FPGA technology to make computing applications faster. The performance achieved by these configurable machines is often one or two orders of magnitude greater than processor-based counterparts. Configurable computers are proven to be the fastest in fields such as RSA decryption, DNA sequence matching, signal processing, microprocessor emulation and cryptography [19].

2.2. FPGA Architecture

A typical FPGA (Field Programmable Gate Array) consists of an array of programmable/configurable logic blocks distributed across the entire chip in a sea of

programmable interconnections, with the entire array surrounded by programmable Input/Output (I/O) blocks, as depicted on Figure 2.1.

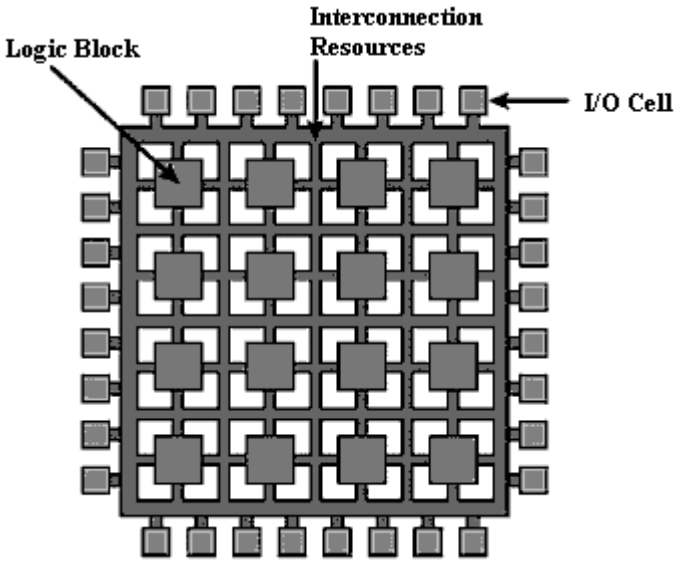


Figure 2.1: Generic FPGA architecture [20]

A logic block is an FPGA unit of area that includes one N-input lookup table (LUT) and one D flip-flop. N-input LUT is basically a memory that, when programmed appropriately, can compute any function of up to N inputs. Configurable logic blocks (CLB), which are made up of groups of LUTs (called slices), are organized in an array on the FPGA and are used to build combinatorial and synchronous logic designs. Each CLB element is tied to a switch matrix to access the general routing matrix. Each CLB also has carry logic to help build fast, compact ripple-carry adders and multiplexers to help cascade LUTs into larger logic structures. Each Input/Output Block (IOB) offers input and output buffers and flip-flops. The output buffer can be 3-stated for bidirectional I/O. The programmable interconnect routes CLB/IOB output signals to other CLB/IOB inputs. It also provides low-skew clock lines, and horizontal long lines which can be driven by 3-state buffers near each CLB. Although LUT and CLB are names given by Xilinx [21], one of the biggest commercial FPGA vendors, and Altera [22] calls these structures Logic Elements and Logic Array Blocks, the structures of the FPGAs are quite similar.

On an FPGA, all layers already exist on the chip and connections are either created or destroyed to implement desired functionality with the aid of Computer Aided Design (CAD) tools. This way, the designers themselves have the advantage to create an IC. Although there are benefits like having a very low NRE cost and a high time-to-market period, it also presents some drawbacks such as the high cost of unit, high power consumption, and slower performance.

Many FPGA vendors now provide FPGA chips with dedicated Block RAM area as well as hardware multipliers and dividers, which prove to be extremely useful in soft processor core design.

2.3. Intellectual Property Cores

Introducing a new design process called core-based system design, embedded devices often come in the form of a System-on-Chip (SoC) in order to keep production and development expenses as low as possible, while complying with various design constraints. A SoC is a concept that integrates the use of pre-designed, pre-verified, re-usable silicon circuitry, called Intellectual Property (IP) cores, to be used as building blocks for large and complex applications on an Integrated Circuit (IC). So, rather than developing every sub-system from scratch, the system is composed by integrating various cores, including the re-use of previously deployed ones (Figure 2.2). SoCs have altered the way commercial, off-the-shelf components are sold: as IP, rather than actual IC. Examples of common IP cores range from a thousand-gate analog circuit blocks to memory controllers, peripheral devices such as MAC (Machine Authentication Code) Ethernet, UART (Universal Asynchronous Receiver / Transmitter), or PCI (Peripheral Component Interconnect) bus controllers, to hundred thousand-gate processor cores.

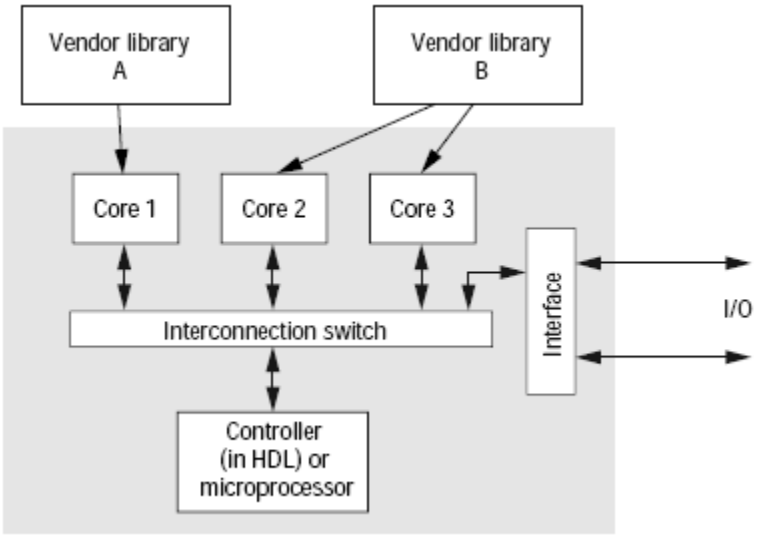


Figure 2.2. Generic system-on-chip architecture [23]

IP cores are processor-level components with behavioral, structural, or physical descriptions, splitting into three primary categories: soft, firm, and hard cores. A *soft core* consists of a synthesizable HDL (Hardware Description Language) description that can be retargeted to different semiconductor processes. HDLs such as VHDL (Very High Speed Integrated Circuit Hardware Description Language) and Verilog increase the range of options available to designers by enabling hardware implementation with the flexibility that language-based design provides, allowing designers to implement flexible Intellectual Property cores. *Firm IP cores* have a higher level of optimization and are often targeted for a specific architecture or device, containing more structure than a soft one, commonly a gate-level netlist that is ready for placement and routing. Firm IP cores traditionally have a mix of higher performance and more efficient resource utilization than soft core implementations of the same functionality, given that these structural cores take advantage of specific device and architecture features, but on the other hand, they are less portable. Soft and firm cores share many common properties and will be considered under the soft core category throughout this thesis.

Hard IP cores are physical descriptions that involve the implementation of a silicon-level circuit within the device fabric. A hard core includes layout and technology-dependent timing information and is ready to be dropped into a system. These components are products of technology, software, and expertise that are subject to patents and

copyrights. Depending on the design constraints, the designer can use the already optimized and synthesized hard cores, or to comply with the constraints, adapt the soft or firm cores to specific limitations.

Because soft cores are not implemented, they are inherently more flexible in function and implementation than hard cores. On the other hand, hard core developers can afford to spend more time optimizing their implementations to be used in many designs. For a SoC that requires the highest performance in current process and design technology, a full-custom hard core is better at meeting these needs by using latches, dynamic logic, 3-state signals and custom memories. However, if the performance target is within the range of a soft core, then the performance goals can be met with a soft core while taking advantage of its inherent flexibility [24].

Soft cores are also technology independent. The high-level Verilog or VHDL does not require the use of a specific process technology or standard cell library, serving the IP core for multiple designs, or for future generations of the current design. A hard core, on the other hand, is very technology-specific. Hard cores can be ported to new process technologies, but the effort to re-optimize full-custom cores is significant and costly. In reality, soft cores are likely designed with one technology and library in mind. The design itself is independent of this choice of technology but it is optimized for a certain technology and library.

A hard core, especially if it was implemented using full-custom techniques, will look like a black-box RAM. This causes difficulties in running gate-level functional simulations on a full-custom hard core which may not have a gate-level netlist, because the design has been done at the transistor level, and gates were not used [25]. Many FPGAs such as the Xilinx Virtex-II Pro and Altera Excalibur are equipped with hard processor cores, namely the PowerPC and the ARM (Advanced RISC Machines), respectively. While the internal hard processors can be fast, small, and relatively cheap, they have several drawbacks. Inclusion of one or more hard processors specializes the FPGA chip, due to the fixed location of each FPGA-based hard processor, it can be difficult to route between the processors and the custom logic. Besides, the performance requirements of each processor in the application may not match those provided by the available FPGA-based hard

processors, and the number of hard processors included in the FPGA device may not match the number required by the application, leading to either too few or wasted hard processors.

Hard cores are optimized once, when they are implemented by the IP provider. Because the core is optimized only once, the IP provider can afford to spend significant resources. Thus, a hard core will typically run faster than a comparable soft core for that one technology in which it is implemented. But, even in that single technology, it is only optimized for one set of goals. If the goal is low area at reasonable performance, the highly tuned performance-optimized hard core may be too large for the application. Soft cores, on the other hand, can be application-optimized: Timing, area and power targets can be adjusted to fit the specific embedded SOC design. For instance, if a SoC uses a 200 MHz clock, then a soft IP core that was designed to run at 250 MHz can be targeted to run at exactly 200 MHz instead. This allows for smaller area and lower power while still meeting the design constraints. If a SoC's speed, area and power targets are exactly what the hard core was targeted for, then that hard core will be viable, however, for the great majority of designs, a soft core will be better optimized for that particular SoC.

Soft cores offer another important advantage over hard cores: compile-time customizations. Custom instructions are algorithm-specific additions of hardware to the soft-core microprocessor's execution pipe, alongside the ALU. These new hardware instructions are used for a time-critical piece of an algorithm, turning the software algorithm into a hardware block. A RISC microprocessor with custom instructions blurs the division between RISC and CISC, because the custom instruction units can be multi-cycle hardware blocks doing quite complex algorithms, but they are embedded in a RISC processor with standard single-cycle instructions. Furthermore several custom instructions can be added to an ALU, (hence questioning the "reduced instruction set") limited only by the FPGA resources and the number of open positions in the soft-core processor's opcodes. The most efficient use of custom instructions occurs when the algorithm to be accelerated operates on data stored in local registers and is a frequently called, relatively atomic operation [26].

Cache memory size is a common compile-time customization. A soft-core processor can be configured for exactly the cache size needed by the specific embedded application. A hard core, on the other hand, cannot be customized in this way. Another customization employed in many soft cores is support for custom instructions. For example, support for external coprocessors may be optionally included if required by the SOC. Special hardware to enable instruction code compression may also be needed in some systems. However, in the systems that do not use these features, the extra hardware could be removed in a soft core, saving area and power. Soft cores may also include implementation configuration parameters. These are special kind of compile-time customizations that help the soft core better match the design requirements.

2.4. The Architecture of Processor Soft Cores

All of the benefits and characteristics of soft IP cores are realized by soft processor cores implemented within FPGA components. Two common soft processor core examples are the Xilinx Microblaze [7] and Altera's Nios II [9]. Both soft processor cores are 32-bit Harvard bus architecture (separate data and instruction memories) Reduced Instruction Set Computer (RISC) systems with 32 general-purpose registers. The Microblaze has a three-stage fetch-decode-execute pipeline, whereas the Nios II has a 5-stage pipeline. The MicroBlaze can operate at 100-200 MHz depending on the FPGA, taking up between 900 and 2,600 Xilinx Look-Up Tables (LUTs), depending on how the processor is configured. The Nios II is able to operate at 150-200 Mhz, consuming between 700 and 1800 Altera Logic Elements (LE). It should be realized that area occupation and performance are completely dependent on the application, target architecture and the speed grade. The Nios II is compatible with Altera Cyclone and Stratix family devices, whereas the Microblaze can be placed on Xilinx Virtex-II, Spartan-II and Spartan-3 FPGAs. Both cores have an address space of 32 bits, handling up to 4 GByte of instructions and data memory, and support for both on-chip Block RAM and external memory. Both cores use memory-mapped I/O, and support word, halfword, and byte accesses to data memory. The performance for NIOS II is 31-218 Dhrystone MIPS (Million Instructions per Second), for Microblaze, it is 92-166 DMIPS.

It is common for commercial soft core vendors to include standardized buses and peripherals on their systems. SoC design includes the incorporation of many cores obtainable from third party vendors, which are designed compatible with soft IP cores, through such standardized buses. On the Microblaze, all peripherals including the memory controller, UART and the interrupt controller run off of the standardized OPB CoreConnect bus. MicroBlaze can be configured with up to eight Fast Simplex Link (FSL) interfaces, which are dedicated unidirectional point-to-point data streaming interfaces each consisting of one input and one output port. The Nios II core has the Avalon Switch Fabric wrapping all peripherals, similar to the CoreConnect. For accesses to memory, the Nios has an interface to fast external memory as well as two master ports for instruction and data, and two optional internal caches for both these memories. Both IP cores also contain an optional Floating Point Unit (FPU) supporting 32-bit IEEE 754 floating point addition, subtraction, and optional division, multiplication and comparison instructions.

For the hardware multiply or the hardware divide units, there are three possible implementation options. The first is to include embedded hardware multipliers in the arithmetic logic unit (ALU). This is usually the choice when targeting devices that have embedded multipliers. The second option is to include LE/LUT-based multipliers in the ALU, for achieving multiplication without consuming hardware multiplier resources. This, of course is a slower option. The third option would be to omit the hardware multiply and emulate the multiplication in software instead. All these choices are also valid for hardware divide. Experiments [27] show that a processor with the multiplier implemented using look-up tables required more area, more energy per cycle, and reduced clock frequency over the same processor with multiplication implemented in the dedicated multipliers.

Both the Microblaze and the Nios II ISA consist of six bit opcodes and five bit register addresses. Both register-immediate type instructions contain an opcode, one destination and one source registers, and a 16-bit immediate value. Register-register type operations contain an opcode, one destination and two source registers, whereas the Nios II ISA has an 11-bit opcode-extension field included for an immediate value. The J-type in Nios II is used to jump to subroutines and has a 6-bit opcode field and a 26-bit immediate data field.

Another soft core that requires attention is the Xilinx PicoBlaze [8], a compact 8-bit RISC microcontroller core optimized for the Xilinx Spartan-3, Virtex-II, and Virtex-II Pro FPGA families, occupying only 96 Xilinx FPGA slices. With 16 8-bit wide registers, an 8-bit ALU responsible for ADD, SUB, AND, OR, XOR, compare, shift, and rotate operations, a single FPGA Block RAM storing up to 1024 program instructions, an internal general-purpose 64-byte RAM, the PicoBlaze can perform 44 to 100 MIPS. The microcontroller also supports up to 256 input ports and 256 output ports or a combination of input/output ports. The basic functionality of the microcontroller can be extended and enhanced by connecting additional FPGA logic to the microcontroller's input and output ports. The microcontroller is useful for control and state machine instructions, and its biggest advantage is its low resource use. The main disadvantage is that it executes instructions sequentially, so the performance degrades with increasing complexity. The program memory requirements also increase with increasing complexity and the response to simultaneous inputs are slower. There are also no hardware multiplier or divider units, and these functions have to be performed using available arithmetic and shift instructions.

The Nios II has three implementation options: for minimal core size (non-pipelined), small core size, and fast execution speed. Features on the Nios II can be added or removed on a system-by-system basis to meet performance or price goals. Designers can also create their own custom peripherals and integrate them into Nios II processor systems. Like custom peripherals, custom instructions are a method to increase system performance by enhancing the processor with custom hardware. The soft-core nature of the Nios II processor enables designers for custom instruction extension, as shown in Figure 2.3, to integrate custom logic into the arithmetic logic unit (ALU). Similar to native Nios II instructions, custom instruction logic can take values from up to two source registers and optionally write back a result to a destination register.

The Fast Simplex Link (FSL) of Microblaze helps customize the processor in a different way. Each FSL provides a 32 bit low latency dedicated interface to the processor pipeline, which can be used for extending the processor's execution unit with custom hardware, providing custom acceleration as a co-processor (Figure 2.3). The *get* instruction in the MicroBlaze ISA is used to transfer information from an FSL port to a general purpose register, the *put* instruction is used to transfer data in the opposite direction [7].

This method is similar to extending the ISA with custom instructions, but has the benefit of not making the overall speed of the processor pipeline dependent on the custom function. Also, there are no additional requirements on the software tool chain associated with this type of functional extension, such as modifications on the control unit. An FSL-based coprocessor can take an arbitary number of inputs and computation can be done in parallel with the Microblaze, without occupying the shared bus.

