Adaptation of OSE_{ck} for an FPGA-Based Soft Processor Platform

Examensarbete utfört i Elektroniksystem vid Tekniska högskolan i Linköping
av

Daniel Staf

LITH-ISY-EX--07/3962--SE

Linköping 2007
Adaptation of OSE\textsubscript{ck} for an FPGA-Based Soft Processor Platform

Examensarbete utfört i Elektroniksystem vid Tekniska högskolan i Linköping
av

Daniel Staf

LITH-ISY-EX--07/3962--SE

Handledare: Kent Palmkvist
isy, Linköpings universitet
Peter Fredriksson
Enea Services Linköping

Examinator: Kent Palmkvist
isy, Linköpings universitet

Linköping, 20 August, 2007
Anpassning av OSE\textsubscript{ck} för en FPGA-baserad mjuk processorplatform

Adaptation of OSE\textsubscript{ck} for an FPGA-Based Soft Processor Platform

Integrated systems become larger and more complicated every day while time to market is shortened. Due to this, there is a need for flexible hardware platforms that use programmable logic not only for custom hardware but also for realizing embedded processors.

This thesis aims to select a suitable, FPGA targeted, soft processor core and adapt the real-time operating system OSE\textsubscript{ck} to run on the selected target. A study of possibilities to integrate setup and configuration of OSE\textsubscript{ck} into the processor’s IDE is also performed.

Studies of OSE\textsubscript{ck} and the two processor candidates MicroBlaze and Nios II have been performed. The processor study showed that MicroBlaze and Nios II have a very similar architecture and both are suitable to host OSE\textsubscript{ck}. MicroBlaze was chosen as target processor mainly because of more available documentation regarding operating system integration.

Performance and footprint was measured with OSE\textsubscript{ck} on MicroBlaze. The performance figures indicate that MicroBlaze can not be expected to have the same processing power as hard processors but works well as a control processor. To achieve high application performance, custom hardware accelerators can be connected. Integration investigations and tests have been performed with the goal of making an interface that conforms to the normal MicroBlaze design flow.

OSE\textsubscript{ck} has been successfully adapted to run on MicroBlaze and integration in the development environment is possible although some steps have to be done manually. Alternative integration options are discussed.

Författare
Daniel Staf
Abstract

Integrated systems become larger and more complicated every day while time to market is shortened. Due to this, there is a need for flexible hardware platforms that use programmable logic not only for custom hardware but also for realizing embedded processors.

This thesis aims to select a suitable, FPGA targeted, soft processor core and adapt the real-time operating system OSE\textsubscript{ck} to run on the selected target. A study of possibilities to integrate setup and configuration of OSE\textsubscript{ck} into the processor’s IDE is also performed.

Studies of OSE\textsubscript{ck} and the two processor candidates MicroBlaze and Nios II have been performed. The processor study showed that MicroBlaze and Nios II have a very similar architecture and both are suitable to host OSE\textsubscript{ck}. MicroBlaze was chosen as target processor mainly because of more available documentation regarding operating system integration.

Performance and footprint was measured with OSE\textsubscript{ck} on MicroBlaze. The performance figures indicate that MicroBlaze can not be expected to have the same processing power as hard processors but works well as a control processor. To achieve high application performance, custom hardware accelerators can be connected. Integration investigations and tests have been performed with the goal of making an interface that conforms to the normal MicroBlaze design flow.

OSE\textsubscript{ck} has been successfully adapted to run on MicroBlaze and integration in the development environment is possible although some steps have to be done manually. Alternative integration options are discussed.
Acknowledgments

I would like to take this opportunity to thank my supervisor at Enea, Peter Fredriksson as well as my examiner at ISY, Kent Palmkvist for their support.

I would also like to thank all inspiring persons working at Enea Linköping for giving me a great master thesis experience. Especially Matthias Bergvall for sharing his experiences in software development and for always giving me good advice.

Finally I would like to thank my love Boel for always supporting me.

Daniel Staf
1 Introduction
  1.1 Background .............................................. 1
  1.2 Objectives .............................................. 2
  1.3 Limitations .............................................. 2
  1.4 Method .................................................. 2
  1.5 Thesis Disposition ........................................ 2

2 FPGAs
  2.1 Circuit Design Methods ..................................... 5
  2.2 Designing with FPGAs ...................................... 6
    2.2.1 Application Areas .................................... 6
    2.2.2 FPGA Types ........................................... 7
  2.3 FPGA Architecture ......................................... 7
    2.3.1 Logic Architecture .................................... 8
    2.3.2 On-Chip Resources .................................... 9
    2.3.3 Specialized On-Chip Hardware .......................... 11
    2.3.4 IP-Cores .............................................. 11

3 Soft Processor Cores
  3.1 Soft Processor Definition .................................. 13
  3.2 Advantages and Drawbacks .................................. 13
  3.3 Xilinx’s and Altera’s Soft Processor Cores ................ 14
  3.4 Basic Architecture ....................................... 15
    3.4.1 Processor Busses ..................................... 15
    3.4.2 Memory ............................................... 15
    3.4.3 Interrupts ............................................ 16
    3.4.4 Registers ............................................. 16
  3.5 Processor Customization ................................... 17
  3.6 Custom Hardware Acceleration .............................. 17

4 Operating System Fundamentals .............................. 19
  4.1 Purpose of Operating Systems ............................... 19
  4.2 Context Switch ........................................... 19
  4.3 Interrupt Processing ...................................... 20
5 Real-Time Operating Systems
   5.1 Real-Time Definition ........................................... 23
   5.2 FPGAs and Real-Time ........................................... 24
   5.3 Real-Time Operating System Properties ......................... 24

6 OSE<sub>ck</sub>
   6.1 The OSE Family .................................................. 27
   6.2 OSE<sub>ck</sub> Fundamentals ..................................... 28
      6.2.1 Basic System Calls ........................................ 28
      6.2.2 Memory Pools .............................................. 28
      6.2.3 Signals .................................................... 29
      6.2.4 Process Types ............................................. 29
      6.2.5 Process Priority .......................................... 29
   6.3 Kernel Configuration ............................................ 30
   6.4 Closely Related Products ...................................... 31
      6.4.1 Timeout Server ............................................ 31
      6.4.2 Heap Manager ............................................. 31
      6.4.3 LINX ...................................................... 32
   6.5 Test Systems .................................................... 33
      6.5.1 Functional Testing ........................................ 33
      6.5.2 Performance Measurement .................................. 33
   6.6 Hardware Requirements ........................................ 33
   6.7 Architecture Dependent Parts .................................. 33

7 Processor Selection .................................................. 35
   7.1 Architectural Properties ....................................... 35
      7.1.1 Predictability ............................................. 35
      7.1.2 Performance ............................................... 36
      7.1.3 Multiprocessor Support ................................... 37
      7.1.4 Interrupt Hardware ........................................ 38
      7.1.5 Available Timers .......................................... 38
   7.2 Development Tools ............................................... 39
      7.2.1 Usability .................................................. 39
      7.2.2 Multiprocessor Support ................................... 40
      7.2.3 Documentation ............................................. 40
   7.3 Conclusion ...................................................... 40

8 Development Environment .......................................... 43
   8.1 Software Tools .................................................. 43
      8.1.1 Xilinx Embedded Development Environment ................ 43
      8.1.2 Platform Description Files ................................ 44
   8.2 Target Boards ................................................... 45
   8.3 Soft MicroBlaze Platforms ..................................... 46
9 Implementation Considerations

9.1 Previous Platforms ........................................... 47
9.2 Kernel Requirements .......................................... 48
  9.2.1 Drivers ......................................................... 48
  9.2.2 Peripherals .................................................... 48
9.3 Driver Compatibility and Reuse ............................... 49
  9.3.1 Timer Drivers ................................................ 49
  9.3.2 Interrupt Driver .............................................. 50
9.4 Non Fixed Instruction Set .................................... 50
  9.4.1 Dynamic Code Selection ................................... 50
  9.4.2 Libraries for All Configurations ........................... 50
  9.4.3 Minimal Configuration ...................................... 51
  9.4.4 Selected Solution ........................................... 51
9.5 Interrupt Driver ................................................ 52
  9.5.1 Hardware configurations .................................... 52
  9.5.2 Priority Levels ................................................ 52
  9.5.3 Interface ....................................................... 53
  9.5.4 Instruction Set ................................................ 54
  9.5.5 User Interrupts .............................................. 54

10 Integration Considerations ................................. 55
10.1 Why Integrating? .............................................. 55
10.2 Functionality to Integrate ................................... 55
10.3 Automatic Library Choice .................................... 56
10.4 Kernel Configuration Integration ............................ 57
10.5 OSEck Design Rule Check .................................... 58
10.6 Driver Configuration .......................................... 58
  10.6.1 Non Fixed Address Space .................................. 58
  10.6.2 Peripheral Hardware Implementation ...................... 59
  10.6.3 Fixed Interval Timer Driver Issue .......................... 59
10.7 OSEck Project .................................................. 60

11 Results and Analysis ......................................... 61
11.1 Achievements .................................................. 61
11.2 Functional Analysis .......................................... 61
11.3 Performance Analysis ......................................... 62
  11.3.1 Method ....................................................... 62
  11.3.2 Measurements ............................................... 62
11.4 Footprint Analysis ............................................ 65

12 Conclusions and Further Studies .......................... 67
12.1 Integration ..................................................... 67
12.2 Performance and Resource Utilization ...................... 67
12.3 Possibilities with OSEck on a Soft Core .................... 68
12.4 The Future of Soft Processors and FPGAs ................... 68
12.5 Further Studies ............................................... 69
Chapter 1

Introduction

In this chapter the reader will be introduced to the thesis objectives and the methods that will be used to fulfill those. Background information is also given as well as limitations. This chapter ends with a thesis disposition that briefly describes the content of each chapter.

1.1 Background

Today, special integrated circuits can be programmable to realize custom logic functions. With programmable logic it is possible to build custom hardware without manufacturing a new chip. Large chips containing programmable logic are called Field-Programmable Gate Arrays (FPGAs) and can be used to implement hardware acceleration of software algorithms.

Commonly processors are implemented in discrete chips with one processor per chip. The FPGA is then connected to the processor to implement hardware acceleration. Today the FPGA allows processors to be implemented in programmable logic in the FPGA. The two largest general purpose FPGA manufacturers Xilinx and Altera offers one 32-bit processor candidate each, MicroBlaze and Nios II.

Embedded processors also need a good software foundation, an operating system. OSE\(_{ck}\) is a Real-Time Operating System (RTOS) capable of managing hard real-time tasks. Because the processors are implemented in logic, they can be customized in many ways not possible in a traditional processor. This poses some challenges to the flexibility of the operating system.

OSE\(_{ck}\) together with the closely related Inter Process Communication (IPC) LINX, makes it possible to set up a location transparent software platform. Software modules can be moved in between processors without affecting communication with other parts of the software. To be able to extend the platform to easily use hardware accelerated software it is important that OSE\(_{ck}\) also runs on processors implemented in logic in the FPGA.
1.2 Objectives

- The soft processors Nios II and MicroBlaze will be examined and compared. The comparison shall address properties that are more or less favorable in terms of hosting OSE\textit{ck}. One of the processors will then be chosen to which OSE\textit{ck} can be ported.

- The OSE\textit{ck} kernel will be ported.

- The performance of OSE\textit{ck} on the chosen processor will be measured and compared with other targets. Footprint figures will also be extracted.

- The possibilities to integrate setup and configuration of OSE\textit{ck} into the processor’s development environment will be evaluated and tested.

1.3 Limitations

Primarily, only the most central parts of OSE\textit{ck} are to be ported. An evaluation of development environment integration possibilities is the primary goal. Practical integration tests will be performed if the time scope allows.

1.4 Method

The work starts with both detailed studies of OSE\textit{ck} and studies of general real-time aspects. This is to get a good understanding of what characteristics are important to consider. Architecture dependent parts of existing OSE\textit{ck} targets are also examined. Architectural and functional studies of the two processor candidates are then performed before the processor comparison is made.

The next part is the implementation. The goal with this part is to design and migrate OSE\textit{ck} to the new platform. The correctness of the implementation will be tested by porting and running a functional test system available for OSE\textit{ck}.

After the implementation step an integration evaluation is performed. The possibility to make OSE\textit{ck} a part of the FPGA and soft kernel design flow is examined. Practical integration is then performed.

The last part includes porting of OSE\textit{ck}’s performance test program and running this using different hardware and memory configurations.

1.5 Thesis Disposition

This section contains a brief description of the chapters in the thesis. Chapters 2 to 6 can be skipped if the reader is familiar with the subject contained within.

- Chapter 1, Introduction, gives a background of the thesis and describes the objectives and methods used.

- Chapter 2, FPGAs, describes what an FPGA is, its uses and architecture.
• Chapter 3, Soft Processor Cores, introduces soft processors with focus on the two candidates for the OSE$_{ck}$ port.

• Chapter 4, Operating System Fundamentals, presents some important operating system concepts that are needed in the following chapters.

• Chapter 5, Real-Time Systems, gives an overview of real-time aspects and how it relates to FPGAs and operating systems.

• Chapter 6, OSE$_{ck}$, briefly describes Enea’s operating system OSE$_{ck}$, its uses, properties, test systems and the operating system’s architecture.

• Chapter 7, Candidate Selection, compares the two processor candidates with respect to how suitable they are to host OSE$_{ck}$.

• Chapter 8, Development Environment, describes the development environment used and how Xilinx’s development tools work.

• Chapter 9, Implementation Considerations, presents some of the issues and solutions that have emerged during the implementation.

• Chapter 10, Integration Considerations, presents some of the issues and solutions that have emerged during the integration.

• Chapter 11, Results and Analysis, presents the achievements and results obtained from the functional- and performance tests as well as footprint figures.

• Chapter 12, Conclusions and Further Studies, concludes the results obtained from the implementation and integration of OSE$_{ck}$ on MicroBlaze.

