Inter-CPU communication platform using embedded Linux for shared memory systems

Utilizing the Nios II CPU and the Avalon Switch Fabric

FREDRIK LINDBERG

Supervisor/Examiner: Ingo Sander
Abstract

Communication between processors is an important aspect of embedded platforms, this project analyzes and improves the synchronization mechanisms of shared memory systems based on the Nios II CPU together with the Avalon Switch Fabric and studies the real-time capabilities of embedded Linux. New hardware synchronization primitives together with software drivers were implemented for efficient inter-CPU communication between embedded Linux and MicroC/OS-II processes executing on different Nios II CPUs. The created hardware primitives allows for asynchronous operation, as oppose to the existing synchronous primitives available for the Nios II CPU. The primitives are performance analyzed with respect to latency both on embedded Linux and on MicroC/OS-II. Further, high-level communication platforms were created based on the primitives and the LINX IPC framework for Linux and subsequently performance analyzed.
## Contents

1 Introduction .................................................. 1

2 Background .................................................. 3
   2.1 Altera .................................................. 3
      2.1.1 Nios II .............................................. 3
      2.1.2 Avalon Switch Fabric ............................... 3
      2.1.3 HAL - Hardware Abstraction Layer ............... 4
   2.2 GNU tool chain and Linux ............................... 4
      2.2.1 µClinux ............................................. 4
   2.3 MicroC/OS-II ............................................ 4
   2.4 LINX .................................................... 5

3 Linux as an embedded operating system .................. 7
   3.1 The Linux kernel and process model.................. 7
   3.2 Real-time Linux ....................................... 9
      3.2.1 Process scheduling .................................. 9
      3.2.2 Preemptiveness in the kernel ..................... 11
      3.2.3 IRQ latency ....................................... 14
   3.3 Minimal Linux system .................................. 15
   3.4 Conclusion ............................................. 16

4 Inter-CPU synchronization primitives .................... 17
   4.1 Existing primitives .................................... 18
   4.2 Avalon Signal core .................................... 19
      4.2.1 Hardware .......................................... 20
      4.2.2 Drivers .......................................... 22
      4.2.3 Summary .......................................... 28
   4.3 Avalon Sleep Mutex .................................... 29
      4.3.1 Hardware .......................................... 29
      4.3.2 IRQ Filter ........................................ 31
      4.3.3 Drivers .......................................... 34
      4.3.4 Summary .......................................... 38

5 High-level communication platform ....................... 39
5.1 LINX ......................................................... 39
   5.1.1 LINX Avalon Connection Manager ................. 40
   5.1.2 Performance ......................................... 42
5.2 Software Message Queue ................................ 43
   5.2.1 Performance ......................................... 45

6 Conclusion .................................................. 47
   6.1 Linux as a real-time system ............................... 47
   6.2 Synchronization primitives for Nios II and Avalon Switch Fabric .... 48
      6.2.1 Avalon Signal Core ................................. 48
      6.2.2 Avalon Sleep Mutex ................................. 49
   6.3 High-level communication platform ...................... 49

Bibliography .................................................. 51
Chapter 1

Introduction

Multi-core CPUs and parallel systems are becoming common both in desktop computers and in the embedded and real-time systems sphere. In contrast to the common desktop market and high-performance cluster computing where SMP (Symmetric Multiprocessing) and homogeneous system and cores constitutes the norm, embedded platforms are often heterogeneous in nature and much more diverse. When working with such heterogeneous systems where different CPUs controls different areas, communication between nodes becomes a key issue as tasks will most likely need to communicate with each other to cooperatively solve a specific function.

Such system could have nodes connected in numerous different way. For example, common configurations and layouts for distributed systems where nodes/CPUs are separated by some distance is to use a protocols such as the CAN-bus, Ethernet or even radio links. A second case is when nodes/devices are clustered together within the same hardware platform, such nodes can utilize a serial protocol such as SPI or I²C for communication of data. A third system type is where all CPUs are integrated into the same chip, for example using an FPGA. Nodes in such system could be completely disjoint without communication capabilities but could also be connected to each other through a common bus connected to for example a shared memory through which they would be able to exchange data, this is the kind of system that have been the sole focus during the project.

The aim of this project has been to foremost investigate the embedded and real-time properties of Linux to evaluate how suitable it would be as a hard real-time system. Secondly, a hardware and software platform have been designed and implemented that allows for efficient communication between processes in different operating systems running on different CPUs in a heterogeneous shared memory multi-core system. The hardware platform have been constructed using a FPGA device from Altera and the Nios II soft-core CPU together with the Avalon Switch Fabric for CPU, memory and peripheral interconnection.

The primary operating system focus have been on embedded Linux and particular the distribution µClinux. Linux was chosen because it is already in use by
several embedded devices, it is capable of running on a range of different architectures and provides a huge set of already existing drivers and applications. As a secondary operating system the real-time system MicroC/OS-II and the Altera HAL framework have been used.

During the project a set of hardware peripherals for the Avalon bus have been implemented for the purpose of allowing more efficient synchronization and communication between Nios II CPUs residing within the same FPGA. Corresponding software drivers and test applications for these hardware devices have been written both for Linux and MicroC/OS-II. The communication platform LINX have also been adapted to the Avalon bus utilizing the created synchronization primitives.
Chapter 2

Background

2.1 Altera

The Altera Corporation manufactures different types of PLDs (Programmable Logic Devices), particularly a set of FPGAs (Field-Programmable Gate Arrays) which are devices that contain programmable logic that allows them to be programmed into performing different kinds of tasks. For example, a FPGA device can be programmed with a so-called soft-core CPU which makes the FPGA device resemble a traditional CPU. In addition to the hardware platform, Altera also provides a set of software tools for system design and development. In particular the "Quartus II" suite which is a tool for analysis and synthesis of HDL designs and the "SOPC Builder" which is a graphical application that automatically generates HDL designs for a complete system that incorporate both CPU, memory and peripherals.

2.1.1 Nios II

The soft-core CPU currently provided by Altera is called Nios II and is a 32-bit RISC-style CPU. The nature of a soft-core CPU allows the system designer to customize many aspects of the CPU and its peripherals including the CPU-peripheral interconnection logic, cache sizes and even custom instructions [2]. Since the Nios II CPU is implemented fully in programmable logic, multiple CPU cores can be placed on a single FPGA thus forming a parallel platform. These CPU cores are independent of each other and there is no need for them to be equal in size or performance.

2.1.2 Avalon Switch Fabric

Peripherals in a Nios II system are not connected to the CPU through a traditional bus, but through what is called the Avalon Switch Fabric which is created as a part of the system generation process performed by the "SOPC builder" tool. The master/slave relation is similar to a bus where a master is able to initiate a bus transaction while slaves only can respond to requests initiated by a master or request...
CHAPTER 2. BACKGROUND

attention by the means of an IRQ line connected to a master. The main difference to a traditional bus is that each master (for example CPU) will be given a dedicated path to each slave (peripheral) it is connected to. When multiple masters are connected to single slave an arbiter is automatically inserted between the slave and the masters and access will be granted in a round-robin manner [1] [4].

2.1.3 HAL - Hardware Abstraction Layer

Altera also provides a software programming library called the Nios II HAL (Hardware Abstraction Layer). The HAL is a single-threaded environment that provides software drivers for the many different hardware peripherals provided by Altera for use together with the Nios II CPU and the Avalon Switch Fabric.

2.2 GNU tool chain and Linux

The Nios II CPU, despite its soft-core nature, acts like a normal CPU and has a RISC style instruction set. A back-end for the Nios II instruction set exists for the GNU Compiler Collection (more commonly known as GCC). This allows normal C code to be compiled into a binary form that can execute natively on a Nios II CPU.

Linux is an advanced, general purpose, kernel with its roots outside the real-time and embedded area. It has however gained popularity as an embedded platform because of the amount of available hardware drivers and its customizability and configurability. The Linux kernel have been ported to the Nios II CPU and incorporated into the Linux distribution µClinux.

2.2.1 µClinux

Operating systems based on the Linux kernel are usually referred to as ”distributions”, they bundle the Linux kernel with the necessary libraries and tools to form a complete system. Once such distribution is the µClinux (Micro Controller Linux) distribution which has its primary focus on small and embedded systems.

A part from the Linux kernel the distribution features a small C library called uClibc aimed at embedded system and also bundled with large set of standard Linux utilities and tools that have been optimized and reduced in size. The distribution contains a build infrastructure for easy cross-compiling for several embedded platforms an CPU architectures.

2.3 MicroC/OS-II

MicroC/OS-II (or µC/OS-II) is a preemptible priority based real time operating system with a small footprint [16] designed for hard real time environments. It ships with a validation suite to prove that it is suitable for safety critical system such as aviation and medical products.
2.4. LINX

2.4 LINX

LINX is an IPC framework for distributed systems created by Enea Software AB. It employs a direct message passing interface were tasks are able to send messages directly to each other without knowledge of the underlying transportation medium [9]. LINX runs primarily on OSE (real-time operating system from Enea) but have been ported to Linux.
Chapter 3

Linux as an embedded operating system

Linux is the kernel part of GNU/Linux based operating system and whilst not a
direct descendant from the original AT&T UNIX it is a clone of the UNIX-like
MINIX operating system [21]. It was originally written for the Intel x86 platform
but is now available for a multitude of completely different architectures.

GNU/Linux is a UNIX-like operating system and is mostly ¹ POSIX compat-
ible. POSIX (Portable Operating System Interface for Unix) is a standard pub-
lished by the IEEE (Institute of Electrical and Electronics Engineers) organization
defines an API (Application Programming Interface), shell and utilities common for
a UNIX-like operating system. The POSIX standard makes it possible to write an
application for one UNIX-flavor and with no or little effort compile and run it on a
different UNIX-flavor.

Another aspect of a (somewhat) normal Linux system, which can be attributed
to its original purpose, is that it requires ² a file system which holds the compiled
code of at least one program (which when loaded and executed becomes a process).
This is fundamentally different from smaller real-time operating systems such as
MicroC/OS-II where one statically compile all tasks and operating system code
into one single binary.

3.1 The Linux kernel and process model

As briefly described earlier, the process (or task) model of Linux is fundamentally
different from smaller real-time operating systems such MicroC/OS-II. A GNU/Linux
based operating system has two distinct operating modes, "kernel space” and "user
space” (or "user land”).