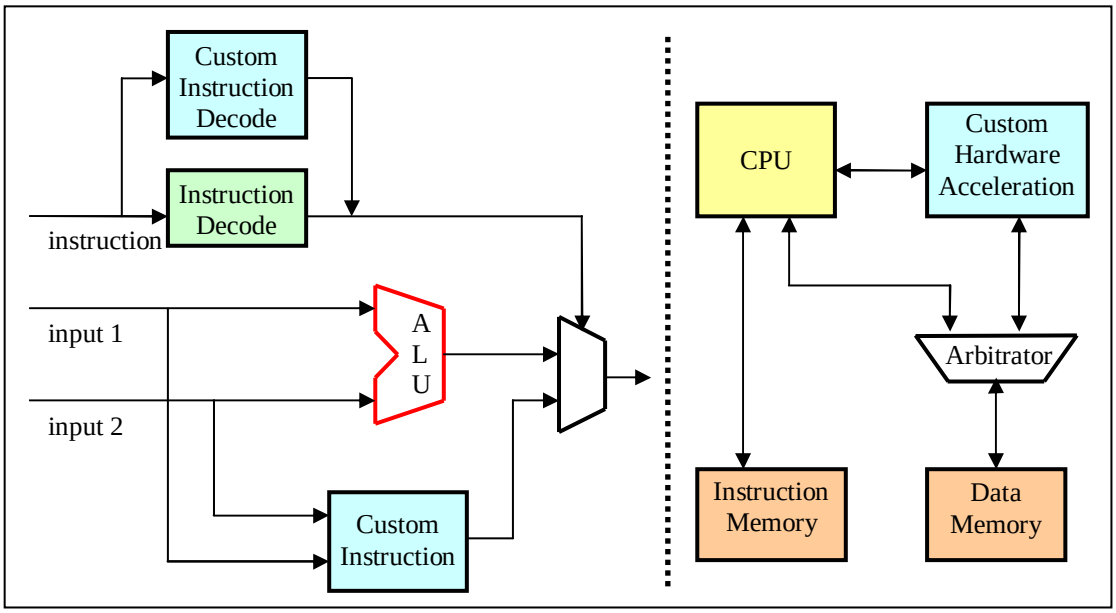


Figure 2.3. Custom instruction extension vs. custom acceleration as a co-processor

The Xtensa Processor Core [5] is a configurable, extensible and synthesizable processor core, designed to enable designers for tailoring each implementation to match the application requirements for the target SoC. Using Xtensa, the processor can be molded to fit the application by selecting and configuring predefined elements of the architecture and by inventing completely new instructions and hardware execution units that can deliver high performance levels. Xtensa has four types of functional blocks:

- "Configurable" function blocks are elements parameterized by the system designer.
- "Optional" function blocks indicate elements available to accelerate specific applications.
- "Optional and Configurable" blocks are optional elements scalable to fit applications, including peripherals.

- "Advanced Designer-Defined" functions are hardware execution units and registers added to the processor by the designer to accelerate specific algorithms for a given design.

Common in all configurations is the base 32-bit Instruction Set Architecture with a 32-bit ALU, 16- or 24-bit RISC instruction encoding, up to 64 general-purpose physical registers, 6 special purpose registers, a 5-stage pipeline and 80 base instructions, which is a superset of traditional RISC instruction sets. The Xtensa configuration parameters include Instruction/data memory size, Size of windowed register file, External bus width, Number of interrupts, Interrupt levels, timers and memory order.

The Tensilica Instruction Extension (TIE) language is a Verilog-like language used to describe new instructions, new registers and execution units, additional data-types, and new I/O ports that are automatically added to the Xtensa processor. Program optimization using TIE is first accomplished by identifying "hot spots", which are performance-sensitive regions of application software [28]. By utilizing an execution profiler, the efficiency of an application program can be analyzed and evaluated where TIE can be used to accelerate the performance of the software. A sequence of processor builds can be iterated while profiling the new processors to compare the benefits of adding instructions and TIE hardware. Aggressive use of parallelism and other techniques can often deliver performance increases using the new TIE instructions.

The Xtensa LX processor [29] implements Tensilica's FLIX (Flexible Length Instruction Xtension) architecture. FLIX is a configuration option that allows designer-defined instructions to consist of multiple, independent operations bundled into a 32-bit or 64-bit instruction word. Wide 32-or-64-bit FLIX instruction formats are intermixed with the base Xtensa ISA's existing 16/24-bit instructions without a mode switch penalty to utilize a FLIX instruction. This performance increase comes with very little overhead, adding 2,000 gates to the size of the processor for instruction decode and control. The FLIX supports data parallelism with SIMD operations using vectors of length 2, 4, 8, and 16, and of width up to 256 bits. There is support for unaligned vector loads and stores, and vector operations including MIN, MAX, ABS, and user-defined operations [30].

While commercial soft cores are commonly deployed, soft core generation and automation is a very active academic research area. Previous research work has been done in attempts to understand and generate soft processor core microarchitecture. Yiannacouras [27] presents an infrastructure for rapidly generating RTL (Register Transfer Level) models of soft processors, as well as a methodology for measuring their area, performance, and power. Plavec [31] developed a replica of the Altera's Nios to be able to closely examine its characteristics. The CPUgen [32] generates customizable RISC CPU cores, allowing customization of address/data/instruction bus size, interrupt handling, indirect addressing, data/instruction latency timings and custom instructions definition.

Two examples of academic open architectures featuring pipelined RISC systems are the F-CPU [15] and the XC16 [33]. The XC16 is a simple processor with 16-bit instruction word that runs C code, compiled by some alterations on the LCC retargetable compiler [34] fitting in less than 400 Xilinx slices. The F-CPU architecture defines a SIMD, superpipelined, 64-bit RISC microprocessor with 64 registers. The priority on the OpenUP project [35], 8-bit FPGA-based CISC microprocessor with 12 bit addressing and 32 registers, is the quick development of applications, as opposed to the quick execution of instructions. The architecture having 8 or 16-bit instruction word and 33 instructions in all, also has external memory access and a DMA channel fitting on 200 Xilinx CLBs. Another open extensible processor is the OPENRISC 1400 [36], a free superscalar 64-bit flexible RISC system featuring different instruction set extensions: a basic Instruction Set, a Floating-Point extension and a Vector/DSP extension, with 32 bits wide instructions aligned on 32-bit boundaries in memory and operating on 8, 16, 32 and 64 bits data.

Not all soft cores implement assembly language instructions. Lavacore [37] is a 32-bit configurable, application-specific, direct execution Java processor for Xilinx Virtex devices, which executes Java Virtual Machine bytecode in hardware, eliminating the need for software-based interpreters or code translators. Another non-RISC architecture is the LEON [38] which implements a 32-bit processor conforming to the SPARC V8 architecture. The LEON processor has separate instruction and data caches, hardware multiplier and divider, interrupt controller, timers, UARTs, watchdog, I/O port and a flexible memory controller. Additional modules can easily be added using the on-chip buses.

Hundreds of processor soft cores have been designed, all with different architectures, functionalities, advantages and drawbacks, with no core completely overpowering the other. Depending on the target application, a simple microcontroller core might be enough, and a full extended commercial soft core would be superfluous. The general-purpose core concept has to make the best use of flexibility in order to fit well for different kinds of applications with different constraints.

3. ARCHITECTURAL EXPLORATION ON SOFT CORE CUSTOMIZATION

As discussed in the previous section, the option of custom unit inclusion to the Instruction Set Architecture is a major contribution to compile-time customizations in soft core processor design. This chapter presents three different architectures of FFT-customized soft cores, namely the SISD, the SIMD and the NoC systems. The FFT was chosen as the specific application because it is a commonly used DSP algorithm that contains a simple atomic operation, namely the FFT butterfly, which is used as the application-specific custom functional unit to the processor architectures. The inclusion of the butterfly unit to the general-purpose ISA provides to be very useful in fast FFT computation, combining many atomic operations into one.

First, the toy algorithm, 8-point-FFT is explained on section 3.1. The first soft core, on Section 3.2, is the SISD system, a simple non-pipelined RISC processor. On Section 3.3, the shared-bus SIMD machine, which was developed using the SISD as a host computer to an array computer is explained. The final architecture explained on section 3.4 is called the NoC, a multiprocessor with a set of on-chip routers, again based on the original SISD.

3.1. Explored Architectures

3.1.1. The Basic Uni-processor

A typical uni-processor consists of an Arithmetic Logic Unit (ALU), memory and a control unit that manages the flow of operation. A CPU basically fetches, decodes and executes. The requested instruction word is brought (fetched) from memory on to the registers of the ALU. Following the execution of the requested operation, which is always a portion of the instruction word, the produced result is saved back, either to memory or one of the registers, ready for possible use in the next operation. The control unit, which manages the memory-ALU duo, is responsible for the complete movement of bits inside the CPU. The set of control bits are used to drive every single component of the processor.

The memory unit of Von Neumann architecture can be thought of in two parts, namely the instruction memory, and the data memory. Obviously, the data memory holds the data to be processed, while the instruction memory holds the instruction program to be run, using the data. Internal registers to keep the program address, the instruction word, the data address and the data word are also necessary. The instruction decoder is responsible for taking the instruction in the IR and decoding it into control bits. A control unit decides on the functionality of each component according to these bits.

3.1.2. Exploiting the Instruction Level Parallelism with SIMD

The use of the data-parallel approach to exploit instruction level parallelism (ILP) is now very commonly used in microprocessor technology. Starting a decade ago with Intel's MMX (Multimedia Extensions) [10], then later moving on to PowerPC's AltiVec [11], Sun's Sparc VIS [12], AMD's 3DNow! [14], and Intel's SSE (Streaming SIMD Extensions) [13], many modern commercial microprocessors have the SIMD capability, which has become an integral part of the Instruction Set Architecture.

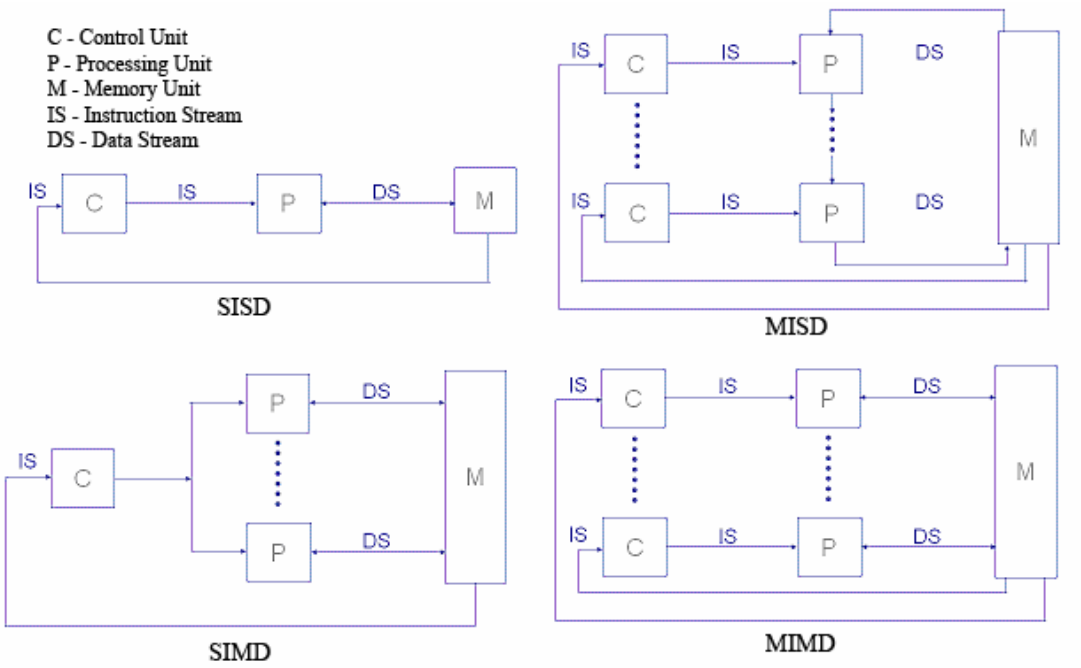


Figure 3.1. Flynn's classification

The idea of using multiple processors both to increase performance and to improve availability dates back to the earliest electronic computers. On 1972, Flynn proposed a

simple model of categorizing all computers that is still useful today [39]. He looked at the parallelism in the instruction and data streams caused by the instructions at the most constrained component of the multiprocessor, and placed all computers in one of four categories:

1) SISD (Single Instruction Stream, Single Data Stream): In classical Von Neumann uni-processor computer, a single stream of instructions is generated by the program. The instructions operate on a single stream of data items. The algorithms for SISD computers do not contain any parallelism.
2) SIMD (Single Instruction Stream, Multiple Data Stream): A specially designed computer in which a single instruction stream is from a single program, but multiple data streams exist. The instructions from the program are broadcast to more than one processor. Each processor executes the same instruction synchronously, but using different data. The processors operate synchronously and a global clock is used to ensure lockstep operation. SIMD is a data-parallel architecture. Two types exist, array computers perform concurrent operations on an array of processors, whereas vector computers use extreme pipeling when applying instructions to a sequence (vector) of data.
3) MISD (Multiple Instruction Stream, Single Data Stream): A computer with multiple processors, each sharing a common memory. There are multiple streams of instructions and one stream of data.
4) MIMD (Multiple Instruction Stream, Multiple Data Stream): General purpose multiprocessor system where each processor has a separate program and one instruction stream is generated from each program for each processor. MIMD is a process level parallel architecture, so each instruction stream operates upon different data. Each processor operates under the control of an instruction stream issued by its own control unit. The processors operate asynchronously in general. MIMD machines with shared memory are described as tightly coupled, whereas MIMD machines with interconnection network are described as loosely coupled.

3.1.3. The Network-on-Chip

Cores do not make up SoCs alone, they must include an interconnection architecture and interfaces to peripheral devices. The interconnection architecture includes physical interfaces and communication mechanisms, which allow the communication between SoC components to take place. Usually, the interconnection architecture is based on dedicated wires or shared busses. *Dedicated wires* are effective for systems with a small number of cores, but the number of wires around the core increases as the system complexity grows. Therefore, dedicated wires have poor reusability and flexibility. A *shared bus* is a set of wires common to multiple cores. This approach is more flexible and is reusable, but it allows only one communication transaction at a time, as all cores share the same communication bandwidth in the system and have limited scalability. Using separate busses interconnected by bridges or hierarchical bus architectures may reduce some of these constraints, since different busses may account for different bandwidth needs, protocols and also increase communication parallelism. Because of the necessity to arbitrate between several requesters, shared bus is slow. When additional components are presented to the system, there is further declination in performance. The ad-hoc routing of wires results in backend complications, lower performance and higher power consumption.

The increasing complexity of integrated circuits drives the research of new intra-chip interconnection architectures. In the context of future ICs with several cores running on high frequencies, a *network on chip* (NoC) appears as a better solution to implement on-chip interconnects. A network on chip uses concepts inherited from distributed systems and computer networks to interconnect Intellectual Property (IP) cores in a structured and scalable way. A NoC is an on-chip network [40] composed by cores connected to switches, which are in turn connected among themselves by communication channels. The structured network wiring gives well-controlled and optimized electrical parameters that eliminate timing iterations and enable the use of high-performance circuits to reduce latency and increase bandwidth.

Sharing the wiring resources between many communication flows makes more efficient use of the wires: when one client is idle, other clients continue to make use of the network resources. An on-chip interconnection network facilitates modularity by defining

a standard interface in much the same manner as a backplane bus, just like VME or PCI. The definition of a standard interface facilitates reusability and interoperability of the modules. The scalability of NoCs in bandwidth and latency allows for any system size implementation and multiple concurrent communications. The flexibility supports multiple applications and configurations for various bit-rates, providing connectivity between each pair of IPs. NoCs are also compositional, allowing a merge of multiple sub-systems.

Several switching methods adapted from traditional network designs are available on NoCs. In circuit switching, a control message is sent from source to destination and a path is reserved. Communication starts, and then the path is released when communication is complete. With packet switching, each switch waits for the full packet to arrive in switch before sending to the next switch. Several novel methodologies are also possible with packet switching, such as cut-through routing or worm hole routing where the switch examines the header, decides where to send the message, and then starts forwarding it immediately.

Interconnection Networks have different architectural variations based on topology, direct or indirect (through switches), static (fixed connections) or dynamic (connections established as required), and the routing type (store and forward/worm hole). The inter-processing element communication can be synchronous or asynchronous. There can be central or distributed control. The efficiency of an interconnection network primarily depends on the delay, bandwidth and the cost.

3.2. The Toy Algorithm: Fast Fourier Transform (FFT)

A Fourier Transform is a representation of a function in terms of a set of sine waves. The set of sine waves of different frequencies is orthogonal, and any continuous function can be represented by summing enough sine waves of the appropriate frequency, amplitude and phase. Fourier Transforms have many scientific applications in areas such as physics, number theory, signal processing, probability theory, statistics, cryptography and geometry.

The Fast Fourier Transform (FFT), invented by Cooley and Tukey [41], is a computationally efficient algorithm for computing a complex Discrete Fourier Transform (DFT), a more sophisticated version of the real DFT. Both of these transforms are named for the way they represent the data; using real or complex numbers. The DFT is extremely important in the area of frequency (spectrum) analysis because it takes a discrete signal in the time domain and transforms that signal into its discrete frequency domain representation. Without a discrete-time to discrete-frequency transform, the Fourier Transform can not be computed with a microprocessor or a DSP-based system.

Decimation is the process of breaking down something into its constituent parts. Decimation in time (DIT) involves breaking down a signal in the time domain into smaller signals, each of which is easier to handle. The idea behind the FFT is the divide and conquer approach, to break up the original N point sample into two (N / 2) sequences, since a series of smaller problems is easier to solve than a large one. This process of decimating each DFT is continued until a series of two-point DFTs (only two input samples) are reached. The DFT requires $(N-1)^2$ complex multiplications and $N(N-1)$ complex additions, whereas FFT's approach only requires a complex multiplication and two complex additions, and the recombination of the points, which is minimal. More information on DFT and FFT can be found in [42].

The Twiddle Factors (TF), which are the frequency resolution of the DFT outputs, stating how far apart each sample is on the frequency domain, can be represented as periodic vectors in the unit circle. The vectors are symmetric, and equally spaced around the circle with spacing $2\pi /N$. The DFT results are periodic around the sampling frequency, so in one round (2π radians), the sampling frequency is obtained. The twiddle frequency is also inversely symmetric about the origin, therefore, the first half (0 to π) of the twiddle factors contain all the necessary information, as the second half is just an inverse of the first half. Thus, four TF's are necessary for an 8-point FFT operation.

The elementary building block of the FFT is the FFT Butterfly. The FFT butterfly is a graphical method of showing multiplications and additions in-between the samples. A signal flow graph of an FFT operation consists of a number of butterflies. Each butterfly takes a pair of input data values A and B and outputs A' and B' as shown in Figure 3.2.

The input data is multiplied by the associated Twiddle Factor W_N^k. The solid dots represent addition/subtraction.