• Appendix A, Abbreviations, contains a list of explained abbreviations used in this thesis.
Chapter 2

FPGAs

This chapter presents a brief history of circuit design methodology that lead to the FPGA concept. Different types and the architecture of today’s FPGA devices is presented as well as common application areas. The chapter ends with a description of the IP-Core concept and a presentation of common specialized hardware blocks found in the FPGA.

2.1 Circuit Design Methods

Early in the era of integrated circuit design, the circuits and the physical design was handcrafted as this was the only option. To manage ever increasing circuit size and complexity, while having limited computer design aid and time to market, different standard circuits emerged.

By using pre-designed logical blocks called standard cells, macro cells or mega cells, depending on the size, design time can be reduced. Cell based design, as it is called, does result in slower and less power efficient circuits but shortens the design time.

Another design approach when the time and costs associated with creating a custom design are too large is array based design. Large volumes of pre-manufactured chips with an array of gates without the interconnect layer can be produced at low cost. The chip design is then carried out by routing interconnects in between fixed transistors. Time and money can then be saved when only the interconnect network has to be custom manufactured.

The next step towards yet more rapid development is to remove the custom manufacturing step altogether. This is done by also including interconnects in the pre-manufactured chip but instead making them programmable. There are different ways of storing the interconnect configuration, both volatile and non-volatile solutions. This is covered more deeply in section 2.2.2. Large programmable chips are called FPGAs and this is the topic of this chapter.

Because of advances in computer design tools and the use of IP-blocks, only small critical parts of today’s chips are made by hand. Newly designed critical parts can later be reused and does not need to be redesigned every time.
2.2 Designing with FPGAs

The Field-Programmable Gate Array (FPGA) is, as the name suggests, programmable in the field. This means that the circuit designer does not have to go through the costly and time consuming process of manufacturing the actual chip. Instead chips containing useful logic resources together with programmable interconnects are mass manufactured. The designer can buy these chips and configure the interconnects with a programming device. This is illustrated in figure 2.1. The major drawback is that chip area, energy efficiency and speed is sacrificed for this flexibility.

![Custom chip FPGA](image)

Figure 2.1. Custom designed chips take a long time to develop and manufacture. FPGAs are pre-produced by the manufacturer and can easily be purchased and then programmed.

2.2.1 Application Areas

FPGAs are typically used in low volume applications because of the high cost per chip. FPGAs are ideal in the prototyping stage of a new product. When the volume gets large it may be wise to migrate the design to a custom manufactured chip. By using a custom chip it becomes more energy efficient, faster and less chip area is consumed. That is, cheaper per chip. The drawback with the custom chip is the long turnaround time and the huge cost involved in the preparation for manufacture of the new design. To address this, some FPGA manufacturers offer opportunities to help taking an FPGA design to mass production without designing a new custom chip. Altera have the option of taking an FPGA design and implement it in a functionality equivalent and pin compatible structured Application Specific Integrated Circuit (ASIC). The structured ASIC contains pre manufactured logic blocks that implement the same functionality as the FPGA. The metal interconnect layers are custom manufactured. This allows for easy migration that gives large power savings[16].

Testing of newly manufactured FPGA chips contributes significantly to the chip cost. By implementing application specific testing of manufactured FPGA chips, Xilinx can offer custom low cost FPGAs in large volumes. Time is saved by only testing the functionality that is used inside the chip. It does not matter if there is an error in silicon, as long as it does not affect the customer’s application. Test data is automatically generated directly from the FPGA development tools used by the customer. This approach provides a cheaper chip with very short lead time but with the same power consumption.[9]
2.3 FPGA Architecture

The major benefit with the FPGA is the possibility to reconfigure the device if the design changes. Even the pin assignment can be changed to some extent if it turns out to be a Printed Circuit Board (PCB) design error. Sometimes it is possible to change the design during runtime to allow for several logic functions to be realized at different instances in time. This has a potential of providing dynamic hardware acceleration in the future.

2.2.2 FPGA Types

Programmability of interconnects can be implemented in different ways. Some techniques allow reprogrammabilility and some allows for the FPGA to retain their contents on power-off. These are the different techniques available:[20]

- **Write-once or fuse-based.** These FPGAs use “fuses” in the interconnect network that are blown by running a high current through them. The fuses have small area overhead but can not be restored once blown. This means that the circuit has to be replaced if the design is changed.

- **Nonvolatile.** By using EEPROM or Flash memory for the interconnects the configuration is preserved without power. The disadvantage is that the memories require a special manufacturing process which makes the chip design complex and thereby expensive. This type of FPGA is appropriate to use when the chip design contains secrets, as opposed to volatile FPGAs where the configuration needs to be stored outside of the chip.

- **Volatile or RAM-Based.** This FPGA type used static Random Access Memory (RAM) to store the configuration. Every time the FPGA looses power, the content is lost. Therefore it needs to be saved in an external non volatile memory and loaded into the FPGA upon system boot. These FPGAs can be reused during prototyping and can be manufactured in a cheap process. This has made them very popular. Another advantage is that the RAM resources can also be used as memory, see section 2.3.2.

- **Volatile + on-chip flash.** As of this writing, Xilinx is preparing to release a volatile FPGA\(^1\) with flash memory included in the same package. This eliminates the need for an external configuration memory and thereby saves PCB space. [30]

2.3 FPGA Architecture

This section intends to briefly introduce the architecture of two modern FPGA devices, as of this writing. Altera’s Stratix and Xilinx’s Virtex series, the latest generations from each manufacturer, are examined. Both are volatile FPGAs. This section uses four main sources for reference material, two from Altera [4][5] and two from Xilinx [32][31].

\(^1\)Spartan-3AN
2.3.1 Logic Architecture

The two families have many similarities in the overall design. They both have a Look Up Table (LUT)-like structure with a register connected at the output as the basic logic building block. See fig 2.2. The LUT can be used to realize logic functions or to work as a small memory if the FPGA is RAM-based. By connecting several LUTs and registers together, more complex functions can be realized. The LUT is the basic building block of today’s FPGAs.

![Figure 2.2. A LUT connected to a register, the basic building block of today’s FPGA devices.](image)

The exact realization of the LUT and the register differs between the two manufactures and both are trying to make the optimal design in terms of reaching high overall utilization and speed. LUT in this section means a LUT-like device. The exact number of LUTs and the number of inputs to them depends on what is thought of as a true LUT.

The Stratix III device groups two merged 6-input LUTs and two registers into an 8-input block called Adaptive Logic Module (ALM). See figure 2.3 This block can be configured to emulate different LUT size combinations. The Virtex 5 device instead groups four 6-input LUTs and four registers into what is called a Slice.

![Figure 2.3. Logic hierarchy in Stratix III and Virtex 5.](image)

To continue up in the hierarchy, Stratix III have grouped ten ALMs into one Logic Array Block (LAB). The ALMs are interconnected to form arithmetic and register chains in the LAB. In Virtex 5, two Slices are grouped and feded with input from a logic block called a Switch Matrix. A Switch Matrix and two Slices are called a Configurable Logic Block (CLB).
The LABs in Stratix II and CLBs in Virtex 5 are placed in a grid like structure. They both have fast connections to neighboring blocks as well as connections to long distance interconnects running over the entire chip. The exact number of LABs and CLBs depend on the size of the FPGA device.

By programming the LUTs to realize different logical functions and interconnect them it is possible to build advanced logical circuits. When different functions are mapped into the FPGA, the computer design tools try to place related logic close to be able to use fast local interconnects. This is true at all levels in the hierarchy.

![Stratix III FPGA architecture](image)

**Figure 2.4.** Stratix III FPGA architecture showing how different resources are placed in silicon. Source: Altera’s website.

### 2.3.2 On-Chip Resources

Apart from logic, specialized resources are embedded in the FPGAs. Both essential blocks like I/O as well as more application specific ones like Digital Signal Processors (DSPs). This section covers the most common resources. Figure 2.4 shows how logic, memory, I/O, DSPs and clocking resources are placed in silicon.

**I/O**

To make a large FPGA useful it needs to be able to connect to the outside world. Therefore several different I/O standards are supported in the Virtex and Stratix families. The pins are designed to be as flexible as possible while still being able to manage high speeds. Different pins manage different standards. Some standard options are:
• Single-ended or differential communication
• Option to add internal pull-up/down resistors and terminations
• Adjustable input delay
• Adjustable output delay, current strength and slew rate
• Support for multiple voltage levels

Support for several memory bus standards and high speed serial interfaces are also common in these families. The next section covers this further.

Memory

Access to low latency memory is also an important resource in calculation intensive applications. There are two types of memory resources in the FPGAs covered.

• Distributed RAM
• Block-RAM

Distributed RAM is the LUTs being used as a RAM instead of as a look up table. This is a benefit of using volatile FPGAs. This memory is consuming logic resources and is preferably only used in small quantities.

Block RAM (BRAM) is Xilinx term for blocks of memory that both Altera and Xilinx have placed in their devices. Figure 2.5 shows BRAMs embedded in logic resources. The BRAMs continue up and down in columns spanning over the entire chip. See figure 2.4. These are accessed as a Static RAM (SRAM) memory and can be configured in different sizes and widths to suit different applications. The memories are dual-port which makes it possible to use them as clock domain crossing First In, First Outs (FIFOs).

![Figure 2.5. Floorplan showing part of a Virtex 5 FPGA with embedded BRAM and DSP cores. The layout is reproduced from the floorplanner tool in ISE.](image-url)
2.3 FPGA Architecture

Clocking

To be able to clock a large design, advanced clock distribution and generation support is needed. Integrated clock generators, PLLs, clock multipliers and phase shifters are used to cope with this issue together with several dedicated clock distribution nets.

2.3.3 Specialized On-Chip Hardware

In the recent FPGA generations, the manufactures have added several specialized hardware blocks into their devices to include dedicated support for different features. These are a few examples:

- High speed Serial Transceivers
- PCI-Express bus endpoints
- Processor cores
- Ethernet controllers
- DSP blocks
- Multipliers

Xilinx and Altera have different series within the Virtex 5 and Stratix III family that are tailored towards particular application areas. Virtex 5 has the families; general logic, serial connectivity, signal processing and embedded systems whereas Stratix III has mainstream, data-centric and high bandwidth versions.

Depending on the target application the FPGA is equipped with different embedded features and different amounts of memory to match the customer’s needs. The embedded hardware features both accelerates functions that can also be realized in logic, as well as enables features not available directly from logic. Less chip area is needed when a dedicated resource can be used as this is optimized for the particular task.

This specialization of logic can also be seen on low levels. Logical resources have special interconnects or are designed to accelerate and compact implementation of certain common logic. Counters are one such example.

2.3.4 IP-Cores

Because today’s designs are very large it is important, if not crucial, to reuse general design parts. This has lead to a business of sharing and selling pre-designed blocks called Intellectual Property Cores (IP-Cores) or IP-Blocks. These blocks contain code describing the hardware in a hardware description language.

IP-Cores come in two main flavors, Hard and Soft. Hard cores describes a complete fixed hardware layout whereas Soft cores only describes the logic functionality. That is, no information on placement in the actual hardware is provided.

One example of a hard IP-Core is the embedded IBM PowerPC 405 in Xilinx Virtex II Pro and Virtex 4 FPGAs.[22]
Soft cores can be used in FPGA designs. Both Xilinx and Altera have lots of cores available on their websites.[18][19] Some are free to use and some require a license fee. Some cores are designed to only work on specific FPGA families or requires special hardware resources. These are sometimes called Firm Cores, which indicates that they are somewhat less portable and optimized for a particular architecture. This thesis will not differentiate in between the two and will refer to soft and firm cores as soft cores. The next chapter covers the soft processor cores available from Xilinx and Altera.
Chapter 3

Soft Processor Cores

This chapter describes what a soft processor is and what its benefits and drawbacks are. The reader will be introduced to the architecture of the two processor candidates, Nios II and MicroBlaze and possible ways of customizing them. Finally, different methods for connecting custom acceleration hardware is presented.

3.1 Soft Processor Definition

A soft processor is a soft IP-Core (see section 2.3.4) implementing a processor. The processor have no fixed hardware layout and is described as code in a Hardware Description Language (HDL) language. HDL-code describes the logic functionality of the processor but not how it is mapped in hardware. Some soft processors are targeted towards a particular FPGA generation where specific embedded hardware is required, like memory or multipliers.

3.2 Advantages and Drawbacks

There are many advantages of using a soft instead of using a discrete processor (in its own package).

- The processor can be customized to only have the features needed.

- Peripherals are only added if needed.

- Custom application-specific hardware accelerators can be added to the processor.

- Busses previously connecting an external processor with the FPGA can be removed and I/O is saved.

- PCB space is saved because fewer components are needed.

- Late changes in software and hardware partitioning are possible.
The advantages has much to do with flexibility, the disadvantages are similar to FPGAs in general:

- Top performance can’t be reached with a soft processor. When there is no major part in the software that can be hardware accelerated, a soft processor may not be fast enough.
- A soft processor is less power efficient.

A combination of some of the benefits from both the soft processor and the discrete solution is the hard processor embedded within the FPGA. It has better performance and still has many of the soft processor benefits listed earlier. The great article FPGA Soft Processor Design Considerations[8] by RC Cofer and Ben Harding is recommended for anyone new to designing with soft processor cores. It lists opportunities, important considerations and common design oversights when starting with soft processor and FPGA design.

**Figure 3.1.** There are several advantages of moving a processor into an FPGA.

### 3.3 Xilinx’s and Altera’s Soft Processor Cores

Both Xilinx and Altera have soft processor IP-Cores available for use in their FPGA devices. The processors and many basic components is included in the development environment’s license fee but more advanced peripherals must be paid for.

Xilinx offers two soft processor cores, MicroBlaze and PicoBlaze. PicoBlaze is a small 8-bit RISC processor that is intended to be programmed in assembler. It delivers 21 to 102 Million Instructions Per Second (MIPS) depending on device according to Xilinx.[25] PicoBlaze is primarily intended to realize complex state machines and is not powerful enough to host an operating system.

MicroBlaze, powerful enough to host OSEck, delivers a maximum of 115 to 240 DMIPS\(^1\) depending on device.