The Linux kernel is a monolithic kernel, but with support for dynamic loadable
device drivers and modules. The kernel and its loadable modules constitutes the

¹The POSIX standard continues to evolve, thus the part implemented varies from time to time.
   It is possible to detect individual features at compile time.
²This is strictly not true. As the source code is available it could be modified to not require a
   file system.
"kernel space" and runs only in the "supervisor-" or "privileged-mode" \(^3\) of the CPU and executes all of the code related to the operating system such as process scheduling and device drivers. Normal processes runs in the un-privileged "user mode" of the CPU and with a, if supported, virtual address space. If no MMU (Memory Management Unit) is available, at least a MPU (Memory Protection Unit) is required to run user space processes in a completely un-privileged mode. If none is present all code will execute in "supervisor-mode"\(^[2]\). The Nios II port of Linux is capable of running without both MMU and MPU, thus all code executes with in the Nios II "supervisor-mode" with full access to the hardware. The distinction between the Linux kernel space and Linux user space still exists even though both are executing within the same privilege mode of the CPU.

The Linux process model is dynamic in nature. "User space" programs are stored in compiled form on a file system, the Linux kernel loads the binary code from file system and creates a running process. The file system does not necessarily need to be on a hard disk or even flash memory, instead RAM based file systems are common in smaller systems.

A part from the user space processes the kernel executes several background tasks commonly referred to as "kernel threads", these run within the context of the kernel.

\[\text{Kernel space}\]

\[\text{IRQ handlers}\]

\[\text{User space}\]

\[P_1\]

\[P_2\]

\[\cdots\]

\[P_{N-1}\]

\[P_N\]

\[K_1\]

\[\cdots\]

\[K_N\]

\textbf{Figure 3.1: User space processes, kernel processes and IRQ handlers}

Since only the Linux kernel has full access to hardware a user processes requiring access to a peripheral is required to enter the kernel through what is known as a "system call". A system call is issued from a user space process by trapping the CPU with a software interrupt. This allows the operating system to take control from the user space process, an identifier written to a register or to the stack allows the operating system to distinguish which system call the user space process is interested in \([2, \text{Chapter 3}]\). On a system with different operating modes a system call implicitly requires a switch from "user mode" to "supervisor mode", such a switch

\(^3\)Also called Ring 0 on some architectures
3.2. REAL-TIME LINUX

can be costly and could have a negative effect on the overall system performance if executed very often.

Boot process

When the Linux kernel boots it starts of in "supervisor mode" and initialize the system and device drivers. When this is complete it launches the first "user mode" process, this process is normally called init and resides somewhere on the file system (the kernel will look at a set of common locations) [7].

In a common Linux distribution this process continues to initialize the system but from "user space" and will also launch login prompt or an interactive shell. An init process is required by default and the Linux kernel will halt execution if there are no init program present on the file system.

3.2 Real-time Linux

It should be stressed that Linux was designed as a general purpose operating system and not as a real-time operating system. General purpose operating systems (such as for servers and desktop computing) are typically tuned to improve the average response time of a system without any guarantees of when a task might complete. Compared to a (hard) real-time operating system where tasks are tuned to complete within a given time frame, and everything else is considered a fatal error.

3.2.1 Process scheduling

The default process scheduler in Linux is priority driven with dynamic process priorities. The scheduler makes no difference between user processes and kernel threads. A process is dynamically classified as "I/O-bound" if it spends most of its time blocked waiting for example user input or disk I/O. If the process spends most of its time running it will be classified as a "CPU-bound" process. I/O-bound processes are given a high priority and CPU-bound processes are penalized with a decreased priority [5]. While such a scheduler might be favored in a general purpose system to improve average response time for I/O-bound processes it is not very suitable for a real-time system where a CPU-bound process might have much higher priority.

Since Linux is mostly POSIX compliant it also implements most of the POSIX 1003.1 real-time extensions [12] which among other things defines fixed priority scheduling for processes.

A scheduling policy dictates how a group of processes with this policy should be scheduled. The POSIX real-time extension defines the following scheduling policies.

SCHED_FIFO

FIFO scheduling policy.
CHAPTER 3. LINUX AS AN EMBEDDED OPERATING SYSTEM

SCHED_RR
Round-robin scheduling policy.

SCHED_SPORADIC
Sporadic server scheduling policy. Not implemented in Linux.

SCHED_OTHER
Vendor specific policy. This is the default scheduler in Linux.

Linux defines a few other policies as well (SCHED_{BATCH, IDLE}), but they are not related to real-time scheduling and thus not covered here.

Each policy has a set of priority levels (PRI_MAX to PRI_MIN) where each level conceptually is a queue of runnable processes with that particular priority and scheduled according to the given policy. The number of priority levels available are vendor specific, but the POSIX standard specifies that at least 32 different levels should exist [12]. Linux 2.6.2x defines 99 priority levels for the policies SCHED_{RR,FIFO} [15].

![Figure 3.2: Processes grouped at priority level within a scheduling policy](image)

SCHED_FIFO - First in-First out Policy, a runnable process with this policy is put at the end of the queue of waiting processes for the selected priority level. Once a process have been selected to run it will run until it blocks (for example I/O request), gets preempted by a higher priority process or gives up the CPU voluntary [15].

SCHED_RR - Round-Robin Policy, works similar to SCHED_FIFO except that processes in this policy with the same priority level are executed in a round-robin manner with a fixed time slice [15].

The Linux scheduler always consults the scheduling classes SCHED_FIFO and SCHED_RR before the default policy SCHED_OTHER. Thus, true fixed priority based scheduling in Linux is possible using any of these two scheduling policies.

Scheduler API
It is possible for a program to alter both its priority and its scheduling policy (with respect to the permissions the program is executing with, normally super-
3.2. REAL-TIME LINUX

user privileges are required) using the API defined by the POSIX specification.
A program can alter its policy and priority with the commands listed below. "pid" refers to a unique process identifier, "policy" is one of the defined policies and "param" is a struct that contains a field that specifies the priority level.

```
int sched_setscheduler(int pid, int policy,
                     const struct sched_param *param);
int sched_getscheduler(pid_t pid);
```

```
struct sched_param {
    ...
    int sched_priority;
    ...
};
```

3.2.2 Preemptiveness in the kernel

Linux gained a preemptive kernel with the release of Linux 2.5 [13], prior to this no kernel thread or user process running in the kernel context (during a system call) could not be preempted at all, except if the CPU was yielded voluntary. The change that allowed preemption of kernel threads introduced a locking mechanism called "spin locks" to protect critical sections\(^4\), this locking mechanism also disables interrupts.

The CPU scheduler runs with the help of a hardware timer interrupt, to see why it is required to disable interrupts and not enough to only rely on the lock, consider what if interrupts were not disabled. The timer interrupt driving the scheduler might then fire in the middle of a critical section and the scheduler might select another process to run and erroneously preemption the process currently executing a critical section. This could lead to a dead lock within the kernel if another kernel thread is waiting for the same lock.

While this change improved the average case, it is still not good enough for real-time systems as disabling interrupts for arbitrary long times becomes a problem as it could result in a priority inversion scenario where a high priority processes might be blocked by a lower priority process that is running within the context of the kernel with such a lock held.

Spin locks

"Spin locks" is a lightweight synchronization primitive that is widely used inside the Linux kernel. They were originally used for SMP (Symmetric Multiprocessing) within the kernel but were extended to also disable interrupts when Linux gained kernel preemption support. Previously there were no need to disable interrupts

\(^4\)A critical section refers to a code sequence that needs to execute from top to bottom in an atomic manner.

with spin locks as everything inside a kernel context executed in a non-preemptive manner [8, Chapter 5].

In essence “spin locks” are an atomic test-and-set value. To obtain a lock the thread enters a busy loop trying write a value to variable, this continues until the operation succeeds and the lock is acquired. This furthers highlight how imperative it is to disable preemption while holding a spin lock. If preemption were allowed, two threads contending for the same lock would easily create a dead lock. Consider the figure 3.3, it illustrates a possible dead lock scenario, process $P_1$ has lower priority than process $P_2$ and are executing with a fixed priority driven scheduler.

![Figure 3.3: Deadlock situation - spin lock with preemption enabled](image)

Both processes are executing within the context of the kernel and are contending for the same spin lock. $P_1$ grabs the lock first because $P_2$ is blocked waiting for some event. $P_2$ becomes ready while $P_1$ still is executing within the critical section, once the scheduler runs it will preempt $P_1$ and allow $P_2$ to run because $P_2$ has a higher priority than $P_1$. But $P_1$ still holds the lock and $P_2$ is stuck in an endless loop, spinning in an attempt go get the lock. Since $P_2$ is of higher priority and is executing an endless loop, $P_1$ will never get the chance to run and release the lock, thus the processes are deadlocked!

**Sleepable spin locks**

As shown “spin locks” needs to execute with interrupts disabled and are widely used within the Linux kernel to protect both small and large critical sections. A negative side effect of disabling preemption while holding a spin lock is that a thread with higher priority not waiting for this particular lock is prevented from running even though it is in a runnable state. This is not much of a problem for a general purpose operating system but becomes more of an issue for real time systems as all these sections interfere with the scheduling of processes and might infer long latencies for processes of higher priority.

Over several years the Linux developer Ingo Molnar have been working on what has been known as ”the RT patch” which address several shortcomings of the kernel concerning Linux as a real time system. One aspect of this patch is that it replaces spin locks with a sleepable mutex, instead of spinning to acquire the lock the calling process will be set to a non-runnable state and blocked from running [19] [18].

This enables critical sections to run with preemption enabled. To see why this is safe, consider the figure 3.3, when $P_2$ wakes up and tries to acquire the spin lock
it will find that it is already locked by $P_1$ and will be put back to sleep again and $P_1$ can finish execution and release the lock. $P_1$ will still be preemted for a slight period of time because of its lower priority, but this is normal lock contention.

![Figure 3.4: Spin lock vs sleepable spin locks. Priority $P_1 > P_2$](image)

Figure 3.4 illustrate the non-preemptable spin locks versus the preemptable (sleepable) spin locks and shows how a lower priority processes holding a spin lock might be preempted by another processes not contending for the same lock. $P_1$ is the high priority process and $P_2$ is a processes with lower priority.