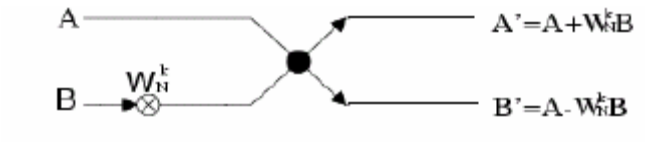


Figure 3.2. The FFT butterfly

Using this notation, we can construct butterfly networks that perform the FFT. Given below is the signal flow graph for the 8-point DIT FFT.

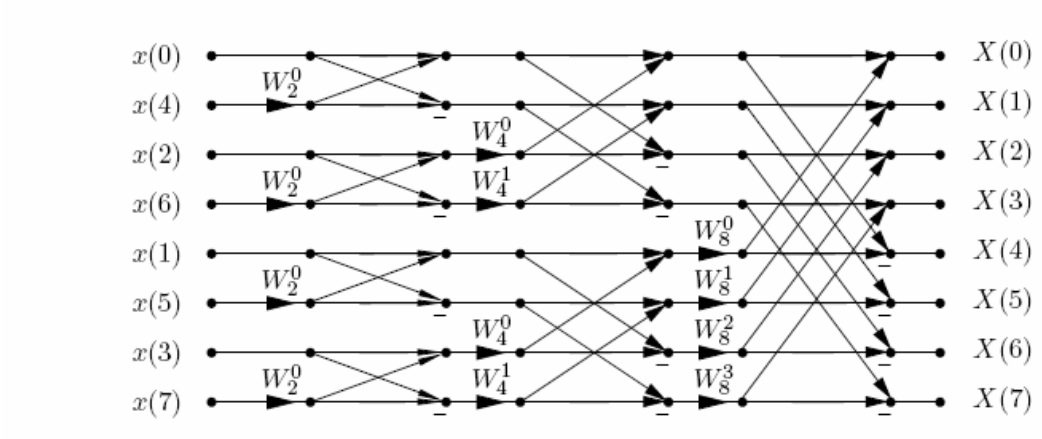


Figure 3.3. Signal flow graph of an 8-point radix-2 DIT FFT

The 8-point radix-2 FFT involves eight complex inputs, four complex Twiddle Factors (TF), and eight resulting complex outputs. As shown in Figure 3.3, the algorithm consists of bit-reversal, followed by three execution stages consisting of four FFT butterfly operations each. The process of decimating the signal in the time domain has caused the input samples to be re-ordered. For an 8-point signal, the original order of the samples is {0, 1, 2, 3, 4, 5, 6, 7}, but after decimation, the order is {0, 4, 2, 6, 1, 5, 3, 7}. The bit patterns representing the sample number has been reversed. This new sequence is the order that the samples enter the FFT, as shown on Table 3.1.

Table 3.1. Bit-reversal

Original Input		Re-ordered Input	
Decimal	Binary	Binary	Decimal
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

3.2.1 The Butterfly Unit

The FFT Butterfly Unit is designed to take the FFT butterfly of two 16-bit fixed-point signed integers, using an FFT twiddle factor, outputting two 16-bit results, one yielding A+WB and the other A-WB. Both the input and output pairs consist of real and imaginary parts of 8 bits each. The twiddle factor, W, is also made up the same way. The computation process, called the FFT butterfly is made up of two complex additions and one complex multiplication to compute A-WB and A+WB. These complex operations have been achieved by real operations instead, as shown in Figure 3.4.

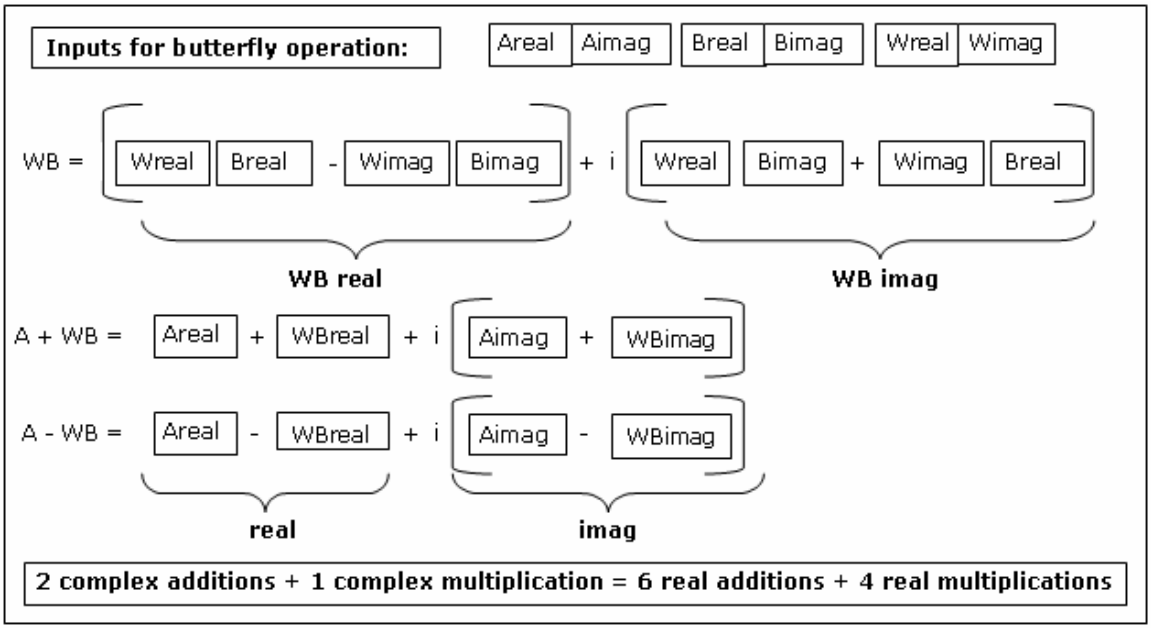


Figure 3.4. The FFT butterfly operation

The FFT Butterfly Unit contains a ROM for the storage of Twiddle Factors, which are brought in during the Butterfly operation. The enable signal set high activates the unit that is shown on Figure 3.5.

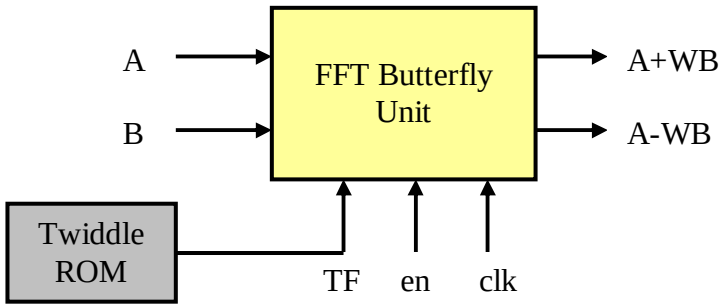


Figure 3.5. The butterfly unit

3.3. The Implementations

The implementations were done on Xilinx Virtex-II FPGA. The VHDL cores were synthesized using the tool Xilinx ISE (Integrated Software Environment) 7.1 [43], and simulated with Modelsim SE 6.0 [44]. The flow of design, as accomplished by ISE takes a VHDL description, synthesizes it, places it on the chip and routes the signals to finally get the FPGA programming file, the bitstream determining the contents of every programmable element inside the FPGA. The ISE provides with information on area utilization of LUTs, IOBs, and dedicated units on the chip. It also provides with timing information and a maximum frequency calculation. The cores were simulated with Modelsim and waveforms were examined to ensure correct functional operation.

The Xilinx Virtex-II 2v500fg256-5 FPGA is a 500K gate FPGA with 24 by 32 CLB columns which have 4 slices each. The chip contains a total of 3072 slices, 32 of 18 Kbit Block RAMs and 32 dedicated multipliers as hardware primitives that do not use up LUT space. These primitives have been extensively made use of in the designs.

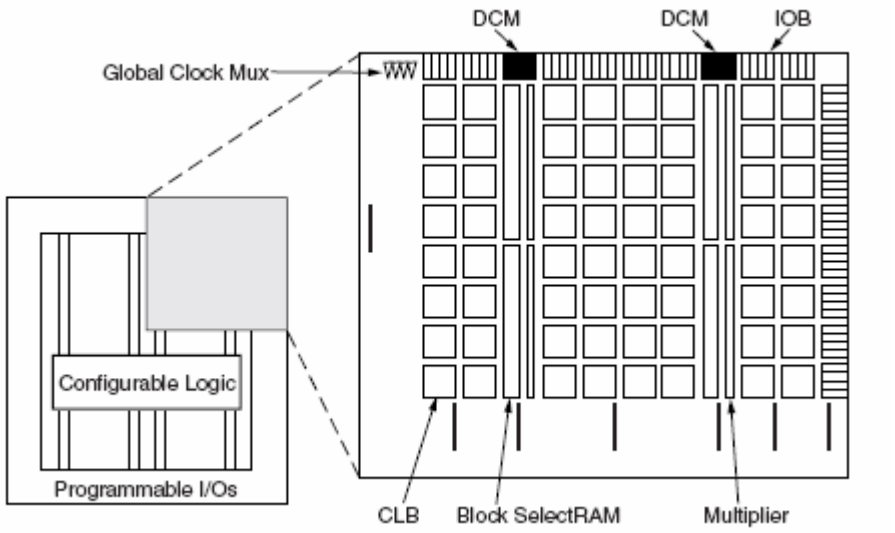


Figure 3.6. The Xilinx Virtex-II [45]

3.3.1. The SISD Model

The SISD model is a RISC (load/store architecture) system with 16-bit instructions operating on 16-bit data words. The SISD processor (Figure 3.7) has 8 general purpose registers and dedicated special registers for the address bits and the data words. It is FFT-application-specific, meaning that a dedicated unit for FFT calculations is available, as well as general-purpose processor functionality.

To serve as Instruction Memory to the SISD, the Xilinx Virtex-II component RAMB16_S18 was chosen. This is a Single-Port Synchronous Block RAM that can be configured to address the data space in different ways. For the SISD-FFT implementation, the instruction word was selected to be 16 bits wide and the addressing is accomplished by 10 bits using this particular unit that has a total size of 16 Kb. The Block RAMs in Xilinx Virtex-II are primitives that use reserved blocks on the FPGA as opposed to general-purpose LUT space, therefore being the best choice for the selection of large RAM. The unit also contains 2 parity bits alongside the 16 bit width; however, these bits were left unutilized. The Data Memory is also a 10-bit addressable, 16-bit Block RAM, same component as the Instruction Memory. The data contained in the Data Memory of SISD_FFT is two's complement, fixed-point, signed or unsigned 16-bit vectors.

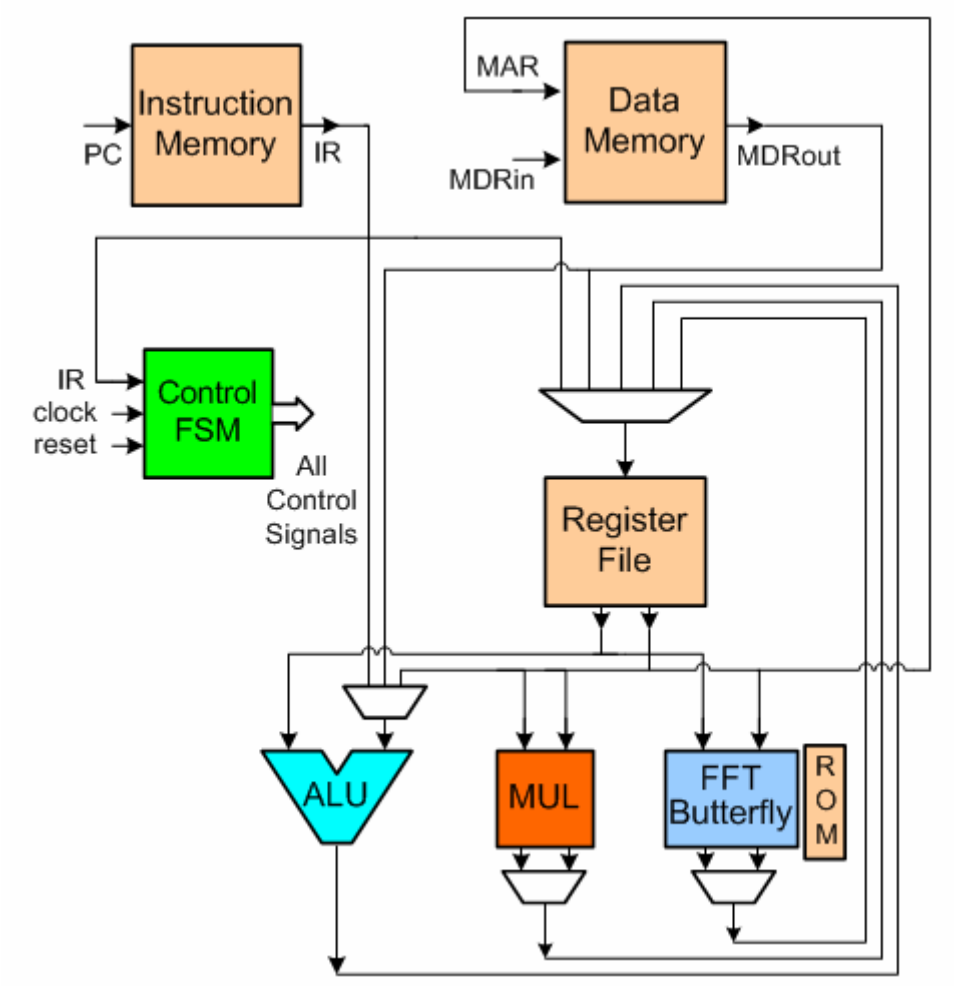


Figure 3.7. The SISD architecture

There are four special registers in the FFT-customized SISD system. The MDR (Memory Data Register) that contains the output of the Data Memory and the IR (Instruction Register) which points at the currently processed instruction are 16-bit registers. The 10 bit wide special registers MAR (Memory Address Register) and the PC (Program Counter) are address-keepers of the Data Memory and the Instruction Memory, respectively.

The Register File (RF) is a unit with a single-port input, and a double-port output, keeping 8 of 16-bit values to be served to the ALU/MUL/BUTF execution trio. The RF is implemented as Select RAM on the Virtex chip, directly utilizing LUTs as RAM. It is preceded by the *Regmux*, which is a 5-to-1 multiplexer with inputs from IR, MDR, ALU, MUL and BUTF. The decision-making of which of these inputs should be directed as an input to the Register File, is again governed by the Control Unit.

The ALU, which is driven with a 3-bit signal that selects between operations, is capable of performing the following seven operations: ADD, SUB, AND, OR, XOR, arithmetically shift right and arithmetically shift left.

Table 3.2. The ALU operations

ALUop	**Operation**
000	ALUout = 0
001	ALUout = ALUin1 + ALUin2
010	ALUout = ALUin1 - ALUin2
011	ALUout = ALUin1 AND ALUin2
100	ALUout = ALUin1 OR ALUin2
101	ALUout = ALUin1 XOR ALUin2
110	ALUout = Shift Right Arithmetic (ALUin1)
111	ALUout = Shift Left Arithmetic (ALUin1)

After each operation, the ALU also outputs a zero bit and an overflow bit that are necessary for the branching instructions. The fetching of an instruction in SISD is as follows. The content of the Program Counter points to Instruction Memory address input. Next, the output of the IM, the IR (16-bit instruction word) is forwarded to the Control Finite State Machine. The Control FSM unit, containing the instruction word, decodes the instruction and sets up the necessary units to perform the correct operation. If a fetch from Data Memory is issued, the address is sent to the MAR, then to the DM address input, and after a read operation is issued, the MDR contains the 16-bit input, which might send it to the Register File and then to the processing elements. If the data is in place at the registers, a processing might be issued, saving the data back in the registers. Also, an immediate operand could be specified, bypassing the MAR/MDR routine.

The Control Unit is a finite state machine completely describing the Instruction Set Architecture of the SISD via hardwired control. As inputs, it has the clock, reset, branch conditions, and the Instruction Word. The outputs are a large set of control signals for all units contained in the SISD, varying from enable signals, resets, multiplexer select signals, to the ALU operation select signal, and the Program Counter increment signal.

The Multiplier unit is the Xilinx Virtex-II primitive MULT18X18s, a hardware multiplier of two 18-bit signed or unsigned numbers. Since the data bus width of the

register file is 16 bits, the 32-bit result has to be sent in 2 cycles, with the low half stored in the desired destination register, and the high half in the register number 8.

The FFT Butterfly Unit, as described previously, takes the FFT butterfly of two 16-bit fixed-point signed integers using an FFT twiddle factor. The inputs are gathered from two registers and the in-place algorithm makes sure to save the outputs on the same register locations. The butterfly unit also contains a ROM for twiddle storage, which is considered to be external to the processor implementation. The processor accesses this ROM by the specified 6-bit address inside the instruction word.

The instruction set of the SISD is summarized on Table 3.3. All 16 possible opcodes have been utilized for different instructions. The 9-bit address fields on load and store instructions limit the use of the DM to the lower half of the unit.

Table 3.3. The SISD instruction set

Opcode	Meaning	Type	Inst.Word Distribution	Explanation	~ [a]
0000	NOP	-	4	No operation	2
0001	ADD	R	4+3reg+3reg+3reg	Add 2 registers, save to a destination register	4
0010	SUB	R	4+3reg+3reg+3reg	Subtract 2 registers, save to a destination register	4
0011	AND	R	4+3reg+3reg+3reg	AND 2 registers, save to a destination register	4
0100	OR	R	4+3reg+3reg+3reg	OR 2 registers, save to a destination register	4
0101	XOR	R	4+3reg+3reg+3reg	XOR 2 registers, save to a destination register	4
0110	SHIFTR	R	4+3reg+5blank+4shift	Arithmetically shift a register right by at most 15 bits	4
0111	SHIFTL	R	4+3reg+5blank+4shift	Arithmetically shift a register left by at most 15 bits	4
1000	MUL	R	4+3reg+3reg+3reg	Multiply 2 regs. Save high half of result in reg 8	5
1001	BUTF	R	4+3reg+3reg+6romadr	Butterfly 2 regs & twiddle. Save result in regs 7&8	5
1010	LD	I	4+3reg+9address	Load a register from low half of the data memory	3
1011	ST	I	4+3reg+9address	Store a register to low half of the data memory	3
1100	BL	J	4+3reg+3reg+6offset	Compare 2 regs, branch if first less than second	5
1101	BEQ	J	4+3reg+3reg+6offset	Compare 2 regs, branch if first equals second	5
1110	ADDI	I	4+3reg+9constant	Add a register with a 9-bit immediate value	4
1111	LDI	I	4+3reg+9constant	Load a 9-bit immediate value to a register	4

a. ~: clock cycles.