Altera offers the soft processor core Nios II which has a similar overall design and performance as MicroBlaze. Altera claims to be able to deliver a maximum of 250 DMIPS [3] which is very close to what MicroBlaze delivers.\(^2\)

---

\(^1\)Dhrystone 2.1 measured by Xilinx.

\(^2\)It should be noted that these performance figures vary with FPGA device and family.
Other familiar cores like PIC16C6x and 80C51 are available from third party vendors as well as processors distributed as open source.

The remainder of this chapter will be devoted to the Nios II and MicroBlaze processors. Note that this thesis is based on version 7.1 for Nios II and version 5.00a for MicroBlaze. As compared to its hard counterparts, soft processors are easy to change after the initial release. Several minor features have changed during the writing of this thesis, which illustrates why it is very important to consult the latest version of specification documents and manuals. Architectural details in this thesis are subject to change when new processor versions are released. The two manufacturers seem to copy good solutions from each other frequently. When using the name Nios, Nios II is referred to unless Nios I is explicitly written.

## 3.4 Basic Architecture

Both processors have a 32-bit Reduced Instruction Set Computer (RISC) harvard architecture. This means that there are separate instruction and data busses which can handle up to 4 GByte respectively. They don’t separate between memory and peripherals, that is, they use memory mapped I/O. This chapter mainly uses the processor’s reference manuals [3, 28] for reference unless other sources are specified.

### 3.4.1 Processor Busses

Altera and Xilinx use two different bus standards as a base for their System-On-Chip (SoC) solutions. Altera have designed the System Interconnect Fabric while Xilinx uses the CoreConnect Bus Architecture designed by IBM together with a fast memory bus.

Altera’s System Interconnect Fabric[6] acts a bus interface in between different Avalon Ports which can be masters or slaves. It connects masters to slaves by generating logic for address decoding, arbitration and pipelining. It is possible for several masters to talk to different slaves at the same time. The System Interconnect Fabric is used both for peripherals and memory.

The IBM CoreConnect[17] architecture was designed to provide some common bus standards to connect components in a SoC environment. It contains three different busses of various complexities to suit the needs for both performance and simplicity. The On-Chip Peripheral Bus (OPB) is one of them which is used by MicroBlaze to connect peripheral and optionally memory.

Xilinx also provides the Local Memory Bus (LMB) and the Xilinx CacheLink (XCL) for memory connectivity.

### 3.4.2 Memory

Memory is needed to store instructions and data. Both processors have a way of connecting FPGA internal memory blocks; on Nios the connection is called Tightly Coupled Memory whereas on MicroBlaze it is connected on the LMB. This is low latency memory and the amount of it varies from device to device.
On MicroBlaze, FPGA external memory can be connected either directly on the cached XCL interface or via the OPB. Nios connects to external memory and peripherals via an Avalon master port. Cache is implemented using FPGA internal memory blocks.

### 3.4.3 Interrupts

Nios has a simple integrated interrupt controller with 32 inputs. MicroBlaze has only one interrupt input to the processor core and must use an external interrupt controller peripheral to expand the number of inputs.[26] One peripheral provides 32 inputs. Cascading of several controllers is possible to further increase the number of available inputs. See figure 3.2. Unused interrupt inputs are optimized away when the hardware in synthesized.

![Interrupt controllers on Nios II and MicroBlaze.](image)

Both processors use one common interrupt entry point for all inputs. Nios does not provide any priority mechanism in hardware. MicroBlaze provides a fixed priority scheme that is decided by the way the interrupting devices are connected to the interrupt controller.

### 3.4.4 Registers

Both Nios and MicroBlaze uses 32 registers divided in four types: [3][28]

- **Volatile (caller-save)** registers are used for temporary variables and for passing parameters to and from subroutines. These registers may change when executing another context and must therefore be saved by the compiler before calling a subroutine. An interrupt routine must make sure to save these registers if they are to be used in the interrupt routine’s run time environment.

- **Non-volatile (callee-save)** registers retain their contents across function calls and must therefore be saved before they are used. Normally the compiler takes care of this but when programming in assembler, this is the responsibility of the programmer.

- **Dedicated registers** have a predefined meaning or purpose. For example to hold the return value from a subroutine. These may be manipulated directly by the hardware. They should not be used for storing software variables.
• **Control (special)** registers are registers that are tightly connected to some hardware. They are used to change or read the status of the processor core. Typical registers are program counter, machine status, exception status, processor version and interrupt enable registers. These registers are not actual registers and therefore special instructions are needed to access them in both Nios and MicroBlaze. This makes them slower to save to memory because they must be transferred via the register file.

### 3.5 Processor Customization

One of the major benefits of soft processors are that they can be configured with only the components and features required for a particular design. Both MicroBlaze and Nios II have many options that can be tweaked. Here are some properties that can be included or excluded for both processors:

• I-Cache, D-Cache
• Multiply hardware support
• Barrel shift hardware support
• Divide hardware support
• Floating Point hardware support
• Bus connections

Nios II comes in three versions with different size and speed properties. One is optimized for speed, one for small size and one for minimal size. DMIPS performance is about seven times lower for the minimal sized kernel compared to the performance optimized version while less than half the logic resources are used. The different Nios versions allows the number of pipeline stages\(^3\), the inclusion of pipelined memory access and the level of branch prediction, to be changed. MicroBlaze has these options fixed\(^4\) but most other options can be changed for MicroBlaze.

In addition to processor core customization comes selection and customization of peripherals, memory and bus architecture.

### 3.6 Custom Hardware Acceleration

FPGA embedded processors have the unique opportunity to easily add custom hardware support for commonly executed software routines. By building specialized hardware it is possible to replace critical software routines with parallel hardware to enhance performance.

---

\(^3\)MicroBlaze have a 5-stage pipeline and Nios II can have 6, 5 or no pipeline stages.

\(^4\)Update: The latest MicroBlaze version (v6) have a 3 or 5 stage pipeline. It has not been verified how this option is changed.
MicroBlaze has eight interfaces called Fast Simplex Links (FSLs). A FSL is a dedicated unidirectional point-to-point data streaming interface that can be used to communicate with custom designed hardware. It is 32 bits wide with one extra bit that determines if there is data or a control word being sent. The FSL is accessed with special put and get machine instructions which make the overhead very low. A FIFO queue in the FSL enables different clock domains to be easily interconnected.

Nios has chosen another solution for connecting custom hardware where it is possible to implement custom instructions directly in the core data path. This enables very tight integration. The drawback is that the custom hardware has to work synchronous with the kernel and be fast enough not to limit the clock frequency of the processor.

Further it is possible to connect hardware modules as peripherals by implementing interfaces for the OPB or the Avalon port.
Chapter 4

Operating System Fundamentals

This chapter covers some of the most fundamental duties of an operating system that needs to be understood to be able to follow the following chapters. Describing operating systems can be the subject of entire books and this chapter only aims to describe low level parts related to context switch and interrupts. This knowledge is crucial when designing architecture dependent parts of an operating system.

4.1 Purpose of Operating Systems

The purpose of an operating system is to provide a platform on which to execute programs.[1] A program is referred to as a process from here on. An operating system is also a resource allocator. The computer system has resources like CPU, memory and peripherals that needs to be shared among several processes.[11] It is the operating system that decides which process in the system that is allowed to use a resource at a specific time. Further is an operating system normally providing many other resources, functionality and hardware abstraction that helps the software developer develop large and complex applications. See Operating System Concepts[1] by Abraham Silberschatz and Peter Galvin for more general operating system information.

4.2 Context Switch

The most crucial duty of a multitasking operating system is to schedule and grant the different processes time to use the Central Processing Unit (CPU). The switch from one to another process is done by saving the process’ state, referred to as the process context, and restore another process’ context. This is called a context switch.

By maintaining a queue of processes ready to run and switching them in and out at a high rate, one at a time, it seems like they are running in parallel.
It is important that the time needed to do a context switch is low because the time spent switching is not useful for the application. If the processes are switched at a high rate, little time is left in between the switches and much time is spent switching. For example; If one context switch takes 1ms and processes are switched in and out 500 times every second that would only leave half the CPU-time for applications.

The state of the processor is mainly described by the contents of the internal CPU registers. The registers contain variables used by the running process and other information like the program and stack pointer. These registers are shared among all processes in the system and must therefore be saved.

The process of saving the state is to copy the contents of all registers into memory. After this, another state is restored by copying a previously saved register content back from another memory location. See figure 4.1. The more time it takes to do this copy, the slower the context switch is.

There are two main types of registers, volatile and non-volatile. All registers does not need to be saved every time, only the ones used. To be able to manipulate with registers, the context switch must be written in assembler. See section 3.4.4 for more information on the registers available in Nios and MicroBlaze.

The compiler can be expected to save the volatile registers before calling the context switch function so these does not need to be saved. On the other hand, when interrupts occur these must be saved. This will be discussed in the next section.

4.3 Interrupt Processing

An interrupt request is a peripheral signaling to the processor that it needs assistance or that something needs to be done. An example is an input peripheral has received data. It is interrupting the processor to tell that the data is available and that it needs to be moved out of the peripheral’s input buffer to make room for
additional data. Interrupts are associated with software interrupt routines that are executed when the interrupts occurs. In the previous example the routine will empty the peripherals buffer.

Interrupts can arrive to the processor at any time and will cause a branch to the interrupt routine. The difference between a normal context switch and an interrupt is that the interrupt is not expected whereas the normal context switch is invoked by the program. An interrupt is much like an ordinary process but its state does not need to be restored on interrupt, it start from the beginning every time. The interrupted process’ state need to be saved because it should later be started again.

Interrupt require a complete context switch, commonly referred to as a large context switch, because the previous process’ state can not be known. A software invoked context switch implies that the process that is switched out is always in the context switch function at the switch moment. With this information, the volatile registers can be safely discarded because the compiler (or the assembler programmer) saves these registers, before the function call, if they are needed later. A context without the volatile registers is called a small context. Care must be taken when designing the context save and restore routines, as the processor’s unsaved state must not be affected by the instructions used to save the state.
Chapter 5

Real-Time Operating Systems

This chapter presents the concept of real-time and constraints on Real-Time Operating Systems (RTOSs). Real-time requirements are important to consider when choosing processor and hardware. The final section will present properties that is important to consider when choosing a RTOS.

5.1 Real-Time Definition

Today, real-time systems exist everywhere in modern society. Most of them are embedded, in cars, MP3-players and many other products. Usually real-time systems are used in a context where they should respond to external events of some sort. Qing Li defines real-time systems like this:

“Real-time systems are defined as those systems in which the overall correctness of the system depends on both the functional correctness and the timing correctness. The timing correctness is at least as important as the functional correctness.”[23]

Where functional and timing correctness means that the system should respond with a correct answer within a strict timeframe. The seriousness of a missed deadline depends of the application but can mean anything from death or huge economical loss, to a lost video frame when watching a movie.

Real-time systems are commonly classified in Hard and Soft real-time systems. Hard real-time means that the timing correctness must be met every time whereas soft real-time might allow for a percentage of missed deadlines without serious damage.
5.2 FPGAs and Real-Time

FPGAs are commonly used as accelerators when high performance is needed. Because of this, FPGAs are often involved in real-time systems where there is a tight time budget.

In case a processor is embedded in an FPGA it is likely that it is involved in real-time tasks. Like controlling and feeding real-time hardware. This makes it important to have a processor and a real-time operating system that together can manage real-time tasks in a controlled and deterministic manner.

5.3 Real-Time Operating System Properties

The RTOS is an operating system capable of managing real-time constraints from the environment. It does not necessarily have to be very fast or have high performance but it always has to be fast enough for the application that it is used in. The remainder of this section will list properties of RTOSs where predictability is the most important one. Qing Li presents these key characteristics of an RTOS[23]:

- reliability
- predictability
- performance
- compactness
- scalability

One should have in mind that these are only technical aspects. If the technical requirements are fulfilled then attributes like:

- run-time royalties
- features
- customer support
- processors supported
- development tools supported

becomes more important. John A. Carbone [24] have collected data from different sources that indicates that it is not having the lowest interrupt latency that is important for most applications when choosing an RTOS, but instead good development tools and customer support.
RC Cofer and Ben Harding also believes that the availability of a full-featured middleware is important when choosing operating system in general, not specific for RTOSs. In addition to what has already been presented, here are some more specific functional features to consider:[8]

- level of Integrated Development Environment (IDE) integration
- kernel robustness
- pre-emption
- resource allocation
- protection schemes
- footprint
- task services
- priority levels
- timer management
- Application Programming Interface (API)
- IPC and synchronization
Chapter 6

\textbf{OSE_{ck}}

This chapter describes fundamental concepts and functionality of the OSE_{ck} operating system. Closely related products will also be described as well as the different kernel versions and how the kernel is configured for use. Finally two test systems will be covered and some hardware and architecture dependencies relevant when migrating OSE_{ck}.

6.1 The OSE Family

OSE Compact Kernel (OSE_{ck}) is a DSP targeted RTOS developed by Enea. It is one RTOS in a family of three where OSE is the largest, OSE_{ck} the small and OSE Epsilon is the smallest. Today the OSE family is widely used in telecommunication equipment, mobile phones and in the automotive industry.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{OSEckArchitecture.png}
\caption{Architecture and dependencies between different parts of the OSE_{ck} kernel, Heap, LINX and the Timeout Server.}
\end{figure}
6.2 **OSE\textsubscript{ck} Fundamentals**

This section will describe some common system calls, the signal IPC concept, memory pools and the process types in OSE\textsubscript{ck}. For more information see the OSE\textsubscript{ck} reference material [10, 11].

### 6.2.1 Basic System Calls

A system call is a call to the operating system kernel. The kernel offers services to the application through system calls. These are some of the basic calls available in OSE\textsubscript{ck}:

- **alloc.** Used for allocating memory and signals from memory pools. See section 6.2.2.
- **delay.** Suspends the calling process for a number of Ticks\textsuperscript{1}.
- **error.** Reports an error to the error handler.
- **receive.** Receives a matching signal from the calling process’ signal queue. This call block until a signal is available.
- **receive\_w\_tmo.** Works like receive but a timeout is used that specifies a maximum number of Ticks that should be waited for a matching signal. A value of zero makes the call return immediately.
- **send.** Sends a signal buffer to another process.
- **sender.** Provided a signal buffer it returns the process who sent the message.