A similar third-party initiative by Wind River System called Wind River Real-Time Core for Linux [22] or RT-Linux address the same issue by replacing the scheduler and allowing the kernel to be preemptive.

Neither the preemptive part of the "RT-patch" nor the RT-Linux scheduler have been ported to the Linux/Nios II architecture and therefore were unable for use during the course of this project.

### Priority Inversion

Preemptive threads naturally leads to priority inversion for kernel threads that shares a lock. The bounded priority inversion between two threads can be limited by keeping the section protected by the lock short and it is also possible to measure the worst case time a lower process might block a higher process.

A issue that is much more worst is unbounded priority inversion involving multiple threads at at different priority possibly resulting in a high priority process becoming blocked for an un-deterministic period of time.

The approach taken by the "RT Patch" is priority inheritance [19]. Priority inheritance results in that if a high priority thread $T_1$ blocks on a lock owned by a low priority thread $T_2$, the thread $T_2$ inherits the priority of $T_1$ thus preventing it from becoming blocked by some other thread with priority $T_1 > P > T_2$. 
3.2.3 IRQ latency

Another important aspect is the performance of interrupt handling within the operating system and particular how much overhead that is required before an interrupt handler begins to execute.

Measurement

Because of its flexibility, the Nios II platform provides an unique opportunity to measure the IRQ latency of an operating system, in a non-intrusive way, from the point when a peripheral asserts its IRQ line until the registered IRQ handler begins to execute.

A special hardware CPU peripheral component was created for the Avalon Switch Fabric that worked together with the "Altera Avalon Performance Counter" [3, Chapter 29] to achieve this. The created hardware block starts the performance counter and simultaneously asserts its IRQ line to interrupt the processor. An IRQ handler registered in the operating system stops the performance counter and resets the hardware block. The number of cycles elapsed from assertion until serviced by the operating system can then be read from the performance counter.

Figure 3.5 shows the IRQ latency in Linux/Nios II versus a basic interrupt handler using the Altera HAL (Hardware Abstraction Layer) executed on the same hardware at a CPU frequency of 50 MHz. The graphs clearly shows that, not only is the Linux interrupt handling slower, but also much more scattered than the HAL based interrupt handler. It should be noted that the HAL based IRQ handler executed dedicated on the CPU without an underlying operating system and represent more or less minimal amount of work required to begin servicing an interrupt on the Nios II platform.

![Figure 3.5: Nios II IRQ latency](image-url)

---

5The performance counter measures the amount of clock cycles elapsed since started.
3.3 Minimal Linux system

Since all source code of a Linux system is available it can be tailored to any need, it is therefore difficult to provide a hard figure for a absolute minimal Linux system.

To establish a RAM usage reference point for a Linux system the Linux distribution µClinux for the Nios II architecture were compiled with different settings and features turned off. No source code of the Linux kernel was modified by hand and only options accessible from the standard kernel configuration interface were modified.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>ELF</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>User space</td>
<td></td>
</tr>
<tr>
<td>SLOB allocator</td>
<td>Dummy init</td>
<td>312K</td>
</tr>
<tr>
<td>printk (4K buf)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nios II JTAG uart</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLAB allocator</td>
<td>Dummy init</td>
<td>316K</td>
</tr>
<tr>
<td>printk (4K buf)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nios II JTAGuart</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLUB allocator</td>
<td>Dummy init</td>
<td>316K</td>
</tr>
<tr>
<td>printk (4K buf)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nios II JTAGuart</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLOB allocator</td>
<td>Dummy init</td>
<td>496K</td>
</tr>
<tr>
<td>printk (4K buf)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nios II JTAGuart</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCP/IP (IPv4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLOB allocator</td>
<td>Dummy init</td>
<td>508K</td>
</tr>
<tr>
<td>printk (4K buf)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nios II JTAGuart</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCP/IP (IPv4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMC 91Cx NIC support</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLOB allocator</td>
<td>Dummy init</td>
<td>552K</td>
</tr>
<tr>
<td>printk (4K buf)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nios II JTAGuart</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCP/IP (IPv4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINX core (default settings)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>µClinux/Nios II default</td>
<td>µClinux default</td>
<td>1368k</td>
</tr>
<tr>
<td>Stratix II dev. board</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The "Kernel space" part of the "Configuration" column lists which options that were enabled, other options were disabled in the configuration system. All different configurations were compiled with the option "Configure for small systems" set.
to true, this explicitly disables several options that commonly aren’t needed for
embedded systems. The "User space" part specifies what the RAM based file system
consisted of. "Dummy init’ is a replacement for the standard init program that
consists of a infinite loop.

The ELF column refers to the size of the ELF file on disk. The Linux build
process compress text sections in the ELF file to achieve a smaller file, these sections
are then uncompressed at boot time.

Further, the Linux kernel relies heavily on dynamic memory allocation, the
RAM value specified in the boot column refers to the value at boot time. Device
drivers and other sub-systems such as TCP/IP may dynamically allocate memory
for buffers and data structures during runtime, therefore additional RAM will most
likely be required for a real-life application.

SLAB, SLOB (Simple Allocator) and SLUB (Unqueued Allocator) refers to dif-
ferent dynamic memory allocators inside the kernel. They have different charac-
teristics and performance and require different amount of book keeping memory.
Which one to select depends heavily on the end application.

Unfortunately the option "optimize for size" in the GCC compiler for Nios II
generated bad code that prevented the Linux kernel from booting properly. All
figures above were thus compiled without this setting, with this fixed even smaller
RAM usage should be possible.

3.4 Conclusion

Without considering the real time aspect, because of its flexibility Linux is suitable
as an embedded system and is in fact widely popular as exactly this.

Linux is with out doubt more heavy weight than small real time systems such as
MicroC/OS-II. The big win for Linux is the amount of available drivers, sub-system
and add-ons that allows developers to reuse code and shorten development time of
complex systems.

Linux is definitely suitable for soft real-time conditions and with extremely care-
ful auditing of possible code paths and the use of all available real time tweaks, such
as the the complete "RT-patch", it is possible to use Linux in a hard real time en-
vironment.
Chapter 4

Inter-CPU synchronization primitives

Most work were concentrated into allowing processes/tasks running within different operating systems on different Nios II CPU connected by the Altera Avalon Switch Fabric. Whenever multiple cores or processes communicate with each other some form of synchronization primitives are needed to prevent race condition and keep shared data structures from becoming corrupted.

To ease synchronization, single CPUs usually provide some form of basic hardware primitive. For example, one is the "test-and-set" operation [20, Chapter 7] which in one operation atomically \(^1\) tests whether a register is 0 and if it is, sets it to a given value. Other common hardware primitives are "atomic-exchange" [11, Chapter 4] in which a value of a register is atomically exchanged with another. The operation "fetch-and-increment" is similar to "test-and-set" but loads a value from memory and increments it in one operation [11, Chapter 4]. These primitives can then be used by software to create more elaborate primitives.

One of the more common primitives is the "semaphore" which acts like an atomic counter, the semaphore is taken by incrementing the variable and released by decrementing it. This is a very crude primitive which can be acquired several times and with no notion of ownership (any process or thread can release the semaphore). A slightly more sophisticated primitive is the "mutex" (or binary semaphore), as the name suggest it is used for strict mutual exclusion and is a counting semaphore restricted to two values, 0 or 1 (unlocked or locked). A mutex also has a notion of ownership this means that a process or threads owns the mutex once it has been locked, this prevents other processes from unlocking it.

The Avalon Switch Fabric and the Nios II CPU essentially provides multiple, separate, CPUs connected to each other via a shared bus. This is radically different from synchronization between processes running on the same CPU for many reasons. Foremost, any hardware assisted synchronization primitive provided by the CPU and/or the operating system running on the CPU is useless as there is no way for another CPU to even read no less manipulate the internal registers of another CPU. Second, the Avalon Switch Fabric does not provide any explicit connection between

---
\(^1\)An atomic operation performs two or more operations in sequence without interference.
CHAPTER 4. INTER-CPU SYNCHRONIZATION PRIMITIVES

multiple CPUs on the same bus and CPUs are essentially unaware of each other.

4.1 Existing primitives

Among the several IP add-ons for the Nios II and the Avalon bus there are a few blocks that are designed for multi-processor synchronization over the Avalon Switch Fabric. The software support for these are provided by the single-threaded Altera HAL (Hardware Abstraction Layer) library.

Mutex core

The Altera Avalon Mutex core is a hardware peripheral that can be used to ensure mutual exclusion to resources shared between different CPUs connected to the same Avalon bus. The mutex consists of two 32-bit registers one is a status register and the other is used for the actual locking mechanism. The 32-bit lock register is divided into two logical 16-bit values.

The top-most 16-bits are the owner field and the lower 16-bits constitutes an arbitrary value field. The lock register is essentially an atomic "test-and-set" register, to lock the mutex a CPU combines its unique owner id with a non-zero arbitrary value to form a unique 32-bit value. This value is then written to the lock register, if the value of the register was 0 it is replaced with the written value, otherwise nothing happens. The CPU then simply reads back the value of the lock register to identify if it owns the mutex or not. To unlock the mutex zero is written to the value part of lock register by the owner [3, Chapter 26].

The problem with this method is that a CPU wanting to acquire the mutex must enter a busy loop attempting to write its unique lock value to the register until it succeeds. This might not be a problem in a smaller single-threaded system but becomes a severe problem for operating systems that are multi-tasking and multi-threaded in nature such as Linux.

While the system will continue to operate, the processes waiting for the mutex will consume all available CPU time thus impacting the performance and response time of the whole system. It becomes even worse if the waiting process is a high priority processes in a real time priority scheduled system as this will prevent all processes with a lower priority to run until the process has acquired the mutex. This also makes the Altera Mutex unsuitable for event notification and signalling between CPUs.

Mailbox core

The Altera Avalon Mailbox core is a hardware peripheral that is designed to send and receive messages between multiple Nios II processors in a Avalon bus based system. Multiple mailboxes can exists in a system and each of them have a dedicated memory area of configurable size. The mailbox memory is divided into slots based on the system bus width (usually 32-bit). The software keeps track of a read


4.2 AVALON SIGNAL CORE

pointer and write pointer which makes it possible to post multiple messages into a mailbox without waiting for the receiver to read a message before a new message can be posted [3, Chapter 27]. To protect the internal data structure and read/write pointers the mailbox is protected by two Altera Avalon mutex cores, one for writing and one for reading. This makes the mailbox core inherit the same shortcomings as the mutex core.

