CHALMERS

Comparison of Synthesizable Processor Cores

KLAS WESTERLUND

Master's Thesis

Electrical Engineering Program

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Division of Computer Engineering
Göteborg 2005

Abstract

The purpose of this thesis work has been to compare two synthesizable processor cores, the LEON2 from Gaisler Research and the NIOS II provided by Altera.

The work consists of three parts:

1. Initial Core Analysis
2. Implementation on a FPGA
3. Performance evaluation by Benchmarking

In the analysis part, the processor architecture of each core and characteristics like pipeline depth, cache sub–system and configurability have been evaluated. Both processor cores have been implemented on the same target FPGA board. In the benchmark part, Dhrystone, Stanford, a typical control application and Paranoia. The first three programs have been executed on two different processor configurations; ”Minimum Area” and ”Maximum Performance” respectively, and in two different frequencies. Paranoia was only executed on the ”Minimum Area” configuration in one frequency. To make the benchmark part possible, a cross–compiler tool chain for each processor system have been used.

The benchmark results are discussed and evaluated. Both processor cores perform equal on Dhrystone and Stanford on the ”Minimum Area” configuration, but LEON2 is the fastest one on the ”Control Application”. On the ”Maximum Performance” configuration, NIOS II is fastest on Dhytstone and Stanford. LEON2 performs best on the ”Control Application” again.

Sammanfattning

Syftet med detta examensarbete har varit att jämföra två syntetiserbara processorer, LEON2 som Gaisler Research har utevecklat och NIOS II som Altera tillhandahåller.

Arbetet består av tre delar:

1. Jämförelse av processorerna
2. Implementering på ett FPGA utevecklingskort
3. Prestantautvärdering med hjälp av benchmarkprogram

I den första delen av arbetet, har processorernas arkitektur, karaktäristiska delar så som antalet pipeline–steg, cachesystem och konfigurerbarhet jämförts och utvärderats. De båda processorerna har implementerats på samma utvecklingskort baserat på en Altera Cyclone FPGA. Som prestandautvädering har fyra program körts på de båda processorerna; Dhrystone, Stanford, en typisk styrapplikation och Paranoia. De tre första programmen har körts på två olika processorkonfigurationer, ”Minimum Area” respektive ”Maximal Prestanda” vid två olika frekvenser. Paranoia kördes enbart på ”Minimum Area” konfiguration vid en frekvens. För att göra prestandautväderingen möjlig måste ett korskompilatorsystem användas för de båda processorerna.

Till sist har prestandautvärderingsresultaten har diskuterats och utvärderats. De båda processorerna har likvärdig prestanda på ”Minimum Area” för Dhrystone och Stanford, medan LEON2 är snabbare på ”Styrapplikationen”. Vid ”Maximal Prestanda” är NIOS II snabbare på Dhrystone och Stanford än LEON2, medan LEON2 är snabbare på ”Styrapplikationen”.

Acknowledgments

I would like to thank my supervisor Jiri Gaisler, Edvin Catovic and the other Gaisler Research staff for supporting me during this thesis work.

At last, I would also thank my examiner Lars Bengtsson at the Department of Computer Engineering at Chalmers for undertaking my thesis work.

Klas Westerlund
Gothenburg August, 2005

Contents

List of Tables

Objectives

Today, synthesizable processor cores are becoming common for embedded microprocessor based applications, where high performance is required. Because of the Field Programmable Gate Arrays (FPGA) are becoming bigger and faster, they can contain a complete microprocessor based system. The first syntesizable processor cores were 8 bit and showed up in the late 1990's, now there are 32 bit processor cores available. In this context, it is of interest to make a comparative analysis with synthesizable processor cores from different providers.

In this thesis work, two syntesizable processor cores have been compared, the LEON2 [1] which is a SPARC V8 compatible processor core developed by Gaisler Research [2] and the NIOS II which is[3] developed by Altera [4].

The work consists of three major parts: processor architecture and system analysis, implementation on a FPGA and benchmarking. Two different processor configurations have been compared and evaluated: minimum area and maximum performance. Both configurations have been executed in two different frequencies: 25 MHz and 50 MHz, respectively. The benchmarks used in this work are Dhrystone, Stanford, Paranoia and a typical control application, the execution results have been discussed for each configuration.

Licensing and Availability

This section contains a evaluation of their license forms, respectively. The LEON2 full VHDL source code is available under the GNU LGPL [5] license, which allows free and unlimited use of the processor core and peripherals. Since it is open source, it is not restricted to a certain technology. LEON2 based systems could be implemented in both FPGA and ASIC. The full LEON2 source code is available through the Gaisler Research homepage [6].

All NIOS II development kits includes a perpetual non–cost license [7] to develop and ship systems using the processor core and peripherals in a Altera™ FPGA. A implementation as a ASIC is also possible. The NIOS II is distributed as a encrypted VHDL file.

1 Initial Architecture Analysis

This section contains the first part of the work. In section 2 the LEON2 is described and analyzed, in section 3 the same kind of description and analysis is done for the NIOS2.

2 LEON2

This section contains a description of the architecture of the processor core, cache hierarchy, the instruction set, available peripherals and configuration options.

2.1 System Overview

The LEON2 [8] implements a 32 bit single issue SPARC V8 [9] compatible processor core.
It is designed for embedded applications, with the following features on–chip:

Separate Instruction– and Data Caches (Harvard Architecture)
Hardware Multiply and Divide
Flexible Memory Controller
Parallel 16/32 Bits I/O Port
Ethernet MAC
PCI Interface
Two UARTs
Interrupt Controller
Two 24–bit Timers
Debug Support Unit with Trace Buffer
Watchdog
Power-down Function
Is Fully Synthesizable VHDL–Code
Can be implemented on both FPGA and ASIC
Support for Different Floating–Point Units. (Not included in this work)

In figure 1 on next page, a typical LEON2 system can be seen.

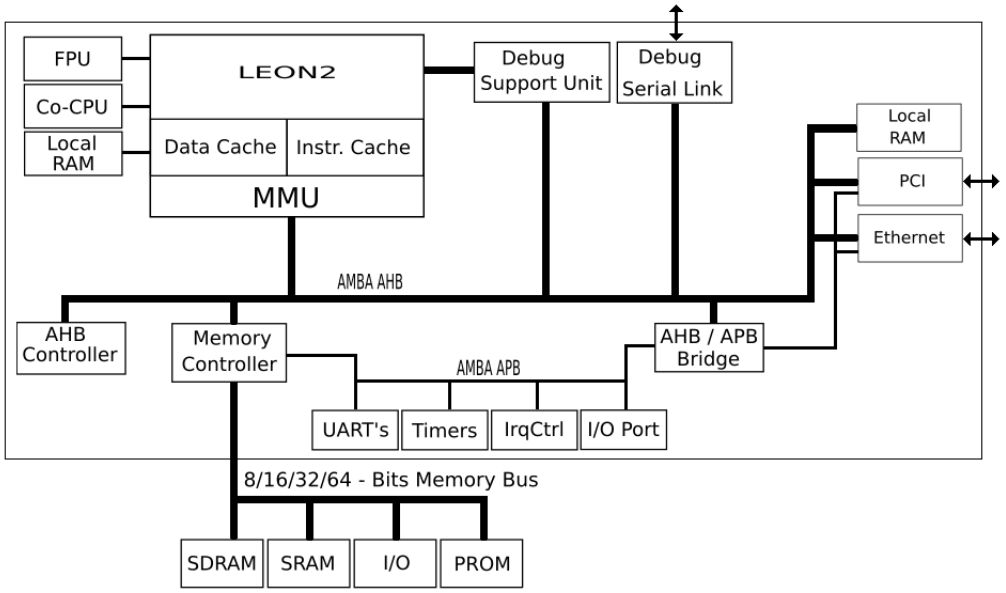

Figure 1: LEON2 System Overview

2.2 Instruction Set Architecture

LEON2 is a SPARC V8 [9] (IEEE 1754) single issue compliant RISC, with a simple five stage pipeline implementation. 32 Bit wide instruction set and few addressing modes:
”Register + Register” and ”Register + Immediate”. Multibyte numbers are stored as big endian.

2.3 Integer Unit

The LEON2 integer unit implements the full SPARC V8 standard, including all multiply and divide instructions. The implementation is focused on portability and low complexity.

The number of register windows is configurable within the limit of the SPARC standard (2–32), 8 is default. Total number of registers by default is 136. Separate instruction and data cache interfaces are provided (Harvard Architecture). The LEON2 is provided with a branch delay slot, more info concerning the delay slot feature can be seen in section 5.1

2.3.1 Pipeline Architecture

The LEON2 integer unit uses a single instruction issue pipeline with 5 stages. The stages can be seen below.

Instruction Fetch
Instruction Decode
Execute
Memory
Write Back

The LEON2 pipeline is stalled until the operation is completed if one of these conditions occurs:

Multi Cycle Instruction
Load or Store from the memory (SRAM or SDRAM)

2.3.2 Multiply and Divide Options

The LEON2 has a variety of multipliers available. In table 1 below, the LEON2 multiplier options can be seen.

Table 1: LEON2 Multiply Options

Configuration	*Result latency cycles*
32 x 32	1
32 x 16	2
32 x 8	4
16 x 16	4
16 x 16 + PIPELINE REG	5
ITERATIVE	35
EMULATED IN SOFTWARE	≈ 40

The LEON2 supports optional signed and unsigned MAC instructions – 16 x 16 bit multiplier with 40 bit accumulator, it executes in one cycle but have two latency cycles. A program that is going to use the MAC instructions should be written in assembly language. A radix–2 hardware divider (non restoring) is also available, with the following characteristics; Input data: 64/32 (Bits) , producing a 32 bit result and takes 35 cycles to compute.

2.4 Cache System

Separate multi–set, instruction– and data caches are provided, each of them are configurable with 1–4 Sets, 1–64 Kbyte/set, 16–32 Bytes/Line. Sub-blocking is implemented with one valid bit per 32–bit word. There are a several replacement policies provided: LRU, LRR and Random. It is possible to mix the policies, e.g. LRU on the instruction cache and random on the data cache. The instruction set provides instructions to flush the caches if it is necessary.