These are enough to manage many tasks but there are about 40 more calls to do more advanced tasks. For a complete reference, see the OSEck Kernel Reference Guide\textsuperscript{[10]}. Only system calls used by the application are included when building the software which means that the size of OSE\textsubscript{ck} varies with the required functionality.

### 6.2.2 Memory Pools

Memory pools are used in OSE\textsubscript{ck} for allocating memory for signals and other uses. When memory pools are created, eight different buffer sizes are specified. Once data is allocated from the pool with the alloc system call, the smallest buffer that is big enough for the requested size is returned. No merging of free small buffers are made, once a buffer is allocated its size is fixed. The pool gets divided into different amounts of different buffer sizes depending on what the application uses. This enables deterministic and effective memory allocation.

One pool must always be available, the system pool, but additional pools can be created. Pool memory must be used for allocating signals but as the overhead is very low, it can be used for other purposes as well.

\textsuperscript{1}System time is measured in Ticks.
6.2.3 Signals

A signal is a message that is sent from one process to another. Signals are used as the primary IPC mechanism in OSE\textsubscript{ck}. Signals are allocated from memory pools that are part of every OSE\textsubscript{ck} system. A signal contains a signal number and user data. The signal size, sender and addressee information is also embedded in the signal, hidden from the user.

Once a signal is allocated it can be sent to another process with the \texttt{send} system call using the Process Id (PID) as destination. If the receiver’s PID is not known, it can be \textit{hunted} for. The name of the process is given as parameter and the PID is returned. If the process does not exist it can be waited for, in which case a message informing of the process availability is received once the process becomes available.

The receiving process receive messages from its message queue with an optionally blocking \texttt{receive} system call. A timeout can also be used.

Ownership of the signal buffer is transferred to the receiver once the signal is sent. The receiver can read the data and de-allocate the signal by returning it to the memory pool or send it on to another process.

6.2.4 Process Types

There are two types of processes. Prioritized processes and Interrupt processes. Both can be created either statically before system start or dynamically in runtime.

Prioritized Processes

Prioritized processes are the most common ones and are written as infinite loops which will run as long as no interrupt process or another prioritized process with higher priority is ready to run.

Interrupt Processes

Interrupt processes are called in response to a hardware interrupt or a software event. They will run from the beginning to the end, if not interrupted by a higher priority interrupt. Two types of interrupt processes exist, OS interrupt processes and User interrupts. OS interrupts have access to system calls and are written in a high level language whereas User interrupts are somewhat faster but must be written in assembler.

6.2.5 Process Priority

Both interrupt processes and prioritized processes must be assigned a priority from 0 to 31, where 0 represents the highest priority. The priority has a different meaning for the two process types. Prioritized processes compete for CPU time where a process with higher priority always executes before any process with lower priority. That is, pre-emptive, priority based scheduling.
When an interrupt occurs, the priorities of the interrupt processes determines which interrupts that is allowed to interrupt the current running interrupt process. Interrupt processes can be nested and runs until all have completed, then prioritized processes are allowed to continue. User interrupts run with interrupts disabled.

Interrupt processes can also be started as a response to a software event. For example when a message is sent to an interrupt process. If another interrupt process with higher priority is already running, the event is queued. The interrupt process is then invoked later when the priority allows.

### 6.3 Kernel Configuration

OSE\textsubscript{ck} kernel is normally delivered in three different library versions.

- Release
- Release with hooks
- Debug with hooks

The release libraries contain few error checks making it faster, smaller and more deterministic. The debug version on the other hand has the possibility to do some runtime checks:

- **Parameter checks.** Verifies parameters like buffer sizes and process identity when using system calls.

- **Validate the caller.** A number of system calls that are not allowed from interrupt context checks that the caller is a prioritized process.

- **Buffer and Stack checks.** Buffer pointers and owners are checked. Buffers are checked for overflow.

- **Initialization checks.** Verifies that semaphores are properly initialized.

It is also possible to include support for hooks in the kernel. A hook gives the possibility to add custom routines that are executed at certain system events. It is possible to execute a routine every time a process switch occurs, for example.

To configure hooks, checks, static processes, memory pools and other kernel settings is a program named **MkConfig** used. A configuration file (appcon.con) is prepared with the appropriate settings. This file is read by the MkConfig program which generates a C file (appcon.c) and sometimes also an assembler file. This file should be linked with the kernel library. See figure 6.2.
6.4 Closely Related Products

Enea develops other products that are normally used and delivered together with OSEck. See figure 6.1 for an overview. In addition to the products listed in this section there are products like the debugger and optimizer tool Illuminator that tracks system level statistics and signal queues. Higher level platforms are also available that ties several operating systems together. The following subsections describe the most common products.

6.4.1 Timeout Server

The Timeout Server provides high resolution timeouts for applications running on OSEck. Timeouts can be scheduled to occur at absolute or relative, one-shot or periodic, points in time. Upon timeout a user supplied callback function is invoked.

The timeout server needs a BSP driver that controls a reprogrammable timer peripheral. Usually this timer is set up to have much higher resolution than the OS-Tick\(^2\) to allow for fine grained timeouts. The Timeout Server manages a queue of pending timeouts and reprograms the timer to interrupt the processor when the next timeout is scheduled to occur. For more information see the Timeout Server Uses’s Guide[12].

6.4.2 Heap Manager

The Heap Manager provides reentrant heap functionality with low overhead and bounded execution time. It is an alternative to using signals for dynamic memory allocation.

\(^2\)The OS-Tick is the system clock.
6.4.3 LINX

“LINX is a technology for location transparent interprocess communication for heterogeneous distributed systems, e.g. clusters and real-time multicore systems. It is based on the standard OSE signal interprocess communication mechanism.” [13]

LINX can be seen as an extension of the standard signal communication within OSE. It enables processes to communicate independently of physical location and on which processor it executes. The same approach is used when sending a message to a local process as to a remote one. LINX is currently available for Enea’s OSE, OSEck and Linux operating systems. On Linux it is a kernel module that provides the LINX API whereas on OSE and OSEck LINX is mostly hidden behind the standard signaling mechanism. LINX for Linux is distributed as Open Source.

![Figure 6.3. LINX transparently delivers messages to processes distributed over several processors.](image)

Different communication medias can be used as physical transportation. Both reliable and unreliable protocols are supported, Thing such as retransmission and link supervision are hidden from the user. LINX automatically forwards messages to the right processor via different medias as is exemplified in figure 6.3.

LINX also contains functionality to set up supervision of processes. When a process dies or all communication paths to a monitored process becomes unavailable it is possible to get a notification. This way it is possible to create a fault tolerant system where applications automatically reconnect to backup processes perhaps on another processor.

Another product, Element, provides a higher level of abstraction on top of LINX and is designed to provide a platform on which to build reliable systems with redundancy.
6.5 Test Systems

For Enea internal testing purposes there are different test systems available. Functional test programs, performance test programs and stress tests. These are used to test the correctness and performance of a new processor port.

6.5.1 Functional Testing

The functional test program runs over five hundred different tests which verifies that the kernel conforms to the functional parts of the requirement specification\(^3\). This program will be used to test the correctness of the implementation.

6.5.2 Performance Measurement

Performance is measured by counting cycles when performing various system call sequences like sending a message, swap process and then receive the message. Different interrupt sequences are also timed. This gives an indication on the performance of doing common tasks but does not show how well a particular application would perform.

To get fair values, several tests should be performed using different combinations of memories and with cache both enabled and disabled. Although it is impossible to guarantee any cycle count with cache enabled, it gives an indication on what can be expected.

6.6 Hardware Requirements

\(\text{OSE}_{ck}\) has very low requirements on the hardware platform. Apart from memory to hold the minimal kernel footprint, only one periodic timer interrupt is required. The timer is needed for generating interrupts for the system clock, called OS-Tick. This timer enables \(\text{OSE}_{ck}\) to employ pre-emptive scheduling and to use system calls like \texttt{sleep} and \texttt{receive\_w\_tm0}\(^4\).

Normally, \(\text{OSE}_{ck}\) uses 32 priority levels, both on prioritized processes and interrupt processes. All priority levels can be changed during runtime. This has to be implemented in software, if not available in hardware, on new target architectures.

\(\text{OSE}_{ck}\) is designed for both little and big endian memory model.

6.7 Architecture Dependent Parts

The parts of the \(\text{OSE}_{ck}\) kernel that is architecture dependent are all collected in a few source files. Mostly they contain parts that relate to interrupts and context switch. A major part of this code is written in assembler and is contained in the arch.asm file. This is some functionality that needs to be rewritten for new architectures:

\(^3\)Enea internal document.

\(^4\)See section 6.2.1 for description.
- A function to switch context. When this function is called from one process, that process doesn’t return immediately. Instead another process returns from the function and continues from where it called the switch context function when it was switched out. Process’ context is saved on the process’ stack.

- A function to set up a context that is used when creating new processes. The purpose is to set up the new process’ stack so the switch context function can be used to swap in the newly created function.

- An interrupt entry point that is branched to on interrupt. It saves a complete context on OS interrupts and switches to interrupt stack if it is not a nested interrupt.

- A soft interrupt entry point, branched to when an interrupt process is to be started as a consequence of some software trigger.

- Routines to find out which process to swap in next.

- Routines to manage the interrupt priority levels.

Two BSP drivers are needed by the kernel, one driver for the interrupt controller and one timer driver to run the OS-Tick. Additionally some files describing data type sizes and alignment must be modified for the new architecture. This information is used when accessing kernel data structures from assembler and by debugger tools.
Chapter 7

Processor Selection

This chapter compares the two soft processor candidates, MicroBlaze from Xilinx and Nios II from Altera. Hardware properties that affects the suitability of hosting OSE\(_{ck}\) as well as how good the IDE supports operating system integration, is examined. The chapter ends with a conclusion and selection of one processor to host OSE\(_{ck}\).

7.1 Architectural Properties

To achieve real-time, timing correctness is crucial according to what was covered in chapter 5. To always meet the required timing constraints, predictability is the most important property to examine for the processors. OSE\(_{ck}\) must be able to execute in a predictable way on the target processor.

When this property is satisfied it is also important to examine what performance differences there are in between the two candidates. OSE\(_{ck}\) together with LINX is well suited to be used in a multiprocessor environment. This makes it interesting to see how well the processors manage in such a configuration.

Some implementation specific details related to interrupts and timer availability are also examined in this section.

7.1.1 Predictability

OSE\(_{ck}\) is built for predictability by having bounded execution time on all system calls. To have predictable execution time is important so that no events can stall the processor at random occasions. Stalls can happen because of unpredictable memory access latency and memory management traps\[7\]. The interrupt latency must also have an upper limit.
Memory

Memory access latency have an upper limit on both processors if the dedicated memory interfaces, LMB on MicroBlaze and Tightly Coupled Memory on Nios are used to access internal memory: Tightly Coupled Memory: “Each tightly coupled memory port connects directly to exactly one memory with guaranteed low, fixed latency.”[3] LMB: “It uses a minimum number of control signals and a simple protocol to ensure that local block RAM are accessed in a single clock cycle.”[28]

To ensure that peripheral memory access does not stall the processor one must take care when setting up the OPB bus on MicroBlaze and the Avalon Switch Fabric on Nios. The real-time parts of the system must have the highest priority on shared busses to be able to guarantee the timing for predictability.

Interrupt latency

Worst case interrupt latency specifications is not available for either of the two processors. On MicroBlaze it is stated that the worst interrupt latency occurs when executing the divide instruction, if the divider is included. The cycle count for the integer divide instruction is 32 cycles at most. Nios specify a 10 cycle count from interrupt to the first instruction is executed in the ISR under certain conditions on code locations in memory. When asking Altera about this they say that the ten cycle count begins after the completion of the instruction in the execution stage of the pipeline. Depending on the Nios configuration, there can be a divide instruction that uses up to 66 cycles in the pipeline. It is probably safe to say that a worst case latency exists as long as the interrupt vector resides in internal memory. No instruction takes forever.

Nios I implements the concept of sliding register windows1. This is highly inappropriate in a real-time system because when all available register windows are used, a memory management trap occurs which must copy the register content to memory. This can happen any time in a multitasking system and thus makes the system indeterministic. In Nios II this functionality has been removed and now has a very similar register bank as MicroBlaze.

7.1.2 Performance

To get good performance with an RTOS, interrupt and context switch latency should be low. A quick way of saving and restoring the processor's state is important. The time taken to save the state depends on the number of registers that needs to be saved as well as how long it takes to save the registers. General performance is not addressed as this is not included in the scope of this report.

---

1 Several register banks are used and the some of the current visible registers change when performing a subroutine call. This way data can be transferred in between functions in a rapid way. See [7] for more information
Register File

Nios’ and MicroBlaze’s register files are very similar. They both have 32 registers where Nios has eight and MicroBlaze has seven volatile registers. The exact number of registers that needs to be saved depends on what features of the processors that is used but the difference is small. The 32 general purpose registers can be written to memory within a single clock cycle each but the special purpose registers needs to use an additional instruction which sums up to two cycles per register. Only one special purpose register needs to be saved in both processors, the status register.

To conclude, interrupt and context switch latencies are very similar in the two processors.

7.1.3 Multiprocessor Support

As long as there is enough space in the FPGA device it is possible to use several processors working side by side. The tricky part is to implement efficient inter processor communication in a simple way. This section will cover some of the interfaces available for inter processor communication.

Shared Memory

Both development environments support the creation of shared memory regions and have the possibility to select this memory as non-cached. This avoids the issues of cache coherence\(^2\). Shared memory can be used as medium for LINX and for setting up shared memory pools in OSE\(_{ck}\). Nios have special instructions that allow non-cached reads and writes to be performed in cached memory regions.