3.3.2. The SIMD Model

On the SIMD-based design for the FFT application-specific general-purpose processor, the SISD design serves as the host computer to the SIMD array unit (Figure

3.8). The SIMD unit consists of 4 Processing Nodes, each containing a Local Memory (LM) coupled with a Processing Element (PE). It is interfaced with the SISD via extending the control unit and using dual-ported Block RAMs for Data Memory. The working principle of an SIMD operation can be observed in 3 parts:

1. The set of data from Data Memory, starting with the source address, is carried to Local Memories.
2. The selected instruction processes on 4 different sets of data on LMs.
3. The results are carried back to the Data Memory, starting with the destination address.

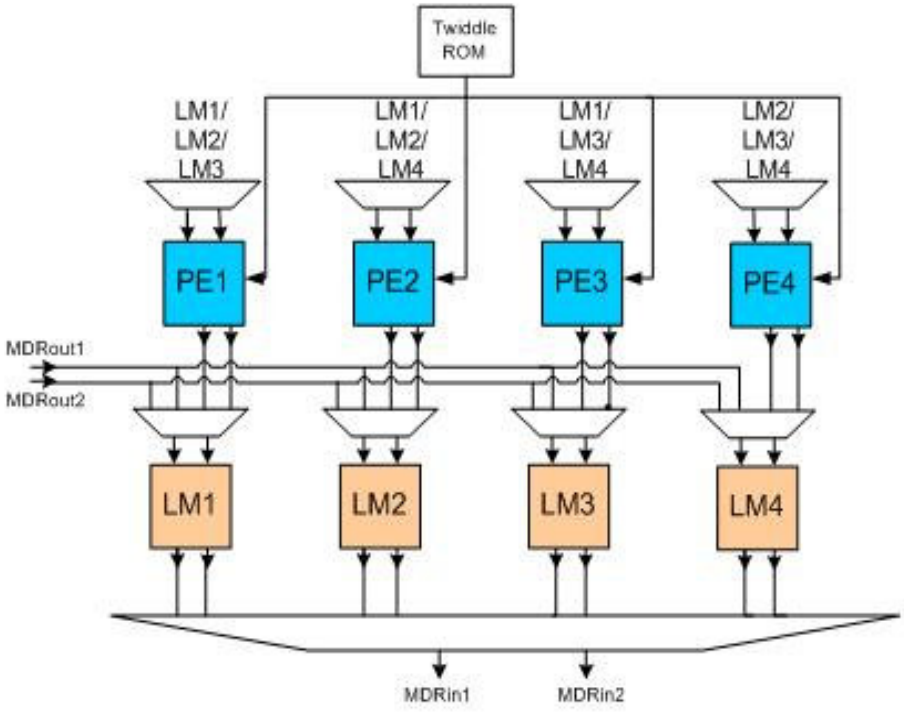


Figure 3.8. The SIMD extension to the SISD

The Processing Elements of the SIMD are made up of the ALU/MUL/BUTF processing trio. Depending on the *PEopsel* control bits that select which operation is to be performed, one of the three units is enabled and the final output is gathered from that unit. The Local Memories of the SIMD are RAMB16_S18_S18 components, dual-ported Block RAMs just like the Data Memory. Although they might seem to be excessively large just for the 8-point FFT, for longer array operations, they are the best solution in the Xilinx

Virtex-II FPGA environment where they do not occupy LUT space. A large (128-to-32) data multiplexer is placed at the end of the Local Memories to select which one to read from when writing back to the main Data Memory.

Since all the 16 different instructions that could be extracted from a 4 bit opcode is occupied, a slight change had to be done on the SISD component: it was decided for the SISD to be stripped off its XOR functionality to compensate for the switch for SIMD instructions, as shown in Table 3.4.

Table 3.4. Changing the SISD opcode table for SIMD mode

Opcode	Meaning	Type	Inst.Word Distribution	Explanation
0101	SIMD	array	16+16+16	Takes the processor into SIMD mode

A SIMD adopts a VLIW (Very Long Instruction Word) approach and the instructions are executed by fetching three 16-bit instructions from Instruction Memory. The 39-bit SIMD instruction depicted in Figure 3.9 is formed the following way. When the opcode “0101” is detected on the fetched instruction, the next two instructions are fetched as well, and saved into the SIMDWORD register. The first 9 bits of the first instruction fetched, including the opcode “0101” and the following 5 bits, are discarded. The remaining 7 bits are merged together with the other two 16-bit instruction words fetched, to form the 39-bit SIMD instruction word.

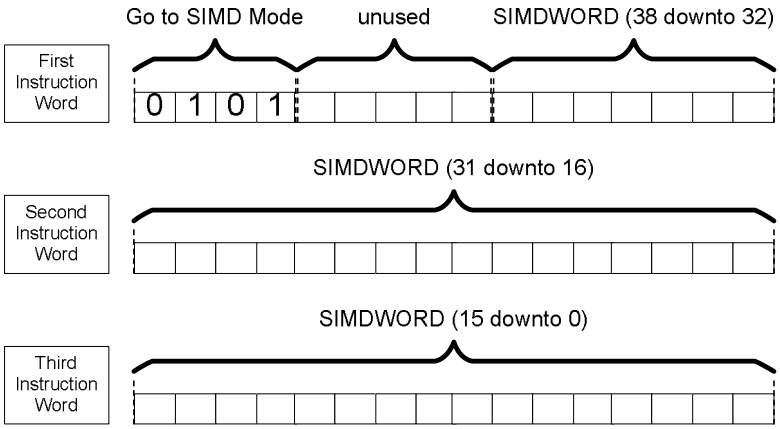


Figure 3.9. Interpreting SIMD instructions

The SIMD instructions contain source and one destination address and a 6 bit field for iterations, which determines the number of executions to be performed on the SIMD unit. Table 3.5 shows the instructions that can be performed on the SIMD.

There are two different types of operations for the SIMD array unit. For arithmetic instructions that use the ALU and MUL, up to 256 calculations can be done with a single instruction, 6-bits specifying the number of iterations on 4 PEs. Afterwards, the results are automatically written back to the Data Memory specified by the destination start address.

Table 3.5. SIMD instruction set (39-bit instructions)

Opcode	Meaning	Instruction Word Distribution
001	ADD	<10addr destination> <10addr source1> <10addr source2> <6iterations>
010	SUB	<10addr destination> <10addr source1> <10addr source2> <6iterations>
011	AND	<10addr destination> <10addr source1> <10addr source2> <6iterations>
100	OR	<10addr destination> <10addr source1> <10addr source2> <6iterations>
101	XOR	<10addr destination> <10addr source1> <10addr source2> <6iterations>
110	MUL	<10addr destination> <10addr source1> <10addr source2> <6iterations>
111	BUTF	<10addr source&destination> <6twiddleaddr> (14 blank) <6iterations>

The other type of SIMD operation is the butterfly operation, whose control is different than the arithmetic operations and is hardwired into the state machine. The system does not perform iterations of the butterfly operation on the inputs, but executes an 8-point butterfly operation on input groups of 8. After receiving the first 8 inputs and bit-reversal, all three phases are accomplished while the selected outputs of the processing elements are sent to each other through the static interconnection until all phases are completed. Then the 8 resulting outputs are sent back to Data Memory and another 8 can be fetched for another 8-point butterfly operation. So, for the BUTF operation, the SIMD is capable of calculating up to 64 three-phase butterflies with a single instruction. To enable the butterfly operation, the Twiddle Factors are automatically distributed to the ROM in advance. The three phase butterfly calculation works as follows:

1. Fetch 8 of 16-bit values from the Data Memory (enumerated 0 through 7).
2. Bit-reverse the inputs.
3. Bring the 16-bit twiddle factors into local twiddle RAM of the PEs (enumerated twiddle0 through twiddle3).
4. Start executing:

   a. butterfly 0 and 4 with twiddle0.
   b. butterfly 1 and 5 with twiddle1.
   c. butterfly 2 and 6 with twiddle2.
   d. butterfly 3 and 7 with twiddle3.
5. Move results around the PEs to:
   a. butterfly 0 and 2 with twiddle0.
   b. butterfly 1 and 3 with twiddle2.
   c. butterfly 4 and 6 with twiddle0.
   d. butterfly 5 and 7 with twiddle2.
6. Move results around the PEs to:
   a. butterfly 0 and 1 with twiddle0.
   b. butterfly 2 and 3 with twiddle0.
   c. butterfly 4 and 5 with twiddle0.
   d. butterfly 6 and 7 with twiddle0.
7. Save the final results into Data Memory.

Keeping in mind that each PE initially deals with the inputs in twos, that is, PE1 initially has inputs 0,1, PE2 has 2,3 PE3 has 4,5, PE4 has 6 and 7, the necessary intra-PE transfers shows that a 2x2 mesh without wraparound connections is the obvious choice for the network topology, as shown in Figure 3.10.

source PE	after first phase	after second phase
PE1	PE3	PE2
PE2	PE4	PE1
PE3	PE1	PE4
PE4	PE2	PE3

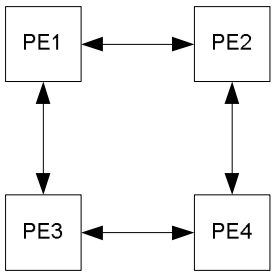


Figure 3.10. The PE interconnection of the SISD-SIMD

The intra-PE routing is managed by the control unit and as shown in Figure 3.8, made possible by a multiplexer, named *PEroutemux*. This unit comes before the PEs, and has three pairs of inputs and one pair of outputs. Two of the input pairs come from neighboring PEs and the third pair comes from the Local Memory. A select signal

PEroutemuxsel helps select which of these should be sent as an input to the Processing Element. By changing this signal, the control unit is able to route the correct outputs to the inputs.

When the system is in SIMD mode, a DM address counter unit is responsible to keep track of the data address incrementation both for reading and writing. Another addition to the SISD are the two multiplexers placed before the address and the input of the Data Memory, distinguishing SIMD mode from the SISD mode, enabling the counter unit to access the DM in SIMD mode, while closing out the SIMD elements in SISD mode.

3.3.3. The Network-on-Chip Model

For the NoC design, the Hermes router infrastructure model [46] was utilized to create a 2x2 router. In this model, for communication, packets are sent in forms of flits instead of wide buses. For each switch, a request and an acknowledge signal is present as well as an input and an output of flit width. Shown in Figure 3.11, an output port in the HERMES switch is composed by the following signals: Tx (data availability), Data_out (data to be sent) and Ack_tx (successful data reception). An input port, on the other hand is composed by the following signals: Rx (data availability), Data_in (data to be received) and Ack_rx (successful data reception).

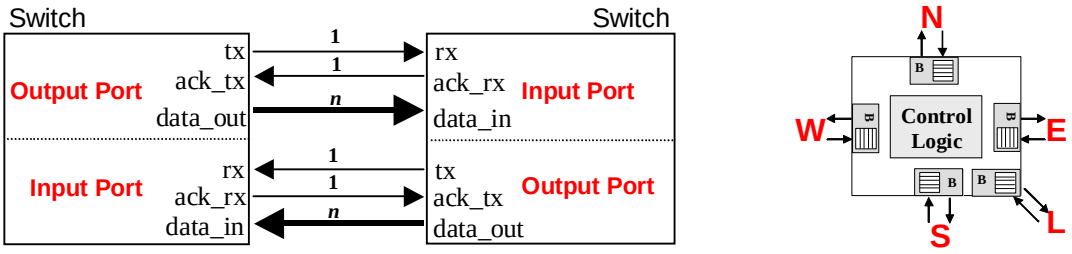


Figure 3.11. A Hermes switch and its ports [46]

A switch of the HERMES router can have up to 5 ports: North, South, East, West and Local. The router is implemented to operate on 2x2 mesh with no wraparound connections, so each switch is a corner switch with 3 available ports. The router was implemented with 8-bit flit size, and 8 input-output buffers. Figure 3.12 presents the developed NoC system.

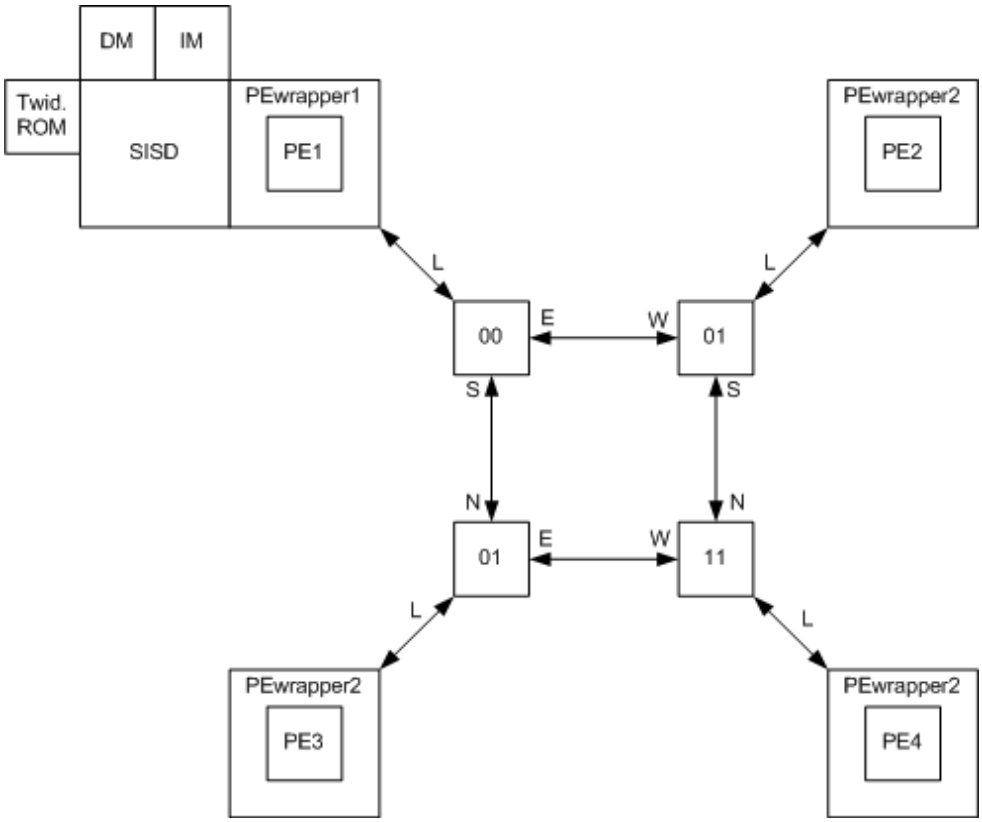


Figure 3.12. The NoC system

Two wrapper modules around the routers had to be constructed for the NoC implementation. *PEwrap1* is the wrapper module for the integration of PE1+SISD+Data Memory+Instruction Memory+Twiddle ROM with the rest of the network. Its primary job is to convert flits into inputs that can be used on the PEs, and to convert the PE outputs into flits that can be routed. Also, it detects from the number of flits received and sent, what the current operation is and what should be done next, i.e. where and how many flits should be sent at the next stage. This is necessary for the implementation of the three-stage butterfly. The wrapper modules keep track of which stage is currently being processed and sends either all or some of the outputs to the correct PE.

PEwrap2 is the wrapper module for PE2, PE3, and PE4. There are 3 instances of this module for each PE. It is very similar to *PEwrap1*, but it is a bit simpler since it does not deal with wrapping the SISD and the memory elements. In short, both the wrappers basically convert packet flits into inputs for the PEs and the PE outputs into flits so that the packet is ready for communication. The PE operations are the same as in the SIMD

module; ADD, SUB, OR, XOR, AND, MUL and BUTF. The wrappers are equipped with the necessary state machines to enable correct routing for the butterfly operation. So, the overall control of the machine is done by three units; *PEwrap1, PEwrap2,* and the control unit of the SISD. The control unit is responsible with the transfer of Twiddle Factors, and the data memory during SIMD mode. It also contains bits coming from the switches to decide when to send the data out to the routers. The butterfly operation at the wrappers can be summarized with Table 3.6.

Table 3.6. Butterfly operation on the NoC

NODE 00 – PEwrap1	NODES 01,10,11 – PEwrap2
1) Send 6 flits containing twiddles.	2) Receive 6 flits, save to internal storage.
3) Send 5 flits of the first operation (PEin1,PEin2,PEop) Execute local PE, route one of the outputs (2 flits)	4) Receive 5 flits, execute with correct twiddle. Send one of the outputs (2 flits) to the target node.
5) Receive 2 flits as one input to the local PE, execute, route one of the inputs.	6) Receive 2 flits, process with correct twiddle, send out one of the outputs (2 flits).
7) Receive 2 flits as one of the inputs. Execute. Send 2 flits.	8) Receive 2 flits. Process with correct twiddle. Send 4 flits (both the outputs of the PE)
9) Receive 4 outputs from each node. Save to Data Memory along with the Local PE output.	

It should also be note that storage of three twiddle factors on each PE is enough for the 8-point FFT, since each PE operates three times to calculate the final values. This is why 6 flits containing these twiddles are initially sent out at first.