2.4.1 Instruction Cache

The instruction cache uses streaming during line–refill to minimize refill latency.
Instruction cache tag layout: (1 Kb/set, 32 bytes/line)

ATAG [31:10]	LRR[9]	LOCK[8]	VALID[7:0]

Only the necessary bits will be implemented in the cache tag, depending on the configuration. The LRR field is used to store the replacement history, if the LRR replacement algorithm was chosen. LOCK indicates if a line is locked or not.

2.4.2 Data Cache

The data cache uses write–through policy and implements a double–word write–buffer.
It can also perform bus–snooping on the AHB bus. A local scratch pad ram can also be added to the data cache controller to allow 0–wait–states access without requiring data write back to external memory.
Data cache tag layout: (4 Kb/set, 32 bytes/line)

ATAG [31:12]	Not Used [11:10]	LRR[9]	LOCK[8]	VALID[7:0]

Only the necessary bits will be implemented in the cache tag, depending on the configuration. The LRR field is used to store the replacement history, if the LRR replacement scheme has been chosen. LOCK indicates if a line is locked or not.

Cacheable Memories: *PROM* and *RAM*
Non-cacheable : *I/O* and *Internal (AHB)*

Write buffer

Consists of three 32–bit registers to temporarily store data until it is sent to the destination, acts like a FIFO.

Cache line locking

If the lock-bit in the cache is set to '1', it prevents the cache line to be replaced by the replacement algorithm (LRR, LRU or Random).

CCR – Cache Control Register

The operation of the instruction and data caches is controlled through a common CCR. Each cache can be in three modes; disabled, enabled or frozen. The register is 32 bit wide.

Disabled:	No caching, all Load/Store requests are passed to the memory controller directly.
Enabled:	Both instruction and data is cached.
Frozen:	As enabled, but no new lines are allocated on read misses.

2.5 Internal Busses

2.5.1 AMBA

The processor has a full implementation of AMBA 2.0 [10], AHB and APB on–chip buses.
The APB bus is used to access on–chip registers on the peripheral functions, while the AHB bus is used for high–speed data transmission.

A flexible configuration scheme makes it simple to add new IP cores. A more detailed description of the internal buses can be seen in section 4.

2.5.2 AHB Bus

LEON2 uses the AMBA–2.0 AHB bus to connect the processor cache controllers to the memory controller and other high-speed units. Default configuration: the processor is the only master on the bus, while there are two slaves; the memory controller and the APB–bridge.

2.5.3 APB Bus

The APB bridge is connected to the AHB bus as a slave and acts as the (only) master on the APB bus. The most on–chip peripherals are accessed through the APB bus, eg UART, I/O, Timer, IrqCtrl.

A detailed bus overview of how the peripherals are connected can be seen in figure 1 on page 2.

2.6 Memory Interfaces

The memory interface provides a direct interface to PROM, memory mapped I/O devices, static RAM (SRAM) and synchronous dynamic RAM (SDRAM). The different controllers can be programmed to either 8, 16, 32, 64 bits data width. Chip–select decoding is done for two PROM banks, one I/O bank, five SRAM banks and two SDRAM banks. The external memory bus is controlled by a programmable memory controller, which acts like a slave on the AHB bus. The function of the controller is programmed through three memory configuration registers through the APB bus.

The controller decodes a 2 Gbyte address space according to the table 2 below.

Table 2: LEON2 Supported Memories and Sizes

Type	*Size*
PROM	512 MB
I/O	512 MB
S(D)RAM	1024 MB

Burst Cycles

To improve memory bus bandwidth, access to sequential addresses can be performed in burst mode. Burst transfers will be generated when the memory controller is accessed using an AHB burst request. These requests includes instruction cache–line fills, double loads and double stores.

2.6.1 SRAM

The memory controller can handle up to 1 GByte SRAM, divided on up to five RAM banks. The bank sizes could be programmed in binary steps from 8 KByte to 256 MByte, while the fifth bank handles the upper 512 MBytes. A read access to the SRAM consists of two data cycles and zero to three wait–states. A write access is similar to the read but takes at least three cycles.

2.6.2 PROM

The PROM banks can be configured to operate in 8– , 16– or 32–bit mode. Because of a read access to the PROM is always done in 32–bit mode, a read access to the 8– or 16–bit mode is done by bursting, in four and two cycles, respectively. A write access will only write the necessary bits.

2.6.3 I/O Devices

The I/O device section can be configured to operate in 8– or 16–bit mode. A I/O device can only be accessed in a single access in 32–bit mode.

2.6.4 SDRAM

SDRAM access is supported to two banks of PC100/133 compatible devices. The controller supports 64–512 MByte devices. The SDRAM controller contains a refresh function that periodically issues an AUTO–REFRESH command to both SDRAM banks, the refresh period could be programmed in the memory controller register. The SDRAM can also be write protected.

2.7 System Interfaces

2.7.1 UART

Two identical UARTs are provided for serial communications. The UART support data frames with 8 data bits, one start bit, one optional parity bit and one stop bit . Hardware flow–control is supported through the RTSN/CTSN hand–shake signals. The two UARTs are possible to run in loop–back mode to ensure a working connection.

2.7.2 Ethernet MAC

A 10/100 Mbps Ethernet MAC is available, it is based on the core from OpenCores [11] , with two AHB interfaces, one master and one slave. The AHB master interface is used by the MAC DMA engine to transfer Ethernet packets to and from memory. The slave handles all configuration . Interrupt generated by the Ethernet MAC is routed to the interrupt controller.

2.7.3 PCI

Primary used for debugging purposes, it supports DSU communications over the PCI bus, if the development board used has a PCI connector. The interface consists of one PCI memory BAR occupying 2 Mbyte of the PCI address space, and an AHB address register.

2.8 Additional Units and Features

The following units and features are provided:

2.8.1 Debug Support Unit

The Debug Support Unit (DSU) allows non–intrusive debugging on target hardware. The DSU allows to insert breakpoints and watchpoints and access to all on–chip registers from a remote debugger. The DSU has no performance impact on the system. Communication to outside debuggers is done by using a Dedicated Communication Link (DCL), e.g UART (RS232) or through any AHB master e.g. Ethernet. The registers of a FPU or Co–processor can also be accessed through the DSU.

2.8.2 Trace Buffer

A trace buffer is provided to trace the executed instruction flow and/or AHB traffic. A 30 bit counter is also provided and stored in the trace as time tag. Its operation is controlled through the DSU control register and the trace buffer control register.

The default size is 128 lines – (2kbyte), could be configured to 8–4096 lines.

2.8.3 Timers

The timer unit implements two 24–bit timers, one 24 bit watchdog and one 10–bit shared prescaler. The prescaler is clocked by the system clock and decremented on each clock cycle. When it underflows, the prescaler is reloaded from the prescaler register and restarted.

2.8.4 Watchdog

A 24–bit watchdog is provided on–chip, it is clocked by the timer prescaler. When the watchdog reaches zero, an output signal is asserted. The signal could be used to generate system reset.

2.8.5 Interrupt Controller

The interrupt controller manages a total number of fifteen (15) interrupts, originating from internal and external sources. Each interrupt can be programmed to one of two priority levels. A chained secondary controller for up to thirty–two (32) additional interrupts is also available. There are a several unused interrupts that can be utilized by other IP–cores and peripherals.

2.8.6 Parallel I/O Port

A partially bit-wise programmable 32–bit I/O port is provided on-chip. It is splited into two parts – the upper 16 bits can only be used when all areas (ROM,RAM and I/O) of the memory controller is in 8- or 16–bit mode. If the SDRAM controller is enabled, the upper 16-bits cannot be used.

2.8.7 Power-down

The processor can be powered–down by writing an arbitrary value to the power-down register. Then the processor will enter the power-down mode on the next load or store instruction. During power-down mode the Integer Unit (IU) will effectively be halted. All instructions that are inside the pipeline will be there until the mode will be terminated. If the mode will be terminated – the Integer Unit (IU) will be re-enabled when an unmasked interrupt with higher level than the current processor interrupt level (PIL) become pending. All other functions and peripherals operate as normal during the power–down mode.

2.9 Co–Processors

2.9.1 FPU

The LEON2 processor model provides an interface to the GRFPU available from Gaisler Research and Meiko FPU-core from Sun Microsystems.

2.9.2 GRFPU

The GRFPU operates on single– and double–precision operands, and implements all SPARC V8 FPU instructions. It is interfaced to the LEON2 pipeline using a LEON2 specific FPU controller (GRFPC). The control unit allows FPU instructions to be executed simultaneously with integer instructions. Only in case of a data or resource dependency the integer pipeline is stalled.

2.9.3 Generic Co–processor

LEON2 can be configured to provide a generic co–processor. The interface allows execution in parallel with the integer unit (IU). One co–processor instruction can be started each clock cycle if there is no data or resource dependency.

2.10 Memory Management Unit

With the optional Memory Management Unit (MMU) it implements a SPARC V8 reference MMU and allows usage of robust operating systems such as Linux. The MMU can have a separate (Instruction and Data) or a common Translation Look-aside Buffer (TLB). The TLB is configurable for 2–32 fully associative entries. When the MMU is disabled the caches operate as normal. When enabled, the cache tags store the virtual address and also include an 8–bit context field.

2.10.1 Translation Look-aside Buffer

The MMU can be configured to use a shared TLB, the number of TLB entries can be set to 2–32. The organization of the TLB and number of entries is not visible to the software and operating system modification are therefore not required.

3 NIOS II

There are three versions of the NIOS II [12] processor core available, one with a single pipeline stage and no cache (NIOS II/e), one with five pipeline stages and instruction cache (NIOS II/s) and the last one with six pipeline stages and both instruction and data caches (NIOS II/f). In this thesis the focus is on the NIOS II/f core, since it is the most extensive one of the available NIOS II cores.

The processor architecture, cache structure, Instruction Set Architecture, peripherals and configuration options are described below.