Posting a message to a mailbox only yields a problem if there are multiple writers on the mailbox, as a writer then might be blocked in a busy loop on the write mutex waiting for a second writer that is currently holding the write lock. Similarly, reading messages from the mailbox yields the same problem for multiple readers as with multiple writers. Additionally, as there is no asynchronous notification mechanism, a reader waiting for a new message on an initially empty mailbox needs to enter yet another busy loop to detect when a new message has been received. This could be solved by a software timer interrupt but with the problem that a too fine grained interrupt interval essentially becomes a busy loop and a too coarse interrupt interval adds a large latency before a new message is detected.

This results in the same problem as the mutex core on a multi-tasking operating system. A high priority process waiting for a new message to be received on a mailbox will block all processes with lower priority from running.

Because of the shortcomings the existing primitives suffer from a set of new hardware primitives for the Avalon bus and the Nios II CPU were designed.

4.2 Avalon Signal core

The foremost obvious shortcoming as evident from both the Avalon Mutex core and the Avalon Mailbox core is the ability for one CPU to asynchronously notify another CPU of some event without involving some form of busy wait or polling method.

In an Avalon system the only external notification mechanism available to an Avalon master, such as a Nios II CPU, are the normal IRQ-lines. The IRQ-lines are unfortunately a quite scarce and inflexible resource and there is no standard way for two masters to interrupt each other which is required if two CPUs are to be notify each other of events. To overcome this obstacle an Avalon slave was created that provides both an external trigger mechanism and a set of virtual IRQ lines using only one real IRQ line at the master.

Figure 4.1 depict a Avalon system with two Nios II CPUs connected to two signal core slave devices (other devices has been left out for clarity) and shows the intended usage for this core. The core has been designed so that each CPU has its own signal core with the IRQ line connected only to it self. The data bus is connected to all CPUs or to those CPUs interested in posting signals/events to the CPU to which the signal core in question is connected to.

19
4.2.1 Hardware

A signal is defined as one (1) bit, thus with 32-bit registers each signal core provides 32 virtual signals per IRQ line. Signals are maskable, i.e. individual signals can be enabled and disabled at will from software and only enabled signals have the possibility to generate a real IRQ. The hardware exposed by the signal core consists of three 32-bit memory-mapped registers that are used for controlling the virtual signals. The register layout is described in table 4.1, offsets are in 32-bit quantities.

Table 4.1: Signal core register definition

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Name</th>
<th>Register Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SIG_MASK</td>
<td>Bit mask of enabled signals</td>
</tr>
<tr>
<td>1</td>
<td>SIG_PEND</td>
<td>Bit mask of pending signals</td>
</tr>
<tr>
<td>2</td>
<td>SIG_ASSERT</td>
<td>Signal source register</td>
</tr>
</tbody>
</table>

**SIG_MASK** contains a 32-bit sized bit mask of which signals the CPU is interested in receiving. A signal not enabled here will not be possible to receive. A logic one (1) is considered a signal to be enabled and a logic zero (0) means that the signal is disabled. Only the CPU considered to be the receiver of the signals from this signal core should write to this register.

**SIG_PEND** is a 32-bit bit mask that contains the set of pending signals intended for the CPU which the signal block is assigned to. The IRQ line connected to CPU owner of the signal core is asserted whenever this register has a non-zero value.

The owner is able to acknowledge and clear pending signals by writing a logic one (1) to the bit position representing the pending signal. This allows the interrupt handler running on the receiving CPU to simply read the value of this register and writing back the read value to clear the pending signals. This effectively eliminates a possible race condition where signals could be missed if new signals were received at the same time this register was cleared.
4.2. AVALON SIGNAL CORE

SIG_ASSERT is a virtual 32-bit register that is used for CPUs to post signals to the signal core. Writing a set of signals to this registers will cause the bits representing these signals to be set in the register SIG_PEND with respect to the configured bit mask configured by the receiving CPU in SIG_MASK. Signals are set in a non-destructive way to avoid losing signals that not yet have been acknowledged by the receiving CPU. The pseudo code below depict how SIG_PEND is assigned with respect to the value written (WRITE_VALUE) to the SIG_ASSERT register and the configured SIG_MASK.

\[
\text{SIG_PEND} = \text{SIG_PEND} \text{ OR } (\text{WRITE_VALUE} \text{ AND SIG_MASK})
\]

Reading the SIG_ASSERT register has no effect and does always return zero.

All register operations inside the hardware block does only require one cycle with additional delay incurred by the Avalon Switch Fabric round-robin arbiter if multiple masters are present. This means that the worst case latency required by the hardware block to read/write a register is equal to the time it takes to access the bus which scales linear with the number of masters connected to the slave device. Manipulating the signal mask and posting signals can be done with a single register operation and thus have a worst case of \(1 \times m\) cycles where \(m\) is the number of masters, this is assuming fair round-robin with 1 transfer share per master.

Receiving a signal requires requires two register operations, one to read the the set of pending signals and one to acknowledge the pending signals. Further, the total latency does also include the number of cycles required for the initial signal post operation, this leads to a hardware only latency of \(3 \times m\) cycles.

In addition to the delay incurred by the Avalon bus, the total latency involves the time required to prepare a memory transfer (devices connected to the Avalon Switch Fabric are memory mapped).

\[
c_{\text{write}} = c_{\text{mem\_write}} + c_{\text{bus}} \times (m - 1) \quad (4.1)
\]

\[
c_{\text{read}} = c_{\text{mem\_read}} + c_{\text{bus}} \times (m - 1) \quad (4.2)
\]

\(c_{\text{write}}\) and \(c_{\text{read}}\) represent the number of cycles for a register operation consisting of the time to issue a write/read memory operation which is equal to the number of cycles required for the instructions \texttt{stwio} (store 32-bit word to memory or I/O peripheral) and \texttt{ldwio} (load 32-bit word from memory or I/O peripheral) [2]. The product \(c_{\text{bus}} \times (m - 1)\) is the worst case additional delay a master might need to wait before it is able to issue its pending memory transfer, \(c_{\text{bus}}\) is the number of cycles or transfer shares assigned to each master in the round-robin arbiter and \(m\) is the number of masters connected to the slave in question.

This makes it possible to express the worst case number of cycles for the operations \textit{post} and \textit{pend} as stated in 4.3 and 4.4. This does only account for the
hardware access time, further software processing required to actually deliver the signal to a process is not included.

\[ c_{\text{pend}} = 2 \times c_{\text{write}} + c_{\text{read}} + c_{\text{irq}} \]  \hspace{1cm} (4.3)

\( c_{\text{irq}} \) in 4.3 represent the IRQ latency which is a combination of a small hardware fraction of 1 cycle and a much larger software fraction that depends on the operating system or software processing the interrupt.

\[ c_{\text{post}} = c_{\text{write}} \]  \hspace{1cm} (4.4)

### 4.2.2 Drivers

The software driver for the signal core needs to be able to perform the following tasks

1. Post new signals to a signal core.
2. Allow manipulation of the signal mask.
3. Wait for one or more signals to be received. The process waiting for a signal must be put to sleep (set to a non-runnable state) by the operating system and woken up when a signal is received. This is required because a process waiting should not use any CPU time during waiting, thus allowing other processes to run even if itself is a high priority process.

Drivers were created for both Linux/Nios II and MicroC/OS-II.

#### Linux

Since Linux has a clear distinction between the kernel and user space applications a kernel driver is required to allow user processes to effectively use the signal core. Support from the kernel is required even when running Linux/Nios II without an MMU/MPU to avoid race conditions between multiple processes accessing the same signal core, but most importantly, only the kernel can (safely) manipulate the state of a process and set it in a blocked state to prevent it from being scheduled to run.

As shown before (section 3.1), for a process running in user space to be able to execute a kernel function it needs to enter the kernel context by using a system call. One solution for user space processes to interact with the signal core hardware is to create a new system call dedicated to it. A system call is a unique number and stored in a table for each architecture For Nios II this table can be found in the file `arch/nios2/kernel/syscalltable.S` in the Linux source tree which is a file written in Nios II assembly language. Adding a new system call involves adding a new call to this file and writing the corresponding function directly inside the kernel. This option is very inflexible as it requires heavy modification of a third-party code base which makes maintenance harder and might also interferes with updates to the Linux source tree.
4.2. AVALON SIGNAL CORE

For these reasons it was decided against a new system call. Instead, a better approach would be to re-use some of the myriad of already existing system calls.

One fundamental part of the design of UNIX-like system is that "everything is a file" [17, Chapter 3]. This can be translated to that (almost) everything ranging from keyboard input to network connections and even mass storage devices (hard drives, USB drives, etc) can, from the application programmers point of view, be treated as a stream of bytes. This also means that (with some exception for Internet Protocol based communication) the file system is used as a global name space with entry points for different devices. Device entry points are usually clustered in the /dev directory but can in theory exist anywhere in the file system. For example, text output to the console is done through the file /dev/stdout and similarly keyboard input can be read from /dev/stdin.

The impact of this is that all I/O to devices and files are abstracted and regardless of what underlying device or file one is manipulating it can be done through the same set of system calls. Kernel drivers associated with the file system entry point are then invoked by these system calls and performs the real device request. Additionally, devices are separated into "character devices" which are byte based devices and "block devices" which are used for devices that work on a block level such as hard drives.

The basic system calls for manipulating character I/O devices are listed below (several other system calls exists for manipulating file attributes, but they are not relevant in this context and thus not covered here). There is also an overlap of the system calls used with character devices and block devices.

- **open** takes a file path as argument and is used to open a device or file. A unique handle called "file descriptor" is returned if the call was successful.
- **close** closes a file descriptor and frees resources.
- **read** reads one or more bytes from a file descriptor.
- **write** writes one or more bytes to a file descriptor.
- **ioctl** performs an arbitrary control request on a file descriptor. Can also be used to transfer small amount of data between user space and kernel space.

A character device meets the requirements for a driver to the signal core. The open and close system calls provides an entry point to the kernel and the ioctl system call allows arbitrary requests to be submitted to a device. This does not require any modification of the Linux source tree, is very flexible and can be installed during run time.