Point to Point Interfaces

MicroBlaze has the FSL interface. It allows for direct 32-bit parallel, processor to processor communication. The FIFO structure allows the link to cross different clock domains. With the instruction level access, this interface could be used to build a high speed LINX interface. Direct links can be created in between processors that needs high throughput while non critical communication can be routed via other processors. There is a good opportunity to easily test different interconnect topologies and add extra communication resources when needed in the FPGA. LINX hides the physical links which makes the application independent of topology changes.

The Avalon architecture also supports a point to point streaming configuration that has some of the properties that MicroBlaze’s FSL interface has.

\(^2\)When several processors share the same memory and have separate caches they might see different data at the same memory location. A write by one processor might only be stored in the cache and is therefore not visible by the other processor. The same applies to reads.
Bus Interfaces

Altera’s Avalon bus architecture allows several masters to talk with different slaves at the same time, something not possible on MicroBlaze. This can enhance performance when several processors are accessing a shared set of slaves. No bridges are needed to separate traffic. The area overhead is not considered here.

Xilinx has chosen to use the IBM designed Bus architecture CoreConnect including the OPB and Processor Local Bus (PLB) busses for their SoC solution. This choice gives support for the PowerPC 405 embedded in some of Xilinx FPGAs. OSE\textsubscript{ck} is available for the PowerPC architectures which gives good possibilities for cooperation between MicroBlazes and PowerPCs in the same FPGA.

Off-Chip Connectivity

Both Xilinx and Altera have high speed serial i/o for their FPGAs which can be used for high speed off-chip connectivity.

7.1.4 Interrupt Hardware

Interrupt latency was covered in 7.1.2 and this section will examine the interrupt hardware and its compatibility with OSE\textsubscript{ck}.

As described in section 3.4.3, Nios uses an internal interrupt controller whereas MicroBlaze uses an interrupt controller peripheral. Neither of the processors provides any auto vectoring\textsuperscript{3} capabilities. One common entry point is used for all interrupt inputs. The software is responsible of making a jump to the correct interrupt routine.

Nios integrated controller does not provide any priority mechanism. This must be handled in software. MicroBlaze’s interrupt peripheral does provide a fixed hardware priority setup and optionally hardware that calculates the highest priority active interrupt input. OSE\textsubscript{ck} needs 32 priority levels that can be dynamically assigned which makes MicroBlaze’s fixed priority scheme unusable. The hardware support for calculating the active interrupt can be used to some extent.

Because of this Nios and MicroBlaze offers about the same interrupt functionality for OSE\textsubscript{ck}. The major difference is that Nios has an internal controller and MicroBlaze an external. Interrupt priorities must be emulated in software for both processors. Currently only one interrupt peripheral is available (opb\_intc) for MicroBlaze. To further enhance performance a custom interrupt controller peripheral could be written that is well matched to OSE\textsubscript{ck}.

7.1.5 Available Timers

To be able to run OSE\textsubscript{ck} and the Timeout Server, timer peripherals are needed that can interrupt the processor. Nios has the highly configurable peripheral \textit{Interval Timer} that connects to the Avalon bus and can be configured for different modes.

\textsuperscript{3}For auto vectoring interrupts, the interrupt controller provides all or some portion of the address of the different interrupt routines.
The *Simple periodic interrupt* timer configuration is appropriate to use as tick generator whereas the *Full-featured* configuration can be used together with the Timeout Server.

MicroBlaze has two useful timer peripherals. The *Fixed Interval Timer* (fit_timer) and the *OPB Timer/Counter* (opb_timer). The fit_timer have no bus connectivity and only generates periodic interrupts which makes a small implementation for the Tick interrupt generation. The Timeout Server can use the opb_timer which can be configured over the OPB bus.

Both processors have peripherals that satisfy the needs from OSE\textsubscript{ck} and the Timeout Server.

### 7.2 Development Tools

The possibilities to integrate OSE\textsubscript{ck} into the IDEs and toolchains will be examined in this section. Both Altera and Xilinx have a graphical development environment that can be used to configure the embedded system including processors and peripherals.

The scope of this thesis does not permit a deeper study of the development environments. It is a major task to compare them. A comparison can easily fill a thesis on its own. The programs take significant time to learn to use and even more time to learn well enough to be able to make additions to the environment.

#### 7.2.1 Usability

It is easy to set up a basic system in both environments. Neither one of them have any significant disadvantage that I have come across. The most important thing when choosing environment is previous experience. They have different ways of doing things but mostly it is a matter of preference.

The Nios II development environment uses a special guide, the System-on-a-programmable-chip (SOPC) Builder, which generates the platform including processors and peripherals. The SOPC Builder outputs the design as HDL-files as well as a .sopc file that describes the system design in a XML-like format. No documentation about this file format has been found.

MicroBlaze’s development environment saves the system design mainly in two files a .mhs and a .mss file. These file formats are well documented and the two files are directly available for manual editing in the development environment.

Most of the time it is probably not necessary to manually edit the design files but it can be a good thing to have the possibility of automatically generating them. It takes some time to build a system in the graphical environments. Especially if it involves multiple processors. When the systems become large and perhaps standardized in a company it can be convenient to be able to generate different platform variations automatically. Automated testing of different configurations is one example.
7.2.2 Multiprocessor Support

There is no problem adding multiple processors in any of the development environments. Altera has some peripherals, not found in Xilinx tool, which is specially designed for multiprocessor systems. These are a Mutex and a Message box peripheral that is designed to help protect common resources like shared memory. In practice these two are almost the same hardware, and the significant part is the software support that is provided. The benefit of using this instead of implementing the same functionality in shared non-cached memory is that it is easy to assign a driver to a physical peripheral. This is not very important for the LINX interface, as it has support for shared memory communication.

7.2.3 Documentation

Documentation on how to integrate a new operating system into the development environments is important.

Several operating systems available for Nios II are listed on Altera’s website. It is also written that “Many of the operating systems are provided as plug-in components to the Nios II integrated development environment (IDE) for seamless configuration.” However, no documentation on how to make such a plugin was found. It was also confirmed by Altera’s support that no such documentation existed.

Xilinx also has a number of operating systems available for MicroBlaze that can be installed as plugins. The Platform Specification Format Reference Manual[29] is delivered with MicroBlaze. It contains documentation on how drivers, libraries and operating system plugins are designed. In practice it describes the syntax for all the file formats used to describe the platform and components.

It would be strange if Altera did not have any documentation on operating system integration. They do have documentation on how to integrate drivers into their hardware abstraction layer[2]. The documentation looked for is probably internal documents available only to Altera partners.

7.3 Conclusion

The two processors are very similar. When internal memory is used and bus priorities are correctly set up, no predictability issues exist. Context switch and interrupt performance are estimated to be almost the same. The bus standard used differs somewhat in between the two candidates. However, none of them have any direct affect on OSE_{ck}. The MicroBlaze FSL interface is an interesting candidate to use with LINX when the FPGAs have grown large enough to contain systems with several processors.

Interrupts are also handled in the same way in the two processors. The only difference is that Nios’ controller is controlled with internal registers whereas MicroBlaze must access registers in a peripheral controller.

Both processors have the timers that are needed. Nios has a universal timer that is highly configurable. MicroBlaze has two separate timers that do about the same job.
The significant difference between the two candidates was the available documentation. Nios did not have any documentation on how to integrate a new operating system publically available. MicroBlaze, on the other hand, did have a reference manual describing the files required.

*MicroBlaze is chosen as the new target for OSE* _ck_. *Nios II is an equally good candidate when equivalent operating system integration documentation is easily available.*
Chapter 8

Development Environment

This chapter describes the development environment setup used when implementing OSE\textsubscript{ck} for MicroBlaze. The software platform description files are presented as well as the library generation. This chapter ends with a description of the development boards and the soft processor platforms used.

8.1 Software Tools

Several software tools have been used. The main editor used for writing code in C and assembler have been Emacs running on a remote server. The OSE\textsubscript{ck} build environment is hosted on a Linux server and this is where the kernel libraries have been built. See figure 8.1. Testing and verification of the libraries in different applications has been done locally on a Windows XP machine connected to a development board.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{development_setup.png}
\caption{Development setup.}
\end{figure}

8.1.1 Xilinx Embedded Development Environment

The Xilinx Embedded Development Kit (EDK) 9.1 has been installed both on the Linux Server as well as on a local Windows XP machine. It includes graphical tools to build the hardware platform and edit software code as well as the underlying command line based programs. Xilinx Integrated Software Environment (ISE) was also installed on both platforms. It is used by EDK for hardware synthesis.
The EDK bundle contains two main development programs, Xilinx Platform Studio (XPS) and Xilinx Platform Studio Software Development Kit (SDK). These programs are all that is needed to get a MicroBlaze system up and running without custom designed hardware. XPS is used to define the hardware platform and can also be used for software development. SDK is a more recently developed program that is intended to replace XPS for software development. Xilinx recommends using SDK for software development instead of the XPS program. Despite of this, XPS was mostly used for the OSEck development because of better low-level debugger control. GNU Debugger (GDB) comes with EDK and was used as debugging tool from XPS.

Xilinx uses a customized version of the GNU compiler collection (GCC) tools to cross compile for MicroBlaze. These were the only EDK tools used on the Linux machine as only software was developed there. On the Windows machine, all compiler and synthesis tools were invoked via the graphical EDK interface.

Xilinx has built a complete graphical interface in the sense that it is possible to build a simple working processor system without knowing much about the underlying toolchain or writing a single bash command. Although it certainly helps to quickly get a test system up and running, it is important to know what is happening in the background to be able to solve problems that might arise.

8.1.2 Platform Description Files

XPS have a guide for creating a new hardware platform, the Base System Builder. This is a step by step guide that allows the user to quickly put together a system with CPU, memory, peripherals and busses assuming you have a Xilinx board description (XBD) file for your physical board.

What this guide does is to populate a set of text files that describes the system. Two of the files that are unique for the EDK flow are the Microprocessor Hardware Specification (MHS) file and the Microprocessor Software Specification (MSS) file. The MHS-file defines the hardware components in the system, their settings and how they are interconnected. The MSS-file describes if and which operating system and drivers that should be used for different components.

The remainder of this section will focus on the software part of the flow. For more information on the hardware tools and flow consult the Platform Specification Format Reference Manual[29] and the Xilinx Embedded System Tools Reference Manual[27].

Software Specification Files

The software platform is described by the MSS-file. The software platform settings can be changed graphically in XPS and SDK. To allow for this, every library and driver has Microprocessor Library and Driver Definition files represented by the Data Definition File (MDD) and the Data Generation File (TCL). The MDD-file contains settings with constraints for libraries, operating systems and drivers that are graphically visualized in the platform settings dialog. When settings are changed, they are written to the MSS-file.
Library Generation

To build the software libraries for all components in the system the LibGen-tool is invoked. This tool reads the TCL file for every driver and library and first executes a Design Rule Check (DRC) and then a Generate section. The Generate section can do everything needed in order to prepare the driver software for compilation. Typically the MHS and MSS settings is accessed and used to generate source code and customize the drivers for the particular implementation. See fig 8.2. The TCL-file is written in a script language.

The libraries and drivers generated form a platform which several applications can use. This was only a short summary of the EDK library generation. To learn more, consult the report [15].

8.2 Target Boards

Most of the testing and development was done on a Spartan-3E Starter Kit development board from Xilinx. The board has an integrated USB JTAG debugger that was used to download the configuration. A serial connection to the board was used to extract debug information and test results.

Another board, a Virtex 5 development Board ML523 was also used for running some performance tests. This board was debugged using a JTAG Parallel Cable IV debugger but the programming was done via a Compact Flash card reader connected to the board.
8.3 Soft MicroBlaze Platforms

Several platforms have been used to test different memory and peripheral configurations. All these platforms have been created using the Base System Builder guide and then modified to suit different needs.

Figure 8.3. The Spartan-3E development board was used for most of the development.
Chapter 9

Implementation Considerations

This chapter presents some of the issues and solutions that have emerged during the implementation. Kernel requirements on drivers and peripherals are described as well as which of Xilinx’s drivers that can be reused for OSEck. MicroBlaze has a non fixed instruction set which poses some challenges that is also addressed. The chapter ends with a more detailed description of the interrupt driver implementation.

9.1 Previous Platforms

OSEck supports several processors, mostly DSPs as this is the primarily target for the compact kernel. Some supported families are: [14]

- Texas Instruments, C6000
- StarCore SC1000, SC2000
- Freescale MSC81xx, MPC5xx
- Analog Devices, TigerSHARC TSx01S, Blackfin
- LSI Logic, ZSPx00
- Agere, DSP16k
- ARM, 9,11 (beta)

No previous port to a soft processor exists but the larger operating system OSE supports the hard embedded IBM PowerPC 405 in Xilinx Virtex II Pro and Virtex 4 FPGAs. This is a good opportunity for cooperation in between the PowerPC and one or several MicroBlaze processors running the same family of operating system.
Most of the supported processors are DSPs and have parallel architectures capable of executing several instructions at the same time, MicroBlaze has not. MicroBlaze is more like a general purpose processor rather than a DSP. On the other hand, because MicroBlaze can be surrounded with custom DSP hardware, it can have the same type of job that a DSP processor would have.

9.2 Kernel Requirements

This section will present the drivers needed by the kernel and the peripheral requirements that follows.

9.2.1 Drivers

The kernel needs a minimum of two drivers to run, a *kernel timer driver* and an *interrupt driver*. The kernel timer driver’s task is to set up an interrupt routine that calls the kernel function `Tick()` which increases the system clock. The interrupt driver’s task is to provide interrupt controller abstraction for the kernel.

9.2.2 Peripherals

Only one peripheral is absolutely necessary; a timer to generate interrupts for the kernel timer driver. There are two timer peripheral IP-Blocks available that can be used: the *Fixed Interval Timer* (fit_timer) and the *OPB Timer/Counter* (opb_timer). The fit_timer can be set up to interrupt the processor at a frequency determined at hardware design time. The opb_timer is more advanced and must be configured in runtime. The Tick should have a fixed period which makes the fit_timer the best choice because it uses the least FPGA resources.