3.1 System Overview

The NIOS II processor is a general–purpose single issue RISC processor core providing:

Full 32 bit instruction set, data path and address space
32 General Purpose Registers (Flat register file)
32 External Interrupt Sources
Barrel Shifter
Avalon System Bus
Instruction and Data Cache Memories (Harvard Architecture)
Access to On–chip Peripherals, and Interfaces to Off–chip Peripherals and Memories
The core is provided as a encrypted VHDL file

A typical NIOS II system can be seen in figure 2 below.

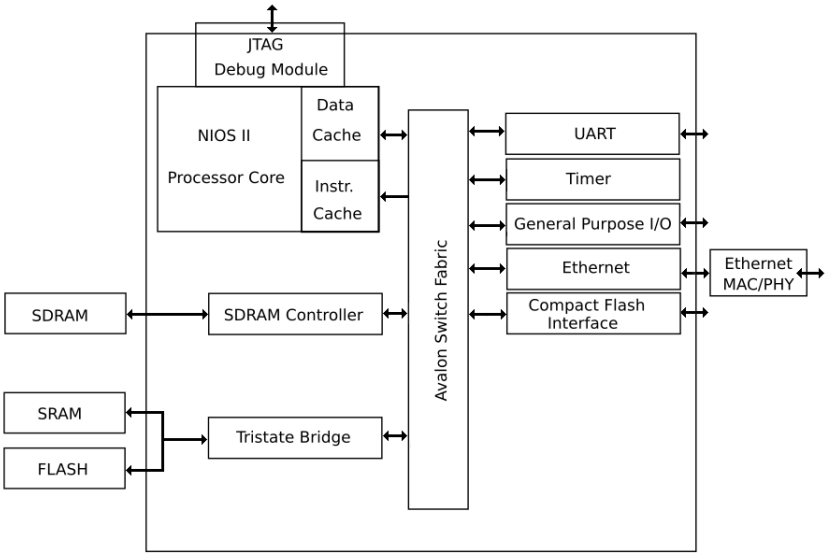

Figure 2: NIOS II System Overview

3.2 Instruction Set Architecture

The Instruction Set Architecture (ISA) is compatible across all NIOS II processor systems. The supported addressing modes are ”register + register” or ”register + immediate”. There is also a possibility to add custom instructions. Multibyte numbers are stored as little endian.

When the processor issues a valid instruction that is not implemented in hardware, an unimplemented instruction exception is generated. The exception handler determines which instruction generated the exception. If the instruction is not implemented in hardware, control is passed to an exception routine that emulates the operation in software, concerning multiply and divide instructions.

3.3 Integer Unit

The integer unit (IU) architecture supports a flat register file, consisting of thirty–two 32-bit general purpose registers. Three control registers are also provided. The architecture is prepared for the future addition of floating–point registers.

All instructions take one or more cycles to execute. Some instructions have other penalties associated with their execution. Late result instructions have a two cycle bubble placed between them and the instruction that uses the result. Instructions that uses Avalon transfers are stalled until it is completed.

3.3.1 Pipeline Architecture

The NIOS II/f core employs a 6–stage pipeline, with following stages:

Instruction Fetch
Instruction Decode
Execute
Memory
Align
Write–Back

The pipeline is stalled when one of these conditions occurs:

Multi-cycle instructions
Avalon instruction master–port read access
Avalon data master–port read/write access
Data dependencies on long latency instructions

When a stall has occurred, no new instructions enter any stage. Only The Decode– and Align stages creates stalls. Up to thirteen (depends on the multiplier latency) instructions can be executed while waiting for the result from a multicycle instruction, if there is no data dependency between the result of the multicycle instruction and the other instructions.

3.3.2 Multiply and Divide Options

The processor supports a variety of multiplication and divide options, mostly depending on the FPGA, according to the table 3 below. No embedded multiplier or divider is provided on the development board used in this thesis work.

Table 3: NIOS II Multiply and Divide Options

ALU option	*Details*	*CPI*	*Result Latency cycles*
NO HW MUL/DIV	EMULATED	≈ 40	N/A
EMBEDDED (STRATIX I & II)	32 x 32	1	2
EMBEDDED (CYCLONE II)	32 x 16	5	2
LE BASED	32 x 4	11	2
HARDWARE DIVIDE	32 / 32	4–66	2

The hardware divide has no exception when a division by zero occurs not on overflow either.

3.3.3 Branch Prediction

The core is provided with a branch predictor to achieve better performance while avoiding stalls during execution. The effectiveness of a branch predictor scheme depends not only on the accuracy, but also on the cost of a branch, if the prediction was wrong. In section 5.1 a comparison of their two different branch handling methods can be seen.

Static prediction – In the NIOS II/s core
Static branch prediction is implemented using the branch offset direction:
A negative offset - predict taken.
A positive offset - predict not taken.

Dynamic prediction – In the NIOS II/f core
Dynamic branch prediction is implemented using a 2–bit branch history table.

Branch Cycles

In the table 4 below, the NIOS II branch cycles are shown.

Table 4: NIOS II Branch Prediction Cycles

Prediction	*Cycles*	*Penalty*
CORRECTLY PREDICTED: TAKEN	2	NO PENALTY
CORRECTLY PREDICTED: NOT TAKEN	1	NO PENALTY
MISPREDICTED	4	PIPELINE IS FLUSHED

3.4 Cache System

The NIOS II/f processor core supports both instruction and data caches. Both caches are always enabled at run–time. Data cache bypass methods are available via software. Cache management and coherency are handled by software, the instruction set provides instructions for cache management. The core supports the 31–bit cache bypass method for accessing I/O on the data master port.

3.4.1 Instruction Cache

The instruction cache has the following features:

Direct–mapped implementation
Critical word first
32 Bytes (Eight words) per cache line
Configurable size: 512 bytes to 64 Kbytes

The instruction byte address has the following fields and sizes for a 8Kbyte cache:

TAG [30:15]	LINE[14:5]	OFFSET [4:2]	00 [1:0]

The offset field is 3 bits wide (an 8 word line), the tag and line sizes depends on the cache size. The maximum instruction address size is 31 bits. The instruction cache is permanently enabled and can not be bypassed.

3.4.2 Data Cache

The data cache has the following features:

Direct–mapped implementation
Write–back
Write–allocate
4 Bytes (One word) per cache line
Configurable size: 512 bytes to 64 Kbytes

The data byte address has the following fields and sizes for a 1 Kbyte cache:

TAG[22:10]	LINE[9:2]	OFFSET[1:0]

The offset field is 2 bits wide, the tag and line sizes depends on the cache size.

In all current NIOS II cores, there is no hardware cache coherency mechanism. Therefore, if there are multiple masters accessing shared memory, software must explicitly maintain coherency across all masters.

3.5 Internal Busses

3.5.1 Avalon On–chip Bus

The Avalon [13] bus is a simple bus architecture designed to connect on–chip processor and peripherals together into a working NIOS II based system. The Avalon is an interface that specifies the port connections between master and slave components, it also specifies the timing by which these components communicate.

The Avalon bus supports advanced features, e.g. latency aware peripherals, streaming peripherals and multiple bus masters. The advanced features allow multiple units of data to be transferred between peripherals during a single bus transaction. Avalon masters and slaves interact with each other based on a technique called slave–side arbitration. Slave–side arbitration determines which master gains access to a slave, if at least two masters attempt to access the same slave at the same time. Both the instruction and data buses are implemented as Avalon master ports. The data master port connects to both memory and peripheral components, while the instruction master port only connects to memory components.

Every peripheral mentioned in the following sections uses the Avalon bus. In figure 2 on page 10, a bus overview can be seen.

3.6 Memory Interfaces

The processor core is capable to access up to 2 GBytes of external address space. Both data memory, peripherals and memory–mapped I/O are mapped into the address space of the data master port on the Avalon interface. Multibyte numbers are stored as little endian.

When sharing memory, the highest performance is achieved when the data master port has been assigned higher arbitration priority on any memory that is shared by both instruction and data master ports.

3.6.1 SDRAM

The SDRAM controller, provides an interface to off–chip SDRAM. The controller supports the standard SDRAM PC100 specification. The controller handles all SDRAM protocol requirements. The core can access SDRAM subsystem with the following data widths: 8, 16, 32, 64 bits, various memory sizes and multiple chip–selects. Up to 4 banks of memory is supported. Because the Avalon interface is latency–aware, pipelined read transfers are allowed.

3.6.2 DMA

The DMA controller performs bulk data transfers, reading data from a source address range and writing the data to a different address range. An Avalon master peripheral (such as the NIOS II), can provide memory transfer tasks to the DMA controller, independently of the processor. The controller is also capable of performing streaming Avalon transactions.

3.6.3 CFI

The common flash interface core (CFI controller) provides connection to external flash memory. The Avalon tristate bridge creates an off–chip memory bus that allows the flash chip to share address and data pins with other memory chips. Avalon master ports can perform read transfers directly from the CFI controller's Avalon port.

3.6.4 EPCS

The EPCS device controller core allows NIOS II systems to access an Altera EPCS serial configuration devices. The EPCS device is able to store non–volatile program data and FPGA configuration data. Boot loading is also provided.

3.7 System Interfaces

3.7.1 UART

The UART core provides a register mapped Avalon slave interface, which allows communication with master peripherals such as NIOS II. It provides configurable baud–rate, parity, start, stop and data–bits and optional RTS/CTS flow control signals.

3.7.2 JTAG UART

The JTAG UART core provides communication between a host PC and a Altera FPGA. Master peripherals communicate with the core by reading and writing control and data registers. The core provides bidirectional FIFOs to improve bandwidth over JTAG connection. The FIFO depth is configurable – could be either in memory or build with registers.

3.7.3 SPI

SPI is a industry–standard serial protocol commonly used in embedded systems to connect the processor to a variety of off–chip devices. The SPI core can implement either the master or the slave protocol. If it is configured as a master, the SPI core can control up to sixteen independent SPI slaves. The core also provides an interrupt output which can flag an interrupt whenever a transfer completes.

3.7.4 Parallel I/O Port

The parallel I/O provides a memory mapped interface between an Avalon slave port and general purpose I/O port. The I/O ports connect either to on–chip user logic, or to external devices. Each core can provide up to thirty–two I/O ports. A bidirectional mode is available with tristate control. The core can be configured to generate a interrupt request on certain inputs.