---

2 The number of system calls are somewhat static but new calls are added when required. A current list of existing system calls for an architecture can be found in the file arch/<arch>/kernel/syscalltable.S in the Linux source code tree.

3 A UNIX file system is modeled as one single tree where / is the root, further disk drives or partitions are mounted as leaf nodes (directories) inside the tree.
For these reasons the Linux driver for the signal core was created as a character I/O device and implemented as a loadable kernel module similarly to any other device driver. A special property of character and block devices is what is called "major numbers" which is a file system property. These numbers allow a file to identify an associated character (or block) driver for a particular file entry. For example, if the major number $N$ was assigned to the device driver $X$ and a file system entry point /dev/Y was created with the major number $N$ the kernel is able to map any system calls performed on /dev/Y to the driver $X$.

Since the Avalon bus does not feature any automatic discovery feature of devices connected to the bus all configuration of devices have to be performed manually. Linux kernel modules allow parameters to be passed during load time, this is used to configure a device. Device configuration parameter syntax for the signal driver were defined as follows:

```plaintext
devs=<dev>[:<dev>]
dev=<[irq,]base_address>
```

The following example will load two signal cores with the given base addresses, the first device is configured with an IRQ (value 5) which makes it able to receive signals while the second only can be used to post signals. `insmod` is a Linux utility that dynamically loads a kernel module into the memory space of the kernel.

```plaintext
insmod avalon_signal.ko devs=5,0x04223880:0x04223888
```

To create the actual file system entry point the utility (or system call by the same name) `mknod` is used. The major number can be obtained from the file /proc/devices after the corresponding module has been loaded. The example shows the major number 254 and the minor numbers 0 and 1, the minor number is used by the driver to identify individual devices of the same type, i.e. different signal cores in this case. The signal core device driver assigns minor numbers in sequence based on the arguments provided to the configuration parameter.

```plaintext
mknod /dev/avalon_signal0 c 254 0
mknod /dev/avalon_signal0 c 254 1
```

The requests required for the signal core are small and does only transfer 32-bits of data at the time. The read/write system calls are are a bit heavy weight for this and does not really fit the device request model either. Therefore, besides the entry points `open` and `close`, only the `ioctl` system call is required. The `ioctl` system call takes three arguments

```plaintext
int ioctl(int fd, int request, ...);
```

The first argument $(fd)$ is the unique file descriptor handle obtained from a successful `open` call, the second $(request)$ is an arbitrary device dependant request...
4.2. AVALON SIGNAL CORE

code and the third is an untyped pointer to a memory location\(^4\). Four request codes were defined for the signal core device drivers, they are listed below:

**AVALON\_SIGNAL\_IOC\_IOCSMASK**
Set the signal mask. The third argument is a pointer to an unsigned 32-bit integer that contains the new signal mask.

**AVALON\_SIGNAL\_IOCG\_MASK**
Get the configured signal mask from a device. The third argument is a pointer to an unsigned 32-bit integer, the integer it points to will be set to the currently configured signal mask.

**AVALON\_SIGNAL\_IOC\_PEND**
Wait for a signal to be received, the calling process is suspended until a signal is received. Does not require a third argument, the received signals are returned by the ioctl function call.

**AVALON\_SIGNAL\_IOC\_POST**
Post a signal to a signal core. The third argument is a pointer to an unsigned 32-bit integer containing the new signals that are to be posted to the signal device.

These ioctl requests together with the system calls open and close are enough to operate the signal core from a Linux user space program.

Figure 4.2 shows a flow graph of how two Nios II CPUs both running Linux interacts with each other using the signal core hardware block and the corresponding Linux driver. Some details of the call path related to return values have been abbreviated for clarity. \(CPU_0\) is pending on a signal, \(CPU_1\) is posting a signal. It is assumed that a proper signal mask already has configured on \(CPU_0\). There is no

---

\(^4\)The ioctl system call appeared in Version 7 AT&T UNIX which dates back to before void pointers were legal in C.
CHAPTER 4. INTER-CPU SYNCHRONIZATION PRIMITIVES

timing dependency between the CPUs and they can each respectively call post and pend whenever they want to.

Processes are blocked from being scheduled and thus prevented from running by using the Linux kernel functions `wait_event_interruptible()` and `wake_up_interruptible()`. The former blocks a process from running until it is woken up by a call to the latter function.

Once CPU₀ enters `wait_event_interruptible()` it becomes blocked from running by the kernel (represented by a dashed line in figure 4.2) and will only resume running if an interrupt is received from the signal core which in turn wakes the suspended task by a call to `wake_up_interruptible()`.

The post by CPU₁ does not never block, thus the ioctl system call returns directly once the new signals have been written to the SIG_ASSERT register.

The signal core hardware block will proceed by generating an interrupt to CPU₀ which will end up executing the IRQ handler installed by the signal core device driver. This handler subsequently acknowledge the pending signals and wakes up any processes sleeping in `wait_event_interruptible()`. Any newly awaken processes now returns from their initial ioctl call and resumes running their respective program in user space.

Figure 4.3 shows the total of cycles required for a signal post from one CPU to resume a suspended process on a second CPU, with both CPUs running Linux. This includes the total call overhead in Linux and is equal to the call path in figure 4.2.

![Figure 4.3: Total signaling latency between two CPUs running Linux](image)

Examples
The following examples shows how a signal core can be manipulated from a program in Linux.

---

5 It will also resume running if a Linux software signal is received, in that case the ioctl system call will be re-started by the C library.
4.2. AVALON SIGNAL CORE

Posting a signal

```c
def = open("/dev/avalon_signal0", O_RDWR);
signals = 1 | 2; /* Post signal 1 and 2 */
error = ioctl(fd, AVALON_SIGNAL_IOCPOST, &signals);
```

Configuring a signal mask

```c
def = open("/dev/avalon_signal0", O_RDWR);
mask = 2;
error = ioctl(fd, AVALON_SIGNAL_IOCSMASK, &mask);
```

Wait for a signal to be received

```c
def = open("/dev/avalon_signal0", O_RDWR);
signals = ioctl(fd, AVALON_SIGNAL_IOCPEND);
```

**MicroC/OS-II**

MicroC/OS-II is a very small operating system designed for real time system and is very different from Linux. The operating system and tasks are compiled into a single binary and there is no distinction between what is kernel space and what is user space. The operating system and tasks all run in "supervisor mode" without an MMU and therefore it is possible to access the hardware devices from anywhere. This naturally leads to a smaller driver.

The approach taken with MicroC/OS-II is to use the existing task synchronization primitives provided by MicroC/OS-II to avoid modification of the operating system source code. One of the primitives provided are "event flags" [14], they allow a task to sleep waiting for some event signalled from either a task or an ISR.

Event flags are accessed through the MicroC/OS-II library functions `OSFlagPost` and `OSFlagPend` [14]. A call to `OSFlagPend` will suspend the calling task until a matching event is posted by some other task using `OSFlagPost`. This is provides the functionality needed for the signal core to work properly, thus the MicroC/OS-II driver is designed using this synchronization mechanism.

The signal core API for the MicroC/OS-II consists of the following functions

**avalon_signal_open**
- Takes a base address and an IRQ as arguments, installs an IRQ handler and performs some initialization of data structures.

**avalon_signal_close**
- De-registers the IRQ handler to prevent further use.

**avalon_signal_pend**
- The task calling this function will be blocked by `OSFlagPend` until a signal is received through the IRQ handler. The received signals will be returned to the caller by the function call.
avalon\_signal\_post

Allows a task to post a new signal to the signal core, takes a 32-bit integer with a bit mask of the signals to be posted.

A call flow similar to the Linux case, but for MicroC/OS-II can be seen in figure 4.4.
The total latency of a signal posted from one CPU until received by the second CPU can be seen in figure 4.5.

Figure 4.4: Signal core call flow between two Nios II CPUs running MicroC/OS-II

Figure 4.5: Total signaling latency between two CPUs running MicroC/OS-II

4.2.3 Summary

The Avalon Signal core provides a lightweight and fast asynchronous notification mechanism that can be used for event signalling between processes running on different operating systems.

The main limitation is the lack of an overall broadcast mechanism, this can however be accomplished by connecting multiple CPUs to the same signal block. All CPUs would then need to receive the same signals and it would not be possible
for individual CPUs to mask different signals in hardware, all masking would have to be done in software.

### 4.3 Avalon Sleep Mutex

Whilst the Avalon Signal Core is more than enough for several synchronization tasks it does not natively provide mutual exclusion for data structures and storage areas residing in memory shared between multiple CPUs.

As described in section 4.1 a mutex hardware block exists for this very purpose but its design requires it to be used with busy loops. By combining the mutual exclusion technique from the original mutex core and the use or IRQ lines, very much like how the signal core uses IRQs for asynchronous notification, a new IP block were created that allows CPUs to pend on a mutex (i.e. waiting for a mutex to become free) without the need to do so in a busy loop. Instead the waiting CPU can block its local process that is waiting for the mutex, thus allowing other local processes to execute, i.e. processes waiting are sleeping on the mutex, hence the name. When the mutex eventually is assigned to the waiting CPU an IRQ line is asserted that allows the CPU to unblock the sleeping process and continue executing.

#### 4.3.1 Hardware

The Avalon Sleep Mutex uses the same definition of a mutex as the Altera Mutex Core (section 4.1), the mutex register is based around an atomic "test-and-set" operation but extended and now contains a queue in hardware of CPUs that are waiting for the mutex. Whenever the mutex is released by its current owner the queue is automatically consulted and the first CPU in the queue is assigned as the new owner of the mutex and an IRQ is generated to notify the CPUs that a new owner has been assigned.

![Internal queue of CPUs waiting for the mutex](image)

Figure 4.6: Internal queue of CPUs waiting for the mutex

By default the queue runs in a FIFO mode, thus the CPU that have been waiting the longest is assigned the mutex. Any new CPUs attempting to acquire the lock will be put at the end of the queue, whenever the current owner releases the mutex the CPU at the top of the queue will be assigned the new owner of the mutex. A simple experimental priority queue mode were also designed that allows mutex owner assignment based on arbitrary priorities. There is however no safe-guards
built into this queue mode and improper use of this mode might therefore cause severe starvation if a high priority CPU access the sleep mutex often.

The sleep mutex core contains three 32-bit registers, they are listed in table 4.2, the offset column is in 32-bit quantities.