The timer’s interrupt wire must be connected to MicroBlaze. MicroBlaze does only have one interrupt input and there would be no interrupts left if the timer was connected directly. A processor without interrupts would not be very useful in many applications. It is possible to implement an interrupt driver without any interrupt controller, supporting just one interrupt, but in practice an interrupt peripheral is also needed by the kernel. Here are the peripheral constraints set on OSE_{ck} MicroBlaze platforms:

- One *OPB Interrupt Controller* must be connected to MicroBlaze running OSE_{ck}.
- One *Fixed Interval Timer* must be connected to the *OPB Interrupt Controller* that is set up to generate periodic interrupts for the OS-Tick.

There is the possibility of using an *OPB Timer/Counter* to run the Timeout Server and in turn, let the kernel timer driver use the Timeout Server for periodic Tick generation. If the Timeout Server is used for other purposes anyway, one *Fixed Interval Timer* can be removed. The resource utilization of the *Fixed Interval Timer* would need to be considered.
Timer was measured\(^1\) and it consumed around 1.5% of the total number of LUTs used by the MicroBlaze core. This affects the system size very little so the Fixed Interval Timer was decided to be a mandatory part of the design to make it simpler and not rely on the Timeout Server. The opb_timer uses around 11% of the LUTs used by the MicroBlaze core.

### 9.3 Driver Compatibility and Reuse

Xilinx does deliver their own Board Support Package (BSP) to MicroBlaze and its peripherals. The BSP is automatically generated depending on what the hardware configuration is. The goal with this port has been to use Xilinx’s drivers when possible. The driver interfaces are more likely to stay the same even if the hardware changes. This reduces the maintenance as new versions of peripherals are introduced. Traditionally a BSP is built for a particular platform and that platform never changes. This is not the case for soft processors where the platform may change with the next service pack.

Drivers that are to be delivered with OSE\(_{ck}\) must also be integrated in Xilinx BSP generation tool-chain if they are to be user friendly to configure.

#### 9.3.1 Timer Drivers

Two different timer peripherals are available on MicroBlaze; the Fixed Interval Timer appropriate for the kernel Tick and the OPB Timer that is suitable for the Timeout Server.

**Fixed Interval Timer**

Xilinx does not provide any driver for the Fixed Interval Timer because it does not have any software settings. A new driver for OSE\(_{ck}\) that sets up the interrupt routine needs to be written.

**OPB Timer**

The OPB Timer/Counter is appropriate for the Timeout Server because of its ability to be reprogrammed. The Timeout Server constantly reprograms the timer for the next scheduled interrupt. The resolution can be as high as the system clock. Xilinx provides both high and low level drivers which can be used to interface a Timeout Server driver as long as the interrupt handling functions are not used. A Timeout Server wrapper driver that uses Xilinx driver can be used.

---

\(^1\) A MicroBlaze core with multiplier but without floating point support was used. The size of other peripherals and busses is not part of the measurement. A 50MHz clock was divided down to a 10ms tick. This measurement does not give exact values but gives an idea of the resource proportions.
9.3.2 Interrupt Driver

The Xilinx provided high level interrupt drivers are not compatible with OSEck because of the fixed eight levels of interrupt priorities. Further is the low level (register wrappers) driver not useful because OSEck have the critical interrupt and context switch parts of the kernel written in assembler. User interrupts requires a jump to the interrupt routine without setting up a C context first to find out the interrupting peripheral. User interrupts must be written in assembler. This means that a full custom OSEck interrupt driver must be written.

9.4 Non Fixed Instruction Set

The OSEck kernel is normally delivered as three pre-compiled libraries to protect the code and functionality of the kernel. MicroBlaze’s instruction set is determined by the hardware configuration selected by the user after delivery.

The MicroBlaze GCC cross compiler takes arguments that determines which hardware resources that are available for the program. If the software is compiled for hardware resources that is not available the program will malfunction and probably crash. Experiments show that executing non-existing instructions will not always halt the processor, but instead continue to execute with an erroneous result as if nothing was wrong. This makes it important to verify that the right hardware is used together with the software. One way of doing this in OSEck is to add code that verifies the Processor Version Registers (PVRs) upon executing the operating system initialization routines. This implies that the PVRs themselves are implemented which can be verified in the Machine Status Register.

When the kernel libraries are compiled it is not known what processor configuration that will be used. There are three ways to solve this:

- The libraries are compiled for all hardware configurations.
- Only one library is compiled that uses a minimal configuration.
- The code is written to read the PVRs and select different code dynamically depending on the configuration at hand.

9.4.1 Dynamic Code Selection

Dynamic code selection is not practical to use as it requires several code versions in memory. Normally, embedded systems do not have any more memory than necessary. Combined with the effort needed to identify and rewrite parts of the kernel manually and the performance penalty added, it is not a good way of handling the problem.

9.4.2 Libraries for All Configurations

To compile for all hardware configurations may seem like it would require a large number of different versions because every options doubles the number of versions
needed. To this come the 3 different libraries needed: release, release with hooks and debug. Having the option to include or exclude support for; Multiply, Divide, Floating point, Barrel Shift and Pattern Compare quickly adds up to 96 different configurations. As multiply actually have three options; none, 32 bit or 64 bit it is actually 144 configurations. In reality, there is no need to have all these options because the OSE\textsubscript{ck} kernel does not use any floating point, divide or 64 bit multiplication instructions. This reduces the number of libraries to 24 which includes all combinations of the 32 bit Multiplier, Barrel Shifter and Pattern Compare hardware.

9.4.3 Minimal Configuration

Multiplications are rare in the kernel and mostly used for indexing data structures where it often can be replaced by a few shift instructions. If no multiply hardware is present, multiplication is emulated in software. Kernel performance measurements were done on different hardware configurations. The measurements showed that the impact of removing the Multiplier, Barrel Shift and Pattern Compare hardware did only give a few cycles penalty on some system calls. Based on these measurement, a minimal configuration will do in most applications. That is, never use the hardware even if it is available.

The code size was also measured to find out the size of the extra libraries needed to do multiply, shift and compare in software. The difference in code size for a kernel compiled with this hardware support and without was about 2Kbyte, kernel only. It is 40\% of a minimally sized OSE\textsubscript{ck} kernel\(^3\) (4.6Kbyte). This is a substantial size increase which is not acceptable. The extra libraries are only used by the kernel while the application use hardware instructions. By studying the c-libraries that come with the compiler it was found out that there are several libraries precompiled for different hardware options. When compiling the kernel without most of the hardware support, one c-library is referenced. When the OSE\textsubscript{ck} library is then used in an application with hardware support another library is referenced which forces the linker to include several versions of the c-library.

9.4.4 Selected Solution

I selected the solution to compile the three different OSE\textsubscript{ck} kernel libraries in eight different versions which are all combinations of Multiply, Barrel Shift and Pattern Compare support. Good practice would be to name the libraries in the same way as Xilinx names the c-libraries by adding the extension \_m\_bs\_p for complete hardware support.

The selection of library can be done using the LibGen functionality in the EDK distribution. This greatly simplifies for the user who does not have to care about changing to the right library when modifying the hardware.

\(^2\)Two to the power of five different configurations plus three library versions of each hardware configuration.

\(^3\)A OSE\textsubscript{ck} kernel must contain the object files alloc, arch, arch\_asm, crepool, error, id, init, process, startose and swap. These give a total size of 4.6kbytes. Multiplier, barrel shift and pattern compare included.
9.5 Interrupt Driver

An interrupt driver was written that supports and requires one OPB Interrupt Controller (opb_intc) to be connected to MicroBlaze. The driver could be further developed to support multiple cascaded interrupt controllers if more than 32 interrupt inputs are needed.

The opb_intc controller can have any number of interrupt inputs less than 32. The driver always implements support for 32 inputs because it makes the code easier and more efficient as a result of the 32-bit architecture.

9.5.1 Hardware configurations

The interrupt controller can be configured with different hardware support enabled that simplifies handling the device. Four additional registers can provide support to easily determine if there is a pending interrupt and which has the highest priority. A fast way of setting or clearing one interrupt enable bit can also be supported replacing the read-modify-write sequence. The interrupt driver uses the optional Interrupt Vector Register (IVR) that provides the number of the highest priority interrupt that is pending. The hardware priority scheme is not used but the register is used to provide one of the pending interrupts. This eliminates the task of calculating the position of a “1” in the Interrupt Pending Register.

9.5.2 Priority Levels

The major task of the interrupt driver is to make the eight hard wired priority levels to work as software configurable 32 priority level inputs. This is managed, as has been done in several previous architectures, by changing the interrupt enable bits for the individual interrupt inputs depending on the current interrupt level. When a medium priority interrupt is running, all low priority interrupts are disabled by clearing the interrupt enable bits for those. All interrupt processes with higher priority have their interrupt enabled.

Because the interrupt enable bits has to be changed every time an interrupt process starts and finishes, it must be fast. By using a pre-calculated table with interrupt enable settings for each interrupt input/process for every interrupt level, the change can be done rapidly. Only a simple move instruction is needed to read one row in the table and write it to the interrupt enable register.

Figure 9.1 shows an example of such a table. Three columns are marked in bold. The left one is an example of an interrupt input that has the priority 2. All interrupts that have the same or lower priority are disabled. The middle column shows an interrupt of the highest priority 0, when it executes are all other interrupts disabled. The right column is a disabled or unused interrupt, it is never enabled. The last row (32) is copied into the interrupt enable register when no interrupt processes are running, that is, when the prioritized processes runs. Here are all interrupts that is used enabled.
9.5 Interrupt Driver

<table>
<thead>
<tr>
<th>Int. Priority</th>
<th>input 0 …</th>
<th>input 1 …</th>
<th>input 2 …</th>
<th>input 3 …</th>
<th>input 31 …</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>1 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 1 0 0 0</td>
<td>0 0 0 0 0</td>
<td>1 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>3</td>
<td>0 1 0 0 0</td>
<td>1 0 0 0 0</td>
<td>1 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>28</td>
<td>0 1 0 0 0</td>
<td>1 0 0 1 1</td>
<td>1 1 0 1 0</td>
<td>0 1 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>29</td>
<td>0 1 0 0 0</td>
<td>1 0 0 1 1</td>
<td>1 1 0 1 0</td>
<td>0 1 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>30</td>
<td>1 1 0 0 0</td>
<td>1 0 0 1 1</td>
<td>1 1 0 1 0</td>
<td>0 1 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>31</td>
<td>1 1 1 0 0</td>
<td>1 0 0 1 1</td>
<td>1 1 0 1 0</td>
<td>0 1 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>Prio Proc (32)</td>
<td>1 1 1 1 1</td>
<td>1 0 0 1 1</td>
<td>1 1 0 1 0</td>
<td>0 1 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
</tbody>
</table>

Figure 9.1. Example of a pre-calculated table of interrupt enable settings for different interrupt levels.

9.5.3 Interface

The interrupt driver is expected to provide some functions that is used by the MicroBlaze OSE\textsubscript{ck} kernel. These are:

- **update\_int\_masks.** Used by the kernel to set the priority of a interrupt process/input. It does so by modifying a column in the table showed in figure 9.1.

- **bspint\_init1.** Used to initialize the interrupt controller and internal data structures in the driver. Called on OSE\textsubscript{ck} boot.

- **krn\_int\_get\_vector.** Called after a interrupt have been asserted to find out which vector that triggered the interrupt. Written in assembler as it is called without a C-context.

- **set\_ipl.** Called before and after every interrupt to set the appropriate interrupt priority level. This function is written in assembler and copies one row in the figure 9.1 table into the Interrupt Enable Register.

The interrupt driver also implements a function to clear edge triggered\footnote{Edge triggered interrupts must be cleared in the interrupt controller whereas level triggered interrupts are cleared by signaling the interrupting device to de-assert the interrupt.} interrupts from user applications. (bsp\_int\_clear)

Two interrupt related kernel arrays are defined in the interrupt driver because the kernel does not know the number of interrupts when the kernel libraries are built. They contain tables to match processes and entry points\footnote{Address in memory where a program or routine starts.} against the interrupt vectors.

By using the interface functions it is possible to write a new interrupt driver to another controller in the future without modifying the kernel. Some cycles are lost in interrupt latency because of this flexibility but it is in the magnitude of a few percent.
9.5.4 Instruction Set

Neither the kernel part nor the driver part of the interrupt handling written in assembler use any of the optional hardware resources. A few cycles can be saved by using a barrel shifter when addressing data on word boundaries instead of doing two one step shifts (multiply by 4). The improvement possibility is very small but alternative instructions can be implemented using preprocessor directives. One drawback is that it increases test complexity.

9.5.5 User Interrupts

User interrupts should be invoked as fast as possible as this is their primarily purpose. The most common way in other OSEck targets is to do a direct jump to the user interrupt routine from the interrupt vector that have a user interrupt assigned. The single interrupt input and interrupt vector on MicroBlaze forces another solution. The interrupt controller must first be asked which vector that was asserted. To be able to call the interrupt driver assembler routines, two registers needs to be saved on stack. One to store the return address of the call and one to hold the vector number. The driver must not use any other than these two registers. When the vector is known, a branch to user code is made. When returning from the user interrupt the two registers must be restored from stack. One simple way is to provide a return address for user interrupts that restores the two registers before continuing the normal execution.
Chapter 10

Integration Considerations

This chapter presents some of the issues and solutions that have emerged during the integration. The goal with the integration is to make an intuitive interface for OSE\textsubscript{ck} that conforms as much as possible to the normal design flow in the EDK.

Make sure that you are familiar with Xilinx EDK or have read section 8.1 to be able to understand the remainder of this chapter.

10.1 Why Integrating?

As presented in section 5.3, development tool support and the level of IDE integration are properties considered when choosing an operating system.

Xilinx puts a lot of effort in making a graphical interface on top of the text based configuration files. It is reasonable to believe that they have done customer investigations that show that having a graphical interface is a good thing. Because of this it seems important that OSE\textsubscript{ck} is also part of the interface.