3.8 Additional Units

3.8.1 JTAG Debug Module

The NIOS II core supports a JTAG debug module to provide JTAG interface to software debugging tools. The core also supports an optional enhanced interface that allows real–time trace data to be routed out of the processor and stored in an external debug probe.

3.8.2 Exception Controller

The architecture provides a simple, non–vectored exception controller to handle all exception types. All exceptions cause the processor to transfer execution to a single exception address. The handler at this address determines the cause of the exception and finishes the appropriate exception routine.

3.8.3 Interrupt Controller

The architecture supports thirty–two (32) external hardware interrupts. The core has thirty–two (32) level–sensitive interrupt request (IRQ) inputs, providing a unique input for each interrupt source. The priority is determined by software. The software can enable and disable any interrupt source individually by masking the IENABLE control register.

4 Bus Comparison

This section contains a more detailed comparison of the internal buses used by each processor core, see table 5 below. The AMBA–AHB and AMBA–APB which LEON2 uses and the Avalon switch fabric which the NIOS II uses.

Table 5: Bus Comparison

Option	*AMBA–AHB*	*AMBA–APB*	*AVALON*
PROVIDER	ARM	ARM	ALTERA
BUS VERSION	REV 2.0	REV 2.0	1.2
DATA BUS WIDTH	8–1026 BITS[1]	8–32 BITS	8–32 BITS
ADDRESS BUS WIDTH	32 BITS	32 BITS	32 BITS
ARCHITECTURE	MULTIPLE MASTER MULTIPLE SLAVE	SINGLE MASTER MULTIPLE SLAVE	MULTIPLE MASTER MULTIPLE SLAVE
PROTOCOL	PIPELINED BURSTING NON TRI–STATE SPLIT TRANSACTIONS	UNPIPELINED NO BURSTING NON TRI–STATE	PIPELINED STREAMING (BURST) TRI-STATE LATENCY AWARE–TRANSFERS
BRIDGING	AHB →APB	No	AVALON →AHB AVALON → TRISTATE
TRANSFER SIZES	8–128 BITS	8–32 BITS	8–32 BITS
TRANSFER CYCLES	1 OR MORE	2	1 OR MORE
TIMING	SYNCHRONOUS	SYNCHRONOUS	SYNCHRONOUS ASYNCHRONOUS[2]
WAIT–STATES SUPPORT	YES	NO	YES, FIXED OR PERIPHERAL–CONTROLLED
OPERATING FREQUENCY	USER DEFINED	USER DEFINED	USER DEFINED

[1] Recommended max = 256 Bits
[2] Asynchronous IP blocks could be connected to the bus

5 Instruction Performance

In this section, the instruction cycle performance for each processor is evaluated. Since both processors are RISC, almost every instruction take one cycle to execute. Some instructions have penalties associated with their execution and takes several cycles to complete. In table 6 below a summary of the instruction performance can be seen.

Table 6: Instruction Cycle Performance

Instruction Type	*Cycles on LEON2*	*Cycles on NIOS II/f*	*Penalties*
MULTIPLY	1,2,4,5,35	1,5,11	NIOS II: LATE RESULT
DIVIDE	35	4-66	NIOS II: LATE RESULT
JUMP	2	3	
DOUBLE LOAD	2	-	
SINGLE STORE	2	-	
DOUBLE STORE	3	-	
ATOMIC LOAD/STORE	3	-	
RET, CALLR	1	3	
CALL	1	2	
LOAD	2	≥ 1	PIPELINE IS STALLED [3]
STORE	3	≥ 1	
READ CONTROL REGISTER	-	1	NIOS II: LATE RESULT

Regarding the NIOS II multiplier performance, it mostly depends on the hardware used, if the FPGA has dedicated multipliers on–chip or not. If a instruction has a late result penalty, it means that the result is available two cycles afterwards, if the result is needed in the next instruction. The penalty may depend on the lack of data forwarding in the part of the pipeline which is associated with the instructions that have the specified penalty. If the pipeline has to be flushed, it takes four cycles to complete.

Since the NIOS II does not have any dedicated double load or store instructions, dealing with data types larger than a word will take at least twice as long time as the single load or store takes.

[3] The pipe line is stalled until the load is completed.

5.1 Branch Delay Slot vs. Dynamic Branch Prediction

This section contains a short comparison of the different branch handling methods both processor cores uses, respectively. The aim of a pipelined processor architecture is to keep the pipeline full of instructions all the time. If not, the performance will decrease, by increasing the Cycle Per Instruction (CPI). When the pipeline depth increases, the cost of a conditional branch will also increase, if the branch is taken.

Branch Delay Slot

The LEON2 uses the branch delay slot feature. A branch delay slot is a single cycle delay that comes after a conditional branch instruction has begun execution.The compiler could insert a instruction in the delay slot, that does not depend on the branch instruction, if it is impossible, a ”no operation” instruction is inserted there. This feature improves performance by having the processor to execute other instructions while waiting for the branch target and condition to be calculated.

Dynamic Branch Prediction

The NIOS II uses a dynamic branch prediction scheme, which is based on a 2 bit branch history table. By using a dynamic predictor, it is possible to look at the outcome of earlier branches to determine whether or not to take coming ones. The efficiency of a dynamic branch predictor depends not only on its precision, but also on the cost of a branch, especially if the prediction was wrong and the pipeline has to be flushed, the longer pipeline the bigger penalty. In section 3.3.3 the NIOS II branch prediction cycles can be seen.

6 Quick Review

This section contains a quick summary of the configuration possibilities for both processors.

FEATURE	*LEON2*	*NIOS II/f*
Integer Unit		
ARCHITECTURE	32–BIT RISC	32–BIT RISC
ISA	SPARC V8	NIOS II–ISA
CUSTOM INSTRUCTIONS	Yes[4]	Yes
PIPELINE STAGES	5	6
ENDIANESS	BIG	LITTLE
REGISTER FILE	WINDOWED	FLAT
NR OF GLOBAL REGISTERS	8	32
REGISTERS / WINDOW	16	–
NR OF WINDOWS	2–32	–
TOTAL NR OF REGISTERS	40–520	32
BRACH HANDLING	BRANCH DELAY SLOT	BHT [5]
FPU SUPPORT	YES	N/A
MMU	YES	N/A
Multiply Options		
SIZE AND (LATENCY)	32 x 32 (1), 32 x 16 (2)	32 x 32 (1+2)
	32 x 8 (4), 16 x 16 (4)	32 x 16 (5+2)
	16 x 16 (5), ITERATIVE (35)	32 x 4 (11+2)
MAC	YES[6]	N/A[7]
Divide Options		
TYPE	RADIX 2	RADIX 2
SIZE AND (LATENCY)	64/32 (35)	32/32 (4-66)[8]

Continues on next page.

[4] Could be added as a Co–Processor instruction
[5] Branch History Table; Dynamic prediction
[6] 16 x 16 multiplier and a 40 bit accumulator
[7] Could be implemented as a custom instruction
[8] The latency depends on the hardware used

FEATURE	*LEON2*	*NIOS II/f*
Cache Options		
INSTRUCTION CACHE		
NUMBER OF SETS	1–4	1
SET SIZE	1–64 KBYTE	0.5-64 KBYTE
POSSIBLE CACHE SIZES	1–256 KBYTE	0.5–64 KBYTE
LINE SIZE	16–32 BYTES	32 BYTES
WRITE POLICY	STREAMING DURING LINE REFILL	CRITICAL WORD FIRST
REPLACEMENT POLICIES	LRU,LRR,RANDOM	N/A
DATA CACHE		
NUMBER OF SETS	1–4	1
SET SIZE	1 – 64 KBYTE	0.5 – 64 KBYTE
POSSIBLE CACHE SIZES	1 – 256 KBYTE	0.5 – 64 KBYTE
LINE SIZE	16 – 32 BYTES	4 BYTES
WRITE POLICIES	WRITE–THROUGH, WRITE BUFFER	WRITE–BACK, WRITE–ALLOCATE
REPLACEMENT POLICIES	LRU,LRR,RANDOM	N/A
Supported Memory Interfaces		
	SRAM, SDRAM PROM MEMORY MAPPED I/O	SRAM,SDRAM FLASH MEMORY MAPPED I/O
Supported System Interfaces		
	ETHERNET, JTAG RS232,PCI	ETHERNET, JTAG RS232,SPI,I^2C,PCI
Software Tool Chain		
COMPILER	GCC 3.2.3	GCC 3.4.1
LIBRARY	NEWLIB 1.12.0	NEWLIB 1.12.0
Supported OS'es		
	ECOS, μCLINUX SNAPGEAR LINUX RTEMS RTOS	μC/OS–II μCLINUX, KROS NORTi,NUCLEUS PLUS prKERNEL

7 Development Tools

This section contains a presentation of the hardware and software tools which have been used to implement each processor system on the same target FPGA.

7.1 Hardware

In this section the target hardware is presented.

Altera Cyclone Development Board

The development board which both processor systems have been executed on, is based on a Altera Cyclone FPGA [14], since the NIOS II cannot be used on other FPGA's than Alteras own. The board consists of the following features:

FPGA:

- Cyclone EP1C20F400C7
- 20060 LEs[9]
- On-chip RAM: 294912[10] Bits
- Two PLL

Memories:

- 1 Mbyte SRAM
- 16 Mbytes SDRAM
- 8 Mbytes Flash
- Compact Flash Interface

Interfaces:

- 10/100 Mbps Ethernet PHY/MAC
- 2 x Serial Ports (RS232)
- Several Expansion Prototype Connectors
- JTAG

Miscellaneous:

- 50 MHz Oscillator
- Push–buttons
- LEDs
- 7–Segment LEDs

[9] A LE is equal to a Xilinx LUT
[10] 64 Blocks, Block Size: 128 x 36 Bits

7.2 Software

In this section the different software tools are evaluated. The different program versions can be seen in Appendix A.