Table 4.2: Sleep mutex core register definition

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Name</th>
<th>Register Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SMUTEX_OWNER</td>
<td>Current mutex owner</td>
</tr>
<tr>
<td>1</td>
<td>SMUTEX_PEND</td>
<td>Virtual registers, writing to this register puts the CPU in queue for a lock.</td>
</tr>
<tr>
<td>2</td>
<td>SMUTEX_STAT</td>
<td>Status register</td>
</tr>
</tbody>
</table>

**SMUTEX_OWNER** contains a globally unique 32-bit integer that identifies the current mutex owner. Similar to the Altera Mutex Core (section 4.1), the register is divided into two halves. The upper 16-bits contains a number that uniquely identifies the owner globally in the system, this is referred to as the owner field. The lower 16-bits contains an arbitrary non-zero number selected by the owner. The mutex is considered to be free when the lower 16-bit value field is zero. An owner with a mutex locked (i.e. the owner field match the process selected owner value) can at any point change the value field as long as it is non-zero.

**SMUTEX_PEND** is a virtual register that has the same layout as the SMUTEX_OWNER register. To lock the mutex a process writes its unique 32-bit identifier (owner field combined with a value field) to this register. This will put the process/CPU in queue for a mutex lock, if the mutex is free it will be directly assigned to the owner. Reading this register has no effect and does always return zero.

**SMUTEX_STAT** is a status register that can be used to query the status of the sleep mutex core.

- **Bit 0**: Reset bit. High after system reset, can be cleared by writing a 1 to its bit position.
- **Bit 1**: Mutex event. Set high when a new mutex owner has been assigned. This bit is cleared by writing a 1 to the bit position. Clearing this bit does also acknowledge the IRQ.
- **Bit 2**: Queue full. Only a limited number of processes/CPUs can be in queue for the mutex at any given point. This bit is set high to indicate when the queue is full.

The sleep mutex operates on 32-bit registers only and is therefore able to perform all its register operations in one cycle similar to the signal core (section 4.2.1).
This makes hardware operations perform in $1 \times m$ cycles assuming a fair round-robin arbiter and 1 transfer shares per master, plus the additional delays incurred by preparing a memory transfer (see equation 4.1 and 4.2).

A major problem with the sleep mutex core is that when the mutex hardware asserts the IRQ line it generates an IRQ on all CPUs, even if the CPU has a process waiting for the mutex or not. This can be somewhat mitigated in the software driver by only enabling the IRQ handler when at minimum one process is waiting for the mutex and disabling it otherwise. This generates a lot of useless interrupts and interferes with the CPUs normal work. A second but slightly more subtle issue is that the IRQ needs to be acknowledged by before the IRQ line is pulled low. If the new owner never acknowledge the IRQ the line will be held high causing all CPUs to enter the interrupt handler over and over again and essentially deadlocking the whole system.

### 4.3.2 IRQ Filter

To work around the issues of the IRQ line being asserted on all CPU whenever the mutex changed owner a small hardware block were created that screens the mutex owner value before allowing the IRQ line to be asserted on a CPU.

This "IRQ filter" is implemented as a dual Avalon master and slave device (the device acts both as a bus master and a bus slave at the same time). The slave side is connected to the final receiver which usually is a Nios II CPU and the master side is connected to the slave device for which IRQ it is suppose to filter, which in this case is the sleep mutex.

Figure 4.8: IRQ and I/O flow between a CPU and slave device connected through an IRQ filter.

As shown in figure 4.8 the task of the filter is to intercept the IRQ generated by the connected slave device and only let it through to the CPU if some condition is
The only visible side effect is that the CPU now receives the interrupt from the IRQ filter device instead of from the original slave device.

The IRQ filter is designed as a generic, re-programmable device and exposes five different 32-bit registers, four of these registers need to be properly configured before the IRQ filter can be used. The registers are listed in table 4.3, offset is in 32-bit quantities.

Table 4.3: IRQ filter register definition

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Name</th>
<th>Register Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IRQF_STATUS</td>
<td>Status register.</td>
</tr>
<tr>
<td>1</td>
<td>IRQF_TADDR</td>
<td>Absolute address of register at slave device.</td>
</tr>
<tr>
<td>2</td>
<td>IRQF_TVALUE</td>
<td>Target value of the slave device register.</td>
</tr>
<tr>
<td>3</td>
<td>IRQF_IADDR</td>
<td>The absolute address of the slaves IRQ acknowledge register.</td>
</tr>
<tr>
<td>4</td>
<td>IRQF_IVALUE</td>
<td>The value to write to IRQF_IADDR to acknowledge IRQ at the slave.</td>
</tr>
</tbody>
</table>

When an IRQ is generated by the slave device connected to the filter it is asserted only at the filter and not at the CPU. The IRQ filter then proceeds by reading a register at a pre-configured address at the slave device and compares the read value with a value already configured into the filter. Only if these values match the IRQ filter generates an IRQ to the CPU, it will also acknowledge the IRQ at the slave device on behalf of the CPU (figure 4.9).

Figure 4.9: Flow chart of the IRQ filter internals

Figure 4.10 shows how a sleep mutex device is connected to two different CPUs each with an IRQ filter device between. If a CPU wants to acquire a mutex lock it configures its IRQ filter with the address of the mutex owner register and its unique owner value, it then proceeds by putting itself in the mutex owner queue. When a
new owner is assigned to the mutex an interrupt is generated to all IRQ filters, each of them will read the mutex owner value and compare it with the pre-programmed value and interrupt their corresponding CPU only if there is a match. The net result is that only the CPU which is the new owner of the mutex will receive an interrupt.

![Figure 4.10: IRQ flow between a sleep mutex core and two CPUs with an IRQ filter](image)

The use of an IRQ filter does incur an additional latency before the IRQ is asserted at the CPU. This additional delay is however significantly smaller than the IRQ hit required for each CPU as shown by the IRQ measurements in section 3.2.3. Assuming fair round-robin on the Avalon Switch Fabric, the maximum number of cycles the IRQ signal might be delayed by hardware is given by 4.5 where \( m \) is equal to the number of masters connected to the sleep mutex device, this includes both CPUs and IRQ filters.

\[
c_{irq} = 1 + c_{bus} \times m
\]  
(4.5)

The signal diagram in figure 4.11 shows how the IRQ filter interacts with the a sleep mutex. When \( smutex_{irq} \) is asserted (signalling owner change of the mutex) a maximum of 2 cycles might pass until \( filter_{irq} \) is raised to interrupt the CPU. Almost 1 cycle before the original IRQ is processed, then 1 cycle to read the register at the sleep mutex, it is this cycle that might be delayed with up to \( c_{bus} \times (m - 1) \) cycles because of the Avalon Switch Fabric round-robin arbitration. \( filter_{read} \) represents the reading of the sleep mutex register by the filter and \( filter_{write} \) represents the IRQ acknowledgement to the sleep mutex by the filter.

![Figure 4.11: Signal diagram of the IRQ filter interacting with a Sleep mutex device](image)
4.3.3 Drivers

Software drivers for the sleep mutex and IRQ filter core must be able to perform the following tasks.

1. A CPU should be able to lock the mutex in both a blocking and a non-blocking way.
2. Unlock the mutex.
3. Check if is the current owner of the mutex.

Linux

The approach for the Linux driver for the sleep mutex is similar to the one taken by the driver for the signal core. The device is exposed as a "character device" and the API is using the POSIX standard I/O routines, see section 4.2.2 for details on these kind of devices.

Just like the signal core driver, this too is implemented as a loadable kernel module. Because of the absence of a slave device discovery system, device base address and IRQ numbers are passed through module parameters during load time. Depending on how parameters are passed an IRQ filter can be enabled for a specific sleep mutex device. Device configuration parameter syntax as defined for the Sleep Mutex driver:

```
devs=<dev>[:<dev>]
devel=irq,sleep_mutex_base[,irq_filter_base]
```

The following is an example of a sleep mutex device with and without an IRQ filter, the only visible change is a different IRQ number and an additional base address referring to the IRQ filter device. The first example is without an IRQ filter while the latter is with an IRQ filter.

```
insmod avalon_smutex.ko devs=5,0x12345
insmod avalon_smutex.ko devs=4,0x12345,0x54321
```

File system nodes for the device major and minor number must be created as well using the `mknod` utility (see section 4.2.2).

As the driver is implemented as a character device the `ioctl` system call is used as a device interface. The list below describes the set of `ioctl` commands were defined for the sleep mutex device. There is no difference in the API depending on if a IRQ filter is used or not, this is handled transparently by the device driver.

**AVALON_SMUTEX_IOCLOCK**

Attempt to lock the mutex in a "blocking" way. This is equivalent of writing to the virtual SMUTEX_PEND register (section 4.3.1) and suspending the calling process.
### 4.3. AVALON SLEEP MUTEX

**AVALON_SMUTEX_IOCTRYLOCK**

Used to try locking the mutex in a non-blocking way, this is equivalent of how the original Altera Avalon Mutex Core works. This will attempt to write the mutex lock value directly to the owner register, it will only succeed if the value field of the register is zero (i.e. the mutex was unlocked).

**AVALON_SMUTEX_IOCUNLOCK**

Unlocks a previously locked mutex by writing a zero to the value field. The original owner value must be passed along so that ownership can be verified by the hardware.

**AVALON_SMUTEX_IOCMINE**

Compares the current mutex owner (SMUTEX_OWNER register) with the value passed to the ioctl call. Returns true/false depending on if the values match.

All ioctl commands defined for the sleep mutex takes the following structure as argument. The unique 32-bit mutex lock value has been divided into two 16-bit members of a c struct data type for easier data manipulation.

```c
struct avalon_smutex_ioctl {
    uint16_t owner; /* Arbitrary globally unique value */
    uint16_t value; /* Lock value */
} __attribute__((packed));
```

![Figure 4.12: Flow of two CPUs interacting through a sleep mutex device](image)

The key performance aspect under Linux is the locking latency - i.e. the time it takes for an application to resume running after waiting for a lock. It is of course difficult to measure the total locking time as it widely depends on the application that is utilizing the mutex and involves longest blocking time a process might hold the mutex and how many different CPUs that are contending for this particular mutex, and how often.