10.2 Functionality to Integrate

This is some of the functionality covered:

- OSE\textsubscript{ck} configuration integrated into the \textit{Software Platform Settings} dialog in XPS/SDK and auto generation of appcon.c.
- Automatic selection of the right OSE\textsubscript{ck} library.
- Hardware OSE\textsubscript{ck} design rule check.
- Automatic configuration of drivers according to hardware design.
- An option to start a new software OSE\textsubscript{ck} project.
10.3 Automatic Library Choice

One of the most important functionalities is to make sure that the right hardware version of the library is used. Normally the user must make sure to include the right library of three possible: release, release with hooks or debug. Now, because the hardware can change it adds up to 24 different versions to choose from. See section 9.4.2.

To ensure that the wrong hardware configuration library is not used it is possible to write a generate section that only copies three libraries, with the right hardware configuration, into the build tree. All 24 libraries are contained in the source folder of the driver and the right ones are copied into the lib directory together with the c-library for easy application access. At the same time they are renamed so the user can use the standard OSE\textsubscript{ck} library names in the Makefile. When the hardware platform is changed are the libraries automatically replaced which ensures that the right ones are used.

Figure 10.1 partly shows the structure of the EDK plugin. Every driver and operating system has the .mdl and .tcl files in a data folder as well as a source folder containing the actual driver. The normal OSE\textsubscript{ck} file structure has been maintained within the src folder. Contained in here are the kernel header files, source and 24 library versions. The MkConfig program is also located here, but not shown in the figure, compiled for Windows, Linux and Solaris.

![Diagram of EDK plugin file structure](image)

**Figure 10.1.** The figure shows parts of the plugin file structure. It contains the 24 kernel libraries among other things.
10.4 Kernel Configuration Integration

OSE\textsubscript{ck} is configured by modifying the configuration file \textit{appcon.con} and then using the MkConfig program to generate the \textit{appcon.c} file that needs to be linked with the kernel. Appcon.c contains parts of the kernel that is configuration dependent.

The MSS-file, containing the software settings, describes a software platform on which several applications can be developed. One MSS-file is created and used by XPS and another independent software platform is described by a MSS-file used by SDK. All applications in XPS share one common software platform and all applications in SDK share one platform. Currently, it is not possible to manage any per application specific settings in the graphical interface.

The OSE\textsubscript{ck} settings in appcon.con are mostly application specific and can be placed in the application platform. It would allow for automatic generation of appcon.c and automatic selection of one of the three kernel library versions. The problem is that this implies that all applications must share the same settings. The settings include such things like which static processes that should be started. That is normally not something that is the same in between all applications. This means that only one application can be used per hardware and software platform if the appcon.con settings are integrated in the MSS-file.

The alternative is to let the user write appcon.con and generate the appcon.c manually by adding the appropriate lines in the Makefile. This is what is normally done for OSE\textsubscript{ck} but it is more complicated than what is normally expected using Xilinx xilkernel\textsuperscript{1} in the EDK environment.

The xilkernel have all its settings in the MSS-file. The difference to OSE\textsubscript{ck} is that the xilkernel does not have as application specific settings. Xilkernel only have the property maximum number of processes while in OSE\textsubscript{ck}, it is also possible to specify actual processes. Two alternatives exist:

- Integrate all settings which gives the possibility to generate the configuration and choose the right version of the OSE\textsubscript{ck} library automatically. Driver initialization can also be added automatically as startup routines in the appcon.con file. This option limits the platform to only host one application in the general case.

- Do not integrate any settings in which case the user has to make an appcon.con file and add invocation of the MkConfig program in the Makefile. This allows having several applications on the same platform.

For the non OSE\textsubscript{ck} experienced user it is probably best to have easy configuration to get started. It is possible to have an option that allows enabling or disabling automatic configuration generation.

\textsuperscript{1}Xilinx small operating system library delivered with EDK
10.5 OSE\textsubscript{ck} Design Rule Check

To ensure that the hardware platform is set up correctly for OSE\textsubscript{ck} it is necessary to do a DRC in the generate section of the TCL-file. The DRC should verify that the mandatory peripherals are connected.

- Verify that an interrupt controller is connected to the processor and that it uses the OSE\textsubscript{ck} interrupt driver.
- Verify that the connected interrupt controller has all required hardware features enabled.
- Verify that a fit\_timer is connected to the interrupt controller.

The verification must be performed by starting at the interrupt pin on the processor and trace the wire to find the interrupt controller. The controller might not be properly connected or there might be several processors and interrupt controllers and the right one must be found. The same applies to the fit\_timer. Some arbitration rule or option must be provided to allow several fit\_timers on the same interrupt controller.

The interrupt controller driver should also trace its wire back to a processor and ensure that OSE\textsubscript{ck} is chosen as the operating system on that processor.

10.6 Driver Configuration

The OSE\textsubscript{ck} BSP is delivered in source code and contains at least the mandatory drivers for interrupt and timer interfacing. The drivers need to be configured to work with all supported hardware configurations. This section presents the parts of the drivers that need configuration. It is the driver’s TCL-file script that has the opportunity to customize the driver before compilation.

10.6.1 Non Fixed Address Space

There is no manual to read where to find peripherals and memory in the address space. That depends on the current hardware platform.

Normally the address map is fixed and specified in the platform documentation. The address map in the soft processor platform is specified in the individual peripheral’s entry in the MHS-file like this:

```
BEGIN peripheral\_name
...
PARAMETER C\_BASEADDR = 0x400c0000
PARAMETER C\_HIGHADDR = 0x400cffff
END
```
The base and the high addresses are used by LibGen to verify that no peripherals have overlapping address space. Most drivers need to know the base address to be able to communicate with the peripheral. The base address can be extracted using the generate section in the TCL-file of the driver to create custom source or include files.

Alternatively can the defines in the xparameters.h file that is generated by the libgen tool be used. They look like this:

```c
#define XPAR_PERIPHERAL_NAME_BASEADDR 0x400c0000
```

Xilinx provides this include file for easy access to common hardware parameters. The driver must use the same name as the peripheral in its defines. This involves using the generate section to customize the name in the driver. It is probably easier to generate the define directly to create a non xparameters.h-dependent implementation.

### 10.6.2 Peripheral Hardware Implementation

It is often possible to customize the hardware configuration of the peripherals which may also affect the drivers. As in the case of the base address discussed in section 10.6.1, these setting needs to be extracted from the MHS-file by using the generate section of the TCL-file.

The generate section of the TCL driver files should collect as much of the needed hardware information as possible. Data that is not possible to collect or calculate must be manually entered by the user in the Software Platform Settings or the MSS-file.

One example is the fit_timer Tick driver. It needs to know the frequency of the timer in order for the driver to know how fast the system clock is. This can be tricky as only the divide factor and the incoming clock wire is known. The wire must be traced and the frequency must be calculated. The hard part is that the wire might be coming from anywhere. Usually it can be traced via some clock module to a chip pin where the clock frequency is specified. But it might not be possible to find out the frequency. Therefore it is easier to let the user specify the incoming frequency in the Software Platform Settings.

### 10.6.3 Fixed Interval Timer Driver Issue

Only peripherals that are connected to a bus are considered part of the Software Platform Settings. This seem natural as a peripherals without bus connectivity, like the Fixed Interval Timer (fit_timer) can not be accessed by the processor. Still it needs a driver in OSEck because the interrupt process that should respond to the timer interrupts must be created. This normally happens in the Kernel Timer Driver’s init function.

The workaround is to put the fit_timer driver as part of the kernel plugin. It is not optimal because it is not easy to switch to another tick timer solution in the future without changing the kernel plugin. However it seems unlikely that the fit_timer peripheral will be removed in the near future as it is so simple.
10.7 OSEck Project

Xilinx has an application wizard that starts automatically when SDK is started. In this wizard it is possible to choose Create a New SDK C Application Project. When the wizard is completed a example main.c-file is created in the new project. This file contains a main-function and some getting-started text.

To quickly get started developing OSEck applications it would be good to add a Create a New OSEck Application Project to the list of options in the wizard. The standard main.c file is not appropriate because no main file is needed when using OSEck and it can therefore confuse the user. Preferably a simple process is generated in one source file and an example appcon.con file is also included.

A OSEck Example Application Project could also be an option where the standard OSEck example application is copied into the new project. Unfortunately the application wizard is currently hard-coded and does not allow any new entries.
Chapter 11

Results and Analysis

This chapter presents the results of OSE\textsubscript{ck}’s performance and functional analysis as well as footprint figures. The performance figures are compared with two other common processors for reference.

11.1 Achievements

- OSE\textsubscript{ck} has been ported to MicroBlaze.
- Functional testing has been performed and passed.
- Performance measurements have been done.
- A plugin for the EDK environment has been made.

11.2 Functional Analysis

All functional tests were executed on the three kernel library versions using hardware divider, barrel shifter and pattern comparator hardware. Both on internal and external memory. Informal functional tests has also been performed using minimal hardware support.

User interrupt tests were not implemented due to lack of time. They require special assembler test routines to be written that generates interrupts while exercising hardware specific functions. All except user interrupt dependent tests passed with the code compiled with O2 optimization.

O3 optimization makes two timing related test cases fail. The compiler rearranges a region where interrupts are enabled over a function call which makes an interrupt routine one OS-Tick late in some rare cases. Most likely this can be solved by placing additional optimization constraints. This happens in kernel libraries that are shared among all architectures and they have been compiled and tested using a considerable amount of compilers before. The scope of this thesis did not allow further studies to isolate the source of the problem.
11.3 Performance Analysis

OSE\textsubscript{ck} performance tests were performed on MicroBlaze to give an estimate on the performance that can be expected while performing various system calls. Performance was measured by counting clock cycles during these calls.

11.3.1 Method

Measurements was done using a free running timer peripheral that counts one step every cycle. The test program reads the value before starting the measurement and reads it again when stopping the measurement. To account for the time taken to read the timer, several measurements were done by only starting and stopping the measurement. The overhead of reading the timer is subtracted from the measured difference. The timer read functions have special tags to prevent the compiler from optimizing away the function calls.

When cache is enabled it is not possible to guarantee exact measurements. By measuring the timer read overhead several times and take an average it is possible to get in-cache numbers. It can be assumed that these functions reside in cache most of the time due to their frequent use.

The test program was compiled with O2 optimization. O3 optimizes over and removes function calls which make the measurements more unreliable.

11.3.2 Measurements

Tests were performed with code contained in both internal BRAM, external cached and external non-cached memory. The two later was done as a reference to see what can be expected from external memory. No cycle count can be guaranteed when using cache. The external memory on the Spartan 3E starter kit used the standard configuration generated by the Base System Builder. 47 different cases was measured. Some common measurements are presented in table 11.1. The cycle count includes the overhead of the function call where applicable.

<table>
<thead>
<tr>
<th>Operation</th>
<th>BRAM</th>
<th>External cached</th>
<th>External non-cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc from pool</td>
<td>106</td>
<td>516</td>
<td>2362</td>
</tr>
<tr>
<td>send without swap</td>
<td>83</td>
<td>503</td>
<td>1934</td>
</tr>
<tr>
<td>receive on existing signal</td>
<td>104</td>
<td>619</td>
<td>2296</td>
</tr>
<tr>
<td>get_ticks</td>
<td>8</td>
<td>69</td>
<td>177</td>
</tr>
<tr>
<td>send - swap - receive</td>
<td>264</td>
<td>1014</td>
<td>6341</td>
</tr>
<tr>
<td>interrupt - small context</td>
<td>163</td>
<td>490</td>
<td>3395</td>
</tr>
<tr>
<td>interrupt - big context</td>
<td>164\textsuperscript{a}</td>
<td>247</td>
<td>3548</td>
</tr>
</tbody>
</table>

\textsuperscript{a}The exact interrupt latency depends on the current running instruction and the processor configuration. See discussion in section 7.1.1.

Table 11.1. Cycle count for common operations in different memory configurations.
It is interesting to note that a small context interrupt takes about as much time as a big context. It would seem natural that a large context takes more time because more needs to be saved. The difference in the number of saved registers is 10, which is completed in an equal amount of additional cycles. This is a very small difference and the extra cycles needed when returning from the wait_fsem system call used in the small context measurement probably uses several cycles.

Table 11.2 shows the proportional increase in cycle time when using external, cached and non-cached, memory. The most interesting figure is the average calculated over all 47 measurements. This gives a more reliable figure when using cache. Many of the system calls are just executed once which gives worse performance than a repetitive application would experience. This can be seen by looking at the cached performance figured for interrupts in table 11.1 or table 11.2. The small context interrupt measurement is executed just before the big context interrupt measurement which shares much code. When the code is already in cache it takes only half the time compared to its first execution.

<table>
<thead>
<tr>
<th>Operation</th>
<th>External cached</th>
<th>External non-cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc from pool</td>
<td>4.9</td>
<td>22</td>
</tr>
<tr>
<td>send without swap</td>
<td>6.1</td>
<td>23</td>
</tr>
<tr>
<td>receive on existing signal</td>
<td>6.0</td>
<td>22</td>
</tr>
<tr>
<td>get_ticks</td>
<td>8.6</td>
<td>22</td>
</tr>
<tr>
<td>send - swap - receive</td>
<td>3.8</td>
<td>24</td>
</tr>
<tr>
<td>interrupt - small context</td>
<td>3.0</td>
<td>21</td>
</tr>
<tr>
<td>interrupt - big context</td>
<td>1.5</td>
<td>22</td>
</tr>
<tr>
<td><strong>Average over all measurements</strong></td>
<td><strong>4.3</strong></td>
<td><strong>23</strong></td>
</tr>
</tbody>
</table>

Table 11.2. Number of times slower than when using internal BRAM.

**Impact of Processor Configuration**

Tests were also performed to measure the impact of different hardware configurations. The hardware Multiplier, Barrel Shifter and Pattern Comparator was tested in different combinations from internal memory. OSE_{ck} does not use division or floating point operations therefore they do not affect the results.