7.2.1 LEON2

Very extensive configuration tool, all necessary details are available through it. E.g. multiplier sizes and latencies, number of cache sets, set sizes, replacement policies and different memory controllers among others. In order to run programs on the target hardware, the BCC [15], a GNU based cross–compiler system has been used. It is based on the GNU GCC 3.2.3 compiler, and uses newLib [16] 1.12.0 as C library.

7.2.2 NIOS II

The configuration of a NIOS II based system is done through SOPC, which is a integrated part of Quartus II™. SOPC – Good, but it would have been much better if the sizes and latencies of the arithmetic options were available explicitly. Now it is like a ”black box” , you know that you get a hardware based multiplier or divider, but you do not know its input and output sizes, features and latencies. Also the NIOS II uses a GNU based tool chain with a Eclipse [17] based GUI. The compiler version is GNU GCC 3.4.1. The newLib [16] 1.12.0 is used as the C library.

Compiler Comments

Due to the different compiler versions [18] each processor system uses, the NIOS II may take advantage of the higher optimization level introduced, in the newer one.

7.3 Implementation

A few things have been done, based on the changes done by De Nayer Instituut [19] on the LEON2, to make it run on the development board. Technology specific ram and the PLL were instantiated, and a new port map was created. The compiling and mapping part of the LEON2 synthesis was done in Synplify Pro 8.0 and the place and route part was done in Quartus II™.

Concerning the NIOS II it was straight forward, all you had to do was to configure the system and then do the synthesis and ”place and route” in Quartus II™. The resulting netlist was downloaded to the target hardware through Quartus™. This was also done for the LEON2.

On LEON2, the benchmark programs were downloaded to the target hardware through GRMON [20], which connects to the DSU and allows debugging of the system. On the NIOS II, the software downloading was done through the provided IDE.

8 Benchmarking

Benchmarking should be an objective, reproducible measure of performance, for example execution–speed comparisons. It must be meaningful and test something relevant to the user. Benchmarks could also be used to monitor performance changes during development. The benchmarks in this thesis work, only consists of integer and emulated floating–point performance.

Two important questions should be asked of any benchmarking activity:
How accurately does the benchmark predict real–world performance?
How reliably can a comparison between competing processors be made?

8.1 Benchmarking considerations

When a set of benchmarks are to be executed on several microprocessor architectures, one must keep a few things in mind, regarding;

1. Which Programming Language
2. Which Benchmark (Program) Version is Used
3. Which Tool Chain (Compiler, Library)
4. Which Optimization Level is Used
5. Which Hardware is Used and How the Processor Core is Configured
6. Which Processor Frequency

Regarding the benchmarks in this report, one must keep in mind that the NIOS II processor is optimized with respect to both FPGA and development board used. There might be some features the LEON2 could not utilize good enough on the FPGA or on the development board used.

In the following tests, all programs have been compiled with the GCC –O2 flag and the –msoft–float flag. All maximum performance executables were compiled with their hardware multiplication and divide specific flags, respectively. If some of these benchmarks are going to be executed on the same target hardware, it is plausible that the results may differ by ± 1 %, since the processor behavior is not deterministic.
All benchmark sets have been executed on both processor cores at two frequencies; 25 MHZ and 50 MHz. Two different frequencies was chosen to see how the execution times are affected when the frequency is doubled. If the frequency is doubled the execution times are not always halved, depending on the new timing criteria.

When a comparison of two or more devices are to be done, one must be sure that the comparison is relevant, you must be very careful of what you are going to compare. It is important to understand how different features affect each other and how the performance is affected, both in a positive and a negative manner.

In this case, regarding the minimum area configurations, each processor core have been configured to be as small as possible, with respect to the number of LE's and the total number of cache bits used. Multiplication and division is emulated in software.

Concerning the maximum performance configurations, the idea was to use as much as possible of all available resources. The multiplier and divider was chosen to give as good timing as possible and the number of cache bits which can be used is set to the maximum available on the FPGA.

It is important to keep in mind that benchmark performance will vary depending on the processor configuration, implementation tools, targeted FPGA architecture , device speed grade, the software compiler and library used.

8.1.1 Floating–point Emulation

Since both processor cores are intended to be used in embedded applications no floating–point unit (FPU) is included by default. To be able to execute programs that contain floating–point arithmetic in the high–level source code, the floating–point part has to be emulated. The compiler has to be informed about it during compilation, by using the "–msoft–float[11]" flag. The compiler then inserts a specified sequence of integer instructions, which behaves like it was done by a FPU. In table 7 below, a list of approximately corresponding number of integer instructions can be seen. The numbers have been taken from the NIOS II instruction set simulator, when a hardware based multiplier and divider were available. The numbers of integer instructions on LEON2 may differ due to the difference in their instruction set.

Table 7: Floating–point operations and their corresponding number of integer instructions when emulated in software.

FLOATING POINT OPERATION	NR OF INTEGER INSTRUCTIONS	NR OF CYCLES
ADDITION	350	600
SUBTRACTION	350	600
MULTIPLICATION	550	1300
DIVIDE	1550	2000

The numbers in table 7 above shows that floating–point emulation takes roughly 50–200 times longer compared to regular integer arithmetics. If no hardware multiplier or divider is available the number of integer instructions will increase, since the multiplication and division instructions themselves have to be emulated.

[11] A GNU GCC specific compilation flag

8.2 The Different Benchmarks Used

In this section, the four different benchmarks used is presented.

8.2.1 Dhrystone

Dhrystone is a benchmark invented in 1984 by Reinhold P. Weicker. The benchmark was first published in ADA, today the C version of the benchmark is mainly used. The current version of Dhrystone, version 2.1 was created in 1988 has been used to measure the integer performance on both processors. The original purpose was to create a short benchmark program, representative of integer programming. Its code is dominated by simple integer arithmetic, string operations, logic decisions and memory accesses, intended to behave like a typical computing application. Most of the execution time is spent in library functions.

The Dhrystone result is determined by measuring the average time a processor takes to perform many iterations of a single loop, containing a fixed sequence of instructions.
The output from the benchmark is the number of Dhrystones per second and the number of iterations of the main loop per second.

8.2.2 Stanford

The Stanford suite is gathered by John Hennessy and modified by Peter Nye. The version of the suite used is 4.2

The suite consists of three major program categories:

Recursion
Loop–intensive
Sorting algorithms

All four loop–intensive programs include multiplication, two of these includes floating–point arithmetics.

All programs perform a check to make sure each program will get the right output, the time spent doing the check is included in the execution time.

The following ten programs are included:

Perm – Calculates permutations recursively
Towers – Solve the Towers of Hanoi problem
Queens – Solve the Eight Queens Problem fifty times
IntMm – Multiply two random integer matrices
Mm – Multiply two random real matrices
Puzzle – A Compute–bound program
Quick – Sort a random array using the Quicksort algorithm
Bubble – Sort a random array using the Bubblesort algorithm
Tree – Sort a random array using the Treesort algorithm
FFT – Calculate a Fast Fourier Transform

After the execution has finished, a kind of mean value is computed, one where all (eight) integer program execution times are included (Non–floating composite) and a second where all ten execution times are included (Floating composite)

8.2.3 Paranoia

Paranoia is the name of a program written by William Kahan in the early 80's. The program used in this benchmark is version 1.4 and converted to C by David M. Gay and Thos Sumner.

Paranoia is designed to characterize floating–point behavior of computer systems.

Here is a part of the tests that Paranoia does:

– Small integer operations
– Search for radix and precision
– Check normalization and guard bits in +, $-$,$\times$ and /
– Check if rounding is done correctly
– Check for sticky bit
– Tests if $\sqrt{X^2} = X$ for a number of integers
    If it will pass monotonicity
    If it is correctly rounded or chopped
– Testing powers Z^i, for small Integers Z and i
– Search for underflow threshold and smallest positive number
– Testing powers Z^Q at four nearly extreme values
– Searching for overflow threshold and saturation
– It also tries to compute 1/0 and 0/0

When all tests have been done, Paranoia prints out a detailed result summary, which tells if the processor fulfil the IEEE754 standard or if there were any failures in the implementation.

8.2.4 Control Application

Since Paranoia does not contain any time measuring, a floating–point program that measures the execution time has been executed on both processors. the program is a kind of control application that does a lot of floating–point calculations. This program reveals the performance of the soft–float part on each processor core, both hardware and the software.

9 Minimum Area

This section and section 10 contains the third part of the thesis work. This section contains the ”Minimum Area” configurations and the results of the benchmarks mentioned in section 8.2. Each processor configuration can be seen in section 9.1 below.

9.1 Processor Configurations

Each processor configuration can be seen in the table 8 below. Additional info concerning the processors, take a look in section 6.

Table 8: Minimum Area Processor Configurations

PROCESSOR CORE / OPTION	LEON2	NIOS II	UNIT
CACHE			
INSTRUCTION CACHE			
SIZE	1024	512	BYTES
ASSOCIATIVITY	1	1	NR OF SETS
CACHE LINES	32	16	LINES
BYTES / LINE	32	32	BYTES
SUB-BLOCK SIZE	1	-	BIT/ 4 BYTE WORD
TOTAL LINE SIZE	291	287	BITS
DATA CACHE			
SIZE	1024	512	BYTES
ASSOCIATIVITY	1	1	NR OF SETS
CACHE LINES	32	128	LINES
BYTES / LINE	32	4	BYTES
SUB-BLOCK SIZE	1	-	BIT/ 4 BYTE WORD
TOTAL LINE SIZE	291	55	BITS
Memory Controller			
SRAM	1	1	MBYTE
ALU			
MULTIPLIER	SOFTWARE[12]	SOFTWARE[12]	-
DIVIDER	SOFTWARE[12]	SOFTWARE[12]	-

[12] Emulated in software

Configuration Comments

All configurable options were chosen to consume as few gates and cache bits as possible. The multiplication and division are emulated in software, it decreases the number of gates used, but the performance is affected in a negative manner. The caches on both processors are direct mapped, since it has a simple implementation and therefor less gates are being used. The memory system consists of 1 MB SRAM, which is enough to the benchmarks that is going to be executed on them later on. These configurations have been at both frequencies, 25 MHz and 50 MHz respectively.