Figure 4.13 shows the locking latency of an sleep mutex under Linux that is idle and only in use by one CPU. The plot shows the number of cycles it takes for
CHAPTER 4. INTER-CPU SYNCHRONIZATION PRIMITIVES

Figure 4.13: Sleep Mutex lock latency - Linux

an Linux application to acquire the lock until it returns to user mode and is able
resume executing. This involves entering the kernel, acquiring the mutex through
hardware and returning to user mode, although it does not include blocking time
as the mutex always is free. At 50MHz the average locking time as shown by 4.13
is 0.1387 ms.

Figure 4.14: Sleep Mutex lock from blocked state - Linux

The measurement shown by 4.13 is a highly optimistic and not a realistic view.
The figure 4.14 instead shows the number of cycles it takes for one CPU to wake
up from being block on a sleep mutex when the mutex becomes available. This was
measured by acquiring the mutex on CPU1, signal CPU0 to attempt to lock the
mutex, delay executing on CPU1 for an arbitrary amount of time so that CPU0
really blocks and suspends the process. A performance counter was then started on
CPU1 at the same time as the mutex was released, the counter was then stopped
by CPU0 when it subsequently acquired the mutex and returned to the running
process. At 50MHz the average time in milliseconds as shown by 4.14 is equal to
4.3. AVALON SLEEP MUTEX

0.7619 ms.

Examples
The following examples show how the sleep mutex can be used from a Linux process.

Locking a mutex, the ioctl call returns when the mutex is locked

```c
struct avalon_smutex_ioctl smtx = { .owner = 0xff, .value = 1 };  
fd = open("/dev/avalon_smutex0", O_RDWR);  
error = ioctl(fd, AVALON_SMUTEX_IOCLOCK, &smtx);
```

Unlocking a mutex

```c
fd = open("/dev/avalon_signal0", O_RDWR);  
struct avalon_smutex_ioctl smtx = { .owner = 0xff, .value = 0 };  
error = ioctl(fd, AVALON_SMUTEX_IOCUNLOCK, &smtx);
```

MicroC/OS-II
Just as with the Linux driver, the driver for MicroC/OS-II is very similar to the one created for the signal core (section 4.2.2). Blocking tasks from running is accomplished by using `OSFlagPost` and `OSFlagPend` [14]. The important factor is that a MicroC/OS-II task will be blocked from running while waiting for a lock, thus allowing other tasks to execute. The task will resume running when the mutex has been assigned to the CPU.

`avalon_smutex_open`
Initializes data structures and installs an IRQ handler for the device.

`avalon_smutex_close`
Disables the installed IRQ handler and resets data structures.

`avalon_smutex_lock`
Attempts to lock the mutex using the CPU unique lock value. The pending CPU will be put on the hardware queue of pending owners, the calling task will be blocked in `OSFlagPend` until a event is received from the installed IRQ handler signalling that the CPU has acquired the mutex.

`avalon_smutex_trylock`
Attempts to lock the mutex in a non-blocking way by simply attempting to write its lock value to the owner register. This only succeeds if the mutex is free.

`avalon_smutex_unlock`
Unlocks a previously locked mutex.
CHAPTER 4. INTER-CPU SYNCHRONIZATION PRIMITIVES

Figure 4.15: Sleep Mutex lock latency from blocked state - MicroC/OS-II

Figure 4.15 shows the time it takes for a task to resume running from being blocked waiting for a lock. This includes both the hardware and software overhead incurred by the MicroC/OS-II drivers and measures the time from the instant CPU\(_0\) issues a \texttt{avalon\_smutex\_unlock} until CPU\(_1\) resumes running from being blocked by \texttt{avalon\_smutex\_lock}. The measurement was carried out with an IRQ filter installed. The total block time can of course not be measured as it depends on the application in question and how many CPUs/processes that are contending on the lock.

4.3.4 Summary

The Avalon Sleep Mutex demonstrates an improvement over the standard Altera Avalon Mutex when used in multi-process systems such as Linux as it allows processes to block while waiting for a lock.

While the concept itself has proven to work, the implementation requires the IRQ filter for effective operation. This is in itself not a problem, but the manual instantiation of IRQ filters in the Altera system generation tool SOPC builder is slightly cumbersome and could possibly be improved by automatically generating IRQ filters through a script.

A second limitation of the IRQ filter is that it can only be used for one unique owner value at any given time, this effectively limits the one process at the time to use the filter. By implementing a mask register to perform masked comparison several processes within the same CPU could utilize the filter at the same time.
Chapter 5

High-level communication platform

While the primitives covered in chapter 4 alone are enough to let multiple Nios II CPUs communicate efficiently with each other a more high-level platform is often convenient. For this reason two different high-level platforms were adapted and/or created to enable communication between Nios II CPUs connected to the same Avalon bus. Only embedded Linux were used for these implementations.

5.1 LINX

LINX is an IPC service framework developed by ENEA designed for heterogeneous systems. It is using a direct message passing system in which tasks running on different system can send messages directly to each other [9]. All communication is based on messages sent between tasks and there is no difference between tasks running on the same CPU, on another CPU in the same system or on a completely different, remote system.

The LINX protocol stack consists to two distinct layers, the RLNH (Rapid LiNk Handler) layer and the CM (Connection Manager) layer. The RLNH layer consists of two parts, an operating system independent part and an operating system dependant part that handles the interaction with the operating system. This layer provides common IPC functions and mapping between remote and local endpoint names so that tasks are able to address each other using named endpoints [10, Chapter 3].

The Connection Manager is located between the RLNH layer and the actual physical medium used for transport. Depending on the underlying medium the Connection Manager needs to implement means such as, but not limited to, message fragmentation, re-transmission and flow control. The physical medium could be anything that connects two systems, for example Ethernet, TCP/IP, shared memory or some other different type of connection [10, Chapter 4].

In essence, this abstracts different types of connections under a unified API and allows tasks to communicate seamlessly across different systems without knowledge of the underlying medium.
LINX runs primarily on OSE (real-time operating system from Enea) but have been ported to Linux. The Linux specific components are licensed under the GPL license while the remaining parts are licensed under the even more liberal BSD license.

5.1.1 LINX Avalon Connection Manager

A LINX connection manager were implemented for Nios II/Linux and the Avalon Switch Fabric. It utilizes shared memory for transportation and the Avalon Signal core for synchronization and message flow control. The connection manager requires two pairs of shared memory buffers and two pairs of signal cores, one for data to be transmitted and one for received data. The buffers does not need to be of equal size, but the TX/RX pairs of two endpoints should match. Parameters identifying buffers and signal cores are passed to the Avalon Connection Manager by the LINX configuration utilities during the creation of a link between two CPUs. The link should be established as a back-to-back connection, i.e. the transmit side on CPU$_0$ should be the receive side on CPU$_1$.

The signal cores are used as a synchronization mechanism and for event signalling, the following different signals are used.

**SYN, ACK** are used during the three-way handshake required to setup a link.

**PKG** notifies the peer that a new packet is available.

**PKGACK** acknowledge the peer that the last packet was received properly.

**HEARTBEAT** signals are sent periodically to detect failures.

**FIN** sent to a peer to gracefully terminate a connection.

To establish a connection over a created link a three-way handshake (figure 5.2) (similar to the handshake used in TCP) mechanism is used. This allows both sides to initiate a connection without any particular timing dependency.
5.1. LINX

As soon as the Avalon Connection Manager have been created by the RLNH layer it will continuously attempt to send a SYN (synchronize) signal to its peer using the signal core. When a SYN signal is received by the Avalon Connection Manager it will respond with an ACK (acknowledge) signal and also send SYN to its peer if not already done so earlier. When a CPU has sent its SYN it will wait for an ACK, when this signal is received it moves to a connected state, and when both sides have received their respective ACK the connection becomes established.

Figure 5.2: Three-way handshake

LINX messages received from the RLNH layer contains, except the message payload, a message size and additional header information. The total message size might be larger than the available amount of configured shared memory, messages are therefore, if needed, fragmented into "packets" according to the size of the memory buffer. Each packet is before transmission prepended with an additional 4 byte header containing the fragment size. The fragments are successively written to the transmit buffer residing in shared memory, once a fragment has been completely copied into the shared memory, the remote endpoint is notified through a PKG signal using the signal core. Upon reception of this signal the receive logic at the peer copies the message fragment from the buffer and responds with the "packet acknowledge" signal PKGACK. The transmit logic is then free to continue with the next fragment. This is a very simple form of flow control but allows messages to be transmitted as fast as they can be received at the remote peer. The receive logic is also simplified as all fragments are received in-order and can be re-assembled into a message on-the-fly as new packets are received, without the need to buffer individual fragments.

Figure 5.3: Message fragmented into packets

To facilitate rapid detection of a peer failure, a periodical heartbeat signal is sent from both endpoints with a 1 second interval. A peer is considered dead if no heartbeat is received for 2 seconds, in such case the connection is explicitly terminated by the connection manager and the RLNH layer is notified about the disconnection. If no heartbeat would be sent peer failures would still be detect during transmission (no packet acknowledge would be received), but an endpoint only receiving data
would never be able to detect a peer failure in this case. Similarly, transmission might be infrequent and it is often desirable to detect failures as soon as possible.

5.1.2 Performance

The LINX Avalon Connection Manager were tested using a program that measured the total round-trip time for a message between two CPUs.

Figure 5.4 shows the round-trip time for different sized messages exchanged between two CPUs, measured using both real-time priority scheduling (SCHED_RR) and the default scheduling provided by the Linux kernel (SCHED_OTHER), with and without additional system load. SCHED_OTHER makes the test program show erratic behavior when subjected to additional system load, while the priority scheduler shows little difference with additional load. System load was achieved by using an extra process that used up all available CPU time. The reason for the behavior of SCHED_OTHER under load can be attributed to the dynamic priority scheduler that is the default scheduler in Linux. It penalize CPU bound processes (section 3.2.1) which results in that the load program will run at different priorities at different times making the real test program run faster and slower depending on this.

The shared memory buffer where set to 1024 bytes causing fragmentation for messages above this size, this can be seen as spikes in the graph and clearly has a negative impact on performance.

Figure 5.4: LINX Avalon-CM round-trip time with 1024 byte buffer

Figure 5.5 shows the same test but with a 4096 byte shared memory buffer which eliminates the need for fragmentation and yields a linear increase in time with increasing message size. SCHED_OTHER is still erratic under load but SCHED_RR shows very little difference with and without additional system load.