3.4. Conclusions

The primary accomplishment on this section was the successful implementation of the 8-point FFT operation, using the FFT Butterfly operation as a custom unit, on three different architectures. The SISD provided with a uni-processor infrastructure, which was used for the creation of the SIMD and NoC models. The SIMD used a simple VLIW technique to create long instruction words by fetching multiple 16-bit instructions. The usage of Block RAMs as Local Memories helped with area-efficient design. In an attempt

to cancel out the shared bus between the Processing Elements and the host computer, the NoC system was formed. Although complete connectivity was established between the elements, the wrapper design was an intricate one to accomplish. Furthermore, the NoC obtained MIMD-like behavior since the PE's did not operate in lock-step manner, they were set to execute whenever they complete receiving their packets. Also, a problem of wasted cycles was caused since the source for packet creation and sending is the SISD, and it is the node that sends packet after packet. As the number of flits in a packet increase, network congestion problem arose because of packet collision. Therefore, extra clock cycles were added to the end of the packet sending state machine to make the switch wait for the input buffers to empty. Eventually, this resulted in slower performance.

The SIMD increased performance over the SISD with each processing inside the array unit taking 2 cycles each. For example, with the 6-bit iterations field set to 4, after the initial 8 cycles of instruction fetch, it takes 16 cycles for moving data to local memories, 8 cycles for processing, and 16 for moving data back to local memory. So, 16 ADD operations of the SIMD-FFT take 48 clock cycles whereas 16 iterations of the load-add-store subroutine on the SISD takes 160 clock cycles. More figures can be examined on Table 3.7.

Table 3.7. Comparisons of SISD, SIMD and NoC

	SISD	**SIMD**	**NoC**
Number of Slices:	489	2028	3028
Number of Slice Flip Flops:	401	1234	2366
Number of 4 input LUTs	789	3441	5134
Number of BRAMs	2	6	2
Number of MULT18X18s	5	25	25
Minimum period (ns)	8.585	15.450	12.326
Maximum Frequency (Mhz)	116.482	64.725	81.129
Clock cycles to complete 1 Butterfly operation	5	5	5
Clock cycles to complete 1 8-pt Butterfly	112	54	91
Clock cycles to complete 16 8-pt Butterfly ops.	1792	218	956
Clock cycles to complete 1 ADD operation	5	5	5
Clock cycles to complete 16 ADD operations	160	48	142

All implementations were able to fit onto the targeted FPGA, the Xilinx Virtex II, which has 3072 overall slices. Looking at Table 3.7, it is convenient to say that SIMD had

the best performance per area for making several iterations of the 8-point FFT. However, for a single 8-point FFT operation and for scalar computations, the SISD was satisfactory.

Hardware units such as the Block RAM and the Multiplier were important for low LUT utilization. It was showed that the equivalent LUT count for the hardware multiplier was 569, and for the Block RAM was 1328 [47]. The Xilinx II FPGA has 6144 LUTs. For the SISD, the design with no hardware primitives would have occupied close to 6200 LUTs instead of 789, where the other two models would never fit onto the selected FPGA.

A weakness in the SIMD model is that since the FFT operation's intra-PE data movements were handled by the control unit FSM, no specific operations to route values around these Local Memories were implemented. The data movement from DM to the LMs is also done automatically with a SIMD instruction being issued, so there are also no SIMD-load or SIMD-store instructions. Although having these instructions would complicate the design, it would help the SIMD to a wider range of possible applications.

There are many reasons why the NoC system seems to have failed. Firstly, the area of the router is high, around a thousand slices. Secondly, the design was open to network congestion, and therefore did not operate as desired. When the flit size was increased to 16-bits, the performance was better, however, the router area increased even further, due to packet buffers. Some of the possible enhancements to the NoC could be to use 16-bit flits or to use packets that contain the next addresses, as well as the Twiddle Factors. On the SIMD, the iterations field can easily be increased when more bits are added to the 39-bit wide SIMDWORD. Each bit added could double the iterative capability of the SIMD instructions. Also with the extra bits, bigger processing elements can be utilized.

Other architectural options that could be explored could include the MISD and the MIMD but these choices were not implemented because they are not exactly suitable with the data-parallel nature of the FFT algorithm. The MISD can be used for systolic implementations, and the MIMD can be accomplished by including a scheduler to manage instruction flow around the system. For the MIMD system, the event-driven routers of the NoC could come in handy.

4. THE SIxD ARCHITECTURE

The SIxD is a configurable, application-specific system with fixed 16-bit instruction word and variable data space, fusing a non-pipelined RISC uni-processor, SISD, with a SIMD multiprocessor. The SIxD supports application-specific custom modules, as well as general-purpose processor functionality. A configuration file governs ALU instructions, branching instructions, data space, and most importantly, activating the SIMD mode of the core with a user-defined number of Processing Nodes (Table 4.1). The SIxD was designed with additions to the SISD system and the SIMD system presented on Section 3.3.1 and Section 3.3.2, respectively. The most important progress is the parameterization and the configuration of the core with a header file written on VHDL that enables compile-time configurability. Instruction set compaction, altered PE interconnection, and the non-VLIW SIMD with SIMD load/store instructions are some of the other changes made.

Table 4.1. SIxD processor features

fixed features	parameterized features
16-bit instruction word non-pipelined 8 registers	data width/length instruction set SIMD Processing Units I/O Ports

Section 4.1 describes the accomplishments of the configuration file of the SIxD and describes how different configurations can be accomplished. Section 4.2 illustrates the architecture of the SIxD in detail, looking at each component. Section 4.3 focuses on the instruction set, and finally Section 4.4 examines the SIMD functionality of the core.

4.1. The Configuration File

The SIxD core is modified at compile-time with the aid of a configuration file written in VHDL. The following types of modifications are supported by the configuration file:

1. Determine the data space (width and length): Depending on the size of data memory needed and the Block RAM available on the FPGA, the user can

specify the data width (as a multiple of 8), and data length (at least 2K words).

2. Select the most appropriate branch and shift instructions: In the default instruction set, BL and BEQ are used. However, if greater-than operations are used more frequently than BL, then the user has the chance of replacing BL with BG. There are six possible branch instructions for two branch instruction slots: BEQ, BNEQ, BL, BLEQ, BG, and BGEQ. As shifts, arithmetic or logical shift instructions can be selected.
3. Remove unused instructions from the instruction set: For example, removing a 16-bit XOR instruction would reduce the area of the core by 11 slices (Table II). This way, the SIxD core can be made to fit in a smaller FPGA.
4. Enable SIMD with a number of Processing Nodes: Depending on the size of chip, the user can enable SIMD instructions, choosing to have any number of PNs that is a multiple of two.
5. Set the number of Input/Output (I/O) Ports, up to a maximum value of four.

Table 4.2. SIxD configuration variables

Constant	**Explanation**
branchsel, branchsel2	Branch selections: Selects between BL/BLEQ/BG/BGEQ and BEQ/BNEQ.
ALUANDen	AND operation: Enables/disables the AND operation of the ALU.
ALUORen	OR operation: Enables/disables the OR operation of the ALU.
ALUXORen	XOR operation: Enables/disables the XOR operation of the ALU.
ShiftRsel	Shiftr select: Shift right arithmetic or logical selection bit.
ShiftLsel	Shiftl select: Shift left arithmetic or logical selection bit.
DATAWIDTH	Data width: The data width as a power of 8.
NoDM	Data length: Specify the address bit length (at least 11).
NoPN	Number of Processing Nodes: The number of PNs of the SIMD.
SIMDenable	SIMD enable: If 0, the SIMD unit is not generated.
SIMDsingleresult	Single/double result SIMD operations.
PORTWIDTH	I/O port width: Width of the I/O ports, as a multiple of 8.
numports	Number of I/O ports: Number of I/O ports with a maximum of 4.

Table 4.2 shows the list of constants defined in the configuration file. An example configuration can be seen below on Figure 4.1.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package SISDconstants is

constant SIMDenable:std_logic:='1';
constant SIMDsingleresult:std_logic:='1';
constant noPN : integer := 4;--8;--2;--128;--64;--32;--16;--256;--

constant datawint : integer := 16;-- 24;-- 32;--8;--
constant no8 : integer := datawint/8;--2
constant datawmul : integer := datawint*2;

constant portw : integer:=datawint;
constant numports : integer:=1; --max:4
constant PORTIN : integer:=(portw*numports);

constant noDM: integer:=1;--2;--4;--

constant DMup: integer:=10;--12;--11;-- addressability

constant ALUANDen: std_logic:='1';
constant ALUORen: std_logic:='1';
constant ALUXORen: std_logic:='1';

--Branches
--slot1
constant branchsel: std_logic_vector(1 downto 0):="00";
--00 BL
--01 BLEQ
--10 BG
--11 BGEQ

--slot2
constant branchsel2: std_logic:='0';
--0 BEQ
--1 BNEQ

constant ShiftRsel:std_logic:='0';
--0 SLL
--1 SLA
constant ShiftLsel:std_logic:='0';
--0 SRL
--1 SRA

subtype PORTWIDTH is std_logic_vector((portw-1) downto 0);
subtype DATAWIDTH is std_logic_vector((datawint-1) downto 0);
subtype DATAWIDTHx2 is std_logic_vector(((datawint*2)-1) downto 0);

end SISDconstants;
```

Figure 4.1. The configuration file

4.2. The SIxD Architecture

The SISD model is a non-pipelined RISC system with 8 general-purpose registers, as well as dedicated special registers such as the Program Counter, the Instruction Register, Memory Address Registers and Memory Data Registers (Figure 4.2).

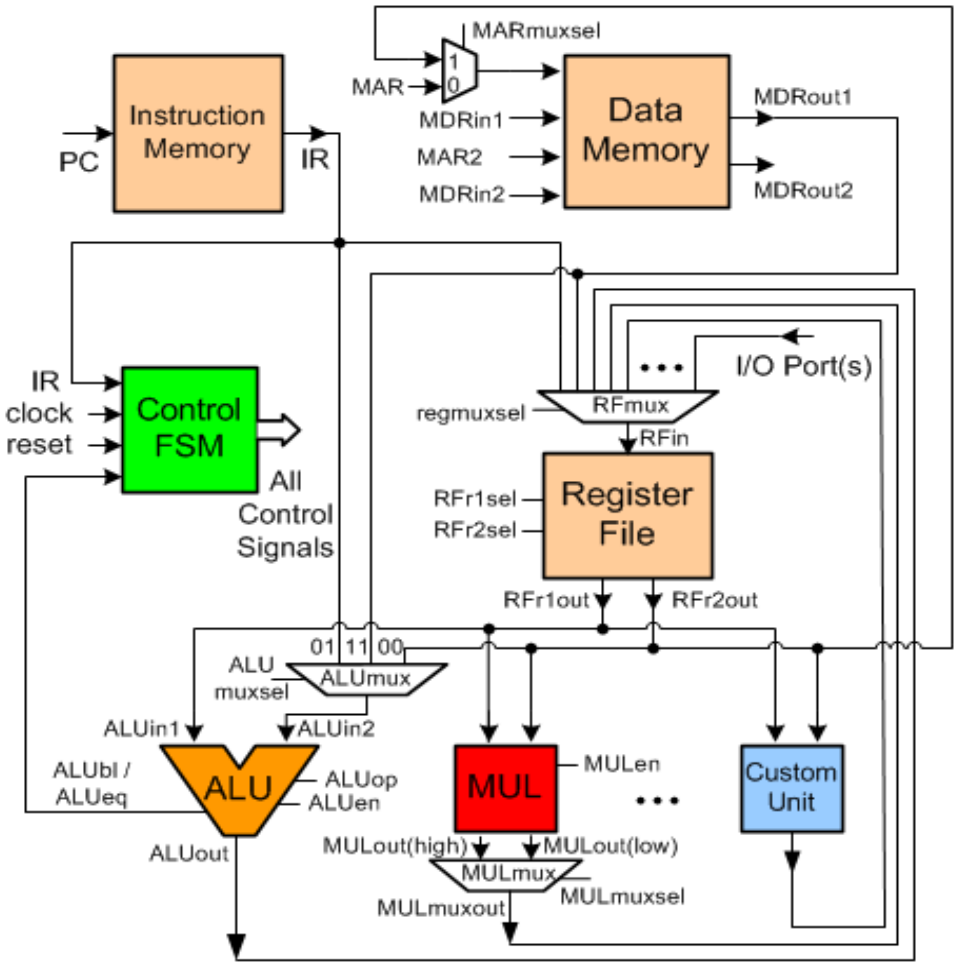


Figure 4.2. The SISD datapath

4.2.1. Instruction Memory

Attempting to make the most out of an FPGA, dedicated Block RAM resources of the FPGA are chosen to serve as Instruction Memory (IM) and Data Memory (DM) of the SIxD. The standard Xilinx component *RAMB16_S18* is a Single-Port Synchronous Block RAM that can be configured to address the data space in different ways. With 10-bit addressing on 16 Kbits, the instruction program consisting of 16-bit instruction words was

located in this particular unit. By common computer architecture definitions, the address of the unit is the Program Counter (PC), while the output is the Instruction Register (IR). The instruction program is presented to the core by initializing the contents of this component with the program code in hexadecimal format, as seen on the left column of Appendix B.

4.2.2. Data Memory

The basic building block of the flexible Data Memory of the SIxD is an 11-bit addressable, 8-bit wide dual-ported Block RAM component, the RAMB16_S9_S9. The dual ported Block RAM, which occupies the same area as the single port type, contains two's complement, fixed-point, signed or unsigned integers. The parameterization of the Data Memory has been accomplished by four constants, namely *datawint*, *noDM*, *DMup*, and *no8*, determining the data width, the data length, the addressing of the Data Memory unit, and the vertical number of 8-bit basic building blocks, respectively. Some configuration options are shown in Figure 4.3, where DM denotes an 8-bit block. Note that when *noDM* is 3 and 4, the 48 or 64 Kbit memory is addressed by 12 bits. For between 64 and 128 Kbit Data Memory configuration, the addressing would have to be done by 13 bits.

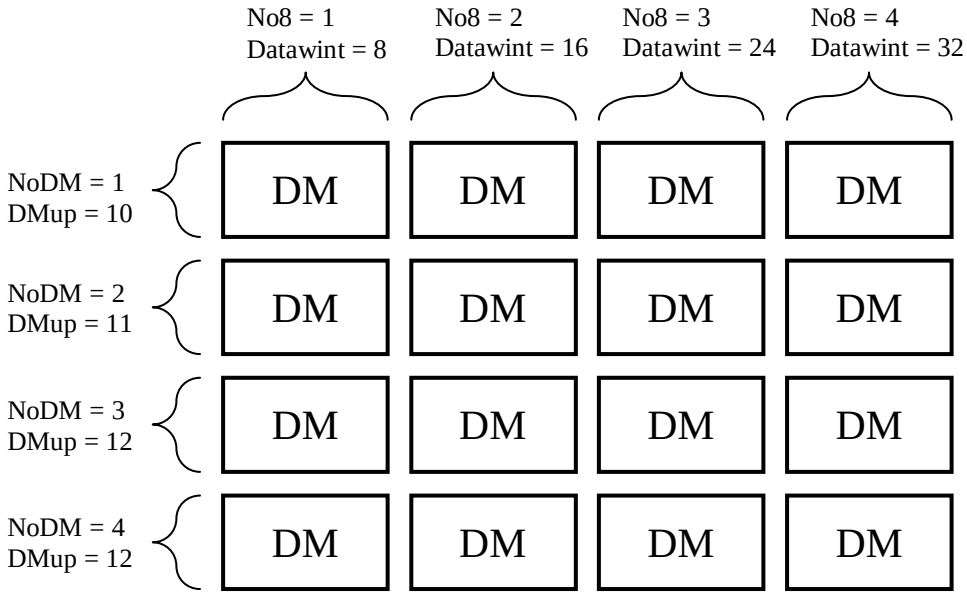


Figure 4.3. Data memory configuration

The concurrent “Generate” statements in VHDL can be used in architecture bodies to describe regular structures, such as arrays of blocks, component instances or processes. Using the constants described above, the situation in Figure 4.3 has been achieved with the VHDL code shown below, on Figure 4.4.