It was found that the Pattern Comparator had no effect on the system calls. Removing the Barrel Shifter had the largest performance impact, 2% slower on average with a median of 1%. This corresponds to one cycle, on average, for many of the calls. When the Multiplier was also removed, the difference in average performance was only about 0.1% worse. Some calls were one or two cycles faster and some were actually slower with the multiplier.
To conclude, the Pattern Comparator and the Multiplier does not give any performance gain for the kernel. Using the Barrel Shifter saves about one cycle per system call. This is only 2% on average which makes OSE$_{ck}$ fairly unaffected by the processor’s configuration. Applications running on OSE$_{ck}$ might be affected much more, especially DSP algorithms.

MicroBlaze Compared to Other Targets

The performance figures were compared to two other targets is order to give an idea of how it matches similar processors. The targets were chosen not to be DSPs. Instead more general purpose targets were chosen, an ARM9 and a PowerPC 555. The performance figures on these processors have been measured on low latency memory and are compared with the figures obtained from the internal memory configuration on MicroBlaze. The numbers are presented in figure 11.1 and the x-axis represents the different measurements marked by dots in the graph. All measurements are sorted by the cycle count on MicroBlaze to make it easier to read. The least time consuming measurements are to the left and the more time consuming ones are to the right. Because MicroBlaze’s figures are sorted it has a smooth curve. This gives the illusion that it is only the reference processors that vary not MicroBlaze.

![Figure 11.1. Cycle count on all measured system call sequences. The measurements are sorted by the cycle count obtained from MicroBlaze.](image)

As can be seen, MicroBlaze is using the most cycles for most of the simple system calls while it is faster than ARM9, on average, on the more time consuming calls. On average over all system calls, ARM9 are nearly 50% faster than MicroBlaze. This might be a result of the more powerful addressing modes found in ARM9.
PowerPC 555 is nearly four times faster on average. This is good to keep in mind when choosing in between using an embedded PowerPC 405 or a MicroBlaze in an FPGA design. The PowerPC has far more performance and can be run at higher frequencies.

Hard processors are easier to clock at higher frequencies which also needs to be considered when comparing performance figures. MicroBlaze has a top clock frequency of 100MHz in the Spartan-3 devices and up to 210MHz in the Virtex-5 devices. This gives an interrupt latency of about 1.64us in Spartan-3 and 0.78us in Virtex-5.

11.4 Footprint Analysis

The size of the implementation was measured to see if it is feasible to run OSE\textsubscript{ck} from FPGA internal memory. The measurements were done on the release library. The debug library is somewhat larger. Only the OSE\textsubscript{ck} features used by the application are linked into the build. This means that the kernel size can vary from 4.6kB in a minimal configuration to around 21kB when using everything available. This includes several system information calls. The Heap and Timeout Server are not included in the figures.

Added to the size of the operating system are the size of the application and the size of the compiler libraries. The application and the operating system share the compiler libraries which make it more complicated to measure the size needed for OSE\textsubscript{ck}. Some libraries used by OSE\textsubscript{ck} are included by default which reduces the size of the application and vice versa. The available hardware also affects the size as this influence which hardware emulation libraries that is needed.

The Spartan 3E evaluation board used for most OSE\textsubscript{ck} development has an FPGA with 360kbit available memory in BRAM blocks. This gives 32kB of available RAM for MicroBlaze. The performance test program, that uses many parts of OSE\textsubscript{ck}, did fit into this memory after removing some parts including the UART drivers. At least 32kB is needed for most applications. In data intensive applications, large memory pools may also be needed. The performance test program uses 8 of the 32kB for memory pools. If performance is not of highest importance, external memory can be used. One possibility is to leave software with real-time requirements in internal memory and place non-critical software in external memory.

The Spartan 3E series is optimized for logic density and have a lower BRAM to Logic Cell ratio. Spartan 3 contains more memory. Table 11.4 contains a list of devices that have the largest BRAM density in each family. The XCE4VFX140 device that has 1242kB of RAM has no problem of hosting several MicroBlazes running OSE\textsubscript{ck} from internal memory if that is needed. The devices listed are also the largest devices in each family. It may not be economical to choose a larger device only because more memory is needed and not more logic. The table also shows the amount of BRAM per logic cells. The more performance tailored Virtex devices have a higher ratio than the Spartan devices.
<table>
<thead>
<tr>
<th>FPGA series</th>
<th>kbit BRAM</th>
<th>BRAM/logic-cells</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartan 3E</td>
<td>648</td>
<td>0,02</td>
<td>XC3S1600E</td>
</tr>
<tr>
<td>Spartan 3</td>
<td>1872</td>
<td>0,025</td>
<td>XC3S5000</td>
</tr>
<tr>
<td>Virtex II Pro</td>
<td>7992</td>
<td>0,08</td>
<td>XCE2VP100</td>
</tr>
<tr>
<td>Virtex 4</td>
<td>9936</td>
<td>0,07</td>
<td>XCE4VFX140</td>
</tr>
<tr>
<td>Virtex 5</td>
<td>8784</td>
<td>0,09</td>
<td>XC5VSX95T</td>
</tr>
</tbody>
</table>

**Table 11.3.** The device with the maximum available BRAM in each FPGA series.
Chapter 12

Conclusions and Further Studies

This chapter concludes the results obtained from the implementation and integration of OSE\textsubscript{ck} on MicroBlaze. A discussion on further possibilities for soft processors and OSE\textsubscript{ck} as well as some possible additional work will end this chapter.

12.1 Integration

This thesis has shown that it is possible to integrate OSE\textsubscript{ck} into the EDK environment. OSE\textsubscript{ck}’s application specific configuration can not be perfectly integrated into the EDK at this moment. One solution to this problem would be if Xilinx provided a way to generate custom entries in the new project wizard which could generate example files and custom compiler settings. As of today, the new project guide is hard-coded. Despite of this, the EDK environment can be used to manage the many versions of OSE\textsubscript{ck}’s libraries that are needed because of non fixed hardware support.

12.2 Performance and Resource Utilization

MicroBlaze is primary a control processor, which is supported by the performance figures. It is not comparable with a hard processor in terms of raw performance and primarily, clock frequency. Top performance is not needed in many applications and here MicroBlaze might be perfect. Calculations that require high performance can be put in hardware instead, making MicroBlaze a control processor, which is what it does best.

The memory requirements of OSE\textsubscript{ck} does allow MicroBlaze to execute from internal memory in some applications. 32kB can be seen as a minimum when using several system calls. If all or some of the code does not have high predictability requirements it can be placed in external memory. The performance figures shows
an average of four time degradation in execution time on cached external memory compared to internal. Real applications can probably execute faster because they experience more cache hits than the test system which runs many of the system calls only once. MicroBlaze provides a good foundation on which to run OSE$_{ck}$.

12.3 Possibilities with OSE$_{ck}$ on a Soft Core

MicroBlaze is well suited as a control processor for application specific hardware. Equipped with OSE$_{ck}$, the hardware can be hidden behind an OSE process which is fed with data and controlled via LINX from any process on any processor. With this approach, modules can be implemented either in software or hardware. If a software process turns out to be too slow, it can be moved to a soft processor which implements the functionality in hardware. Other parts of the system remain unchanged; LINX makes sure that the process is available as if it was executing on the original processor.

12.4 The Future of Soft Processors and FPGAs

As products are developed at higher and higher rate, the need for flexible platforms will certainly increase. The FPGA, used either in its own package or as a part of a SoC, can not only provide custom hardware acceleration and glue logic but also processors if needed.

There is no doubt that technical evolution will continue and lead to larger systems with more transistors contained on the same chip. Soft processors are energy and area inefficient compared to hard processors and they do not have the potential of becoming the main processor in a SoC. Especially in battery powered and cost sensitive applications. But, future chip generations can be very large compared to the size of the processors covered in this thesis. A couple of energy inefficient soft processor might then not be very significant compared to the system as a whole. The flexibility they offer to put software intelligence around any hardware might be more important.

Today, soft processors offer a great opportunity to replace simple FPGA external processors with embedded soft processors. These processors consume expensive logic resources but offers great PCB flexibility and allows for late changes.

One of the key components of future system development is a framework for hardware and software that ties all parts of the system together. Today there exists many different standards that defines how to interconnect hardware IP-blocks and software modules. Only the future will tell if any standard will be dominating and which. Nevertheless it will be important both for operating system vendors and processor vendors to support standards if not large enough to create them.
12.5 Further Studies

LINX has good potential to be used as IPC both outside as well as within the FPGA. There is already room for several MicroBlaze instances together with embedded PowerPCs. There is a good opportunity to create a LINX EDK plugin that allows for easy set up and automation of the LINX configuration. With LINX, several different connection topologies can easily be tested for throughput, latency and other performance figures.
Conclusions and Further Studies
Bibliography


Appendix A

Abbreviations

An alphabetized list of explained abbreviations used in this thesis.

**ALM** Adaptive Logic Module. Logic building block found in Altera’s FPGAs. Consists of LUTs and registers.

**API** Application Programming Interface. Specification of a software interface.

**ASIC** Application Specific Integrated Circuit. Custom designed integrated circuit in its own package.

**BRAM** Block RAM. Xilinx’s name for their SRAM blocks embedded in the FPGA.

**BSP** Board Support Package. Collection of drivers and configurations for a certain board or platform.

**CLB** Configurable Logic Block. Logic building block found in Xilinx’s FPGAs. Consists of two Slices.

**CPU** Central Processing Unit. Synonym for processor.

**DMIPS** Dhrystone MIPS. Dhrystone is a measurement standard that simulates common computer activities to evaluate performance. DMIPS is Dhrystone multiplied by a constant.

**DRC** Design Rule Check. Verification of a design to see that is conforms to applicable design rules.

**DSP** Digital Signal Processor. Processor designed specifically for digital signal processing.

**EDK** Embedded Development Kit. Tool suite containing tools for developing systems with MicroBlaze and PowerPC.

**EEPROM** Electrically Erasable Programmable Read Only Memory. A non volatile memory that can be erased and reprogrammed electrically.
Abbreviations

**FIFO**  First In, First Out. Queue implementation.

**FPGA**  Field-Programmable Gate Array. Large programmable logic circuit.

**FSL**  Fast Simplex Link. Point to point high speed 32-bit bus with support for FIFO queue clock domain bridging. Used by MicroBlaze primarily to connect dedicated hardware.

**GCC**  GNU compiler collection. Set of programming language compilers produced by the GNU Project.

**GDB**  GNU Debugger. Debugger tool.

**HDL**  Hardware Description Language. Language syntax in which to describe hardware logic in code.

**IDE**  Integrated Development Environment. Development environment that helps developing hardware or software.

**IP-Block**  Intellectual Property Block. See IP-Core.

**IPC**  Inter Process Communication. The way processes communicate locally within one processor or remotely. LINX is an IPC-library.

**IP-Core**  Intellectual Property Core. Hardware design protected by copyright laws. Available for free or for a license fee from the creator. Used as a building blocks to create large systems.

**ISE**  Integrated Software Environment. Program used to manage Xilinx hardware design flow.

**JTAG**  Joint Test Action Group. Standard for test access port and boundary-scan architecture. Can be used for debugging.

**LAB**  Logic Array Block. Logic building block found in Altera’s FPGAs. Consists of ten ALMs.

**LMB**  Local Memory Bus. Bus developed by Xilinx used for connecting low latency memory to MicroBlaze.

**LUT**  Look Up Table. Hardware or software component that returns a value from a specified address in a table. Normally implemented in memory. Often used to store pre-calculated values for fast retrieval.

**MDD**  Data Definition File. Xilinx file format that defines the settings for a driver, operating system or library.

**MHS**  Microprocessor Hardware Specification. Xilinx file format that contains the description of how to build the hardware platform.

**MIPS**  Million Instructions Per Second. Measure of processor speed which is not comparable in between different architectures.
**MMU** Memory Management Unit. Peripheral or part of processor that manages memory access right and translation of memory addresses.

**MSS** Microprocessor Software Specification. Xilinx file format that contains the description of how to build the software platform.

**OPB** On-Chip Peripheral Bus. Standard bus among IBM’s CoreConnect bus architectures used by Xilinx.

**OSE** Operating System Enea. The largest one of three operating systems developed by Enea.

**OSE\(_{ck}\)** OSE Compact Kernel. DSP optimized RTOS developed by Enea.

**PCB** Printed Circuit Board. The board which electrically connects all chips and components to each other.

**PCI** Peripheral Component Interconnect. Bus standard.

**PID** Process Id. Used to identify processes.

**PLB** Processor Local Bus. Standard bus among IBM’s CoreConnect bus architectures used by Xilinx.

**PLL** Phase-Locked Loop. Feedback system that is used to synchronize a clock to a reference clock.

**PVR** Processor Version Register. The processor version registers in MicroBlaze are 12, 32 bit, read only registers that contains information on the hardware configuration.

**RAM** Random Access Memory. Volatile electrical memory that have constant access time to all memory addresses.

**RISC** Reduced Instruction Set Computer. The instructions are few and simple to be able to use high clock frequencies.

**RTOS** Real-Time Operating System. Operating system designed to handle real-time tasks in a predictable manner.

**SDK** Xilinx Platform Studio Software Development Kit. Program made to ease software development for MicroBlaze and PowerPC. Eclipse based.

**SoC** System-On-Chip. The concept of integrating processors, memory and application specific hardware on only one chip instead of using several discrete components.

**SOPC** System-on-a-programmable-chip. Altera’s name on a system created with the SOPC-Builder.

**SRAM** Static RAM. Volatile RAM that saves the contents by using a flip-flop for every memory element. Does not need to be refreshed.
**TCL** Data Generation File. Xilinx file format that contains code that does DRCs and generate files needed for a driver, operating system or library.

**TCP** Transmission Control Protocol. Reliable network protocol normally used in combination with IP.

**USB** Universal Serial Bus. Serial bus standard primarily used to connect peripherals to a personal computer.

**XBD** Xilinx board description. File format that describes the FPGA external peripherals. Used by the Base System Builder Guide.

**XCL** Xilinx CacheLink. Bus developed by Xilinx used for connecting cached external memory to MicroBlaze.

**XML** Extensible Markup Language. Language for structuring data using tags.

**XPS** Xilinx Platform Studio. Program to graphically create hardware platforms for MicroBlaze and PowerPC.
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Daniel Staf