9.2 Synthesis Results

In the table 9 below, the synthesis results can be seen. The ”Total Mem bits”–section contains both cache bits and the register file bits that each processor core utilizes. The number of LE’s used is without the debug unit each processor core uses, respectively. The number inside the parenthesis is the percentage of the maximum available option.

Table 9: Minimum Area Synthesis Results

PROCESSOR CORE	LEON2	NIOS II	LEON2	NIOS II
FREQUENCY	25 MHz	25 MHz	50 MHz	50 MHz
LE’S	5189 (25 %)	2167 (10 %)	5259 (25 %)	2181 (10 %)
M4K BLOCKS[13]	10 (15 %)	10 (15 %)	10 (15 %)	10 (15 %)
TOTAL MEM BITS	34 688 (11 %)	13 872 (4 %)	34 688 (11 %)	13 888 (4 %)

Synthesis Result Comments

Concerning the LEON2 ”Total Mem Bits”, Synplify Pro™ and Quartus II™ reports different number of memory bits used. In the table 9 above the number used is that the Quartus II™ reports. The ”Mem bit” section contains the memory bits used by the caches and the register file. As shown in table 9, the LEON2 is almost two and a half times bigger than the NIOS II core. The NIOS II core is vendor optimized with respect to the FPGA which have been used. The difference in number of LE’s used, for each processor at the two different frequencies mostly depends on the the timing criteria, which could be harder to fulfil with the same number of LE’s used.

[13] On–Chip RAM, Total 64 Blocks, Block size: 128 x 36 Bits

9.3 Benchmarking

This section contains the benchmarking results and conclusions. As mentioned in section 8.1 all benchmark sets have been executed in two frequencies, 25 MHz and 50 MHZ, respectively. It is important to notice that the NIOS II caches are only 0.5Kbytes each in this ”Minimum Area” part of the complete benchmark set, therefore the numbers in this section should not be taken at face value.

9.3.1 Dhrystone

In table 10 below, the Dhrystone results are shown.

Table 10: Dhrystone Results – Minimum Area

PROCESSOR CORE	LEON2	NIOS II	LEON2	NIOS II
FREQUENCY	25 MHz	25 MHz	50 MHz	50 MHz
1 ITERATION (MS)	68.2	69.8	33.4	34.9
DHRYSTONES/SEC	14 652	14 301	29 925	28 653
DHRYSTONES/SEC/MHZ	586	572	599	573

Dhrystone Result Comments

The bigger caches on the LEON2 shows that the performance impact on the execution time is roughly 4% for such a big cache system compared to the NIOS II. Since the caches are small and despite the fixed sequence of instructions, there will be a lot of accesses to the main memory which will affect the execution time in a negative manner. The frequency doubling increased the performance on LEON2, but the NIOS II has almost the same performance at both frequencies.

9.3.2 Stanford

The Stanford benchmark set was executed on both processor cores at two frequencies. The results can be seen in table 11 below. In this benchmark set, the execution times should be as short as possible.

Table 11: Stanford Results – Minimum Area

PROCESSOR CORE	LEON2	NIOS II	LEON2	NIOS II	
FREQUENCY	25 MHz	25 MHz	50 MHz	50 MHz	
PROGRAM					UNIT
PERM	66	80	33	40	ms
TOWERS	116	150	66	82	ms
QUEENS	50	52	33	26	ms
INTMM	316	707	150	345	ms
MM	3633	5281	1816	2727	ms
PUZZLE	483	498	266	249	ms
QUICK	66	120	33	60	ms
BUBBLE	84	120	50	60	ms
TREE	500	198	250	98	ms
FFT	3417	5003	1734	2687	ms
COMPOSITES					
NONFLOATING	270	279	137	140	
FLOATING	2848	4043	1397	2129	
COMPOSITE SUM	3118	4322	1534	2269	

Since the Stanford benchmark set contains various types of applications, a graphical overview was made to make it easy to compare their execution times, it can be seen in figure 3 on the next page.

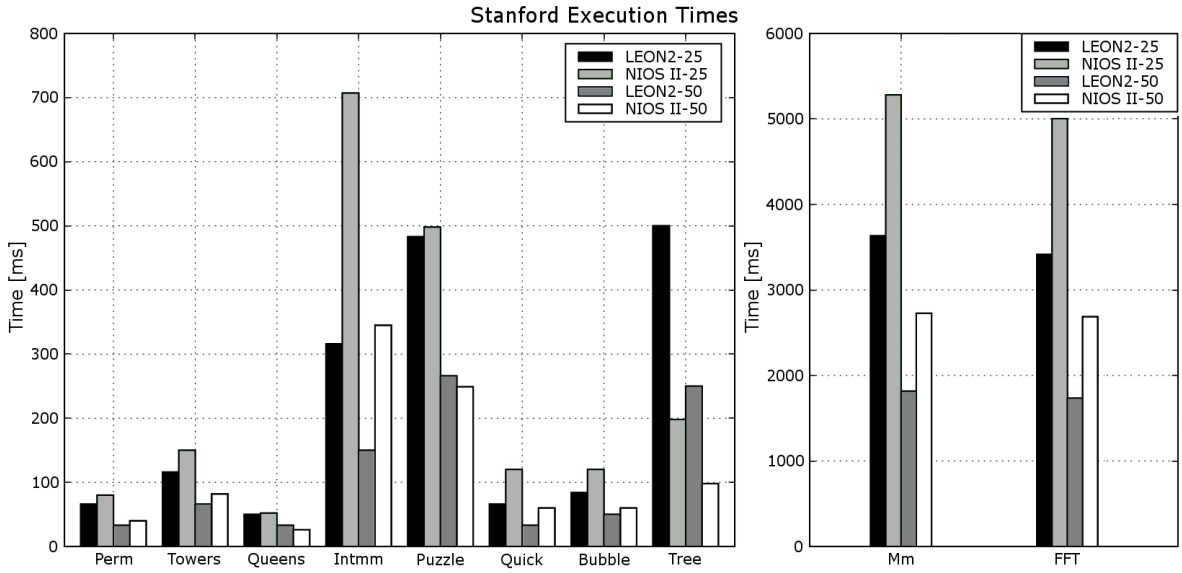

Figure 3: Minimum Area Stanford Execution Times

In figure 3 the integer program times are to the left and the emulated floating–point programs (Mm and FFT) times to the right. Notice the different time scales.

Stanford Result Comments

As mentioned before, the NIOS II caches are only 0.5 Kbytes each. Regarding the first three programs: Perm, Tower and Queens,the LEON2 is the fastest due to its windowed register file, which speeds up execution of programs containing a few function calls, compared with the flat register file that the NIOS II uses.

Concerning the Intmm and Puzzle results, when two matrices are to be multiplied or dealing with matrices and loop–intensive algorithms in general, will cause a lot of both instruction and data transactions. This will stress both caches, the memory system and the system bus quite a lot. If the processor is equipped with a branch predictor, its accuracy will also affect the execution times, especially when it predicts wrong.

There is a noticeable execution difference concerning the Tree-program, which includes recursion, iteration and selection. The recursive part causes register window overflow on LEON2, by spending much time in the trap routine, which has a negative impact on the performance. Every time the same function is called after the first overflow has occurred, the trap function will be executed. The bigger the tree is, the more time is spent in the trap routine. Deeply recursive algorithms is a disadvantage for a processor with a windowed register file compared with a processor that uses a flat register file.

The equal execution times of the sorting algorithms; Quicksort and Bubblesort on the NIOS II, probably depends on the small caches. Since the array contains 5000 random numbers, combined with a data cache of only 512 bytes. This combination will cause a high load on the memory system and the system bus as well, which will increase the execution times.

Concerning the floating–point programs Mm and FFT, where the floating–point arithmetics have to be emulated in software, the LEON2 execution times are roughly 30 % shorter. One obvious reason is of course the cache size difference. But to try to find out other possible reasons, the assembly code from the two compilers were compared and evaluated. The assembly code contained a lot of load, branch and multiply instructions and a emulated floating–point multiplication or divide will need some extra instructions since they both have to be emulated due to no hard multiplier nor divider is available in these configurations. All load instructions will stall the NIOS II pipeline, due to its load delay of two cycles. The small cache system causes a lot of replacement conflicts, then there will be a higher load on the system memory and on the system bus as well. In this program, the branch handling capabilities has a impact on the execution performance, especially on the NIOS II, if its predictor predicts wrong, the pipeline has to be flushed. Pipeline flushing could be time consuming, if it happens too often, since the execution has to restart from the instruction that comes after the branch instruction.

Finally, their non–floating composite values are quite equal, despite the cache size differences, but the difference concerning the floating–point composite is approximately 30 %, in this case, the cache sizes and the write buffers that LEON2 uses speeds up the execution and the load delay as mentioned above affects the execution times on the NIOS II.

9.3.3 Control Application

To find out how good each processor is when dealing with soft–float operations and as a complement to the floating–point programs in the Stanford benchmark set, the control application has been executed on both processors. This application reveals more about their floating–point performance. The results can be seen in table 12 below.

Table 12: Control Application Results – Minimum Area

PROCESSOR CORE	LEON2	NIOS II	LEON2	NIOS II	
FREQUENCY	25 MHz	25 MHz	50 MHz	50 MHz	
PROGRAM					UNIT
CONTROL APPLICATION	487	1250	251	620	SEC

Control Application Result Comments

Floating–point emulation in software , as mentioned in section 8.1.1 causes a lot of instructions to be executed by the integer part of the processor. In table 12 above, the LEON2 is almost 2.5 times faster than the NIOS II. This program includes more instructions than the floating–point programs included in the Stanford benchmark set do. The combination of many instructions and a relatively small cache system will cause a high load on each processor and on the cache and memory system as well. In this situation, the data handling capabilities of the processor cores are revealed. In this case the LEON2 is the better one.

9.4 Minimum Area Conclusions

Concerning the results in this ”Minimum Area” section, their performance are quite equal while comparing their integer performance. In the floating–point part of the benchmarks the performance on the LEON2 is the better one. The difference may depend on the bigger cache system and the write buffers that LEON2 uses.