```
datalength: for i in 0 to noDM-1 GENERATE
        datawidth: for j in 0 to (no8-1) GENERATE
                DM:dmem port map (
                        DMADDRA =>DMADDRA(10 downto 0),          --PortA address
                        DMADDRB =>DMADDRB(10 downto 0),          --PortB address
                        DMclk => DMclk,                          --clock
                        DMinA => DMinA(((j*8)+7) downto (j*8)),  --PortA input
                        DMinB => DMinB(((j*8)+7) downto (j*8)),  --PortB input
                        DMENA => enA(i),                         --PortA enable
                        DMENB => enB(i),                         --PortB enable
                        DMr => DMr,                              --reset
                        DMWEA => weA(i),                         --PortA write enable
                        DMWEB => weB(i),                         --PortB write enable
                        DMoutA => DMouttA(i)(((j*8)+7) downto (j*8)),  --PortA output
                        DMoutB => DMouttB(i)(((j*8)+7) downto (j*8))   --PortB output
                );
        end generate datawidth;
end generate datalength;
```

Figure 4.4. VHDL code of the flexible data memory that generates block RAMs

4.2.3. Register File

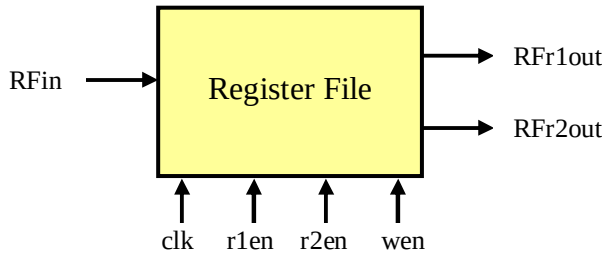


Figure 4.5. Register file

The Register File (RF) with a single-port input and a double-port output has been implemented as a Select RAM on the FPGA, directly utilizing the LUTs as RAM. The 8-bit wide component RAM16X8D is instantiated similar to the Data Memory implementation, as a building block for wider data width configurations of the SIxD. As shown in Figure 4.5, the unit is run by two 3-bit select signals for both ports, named *r1en*, and *r2en*, and a write enable signal depicted *wen*, all of which are controlled by the Control

Unit. It is important to note that a simple in-order issue processor that only use on-chip memory is considered, so there is no cache. The memory on the FPGA is faster than a typical processor implementation, eliminating the need for further exploring the memory hierarchy.

4.2.4. Arithmetic Logic Unit

The ALU of the SIxD is similar to the one explained on Section 3.3.1. However, the shift instructions can be configured in compile-time to run logically or arithmetically. Also, as stated on Section 4.1, unused AND, OR, XOR instructions can be removed for lower area utilization. The ALU also contains two comparators, so that when a branching instruction is issued, it determines if the branch is to be taken or not.

Table 4.3. The ALU of the SIxD

ALUop	**Operation**
000	ALUout = 0
001	ALUout = ALUin1 + ALUin2
010	ALUout = ALUin1 - ALUin2
011	ALUout = ALUin1 AND ALUin2 (optional)
100	ALUout = ALUin1 OR ALUin2 (optional)
101	ALUout = ALUin1 XOR ALUin2 (optional)
110	ALUout = Shift Right A/L[a] (ALUin1)
111	ALUout = Shift Left A/L[a] (ALUin1)

a. A/L: Arithmetic or Logical.

4.2.5. Multiplier Unit

The primitive Hardware Multiplier, MULT18X18S, which is standard in Xilinx Virtex family and Spartan 3 FPGAs, is the obvious choice for multiplier, requiring no LUT space on the chip. As discussed on Section 2.4, if no Hardware Multiplier is present on the target FPGA, the user must either remove the multiplication instruction or provide with a custom generic multiplier. It is a unit that takes two 18-bit inputs, outputting a 36-bit result. For the 8 and 16 bit data width configuration, a single multiplier would suffice, however, for the 24 and 32 bit SIxD implementations, four hardware multipliers could be combined together, as suggested in [48].

4.2.6. Control Unit

The hardwired Control Unit (CU) is a Finite State Machine, implementing a sequential circuit based on different states in the machine, issuing a set of control signals at each state. Depending on the instruction word input, the Control FSM completely describes the Instruction Set Architecture of the SIxD, by setting up the necessary bits for corresponding units to perform the correct operation, as shown on Table 4.4.

4.2.7. Custom Unit

A custom unit can be a user-defined instruction or a custom-designed hardware module, as described on Section 2.4. The bare SISD core has an ALU, a multiplier and no custom unit. In order to create a custom instruction on the SISD, the custom unit must be placed with inputs from the register file, and output(s) to the input multiplexer of the RF. A reserved opcode in the ISA must be associated with the control on the new unit and corresponding states must be revised for the proper enabling of the unit, and the correct choice of the select signal of the RF input multiplexer to assure the proper propagation of the outputs. The “reserved” places are used for custom instruction extension to the ISA of the SIxD. More custom units could be added to the ISA by having selection bits distinguish different instructions, like done in the shift instructions; two instructions for a single opcode. Since there are eight registers, the total number of input and outputs to the custom unit can be thought of as eight. The new states added must load all the necessary registers to the unit in advance, and then set the enable signal of the custom unit. The outputs have to be multiplexed out one by one, and saved the the desired registers. More information on custom unit implementation to the SIxD, as well as an example implementation can be found in Appendix A.

4.2.8. Interrupt and Exception Mechanisms

Up to four interrupts are implemented on the SIxD, as well as an interrupt enable/disable instruction. The core has no stack or accumulator, so only a single level of interrupt handling is possible. The SIxD gives an exception when an invalid opcode is read, and the exception handler forces a skip to the next instruction. An interrupt appears

when interruptreq signal is high, with the IRQ signal identifying which of the interrupts is disrupting the system. The interrupt is acknowledged by the system, which sets the interruptack signal high, after which the exception/interrupt address is stored into the EPC (Table 4.4). Following the Interrupt Service Routine (ISR) that describes the actions to be performed for the specific interrupt, finally the value in EPC is stored back into the Program Counter so that the program continues from where it left off. The interrupt/exception addresses are at the start of the Instruction Memory, at hexadecimal addresses “0000” and “0010” respectively, and can easily be changed by altering the initialization data of the IM. This can be useful because interrupts do not automatically store the contents of the registers and if needed, this could be accomplished by modifying the ISR.

4.3. The SIxD Instruction Set

Using the components described previously, the resulting instruction set is described on Table 4.5. The instruction formats supported by the SIxD are:

- Register-register (direct)
- Register-constant (immediate)
- Register-memory (indirect)
- Branch format

The SIxD is a non-pipelined system primarily because the design is simpler and cheaper to manufacture, lowering NRE and unit costs. Although with pipelining the instruction bandwidth is increased by reducing the cycle time of the processor, there are certain advantages of not pipelining as well. The execution of only a single instruction at a time prevents branch delays, data and branch hazards. A pipelined design also requires more area related to pipeline registers and extra control logic. The instruction latency in a non-pipelined processor is slightly lower than in a pipelined equivalent, due to the fact that extra flip-flops must be added to the data path of a pipelined processor. Also, the performance of a pipelined processor is much harder to determine and varies widely between different programs, but a non-pipelined processor will have a fixed instruction bandwidth.

Table 4.4. The control signals of the SIxD

Associated Unit	Explanation	Name of signal	In/Out	Width
	Clock signal	clock	in	1
	Reset	Rst	in	1
	Inst. Reg.	IR	in	16
ALU	Branch if Less	ALUbl	in	1
	Branch if Equal	ALUeq	in	1
	ALU operation	ALUop	out	3
	ALU enable	ALUen	out	1
	branching	branch	out	2
MUL	MUL enable	MULen	out	1
	MUL reset	MULr	out	1
Register File	RF reset	RFrst	out	1
	RF write select	RFwsel	out	3
	RF read1 select	RFr1sel	out	3
	RF read2 select	RFr2sel	out	3
	RF write enable	RFwe	out	1
	RF read1 enable	RFr1en	out	1
	RF read2 enable	RFr2en	out	1
Instruction Memory	IM read enable	IMen	out	1
Data Memory	DM read enable	DMen	out	1
	DM reset	DMr	out	1
	DM write enable	DMwe	out	1
Program Counter	PC	PC	out	10
Memory Address Reg.	MAR	MAR	out	9
Multiplexers	RegFile MUX select	regmuxsel	out	3
	MAR MUX select	marmuxsel	out	1
	MUL MUX select	mulmuxsel	out	1
	ALU MUX select	alumuxsel	out	2
Exception	Exception	exception	out	1
	Exception cause	excause	out	2
	Exception PC	EPC	out	10
Interrupt	Int. Request	interruptreq	in	1
	Int. Acknowledge	interruptack	out	1
	IRQ channel	IRQ	in	2
I/O Ports	I/O port read enable	Port_en	out	4
	I/O port write enable	Port_we	out	4
SIMD	SIMD mode enable	SIMD_en	out	1
	SIMD PE operation sel.	PEopsel	out	3
	SIMD iter	Iter	out	9
	SIMD unit done	SIMDdone	in	1
	DM serves SISD/SIMD	DMmuxsel	out	1

Table 4.5. The SIxD instruction set

Opcode	Meaning	Inst. Word Distribution	Explanation	~ [a]
0000	NOP	4+“000”+(9)	No operation	2
0000	Interpt.En.	4+“001”+(5)+4IEn	Enable/Disable Interrupts	2
0000	RET	4+“010”+(9)	Return from exception/interrupt	2
0001	ADD	4+“001”+3reg+3reg+3reg	ADD 2 registers, save to a destination register	4
0001	SUB	4+“010”+3reg+3reg+3reg	SUB 2 registers, save to a destination register	4
0001	AND	4+“011”+3reg+3reg+3reg	AND 2 registers, save to a destination register	4
0001	OR	4+“100”+3reg+3reg+3reg	OR 2 registers, save to a destination register	4
0001	XOR	4+“101”+3reg+3reg+3reg	XOR 2 registers, save to a destination register	4
0010	ADDimm	4+“001”+3reg+6imm	ADD 2 registers, save to a destination register	4
0010	SUBimm	4+“010”+3reg+6imm	SUB 2 registers, save to a destination register	4
0010	ANDimm	4+“011”+3reg+6imm	AND 2 registers, save to a destination register	4
0010	ORimm	4+“100”+3reg+6imm	OR 2 registers, save to a destination register	4
0010	XORimm	4+“101”+3reg+6imm	XOR 2 registers, save to a destination register	4
0011	ADDindt	4+“001”+3reg+3reg+(3)	ADD 2nd reg. with data pointed by 1st reg.	5
0011	SUBindt	4+“010”+3reg+3reg+(3)	SUB 2nd reg. with data pointed by 1st reg.	5
0011	ANDindt	4+“011”+3reg+3reg+(3)	AND 2nd reg. with data pointed by 1st reg.	5
0011	ORindt	4+“100”+3reg+3reg+(3)	OR 2nd reg. with data pointed by 1st reg.	5
0011	XORindt	4+“101”+3reg+3reg+(3)	XOR 2nd reg. with data pointed by 1st reg.	5
0100-0110	reserved	4+…	Reserved for custom instructions	-
0111	SHIFTR	4+“000”+3reg+6shift	Shift the contents of a register right	4
0111	SHIFTL	4+“001”+3reg+6shift	Shift the contents of a register left	4
1000	MUL	4+3reg+3reg+3reg+3reg	Multiply 2 regs. Save in 2 regs.	5
1001	SIMDex	4+3reg+3reg+3reg+3op	SIMD execute two vectors, save into another	6+2i [b]
1010	LD	4+3reg+9address	Load a register from data memory	3
1011	ST	4+3reg+9address	Store a register to data memory	3
1100	BL [c]	4+3reg+3reg+6offset	Compare 2 regs, branch if first less than second	5
1101	BEQ [c]	4+3reg+3reg+6offset	Compare 2 regs, branch if first equals second	5
1110	SIMDld	4+3reg+3reg+(3)+“000”	Load an array to SIMD registers	5+3i [b]
1110	SIMDst	4+3reg+3reg+(3)+“001”	Store an array from SIMD registers	5+3i [b]
1110	SIMDrt	4+3reg+3reg+3dir+“010”	Route a value in SIMD regs. according to direction	6
1110	SIMDiter	4+9iter+“011”	Set the number of iterations to perform SIMD oper.	2
1111	MOVimm	4+“000”+3reg+6imm	Load a 6-bit immediate value to a register	4
1111	MOVind1	4+“001”+3reg+3reg+(3)	Load 1st reg with data pointed by 2nd reg.	5
1111	MOVind2	4+“010”+3reg+3reg+(3)	Store 2st reg into data mem. location pointed by 1st reg.	5
1111	LDport	4+“011”+3reg+3sel+“000”	Load from port selected by 3sel signal	4
1111	STport	4+“011”+3reg+3sel+“100”	Send to port selected by 3sel signal	4

a. ~: clock cycles.
b. i: iterations.
c. Can be altered with the configuration file.

The area occupation of the core on different data width configurations can be observed on Table 4.6, where it can be seen that these changes do not alter the size of the Control Unit, whereas the other hardware modules are proportionally increasing in size as a larger width is chosen. The effect of the dedicated FPGA modules can also be observed

here, as the multiplier and the memory units cause negligible area utilization, and are not included.

Table 4.6. SISD area information (slices)

Unit	Data Width			
	8-bit	16-bit	24-bit	32-bit
SISD (all)	279	426	573	689
ALU	68	148	231	310
Register File	20	40	60	80
Reg. MUX	22	44	66	88
Control Unit	186	186	186	186

4.4. The SIMD functionality

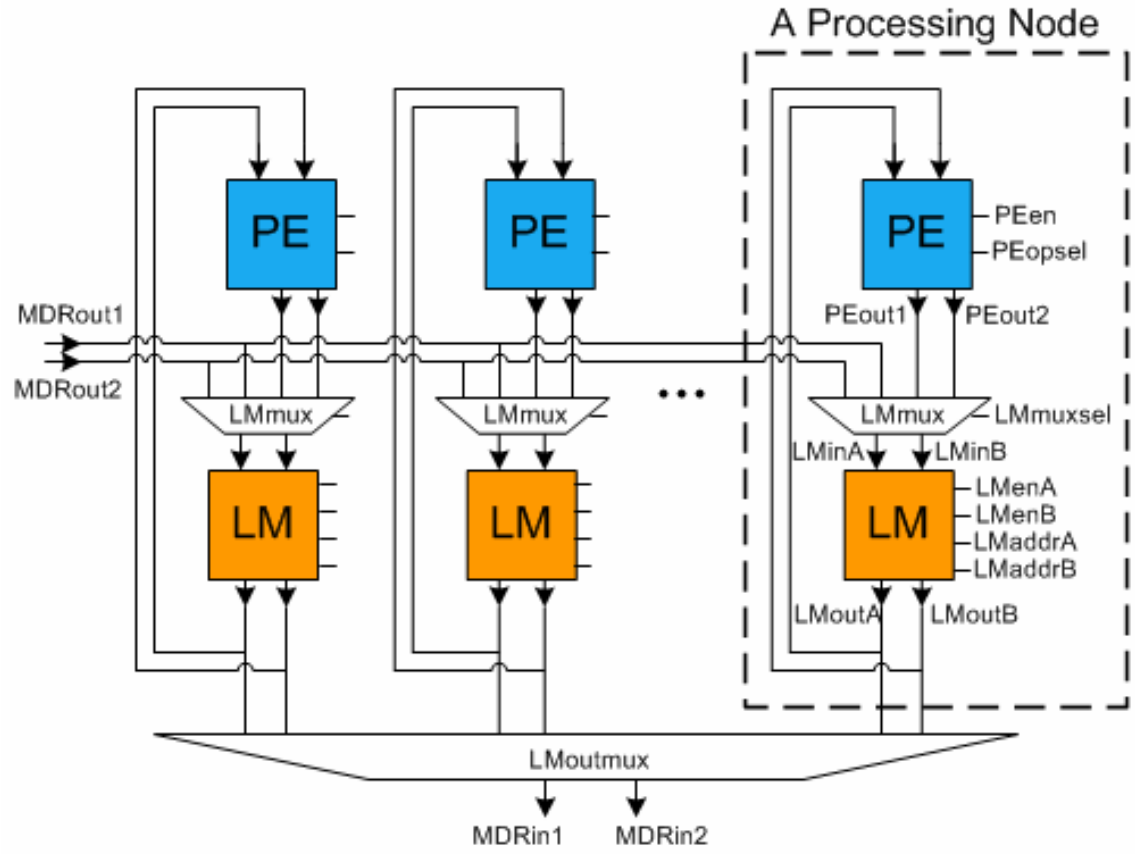


Figure 4.6. The SIMD unit

The SIMD functionality of the SIxD, depicted in Figure 4.6, is initiated within the configuration file by the *SIMDenable* signal set high. This produces the generation of a

number of user-defined Processing Nodes. A SIMD Processing Node is made up of a Local Memory (LM) and a Processing Element (PE), thus providing with the most elementary units for computation. The Local Memory (SIMD registers), which is implemented in a dual-ported Block RAM also needs to be able to take inputs (MDRout1 and MDRout2) from the main Data Memory of the SISD, causing the need for a multiplexer, completing a Processing Node (PN). The control of the SIMD, including the PEs, the LMs, and the multiplexers, are not managed by the Control Unit, but inside the SIMD itself.

The instructions that emerge with the activation of the SIMD are as shown in Table 4.7. Since VLIW is not used for these SIMD instructions, only register indirect addressing can be used, since otherwise there would be no way of issuing a SIMD instruction inside a 16-bit instruction word. In these instructions, the register contents act as address locations for the DM and the LMs. The *set_iter* instruction sets an iterations register, *iter*, with a constant, determining the number of times to iterate the given SIMD instruction. Let's say that a load vector instruction is being issued on the SIMD with four processing elements, and iter is set to eight. In this case, thirty-two values inside the DM are distributed dually (via dual-port Block RAMs) to eight consecutive locations in the four PEs. This way, the iterations register, *iter* achieves the speeding up of data transfers between the DM and the LMs. A large data multiplexer is placed at the output to select which Local Memory to read from when writing back to the main Data Memory.

Table 4.7. SIMD instructions of the SIxD

SIMD instruction	**Explanation**	**~** [a]
SIMDld	Loads a vector from DM to the LMs.	5+3i [b]
SIMDst	Stores a vector from LMs to the DM.	5+3i [b]
SIMDex	Execute two vectors based on the op_sel signal, save to a vector in LMs.	6+2i [b]
SIMDrt	Route a scalar from one LM to another.	6
SIMDiter	Set the number of iterations to perform.	2

a. ~: clock cycles.
b. i: iterations.

4.4.1. SIMD Customization

For the SIMD part, the customization process is similar to as explained on Appendix A, but differs in the sense that the new custom functional unit is added inside the Processing Element with a certain bit of the *op_sel* signal enabling it. For a double-input, single-output, single-cycle custom operation, no other modification of the core is necessary. If the custom unit outputs two results, setting the *double_output* bit high on the configuration file lets results to be written on two consecutive memory address locations.

4.4.2 Intra-PE communications

It is easy to implement a SIMD with communication autonomy [49]: by having the LM input multiplexers (Figure 4.5) take their inputs from all outputs of all PEs. Of course, especially for larger array units, this results in very large multiplexers and increased area usage. If an on-chip network is formed with routers in-between the LMs, these multiplexers can be removed. However, this method was shown to be insufficient in Section 3.3.3.

The slow but small method implemented in the SIxD employs none of these, but uses a pre-defined look-up table that contains encoded direction information for the movement of data between the LMs. The table consists of encoded bits of source and destination PEs, requiring no extra hardware, and only a few slices. For example, if the user defines “101” as a transfer from LM1 to LM4 in compile-time, using this bit sequence in the “route” instruction, along with the source LM address and the destination LM address, results in a data movement between these two Processing Units.