Figure 5.5: LINX Avalon-CM round-trip time with 4096 byte buffer
5.2 SOFTWARE MESSAGE QUEUE

A simple proof-of-concept message queue for Linux were implemented using shared memory, the Avalon Sleep Mutex core and the Signal Core. The memory is divided into slots of fixed size, the slot size is configurable at initialization but depends on the size of the shared memory. Additionally a write and read pointer is kept in shared memory to keep track of the current read and write position. This eliminate the requirement for a message to be read before a new one can be posted, thus multiple messages can be posted in sequence. The Avalon Sleep Mutex is used to protect the integrity of the queue, the Avalon Signal core is used to create a software based ”condition variable” to eliminate the need for polling mechanism by a reader to detect when new messages arrive when the queue is empty.

A ”condition variable”, or ”monitor” which is the original name, is a synchronization scheme that provides mutual exclusion to a procedure associated with the monitor. The condition part of this pattern consists of the two constructs wait and signal where wait suspends the calling process until signal is called by some other process [20].

The practical approach taken here is similar to the one used by POSIX threads [6]. A sleepable mutex provides the mutual exclusion part of the monitor pattern and a process signalling mechanism represents the conditional part. Consequently, the Avalon Sleep Mutex and the Avalon Signal core was used in this implementation to create a inter-CPU condition variable.

A condition variable is initialized with a mutex and a signal core and after that only manipulated using the following operations.

**cond_wait** suspends the calling process and automatically unlocks the mutex associated with the condition variable (the mutex must therefore be locked when calling this function). The process will block until signalled by some other process and will acquire the associated mutex **before** returning to the calling
process. The blocking is accomplished by pending on a signal using the signal core (section 4.2).

**cond_signal** wakes up a process blocked on the condition variable. This call has no effect if there are no processes currently blocked in **cond_wait**. This works by posting a signal to the signal core hardware block (section 4.2) configured with the condition variable.

To use this condition variable with more CPUs a signal core that supports a broadcast mechanism is required.

The message queue itself is built around the condition variable and consists of a send and receive construct. As the message size is fixed these calls only takes a pointer to a buffer containing the message or buffer large enough to hold a new message.

**mqueue_send** posts a message to the given queue. This is accomplished by acquiring the mutex protecting the queue and then writing the message into the shared memory buffer. Once this is complete the mutex is released and **cond_signal** is called to unblock any readers waiting.

**mqueue_receive** receives a message from the queue. The associated mutex is locked and the queue is checked for messages, if the queue is empty **cond_wait** is called to suspend the process, this also unlocks the mutex. When a new message is received, **cond_wait** returns with the mutex locked, the message can then simply be copied from shared memory into the local buffer, the mutex unlocked and the **mqueue_receive** call can return.
5.2. SOFTWARE MESSAGE QUEUE

5.2.1 Performance

The message queue was tested with a slot size up to 2000 bytes using a shared memory on 4096 bytes, yielding minimum of 2 slots. Similarly to the performance analysis of the LINX Avalon Connection Manager (section 5.1.2), the message queue were tested using both SCHED_OTHER and SCHED_RR with and without additional system load. SCHED_OTHER shows the same behavior as with the LINX test and SCHED_RR shows no significant difference with or without load.

![Software message queue](image)

Figure 5.6: Software message queue

The message queue has fixed slots and does not support fragmentation nor requires and additional processing of the data besides a memory copy and should therefore scale in $O(n)$ where $n$ is the number of bytes in the message. This seems to be supported by the practical test as the time increases linear (although at a very shallow rate) with increasing message size.

The shallow increase in time also suggest that the the memory copy operation does not add any significant time with increasing messages, but instead the larger part of the time comes from the inter-CPU synchronization mechanisms used to construct the message queue.
Chapter 6

Conclusion

6.1 Linux as a real-time system

Linux and Linux-based operating systems are highly flexible with existing support for different CPU architectures, support for a myriad of different hardware devices and support for many different protocols such as various networking protocols like IPv4/IPv6, TCP/UDP, network file systems like NFS and AFS, and peripheral protocols like USB and Firewire. Additionally, the number of available applications for Linux and POSIX environment are enormous\(^1\) and most of them are open source programs.

Linux is without doubt more heavy weight than small real time systems such as MicroC/OS-II (section 3.3), but the amount of already existing support and application together with the possibility of modifying all the source code, if needed, makes Linux an attractive choice for (soft real-time) embedded platforms as existing code and applications can be reused.

Using Linux in a hard real-time environment requires significantly more work and auditing because of the sheer size of the Linux kernel and the fact that it never was designed as a hard real-time system. While user land processes in Linux are able to be scheduled with real-time priority scheduler, (section 3.2.1) problems still arise because of the clear distinction between the ”user space” and ”kernel space” (section 3.1) particularly related to synchronization mechanism used inside the kernel. For a priority driven scheduler to be effective the kernel needs to be fully preemptable to allow more important user land processes to preempt the kernel.

Work by Ingo Molnar et al. in the form of the ”RT patch” tries to address these issues and enables kernel preemption as far as possible by replacing non-interruptible kernel synchronization mechanism called spin-locks with sleepable, preemptable, locks. This patch also addresses scheduling of interrupt handlers to defer execution which further helps in scheduling predictability of low priority interrupts. A similar third-party initiative by Wind River System called Wind River Real-Time Core for Linux or RT-Linux address the same issue by replacing the scheduler and allowing

\(^1\)Websites like Freshmeat and Sourceforge hosts and lists popular applications
the kernel to be preemptive.

The conclusion is that while it is fully plausible to run Linux as a hard real-time system, very careful auditing of all involved kernel components would be required, especially with consideration regarding sections executing with interrupts and preemption disabled.

A more sensible approach would be to use a multi-core system with one core dedicated to running tasks requiring serious hard real-time operating conditions. This core would be controlled by a smaller RTOS like MicroC/OS-II or any other dedicated real-time system that have been verified for safety critical operation. A secondary core would then run Linux and primary control the system interface such as an LCD display or a TCP/IP-based interface.

Several aspects of the Nios II/Linux port were insufficient at the time of the project and needs improvements, particularly MMU/MPU support and a port of the "RT patch" to Nios II/Linux to enable kernel preemptiveness as far as possible.

### 6.2 Synchronization primitives for Nios II and Avalon Switch Fabric

Two new hardware blocks for the Nios II and the Avalon Switch Fabric for inter-CPU synchronization purposes were constructed during the project.

#### 6.2.1 Avalon Signal Core

The Avalon Signal core fills the need of an asynchronous event mechanism between multiple Nios II CPUs which previously did not exist. By using this hardware block a process on CPU0 can notify a process on CPU1 that some asynchronous event occurred without the need for CPU1 to poll a memory location to detect events. Both the hardware and the software drivers for this block execute in $O(1)$ making only the bus access time and IRQ latency of the operating system an issue. This makes the hardware block suitable for asynchronous non-periodic events, for extremely frequent events a polling method might be more suitable because of the IRQ overhead associated with each delivered signal.

The main limitation is the lack of a broadcast mechanism because of lack of support from the underlying bus. An broadcast-like mechanism could be implemented in software by linearly sending events to those CPUs in question, this does however introduce a strict order in time of the delivered signals. Another method would be to create one signal block and connect all CPUs to the same block, this would however not make it possible for individual CPUs to mask specific signals. Yet another possibility is to extend the core and dynamically generate multiple registers for each CPU connected to the device using the scripting capabilities of the Nios II system generation tool "SOPC builder".
6.3. HIGH-LEVEL COMMUNICATION PLATFORM

6.2.2 Avalon Sleep Mutex

The second hardware block created was the Avalon Sleep Mutex which attempts to improve the standard Altera Avalon Mutex by making it possible for CPUs to “sleep” when waiting for a lock on a busy mutex. By maintaining a hardware queue of CPUs waiting for a lock, a CPU can be notified when it is assigned the lock instead of the need for it to enter into a busy loop attempting to acquire the mutex.

This makes it possible for the operating system to suspend the calling task/process from running and resume it when the mutex have been locked. This could improve throughput in priority driven multi-tasking operating system where a high priority task would block waiting for a mutex lock allowing lower priority tasks to run, compared to a polling method like the one deployed by the original Altera Avalon Mutex where the high priority task would block all tasks of lower priority from running during the time the high priority task attempts to acquire the mutex.

However, this also increases system complexity which must be taken into consideration when designing an application using this hardware block.

The concept has proven itself to work, but because of how the Avalon Switch Fabric works an extra hardware device called IRQ filter had to be created. The filter is required to avoid having the Sleep Mutex generate an IRQ to all CPUs involved, instead one IRQ filter per CPU individually screens the Sleep Mutex and only interrupts its assigned CPU if it really is the new owner. This essentially moves the process of finding which CPU that is the owner from software into hardware.

This has a positive impact on a running system but further increases the complexity of the Sleep Mutex during system generation. Particularly the manual instantiation of IRQ filters in the Altera system generation tool “SOPC builder” is slightly cumbersome. This could be improved by automatically generating IRQ filters using the scripting capabilities of the “SOPC builder” tool.

A second limitation of the IRQ filter is that it can only be used for one unique owner value at any given time, this effectively limits the one process at the time to use the filter. This could be improved by allowing a masked comparison by adding a new mask register which essentially would allow multiple processes within one CPU to use the same filter by treating different parts of the mutex owner value as CPU unique and process unique and interrupt whenever the CPU unique part match.

6.3 High-level communication platform

Both the platform based on the existing LINX framework and the created software message queue based solely around the created hardware synchronization primitives proved to be viable methods for inter-CPU communication between processes running on different CPUs on the same shared memory system.

The LINX framework together with the created Avalon Connection Manager is a very flexible solution and comes with an easy to use API. This flexibility does however come with a price as the software message queue performed better as shown by the graphs in figure 5.6 and 5.5. Additionally, LINX comes with a larger code
footprint, thus increasing the amount of RAM required to run the Linux kernel (table 3.3).

While LINX definitely remains an option and would be suitable to bridge multiple Nios II CPUs on different hardware systems using other existing connection managers (such as Ethernet), it might come with a too high overhead when only two Nios II CPUs within the same FPGA are involved. For such a case a solution based on the software message queue or synchronization with a plain Sleep Mutex or Signal Core might provide a more lightweight communication mechanism.
Bibliography


BIBLIOGRAPHY