A relatively small cache combined with multiplication and divide emulated in software while executing a program like the Control Application, will reveal the total system performance, then the processor has to work with a high load during a longer time.

10 Maximum Performance

This section contains the last part of the thesis work. This part contains the ”Maximum Performance” configurations and the results of the benchmarks mentioned in section 8.2.

10.1 Processor Configurations

Each processor configuration can be seen in the table 13 below. Additional info concerning the processors, take a look in section 6

Table 13: Maximum Performance Processor Configurations

PROCESSOR CORE OPTION	LEON2	NIOS II	UNIT
Cache			
INSTRUCTION CACHE			
ASSOCIATIVITY / SET SIZE	2 / 4096	1 / 8192	NR OF SETS / KBYTES
CACHE SIZE	8192	8192	BYTES
REPLACEMENT POLICY	LRU	N/A	
CACHE LINES	256	256	Lines
BYTES / LINE	32	32	Bytes
SUB-BLOCK SIZE	1	-	Bit/ 4 Byte Word
TOTAL LINE SIZE	294	278	BITS
DATA CACHE			
ASSOCIATIVITY / SET SIZE	2 / 4096	1 / 8192	NR OF SETS/ KBYTES
CACHE SIZE	8192	8192	BYTES
REPLACEMENT POLICY	LRU	N/A	
CACHE LINES	256	2048	LINES
BYTES / LINE	16	4	BYTES
SUB-BLOCK SIZE	1	-	BIT/ 4 BYTE WORD
TOTAL LINE SIZE	155	55	BITS
Memory Controller			
SRAM	1	1	MBYTE
ALU			
MULTIPLIER SIZE (LATENCY)	16 x 16 (5)	32 x 4 (11+2)	-
DIVIDER SIZE (LATENCY)	64/32 (35)	32 / 32 (N/A)	-

Configuration Comments

Both processor cores have be configured to achieve as high performance as possible. The cache sizes has increased, multiplication and divide is performed in hardware. The size of the LEON2 multiplier was set to 16 x 16, with a latency of 5 cycles, which gained the best timing. Regarding the data cache ”bytes / line” option, on the LEON2, it was chosen to 16, since it will improve the associativity, but it consumes more gates which is not a problem on this FPGA. Concerning the replacement policy, the LRU and the random algorithms were tested, they performed quite equal, but the LRU had the best performance in the Control Application part.

The NIOS II configuration options are limited to the FPGA used, since it does not have any dedicated multiplier on–chip and there was only one LE based multiplier available. The cache sizes are the only part which is configurable.

10.2 Synthesis Results

The synthesis results can be seen in table 14 below. The LE part of the table contains the processor core, timer, UART and the memory controller. The debug unit which each processor uses is not included in the numbers. The number inside the parenthesis is the percentage of the maximum available option.

Table 14: Maximum Performance Synthesis Results

PROCESSOR CORE	LEON2	NIOS II	LEON2	NIOS II
FREQUENCY	25 MHz	25 MHz	50 MHz	50 MHz
LE'S	7389 (36 %)	3057 (15 %)	7554 (37 %)	3058 (15 %)
M4K BLOCKS[14]	42 (65 %)	43 (67 %)	42 (65 %)	43 (67 %)
TOTAL MEM BITS	167 168 (56 %)	158 976 (53 %)	167 168 (56 %)	158 976 (53 %)

Result comments

Concerning the LEON2 "Total Mem Bits", Synplify™ and Quartus II™ reports different numbers of memory bits used. In the table above the number used is that the Quartus II™ reports. The "Total Mem Bits" includes both instruction and data cache bits and the register file. As shown in the table 14 above, the LEON2 uses more than two times more LE's than the NIOS II, where the LEON2's multiplier, divider, two cache controllers and the windowed register file consumes a lot of LE's compared to the vendor optimized NIOS II. The difference between the number of LE's used by each processor at the two different frequencies is because of the new timing criteria to fulfil, which may affect the place and route part of the synthesis chain.

[14] On–Chip RAM, Total 64 Blocks, Block size: 128 x 36 Bits

10.3 Benchmarking

This section contains the benchmark results and conclusions for the maximum performance part of the work. Both processor configurations have been executed in two frequencies, 25 MHz and 50 MHz, respectively.

10.3.1 Dhrystone

In table 15 below, the execution result of Dhrystone on both processor cores at both frequencies can be seen.

Table 15: Dhrystone Results – Maximum Performance

PROCESSOR CORE	LEON2	NIOS II	LEON2	NIOS II
FREQUENCY	25 MHz	25 MHz	50 MHz	50 MHz
1 ITERATION (MS)	26.8	23.6	13.1	11.8
DHRYSTONES/SEC	37 383	42 299	76 433	85 030
DHRYSTONES/SEC/MHZ	1495	1692	1529	1701

Dhrystone Result Comments

The results are shown in table 15 above. A processor system with a big cache and a program where the main part is a loop with a fixed sequence of instructions the cache hit rate will go towards 100 %. Increasing the cache size will not give a better result in this benchmark. Requiring no main memory access thus becoming more representative of the processor, rather than system performance. If the results are compared with the execution times in the minimum area section, one can see that the cache impact on integer programs are enormous, almost three times faster, see table 10. One interesting question is: ”How much does the compiler affect the execution times?” No assembly code study has been done in this section, since it is a very complex and time consuming task to evaluate the compiler efficiency.

10.3.2 Stanford

The Stanford benchmark set has been executed on both processors, at both frequencies. The results can be seen in table 16 below. The shorter execution times the better performance is achieved.

Table 16: Stanford Results – Maximum Performance

PROCESSOR CORE	LEON2	NIOS II	LEON2	NIOS II	
FREQUENCY	25 MHz	25 MHz	50 MHz	50 MHz	
PROGRAM					UNIT
PERM	66	79	33	39	ms
TOWERS	100	95	50	47	ms
QUEENS	50	49	33	25	ms
INTMM	50	67	17	33	ms
MM	1183	1338	583	667	ms
PUZZLE	400	384	184	192	ms
QUICK	50	61	34	30	ms
BUBBLE	67	78	33	39	ms
TREE	367	105	183	54	ms
FFT	1483	1543	733	772	ms
COMPOSITES					
NONFLOATING	184	125	93	62	
FLOATING	1188	1201	589	600	
COMPOSITE SUM	1372	1326	676	662	

In order to make the comparison of the Stanford benchmark set easier, a graphical overview has been made, it can be seen in figure 4 on the next page.

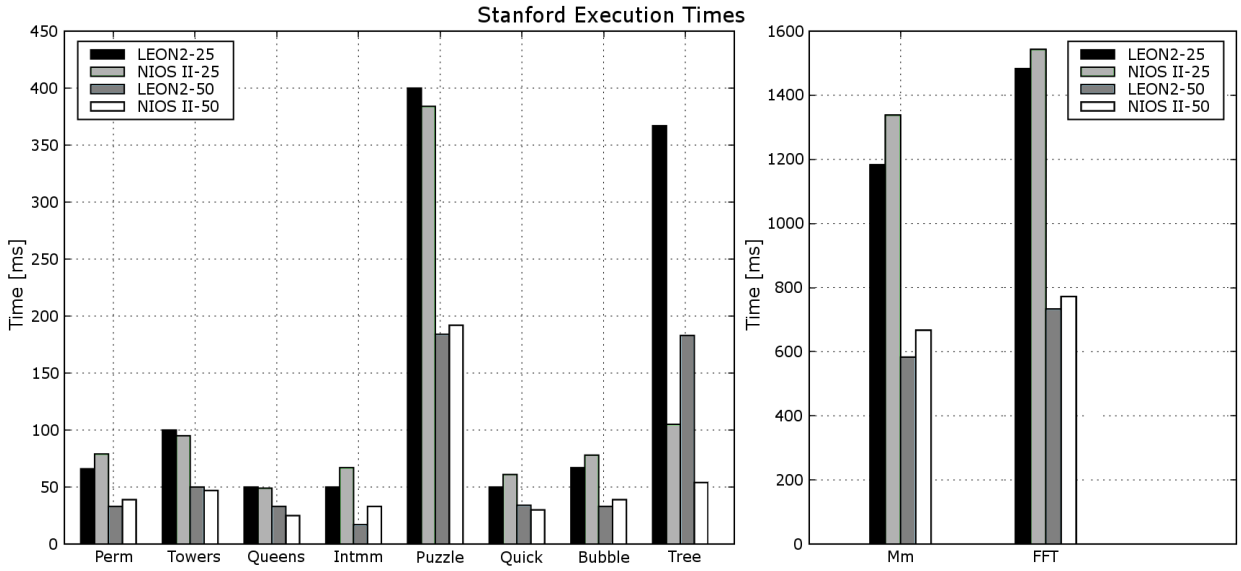

Figure 4: Maximum Performance Stanford Execution Times

In figure 4 the integer program times can be seen to the left and the emulated floating–point times to the right. Notice the different time scales.

Stanford Result Comments

When comparing the results of the first three programs, Perm, Towers and Queens, with the result of same three programs in the minimum area section 9.3.2, the difference is not as big as one may assume. It is because of the caches have not been filled up yet and they do not take advantage of the multiplier nor divider which is included in hardware in this configuration.

Regarding the next seven programs: Intmm, Mm, Puzzle, Quick, Bubble Tree and FFT, the difference is obvious, since they all contain a lot of data to be processed, where the bigger cache system is a advantage. Concerning the matrix multiplication programs (Intmm and Mm) and Puzzle, which are loop intensive, the speed up is caused by the hardware multiplier and the bigger caches, where the temporal and spatial locality are improved, then it is a bigger possibility that the desired data or instruction already is in the caches, respectively, and no fetching from the main memory is needed.

Their multiplier performance decreases the execution times almost three times. Especially the NIOS II multiplier performs really good, even if it has such a big latency compared to the LEON2 multiplier. (Thirteen cycles compared with five on the LEON2).