The area, performance and power metrics of the SIxD are completely dependent on the application, since the number and the contents of the Processing Elements determine the size of the SIMD mode of the SIxD core. Section 5 gives better insight on the utilization and performance of the core with an implementation of MPEG-7 Motion Activity Descriptors on the SIxD.

5. IMPLEMENTATION EXAMPLE ON THE SIXD: THE MPEG-7 MOTION ACTIVITY DESCRIPTORS

Although a detailed explanation can be found in [50], The MPEG-7 Motion Activity Descriptors (MAD) considered here are for motion intensity and spatial distribution of motion activity. The algorithm, depicted on Figure 5.1, can be summarized as follows: In a set of motion vectors, first the hypotenuses of 8-bit input pairs are calculated, and the average of the results are found. Then, the lower-than-the-average values of the vectors are set to zero, resulting in the spatial activity matrix, and another averaging of this matrix gives us the intensity of motion for the frame. The descriptor implementation which works with pre-calculated motion vectors has been employed on 16x16 motion vectors, requiring two sets of 256 elements each. Although the inputs are 8-bits wide, a 16-bit CPU was used because of the usage of multiplication and averaging. For the sake of processing comparison, data is previously loaded from the input ports, and is available in Data Memory.

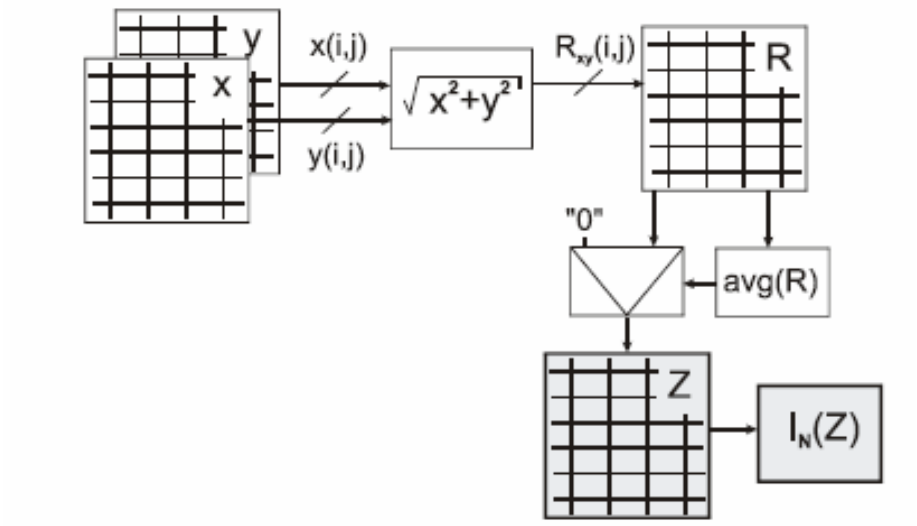
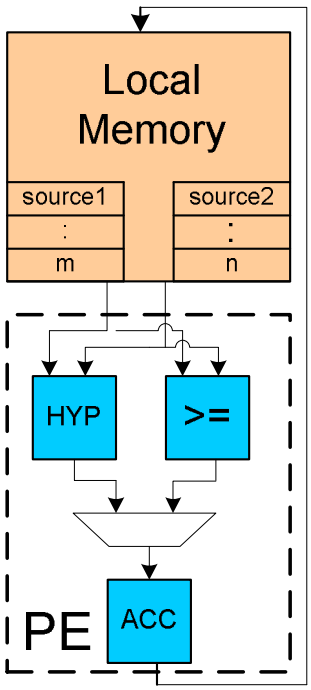


Figure 5.1. The MAD algorithm [50]

5.1. The MAD Algorithm Implementations

Fig. 5.2 shows how the PE operation bits (op_sel) are used to manage the hypotenuse unit, the compare unit, and the accumulator. In the algorithm, first the hypotenuses are calculated in parallel, after which, the outputs are collected and summed together by the accumulator. Then, the final accumulator results are gathered by the SISD, added and shifted to compute the average, and sent back to the PNs of the SIMD. Afterwards, the compare instruction comes into play, comparing the data elements with the average, outputting zero if less than the average, or the input data itself if greater than or equal to the average, forming the spatial activity matrix. It also accumulates its results, which are finally saved, added and scaled together for the final result, the intensity of motion for the frame.



Op_sel (3bits)	PE operation
001	Hypotenuse
010	Output accumulator
011	Hypotenuse+accumulate
100	Compare with average
110	Compare+accumulate

Figure 5.2. The processing node for the MPEG-7 motion activity descriptors and the PE operation table

There are many ways to implement this “process-and-accumulate-intensive” application. Some options are: custom square root unit, custom hypotenuse unit and SIMD customization. On the bare SISD core without a custom unit, an 8-bit square root

calculation takes about 280 clock cycles. On one implementation, a custom unit designed for the 8-bit square root calculation and placed in opcode “0100” takes 5 cycles (including instruction decode) and 55 slices to compute the square root. The whole algorithm runs for a total of 21,286 clock cycles, 468 us on an 11ns (90 MHz) clock cycle.

Another method is to use the specially-designed hypotenuse custom unit, taking 76 slices, and 5 clock cycles to execute. Using this unit, two MUL, one ADD, and one SQRT operation is combined into one HYP instruction, with the total MAD runtime reduced to 19,413 clock cycles, 427 us on a 11ns (90 MHz) duty cycle.

Finally, enabling the SIMD and using vector instructions on specially designed Processing Elements that execute in two clock cycles (Figure 5.2) and occupying 111 slices, the results are shown in Table 5.1.

Table 5.1. The MPEG-7 motion activity descriptors on the SIxD

No. PN	Iter	Total slices	SIMD slices	Max. freq. (MHz)	Runtime (clock cycles)	Power cons. (mW)
2	128	963	475	109	1833	521
4	64	1205	717	94	1196	548
8	32	1647	1163	94	882	542
16	16	2523	2049	89	734	566
32	8	4266	3800	94	678	606
64	4	7754	7288	83	686	647
128	2	14780	14308	108	762	689
256	1	28,720	28243	100	944	713

The cost vs. performance graph on Figure 5.3 shows how the performance is inversely proportional to the area occupied by the core. However, when the chosen number of Processing Nodes of the SIxD is redundant for the algorithm, the design not only takes up too much space and consumes more power, but also is slower and inefficient. Experiments done with 128 and 256 PEs resulted in 14,780 and 28,720 total slices and 762 and 944 clock cycles to execute, respectively. This is due to the fact that the data transfer starts taking more time than the processing. The ideal number of PNs seems to be 16 or 32 (16 and 15 us, respectively on 11 ns clock cycle) for the Motion Activity Descriptor on a 16x16 motion vector frame, depending on area or time being a constraint. Furthermore, the 256-PE SIMD (depicted SIMD-256) does not fit even fit the biggest Xilinx Virtex-II chip

with eight million gates because it occupies 259 Block RAMs and 513 MUL units, more than the chip resources.

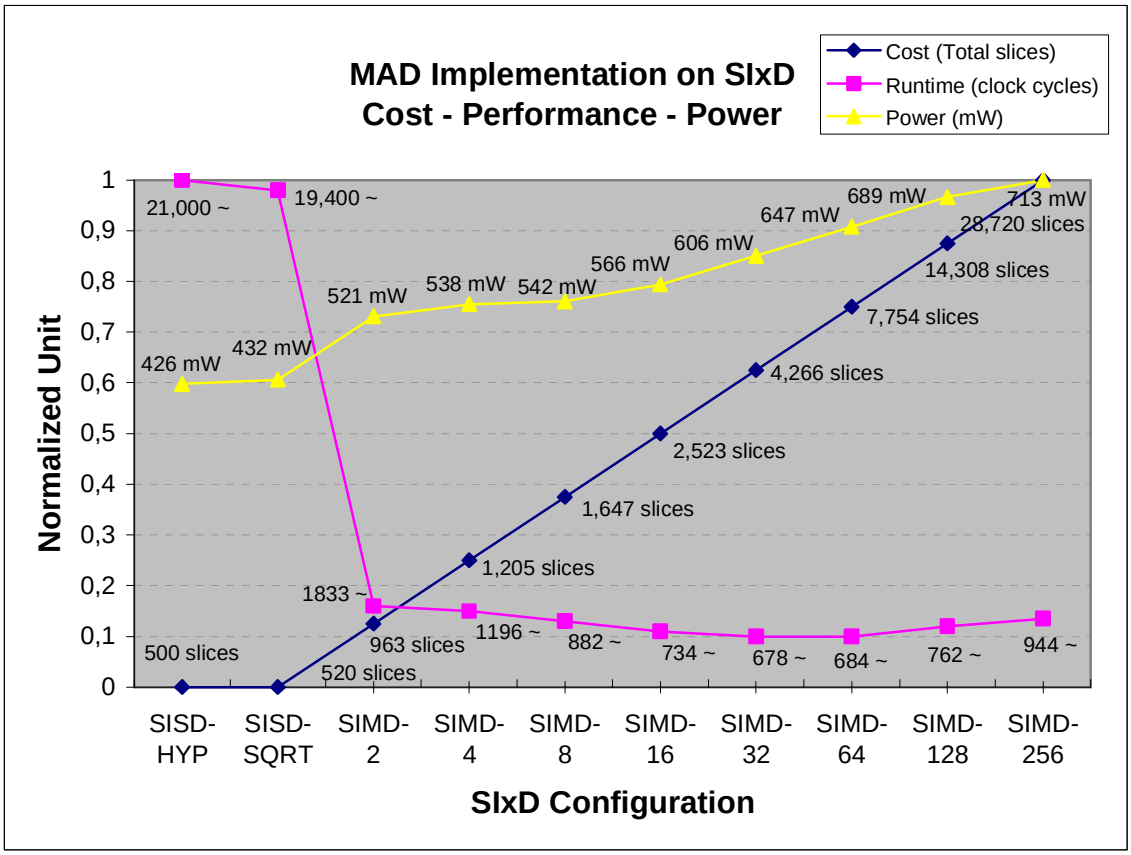


Figure 5.3. Cost-performance-power graph of the MAD algorithm optimized SIxD core

The MAD program using scalar instructions (Appendix B) takes between 4310 and 4566 instructions and 21,286 clock cycles to run, so the resulting average CPI (Cycles per Instruction) is between 4.94 and 4.66 and 2.1 MIPS. The MAD program using SIMD instructions (Appendix C) takes 42 instructions, regardless of the number of Processing Nodes selected. The system is able to run at 66 MIPS and the average CPI is 16.1. This means that although the time to complete an average instruction is about 3-folded, the SIMD instructions do more, accomplishing a 31-fold MIPS speedup, for an area increase of 8.5 times of the SISD.

6. CONCLUSIONS AND FUTURE WORK

6.1. The Partial Runtime Reconfiguration of the SIxD

Partial reconfiguration is useful for applications that require different designs to be loaded into the same area of a chip, or that require the ability to change portions of a design without having to reset or reconfigure the entire chip [51]. There are two methods of the partial reconfiguration of a Xilinx device, the difference-based and the module-based methods. The difference-based method uses saved partial configuration bitstreams for making small changes in the FPGA, such as replacing LUT or Block RAM contents. The module-based method makes use of defined reconfigurable modules that communicate by boundary-defining bus macros on the FPGA, such that no signal crosses the module boundaries except for communications signals via these bus macros and global clocks.

In an attempt to increase the performance and the capabilities of the system, the partial dynamic reconfiguration (Figure 6.1) of some functionalities of SIxD can be considered:

- The contents of the Custom Units (Section 4.2.7) and the Processing Elements (Section 4.4) could be reconfigured from a hardware library, which would enable a wider range of algorithms to be run on the SIxD. Depending on the instruction needed, the necessary unit(s) can be fetched at runtime into the Custom Unit and/or Processing Element spaces, whose boundaries are defined by bus macros, ready for computation. The reconfiguration time would depend solely on the size of the Custom Unit/Processing Element and the overhead required for reconfiguration.
- Intra-PE communications (Section 4.4.2) could be reconfigured during runtime, since as the number of PNs increases, a greater variety of intra-PE communications is necessary. The look-up table that keeps the encoded routing information occupies only a few slices, therefore its reconfiguration could be done very quickly using the difference-based method.

- The ability to change the number of PEs during runtime could also be of use. Increasing the number could result in a performance increase in many cases, on the other hand, having less PEs could provide with more space on the FPGA for other applications. Additional bus macros would be necessary to define the boundaries where the extra PEs will be placed.

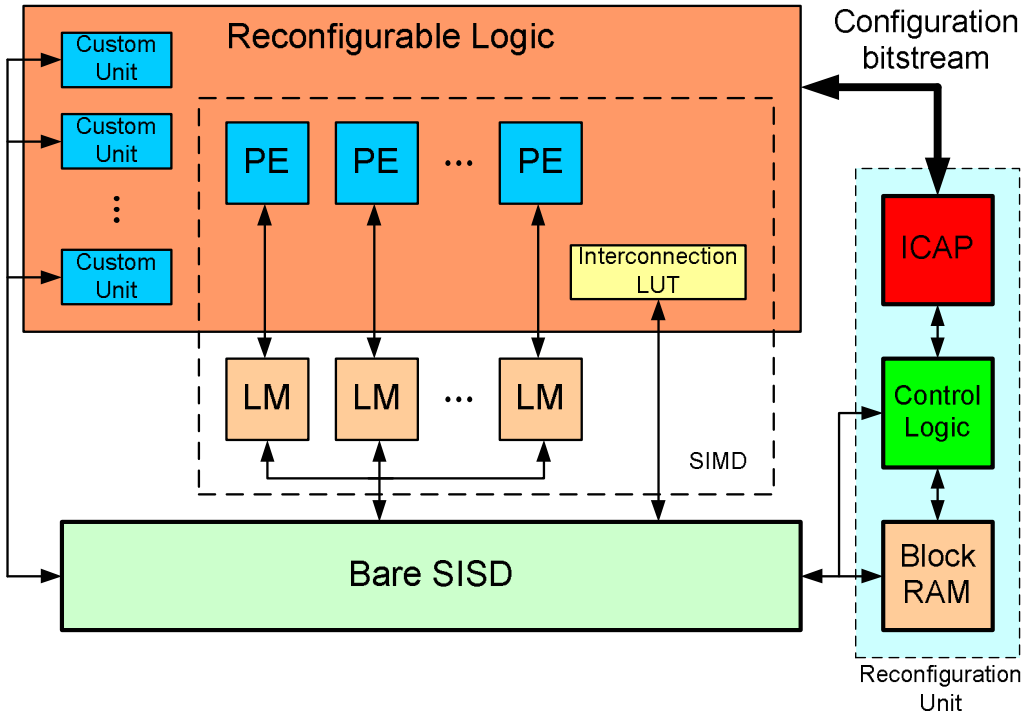


Figure 6.1. The reconfiguration system

A PROM initially configures the FPGA; however bitstreams that are small enough could be allocated on on-chip Block RAM, so that the reconfiguration can be done on the FPGA itself, improving the reconfiguration time. Xilinx Virtex devices have an Internal Reconfiguration Access Port (ICAP), which can be controlled by internal FPGA logic [52]. ICAP provides an 8-bit input data bus and an 8-bit output data bus which can be used by internal logic to reconfigure and read back configuration memory. Data is loaded into the FPGA on a column basis, with the smallest unit being a configuration frame of the bitstream. There are multiple frames per CLB column, so the frame size is device size dependent. For example, there are 3392 bits per frame in a XC2V1000 [53], which has 5120 slices in total. Since the maximum frequency that the ICAP interface can be clocked at is 66 MHz, the minimal time to reconfigure over ICAP a single XC2V1000 frame at 66

MHz is 13us [54]. The Virtex-II 1000 having a total of 1536 frames to configure for its 5120 slices, yields to a rough estimate of a thousand bits to reconfigure on average per slice, for which the minimum time to reconfigure is about 4 us. Therefore, the reconfiguration of the PE used at Section 5 (92 slices) would take about 360 us per PE, while the insertion of the SQRT unit (55 slices) would take about 215 us.

In the reconfiguration system proposed in Figure 6.1, the bare SISD core does the job of running the software for reading and writing the reconfiguration data that is kept inside the Block RAM, which is used as a cache for partial bitstreams. The control logic acts as a software driver for reading and writing to ICAP interface, while the ICAP accesses the internal configuration of the FPGA. The reconfigurable logic part, which incorporates the units to be altered by the loaded bitstreams, should be well-placed on the FPGA using bus macros [51].

Obviously, the time for reconfiguration takes too long for a single iteration of the MAD algorithm on the SIMD, but the introduction of the SQRT or the HYP custom units to the bare SISD makes a positive impact on the algorithm runtime. Multiple iterations of any custom-unit-aided complex algorithm run on the runtime-reconfigured SIxD could result in a performance increase compared to when no reconfiguration methods are used, provided that essential units suitable for the algorithms are readily available inside the hardware library.

6.2. SIxD Conclusions

FPGA-based embedded systems design methodologies have helped software flexibility to represent hardware functionality, providing the designer with many different modular and architectural options. The SIxD has been prosperous in representing different computer architectures with a single program, as well as configurations based on the requirements of the target application, providing several options for the set of instructions and the representation of data.

In this study, the implementation of a flexible and customizable soft processor was presented. The soft core can be configured to run in scalar mode and fit in small FPGAs or

run with vector processing (SIMD) capability for higher performance on larger chips. A look at the architectures of modern commercial and academic soft cores facilitated to perform an architectural exploration on three customizable soft core examples. Then the configurable customizable soft core, the SIxD was presented and was demonstrated with an implementation of the MPEG-7 Motion Activity Descriptors, showing how the core might be configured for up to 31-fold speedup of algorithm runtime. A method for the dynamic reconfigurability of the core, and how this might improve its processing capabilities, was finally discussed.

This thesis has also presented in detail with how such a soft core is developed in VHDL, but does not necessarily mark the end of related studies at all. Thanks to the flexibility of VHDL and Verilog, the infinitely many design options are only limited by the core designer's interests, capabilities, and time.

Architecturally, the addition of MIMD and MISD options to the core would complete the taxonomy of Flynn. Furthermore, the comparisons between the SIxD and its pipelined version could also be of interest. A compiler to enable the conversion from high level languages to the SIxD assembly language is absolutely essential. This compiler could also parse the code and decide which architectural option best fits the requirements. Furthermore, critical kernels in the code could be detected, and custom instructions to be mounted onto the architecture could be provided. Ideally the inputs to the compiler should be the program code, the constraints for size, performance and power, and the target FPGA. Containing the FPGA device information, the compiler can come up with the most appropriate system for the application.

Although size and performance has been of primary importance in this study, the low-power design considerations that are quite important for embedded systems design have not been discussed. Also, more peripheral units such as timers, watchdogs, and UART (Universal Asynchronous Receiver/Transmitter) can be included in the core. Another addition to the system could be to have the SIMD operate parallel to the SISD, as long as distinct data addresses are operated on. This is not a very difficult implementation option that could provide with even higher performance. Also, the instruction word can be easily upgraded to 18 bits of width, since the Block RAMs are actually that wide. The extra

two bits could be used to increase the number of registers to 16, or to create more space for custom instructions.

APPENDIX A: THE SIxD USER'S MANUAL

1) Open Project in Xilinx ISE 7.1.
2) Open sisdconst.vhd, set up all necessary constants.
3) Custom Unit extension:
    a. Add the custom VHDL file to the ISE Project.
    b. Open sisd.vhd.
    c. Add the component declaration of the custom unit VHDL file among the other component declarations in sisd.vhd. Also declare the necessary new signals.
    d. Instantiate the new custom component.
        i. Connect the previously declared signals:
            1. As the clock signal, use *clk*. For reset, use *rst*.
            2. As the inputs of the custom unit, use *RFout1* and *RFr2out*, the outputs of the register files. For a single output, use *RFr1out*.
        ii. Connect the output of the custom unit to an unused RFmux input, named *regmuxin0* through *regmuxin7*. Note that the *regmuxsel* signal that is used to enable the correct multiplexer output is the three-bit representation of the number at the end of this signal name.
        iii. Editing control.vhd for the enable signal and other control signals:
            1. Open control.vhd.
            2. Add the new ports related to the custom unit to the entity. Have the new ports to control.vhd the same size, same unit and for the sake of simplicity, the same name as of the custom unit.
            3. Add new states to the control Finite State Machine. This is done by finding the state declaration line: "type statet is (S1,S2,S3,S4a,S4b,S4c,S4d,S4e…", and by adding new states to the list in parentheses, such as S30, S31 etc.
            4. Associate the new instruction with an opcode.

a. Find the line “case IR(15 downto 12) is”, followed by “when "0000"=>”. This is the case statement that declares the opcodes.
b. Choose an unused opcode, for example, “0100”.
c. Following the line “when "0100" =>”, place the necessary state information. Some common action examples are:
    i. “RFr1sel_ctrl <= IR(8 downto 6);”, selecting to output the contents of the register file through the first port, addressed by the bits 8 through 6 of the instruction word.
    ii. “RFr1en_ctrl <= '1';”, enabling the RF first port output.
    iii. “ALUen_ctrl <= '0';”, disabling the ALU, since it is not being used at the current state.
    iv. “State <= S31;”, next state information.
d. Declare all necessary states with correct state transitions, and proper enabling and disabling of signals. For a write back to the Register File, use “RFwe_ctrl <= '1';” for write enabling. On the final state of the custom operation, make sure to include, “PCC:=PCC+1;”, incrementing the program counter, and “state <= S1;”, for the return to the initial state of the Control FSM.

5. Connect the newly added control outputs to the appropriate custom unit signals.

APPENDIX B: THE MAD PROGRAMS