Regarding the sorting algorithms: Quick, Bubble and Tree, which also takes advantage of the big cache system the performance has increased. The equal execution times on the Quick – and Bubble sort algorithms on LEON2 at 50 MHz, probably depends on the data which the random function generates, otherwise the Quicksort algorithm would be the fastest one, as it is on the NIOS II and on the LEON2 in the ”Minimum Area” section.

The Tree sort algorithm is deeply recursive, which causes register window overflow on the LEON2, most of the execution time is spent in the trap routines, but the NIOS II handles the recursive part very good, indeed. The difference in the FFT program is only about 4 %, one reason could be the difference in the multiplier latency, since it is not as multiplication intensive as the Mm program, where the difference is about 10 %.

Overall, when comparing the composite sum, the NIOS II is roughly 30 % better. The cache system is big enough to contain all necessary instructions and data, since a majority of the programs are loop–intensive integer programs. The LEON2 on the other hand has the best floating–point performance, but not as big as in the minimum area section.

10.3.3 Control Application

As a complement to the floating–point programs in the Stanford benchmark set, the Control Application was executed to reveal emulated floating–point performance differences. In table 17 below the result of the execution of the Control Application can be seen.

Table 17: Control Application Results – Maximum Performance

PROCESSOR CORE	LEON2	NIOS II	LEON2	NIOS II	
FREQUENCY	25 MHz	25 MHz	50 MHz	50 MHz	
PROGRAM					UNIT
CONTROL APPLICATION	200	293	107	141	SEC

Control Application Result Comments

This time, the floating–point emulation performance has increased, by beneficiation of the bigger cache system and the hardware based multiplier and divider used. When a more computing intensive program is executed, it will reveal a more realistic work load, on the processor as well as on the memory system. As the numbers in table 17 shows, the LEON2 performs about 30% better than the NIOS II. The difference could depend on the multiplier latency,which is six more cycles on the NIOS II and the load delay, which is one cycle on LEON2 and two cycles on the NIOS II.

10.4 Maximum Performance Conclusions

As one could see in the result sections above, the execution times has decreased, compared with the ”Minimum Area” results. When a hard multiplier is available on–chip it improves the execution speed, compared to software emulation.

When ”small programs” like the Stanford benchmark set is executed, a bigger cache system in not always a advantage. If it is too big, it will introduce some overhead by checking empty places, while accessing the caches. If the cache is to small there will be replacement conflicts, which will decrease the execution performance, since the data and the instructions have to be fetched from the main memory.

In loop intensive applications, the performance will be improved, as seen in the results above, since the temporal and spatial locality in the bigger caches will be improved, then data and instructions does not have to be fetched from the main memory that often.

11 Paranoia

In this section the results from floating–point test program Paranoia can be seen.

11.1 Results – NIOS II

When Paranoia was executed on the NIOS II, the program reported one failure in the multiplication part of the test.

A part of the output from Paranoia on NIOS II:

```
Multiplication is neither chopped nor correctly rounded.
Sticky bit used incorrectly or not at all.

The number of FLAW[15]s discovered = 1.
The arithmetic diagnosed seems Satisfactory though flawed.
```

Possible Failure Sources

The failure is probably caused by the code generation in the soft–float part of the compiler. There where no error when the program was executed when it was compiled without optimizations. But the performance will drop by a certain amount, without optimizations which is not satisfactory. Especially when it will be used to do a lot of emulated floating–point calculations.

Result without optimizations

```
No failures, defects nor flaws have been discovered.
Rounding appears to conform to the proposed IEEE standard P754,
except for possibly Double Rounding during Gradual Underflow.
The arithmetic diagnosed appears to be Excellent!
```

11.2 Results – LEON2

When Paranoia was executed on the LEON2 neither failures nor flaw's were detected.

A part of the output from Paranoia on LEON2:

```
No failures, defects nor flaws have been discovered.
Rounding appears to conform to the proposed IEEE standard P754,
except for possibly Double Rounding during Gradual Underflow.
The arithmetic diagnosed appears to be Excellent!
```

[15] FLAW: lack(s) of guard digits or failure(s) to correctly round or chop

12 Summary

Analyzing synthesizable processor cores performance is not an obvious task, since there are several things depending on each other. The hole synthesis chain which is complex, begins with the processor VHDL source code and ends up with the netlist after the place and route part has finished. All steps included in the synthesis affects the overall system performance, but the major impact of the final performance is probably depending on the software compiler and the application that is going to be executed on the target.

Syntesizable processor cores are in general configurable in some way. Mainly, both the instruction and the data caches sizes and the multiplier size and latency could be customized, since they affect the overall performance most. Another feature they have is that hardware migration is possible, which make them reusable and flexible. A disadvantage is that some of the processor cores are software tool dependent and hardware dependent, as well. Which will force one to use certain software tools and FPGA's.

In the "Minimum Area" section, each processor core has been configured to utilize as few gates and cache bits as possible. To save gates, multiplication and divide was emulated in software. Regarding the results of the benchmarking, their integer performance are quite equal, despite the cache size differences. When dealing with emulated floating–point applications, LEON2 is faster, by taking advantage of its write buffers and the bigger cache system. In the NIOS II case where the load delay, which is two cycles affects the performance negatively, by stalling the pipeline. If a stall occurs many times, especially when dealing with floating–point emulation combined with a relatively small cache system, its performance will drop by a certain amount.

In the "Maximum Performance" section, where the aim was to configure both processor cores to achieve as high performance as possible. Multiplication and divide is performed in a hardware based multiplier and divider, respectively. Their sizes and latencies have been configured to gain as high overall performance as possible. The NIOS II has the best integer performance, especially on Dhrystone, which contains a fixed sequence of instructions, where the hole sequence more or less fits in the instruction and data caches, respectively.

The bigger cache system improves the performance, by improving the cache hit rate, which on such small applications, like Dhrystone and Stanford is almost 100 %, on both processors. Then the benchmark results will be representative of integer performance rather than the overall system performance. In the emulated floating–point application part, LEON2 once again is the fastest one. Its data handling capabilities is better than on the NIOS II

By respect to their sizes, it is noticeable that the NIOS II core is vendor optimized and the so- urce is encrypted, which is a limit to portability when it only could be used in Altera FPGA's. The LEON2, which has no vendor restriction, fits well in a low–end FPGA like the one used in this work, even if it is not optimized with respect to a certain technology.

To achieve best performance when dealing with embedded systems, the hardware and software have to be designed together. When having a FPGA based platform, the whole system can be re–configured, by respect to the FPGA capability, to change characteristics if its performance is not good enough.

With respect to their configurablity the LEON2 is the best, by providing multi set cache system with configurable sizes and ”bytes/line” and a variety of cache replacement policies and multipliers. On the NIOS II, only the cache sizes are configurable, the multiplier option depends on the target FPGA used. One NIOS II advantage is that it supports custom instructions, which could speed up applications, where a certain task is dominating.

13 Appendix

Appendix A

Program Versions

This section contains the different development tool versions used in this work.

Provider	*Program*	*Version*
SYNPLICITY INC.	SYNPLIFY PRO	VERSION 8.0, BUILD 189R, BUILD JAN 17, 2005
ALTERA	QUARTUS II	4.2 BUILD 157 12/07/2004 SJ FULL VERSION
GAISLER RESEARCH	GRMON	1.0.6 PROFESSIONAL EDITION

Appendix B

Stanford Weight Values

The Non–Floating point composite is calculated as the sum of the execution time for each program multiplied by each program's weight value, and divided by the number of integer programs (eight of ten). The floating point composite is calculated in the same way but the values of all ten programs are included.

Program	*Weight*
PERM	1.75
TOWERS	2.39
QUEENS	1.83
INTMM	1.46
MM	2.92
PUZZLE	0.50
QUICK	1.92
BUBBLE	1.61
TREE	2.50
FFT	4.44

14 References

[1] LEON2 Processor Overview
Url: *http://www.gaisler.com/products/leon2/leon.html*

[2] Gaisler Research, Första Långgatan 19 Gothenburg, Sweden
Url: *http://www.gaisler.com*

[3] NIOS II Processor Overview
Url: *http://altera.com/products/ip/processors/nios2/cores/ni2-processor_cores.html*

[4] Altera Corporation, 101 Innovation Drive,
San Jose, California 95134, USA
Url: *http://www.altera.com*

[5] The GNU LGPL License form
Url: *http://www.gnu.org/copyleft/lesser.html*

[6] The LEON2 Full Source Code
Url: *http://www.gaisler.com/products/leon2/leon_down.html*

[7] NIOS II Licensing Info
Url: *http://www.altera.com/products/ip/processors/nios2/features/ni2-q_and_a.html*

[8] The LEON2 Processor User's Manual XST Edition Version 1.0.27 January 2005

[9] The SPARC Architecture Manual Version 8, Revision SAV080SI9308
SPARC International Inc. 535 Middlefield Road, Suite 210
Menlo Park, CA 94025, 415–321–8692
Url: *http://www.sparc.org*

[10] AMBA AHB and APB Specification Rev 2.0, ARM IHI0011A, 1999
Url: *http://www.arm.com*

[11] The OpenCores Ethernet MAC
Url: *http://www.opencores.com/projects.cgi/web/ethmac/overview*

[12] The NIOS II Processor Reference Handbook NII5V1-1.2 September, 2004

[13] The Avalon Bus Specification, Reference Manual version 2.3, July 2003

[14] Additional Development Board Info
Url: *http://altera.com/products/devkits/altera/kit-nios_1c20.html*

[15] BCC, A GNU based Cross–Compiler System, Used by LEON2
Url: *http://www.gaisler.com/doc/libio/bcc.html*

[16] Newlib, a C Library Supported by Redhat
Url: *http://sources.redhat.com/newlib/*

[17] The Eclipse IDE GUI
Url: *http://www.eclipse.org*

[18] The GNU GCC Release History and Change Logs
Url: *http://gcc.gnu.org/releases.html*

[19] LEON2 Changes Done by De Nayer Instituut, Belgium
Url: *http://emsys.denayer.wenk.be/?project=empro&page=cases&id=14#ls*

[20] GRMON, A Combined Debug Monitor and Simulator for LEON Processors
Url: *http://www.gaisler.com/products/grmon/grmon.html*