```
F000    MOVimm R0, 0          --mempointer=0
F280    MOVimm R1,32          --memptrend=32
7210    SLA R1,4              --shift left R1 by 4 bits=MUL by 16
FE00    MOVimm R7, 0          --memflag=0
E440    Ien 0001              --enable interrupts for port A
DFBF    BEQ R7,R6,0           --wait while memflag=0
Interrupt (save regs)
F403    MOV R2, PortA         --move portA input to R2
F082    MOV (R0), R2          --store R2 into (pointed by R0(which is the mempointer))
1005    ADDimm R0,1           --memptr++
D041    BEQ R0, R1, 1         --memptr=?memptrend
(restore regs)
E000    RET                   --return from interrupt
FE04    MOVimm R7,1           --memflag=1
E400    Ien 0000              --disable all maskable interrupts
(restore regs)
E000    RET                   --return from interrupt
```

Figure B.1. Single port data load for the MAD algorithm

```
F000    MOVimm R0, 0          --memptr=0 x values
F280    MOVimm R1,32          --memptrend=32
7210    SLA R1,4              --shift left R1 by 4 bits=MUL by 16
FA00    MOVimm R5, 0          --Accumulator for Rxy's
F404    MOVimm R2, 1          --memptr2=1 y values
FD00    MOVimm R6, 64         --start address for Rxy=64
7C0C    SLA R6,3              --shift left R6 by 3 bits=MUL by 8
F601    MOVindrt R3, (R0)     --first x value
F881    MOVindrt R4, (R2)     --first y value
86D8    MUL r3,r3,r3          --x^2
8920    MUL r4,r4,r4          --y^2
16E0    ADD r3,r3,r4          --x^2+y^2
96C0    SQRT r3,r3            --SQRT(x^2+y^2)
FCC2    MOVindrt (R6), R3     --save result in Rxy table
1009    ADDimm R0, 2          --increment x value pointer by 2
1409    ADDimm R2, 2          --increment y value pointer by 2
1C05    ADDimm R6, 1          --increment Rxy table pointer by 1
1B58    ADD r5,r5,r3          --accumulate Rxy result
C474    BL R2, R1, (-12)      --loop if not end of table
6A20    SRA R5, 8             --dvide the accumulator for the average
FE00    MOVimm R7, 0          --accumulator for AVG2
F000    MOVimm R0, 0          --initialize counter
FD00    MOVimm R6, 20H        --point to beginning of Rxy table
7C0C    SLA R6,3              --shift left R6 by 3 bits=MUL by 8
F781    MOVindrt R3, (R6)     --move value to R3 to be added to AVG2
C742    BL R3, R5, 2          --if value lower than avg (R5), save a 0 instead
1F08    ADD R7,R7,R3          --accumulate AVG2
D001    BEQ R0,R0,1           --jump 1
FC02    MOVindrt (R6), R0     --saving a 0 for the lower-than-average value
1C05    ADDimm R6,1           --increment Rxy counter
2209    SUBimm R1, 2          --decrement counter
C078    BL R0,R1,(-9)         --if counter=0 stop
6E20    SRA R7,8              --Divide the final result of the accumulator to obtain avg.
1C05    ADDimm R6,1           --increment Rxy counter
FDC2    MOVimm (R6), R7       --store the result
```

Figure B.2. The MAD algorithm program on the SISD

```
--There are four Processing Nodes.

F000    MOVimm R0, 0                --R0=0
F300    MOVimm R1,32                --R1=32
1249    ADD R1, R1, R1              --R1=64
1289    ADD R2, R1, R1              --R2=128
1389    ADD R6, R1, R2              --R6=R1+R2=192
13D2    ADD R7, R2, R2              --R7=256
E1FB    iter=63                     --iter=63 (64 iterations)
E000    SIMDld R0 R0                -- load 128*4=512 values to SIMD regs
940B    SIMDexec R2 R0 R1 "011"     --(op:011-hyp_acc)
E003    iter=0                      --one iteration
9C02    SIMDexec R6 R0 R0 "010"     --SIMD execute  (op:010-acc_out)
E181    SIMDstore R0 R6             --store from LM(R6) to DM(R0), R0+1,R0+2,R0+3
FA00    MOVimm R5, 0                --R5=0
F340    ADDindrt R5 (R0)
2201    ADDimm R0, 1                --R0=+1
F340    ADDindrt R5 (R0)
2201    ADDimm R0, 1                --R0=+1
F340    ADDindrt R5 (R0)
2201    ADDimm R0, 1                --R0=+1
F340    ADDindrt R5 (R0)
7142    ShiftR R5, 2                -- divide R5 by 4
BA00    store R5, 0
BA01    store R5, 1
BA02    store R5, 2
BA03    store R5, 3
EC00    SIMDld R6, R0               --DMsrc=R0, LMdest=R6
E1FB    iter=63                     --iter=63
9036    SIMDexec R0 R0 R6 "110"     -- (op:110-compare_acc)
E003    iter=0                      --iter=0
9C32    SIMDexec R6 R0 R6 "010"     --(op:010-acc_out)
E203    iter=64                     --iter=64
F180    MOVimm R6, 0                --R6=0
EC81    SIMDstore R6, R2
33B8    ADDindrt R6 (R7)
23C1    ADDimm R7, 1                -- R7=+1
33B8    ADDindrt R6 (R7)
23C1    ADDimm R7, 1                -- R7=+1
33B8    ADDindrt R6 (R7)
23C1    ADDimm R7, 1                -- R7=+1
33B8    ADDindrt R6 (R7)
7188    ShiftR R6, 8                --div by 256 for overall average
F5F0    MOVindrt2 (R7) r6           --store the average
```

Figure B.3. The MAD algorithm program using SIMD functionality

APPENDIX C: RELATED PAPERS

The following paper was accepted for publishing as a result of the work described in this thesis:

Sonmez, N. and A. Yurdakul, “SIxD: A Configurable Application-Specific SISD/ SIMD Microprocessor Soft-Core”, *Proceedings of International Symposium on System-on-Chip*, Tampere, Finland, November 2006, to appear.

REFERENCES

1. Vahid F. and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*, John Wiley and Sons, 2001.

2. Gokhale, M. et al., "Splash: A reconfigurable linear logic array", *Proceedings of the International Conference on Parallel Processing*, Urbana-Champaign, IL, USA, pp. 526-532, 1990.

3. Smith, A. et al., "PRISM II Compiler and Architecture", *Proceedings of IEEE Workshop on FPGA-based Custom Computing Machines*, 1993.

4. Lu, G., H. Singh, M.Lee, N. Bagherzadeh, F.Kurdahi, and E. M. C. Filho. "The Morphosys parallel reconfigurable system", *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, Toulouse, France, pp. 727-734, 1999.

5. Gonzalez, R.E., "Xtensa: A Configurable and Extensible Processor," *IEEE Micro*, vol. 20, no. 2, pp. 60-70, 2000.

6. Lysecky R. and F. Vahid, "A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning", *Design Automation and Test in Europe Conference (DATE)*, pp. 480-485, 2004.

7. Xilinx Inc., *MicroBlaze Processor Reference Guide*, http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf.

8. Xilinx Inc., *PicoBlaze 8-bit Embedded Microcontroller User Guide*, http://www.xilinx.com/bvdocs/userguides/ug129.pdf.

9. Altera Inc., *Nios Embedded Processor*, http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf.

10. Peleg A.and U. Weiser, “MMX Technology Extension to the Intel Architecture,” *IEEE Micro*, vol. 16, no. 4, pp. 42-50, Aug. 1996.

11. Diefendor K., P. Dubey, R. Hochsprung, H. Scales, “AltiVec Extension to PowerPC Accelerates Media Processing”, *IEEE Micro*, vol. 20, no. 2, pp. 85-95, Mar/Apr 2000.

12. Sun Microsystems, Sparc VIS, http://www.sun.com/processors/vis/.

13. Klimovitski, A., “Using SSE and SSE2: Misconceptions and Reality”, *Intel Developer Update Magazine*, 2001.

14. Oberman, S., G. Favor and F. Weber, “AMD 3DNow! Technology: Architecture and Implementations”, *IEEE Micro*, vol. 19, no. 2, pp. 37-48, 1999.

15. Freedom CPU, http://f-cpu.seul.org/.

16. Etiemble, D. and L. Lacassagne, “Introducing image processing and SIMD computations with FPGA soft-cores and customized instructions”, *Proc. 1st International Workshop on Reconfigurable Computing Education*, Karlsruhe, Germany, 2006.

17. Jones, A. K., R. Hoare, D. Kusic, J. Fazekas, and J. Foster, “An FPGA-based VLIW Processor with Custom Hardware Execution”, *Proc. ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 107-117, 2005.

18. Compton, K. and S. Hauck, “Reconfigurable Computing: A Survey of Systems and Software”, *Proc ACM Computing Surveys*, Vol. 34, no. 2. pp. 171-210, 2002.

19. DeHon, A., *Reconfigurable architectures for general-purpose computing*, Ph.D. dissertation, Massachusetts Institute of Technology, 1996.

20. Zeidman, B., “Introduction to Programmable Systems on a Chip”, *EEtimes*, http://www.eetasia.com/art_8800373653_499485_c4c4d1f8200508.htm.

21. Xilinx Inc., http://www.xilinx.com.

22. Altera Inc., http://www.altera.com.

23. V. Madisetti and L. Shen, “Interface Design for Core-Based Systems”, *IEEE Design & Test of Computers*, vol.14, no.4, p.42-51, 1997.

24. Cofer R.C. and B. F. Harding, *Rapid System Prototyping with FPGAs: Accelerating the Design Process*, Newnes, 2005.

25. Mosher, R., “Choosing hardware IP”, *Embedded Systems Design*, http://www.embedded.com/showArticle.jhtml?articleID=178600378, 2006.

26. Lau, D., J. Blackburn and J. A. Seely, “The use of hardware acceleration in SDR waveforms”, *Altera Corporation SDR Forum*, November 2005.

27. Yiannacouras P., *The Microarchitecture of FPGA-Based Soft Processors*, Masters Thesis, University of Toronto, 2005.

28. Tensilica Inc., TIE compiler, www.tensilica.com/pdf/TIE_V6.qxd.pdf.

29. Tensilica Inc., *Xtensa LX Configurable Processor Core*, http://www.tensilica.com/products/lits_docs.htm.

30. Jani D., G. Ezer and J. Kim, "Long Words and Wide Ports: Reinventing the Configurable Processor", *Hotchips*, August 2004.

31. Plavec, F., *Soft Core Processor Design*, Masters Thesis, University of Toronto, 2004.

32. CPUgen, http://www.opencores.org/projects.cgi/web/cpugen/overview.

33. Gray, J., “Building a RISC System in an FPGA: Part 1: Tools, Instruction Set, and Datapath; Part 2: Pipeline and Control Unit Design; Part 3: System-on-a-Chip Design”, *Circuit Cellar Magazine*, no. 116-118, March-May 2000.

34. LCC, A Retargetable Compiler for ANSI C, www.cs.princeton.edu/software/lcc/.

35. Project OpenUP, www.dte.eis.uva.es/OpenProjects/OpenUP/index.htm.

36. OpenRISC 1000, http://www.opencores.org/projects.cgi/web/or1k/overview.

37. Bose B., M. Tuna, and J. Nagy, “Lavacore configurable java processor core”, *Aerospace Conference Proceedings*, vol. 4, pp. 1953-1959, 2002.

38. Gaisler, J., “A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture”, *International Conference on Dependable Systems and Networks*, pp. 409, 2002.

39. Flynn, M. J., “Some Computer Organizations and Their Effectiveness”, *IEEE Transactions on Computing*, vol. 21, no. 9, pp. 948-960, September 1972.

40. Dally, W. J., and B. Towles, “Route Packets, Not Wires: On-Chip Interconnection Networks”, *Proceedings of the 38th Design Automation Conference (DAC)*, Las Vegas, June 2001.

41. Cooley, J. W. and O. W. Tukey, “An Algorithm for the Machine Calculation of Complex Fourier Series.” *Mathematics Computation*, no. 19, 297-301, 1965.

42. Chu, E. and A. George, *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*, Boca Raton, FL, CRC Press, 2000.

43. Xilinx Inc., ISE Foundation, http://www.xilinx.com/ise/logic_design_prod/foundation.htm.

44. Modelsim SE 6.0, http://www.model.com.

45. Xilinx Inc., Virtex II Complete Data Sheet, http://direct.xilinx.com/bvdocs/publications/ds031.pdf.

46. Moraes, F.G. et al. "HERMES: an Infrastructure for Low Area Overhead Packet-Switching Networks on Chip". *Integration, the VLSI Journal*, vol. 38-1, pp 69-93, 2004.

47. Sheldon D., R. Kumar, F. Vahid, R. Lysecky, and D. Tullsen. "Application-Specific Customization of Parameterized FPGA Soft-Core Processors", International Conference on Computer-Aided Design, San Jose, November 2006.

48. Xilinx Inc., *"Xilinx Application Note XAPP467: Using Embedded Multipliers in Spartan-3 FPGAs"*, v1.1, May 13, 2003.

49. Narayanan, P. J., "Processor Autonomy on SIMD Architectures", *Proc. ACM International Conference on Supercomputing*, pp. 127-136, July 1993.

50. Savakis A., P. Sniatala and R. Rudnicki, "Real-time video annotation using MPEG-7 motion activity descriptors", *MIXDES 2003*, Lodz, June 2003.

51. Xilinx, Inc., "Two Flows for Partial Reconfiguration: Module Based or Difference Based", v1.1, Nov 2003.

52. Blodget B., S. McMillan, and P. Lysaght. "A lightweight approach for embedded reconfiguration of FPGAs", *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pp. 399 - 400, 2003.

53. Blodget B., P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan “A Self-reconfiguring Platform”, *Proceedings FPL 2003*, Lisbon, Portugal, September 2003.

54. Ullmann M., B. Grimm, M. Hübner and J. Becker, “An FPGA Run-Time System for Dynamical On-Demand Reconfiguration”, *RAW/IPDPS 04*, Santa Fe, New Mexico, USA, April 2004.

REFERENCES NOT CITED

Ashenden, P. J., The *Designer's Guide to VHDL*, Morgan Kaufmann Publishers, 1995.

Flynn M. J., *Computer Architecture: Pipelined and Parallel Processor Design*, Jones and Bartlett, Boston, 1995.

Hwang K., *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw Hill, 1993.

Patterson D.A. and J.L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 1994.