The University of Manchester
Department of Computer Science
Project Report 2024

My Little Operating System

Author: Thomas H. Jones

Supervisor: Dr James Garside

Abstract

My Little Operating System

Author: Thomas H. Jones

Supervisor: Dr James Garside

Contents

List of Figures

List of Tables

Glossary

ARM A semiconductor design company who design a family of RISC architectures. 8

RISC-V An open standard reduced instruction set computer architecture developed by the University of California, Berkley. 7, 8

Abbreviations

CLINT core local interruptor

CSRs control and status registers

IO Input/Output

ISA instruction set architecture

PLIC platform level interrupt controller

RISC reduced instruction set computer

Chapter 1

Introduction

There are many low level concepts that are discussed over the course of higher education. This project sets out to explore some these concepts through basic implementation and experimentation, using simple hardware to allow full comprehension of the system being used during development. This chapter will introduce the aims of the project, and describe the motivations behind the project.

1.1 Aims

The overall aim of this project is to produce an operating system for a RISC-V based microcontroller, that allows the execution of multiple processes. To do this, a RISC-V machine must be selected, and a toolchain to develop, load, and execute/debug code on the machine must be made. The elements of the operating system will be split into three main aims, the exectution and scheduling of processes, the implementation of basic memory management to restrict each processes memory access, and the development of Input/Output (IO) to allow interaction between the processes and a user. The effectiveness and limitations of the system will then be evaluated, and the RISC-V architecture will be compared with other reduced instruction set computer (RISC) architectures.

1.2 Motivation and Challenges

The basis of this project is to facilitate experimentation with operating system concepts, as while in a university level course the concepts are explained, it is often not possible to showcase these concepts in a practical enviroment, limited

often to simulations. By using a bare metal system, this project will allow full access to a machine with no other system software, allowing complete control over the enviroment, which allows the implementation of concepts like processes and memory management.
This has also been used as an opportunity to explore the RISC-V architecture, as the prevalent architecture used in microcontroller systems are ARM based, so the use of RISC-V in this project will enable the comparison of the two architectures, and evaluate the effectiveness of RISC-V as an architecture.
Since the project is being developed on a bare metal system, considerations like the toolchain used to develop and debug code are important as they will have to be configured for the specific system being developed on, which may be more challenging as the userbase of RISC-V is more limited, so the tools required will be smaller and less well documented. The loading of code onto the machine is also important to consider, as it is easily possibly that faulty or buggy builds could cause serious malfunction in the machine, some of which could not be reverible, so the loading and debugging of code must be done with care.
Other challenges include system limitations, as unlike a modern machine that would be used for general use, a microcontroller will have vastly limited resouces in comparison, resulting in a need for greater optimisation and careful planning, so the requirements of the system do not exceed the resouces of the machine. Another limitation is the machine IO capabilities. This limitation does not only affect the end result of the systems IO, but also increases the difficulty of development, as some features such as time based interrupts can be difficult to develop without physical feedback from the machine.

1.3 Report Outline

- Introduction: This chapter gives the basic overview of the planned project
- Background: This chapter will introduce key concepts to enable a more comprehensive understanding of the projects details
- Design: This chapter will outline the key design decisions made before and during the project
- Implementation: This chapter will give details on the practical implementation of concepts and designs mentioned in previous chapters

- Evaluation: This chapter will discuss the implementation and findings made during the implementation
- Conclusion: This chapter will give a summary of how the aims of the project were met

Chapter 2

Background

This chapter will go over the background of this project, which will cover basic operating system fundamentals, a brief overview of the RISC-V architecture, and the specifics of the board that will be used, which is the Sifive Hifive1 RevB. By the end of the chapter, the reader should be able to understand how parts of the operating system should function, and be able to understand the RISC-V architecture in comparison to the ARM architecture.

2.1 Operating System Fundamentals

An operating system's purpose is to provide an interface between user code and the hardware, such as memory and IO, to allow the user code to function seamlessly, and allow interaction with users. This has been split into three key sections, as listed by the aims of the project, as process management, memory management and IO management.[2]

2.1.1 Process Management

The process model is where all software is organised as seperate code, which is scheduled in a fashion to give the illusion of multiproccessing. While multiproccessing is possible with multiple hardware threads, this project is limited to microcontroller chips, which mainly only support a single hardware thread. To give the illusion of multiproccessing, the information to load and run individual sections of sequenctial code is stored, so that an algorithm can be used to load, run, and possibly pause these execution, to allow them to be swapped out at a

rate that makes it imperceivable that they are not running concurrently. This algorithm is the scheduling algorithm, and there are several approaches to how this is handled.[2]

Scheduling Algorithms

In an interactive system, processes are ran in a way that ensures each process will get a fair amount of execution time. This is done simplest using a round robin algorithm. Due to the interactive requirement, processes that take a large amount of execution time need to be pre-empted, so one process cannot halt other processes for too long. This is done mainly by pausing a running process after a certain amount of time (which is known as the algorithms quantum).

2.1.2 Memory Management

The goal of memory management is to streamline how a process can use memory, while at the same time protecting critical sections of memory from faulty or malicous user code. In a system with multiple processes being executed and no memory abstraction, there exists the possibility that two processes attempt to use the same section of memory, creating a conflict that would cause both processes to run incorrectly. This occurs as different processes cannot be aware of each other, and have no choice but to use memory without knowledge of which sections are in use. This can be solved using memory abstraction. The simplest method of this is using address spaces, where each process is given permission to access only segments of directly addressed memory. This prevents each process modifying other processes memory, and allows the process to behave individually, as long as it is provided with the location of its address space.[2]

2.1.3 IO Management

2.2 Information on board

2.3 RISC-V

RISC-V is RISC instruction set architecture (ISA), built with the goal to be completly open. The ISA uses a base interger ISA which can be used on its own or with a number of optional standard extensions, which is part of a goal to avoid

'over-architecting' for particular micro-architectures. There are variants for both 32 bit systems and 64 bit systems however for this project only the 32 bit system will be considered.[3]

2.3.1 Registers

RISC-V implenents 32 general purpose registers, which in our case will be 32 bits wide, and labeled x0 to x31. The x0 register is hardwired to zero, and all other registers may be used. There are no specific registers used for storing information like the stack pointer or return address, however in this project the standard calling convention will be used. This specifies how each register should be used, how the register should be saved, and gives each register a pseudonym which correlates to its purpose. Important registers are ra, which is used to store the return address produced by the jump and link instruction, sp and gp, which are used to store the stack pointer and the global pointer. The stack pointer is used to store the most recent item in the current processes stack, and the global pointer is used to point to the address space where a processes global variables will be stored. The collection of registers a0-a7 are used to pass arguments to functions, as well as a0 serving as a return value. For general use there are registers t0 to t6 and s0 to s11, the difference being that t0-t6 are temporary registers that are not required to have their value saved before use, whereas s0-s11 does require saving the value. This allows the use of temporary registers to store intermeddiate values or value not needed to be stored, without needed the overhead of adding them to the stack, and the saved registers allow important information not to be overwritten by called functions.[3]

2.3.2 Base ISA

The base ISA, known as RV32I, specifies the set of instructions that all RISC-V systems implement, which includes both priviledged and unpriveledge instructions. The load and store operations allow loading and storing one register to or from memory. This registers can be treated as a word, half word or as a byte. There are the standard arithmetic, logic and shift operations, however immediate values are limited to 12 bits, so operations such as LUI (load upper immediate) may be used to load larger immediate values into a register.
Unlike ARM, the RISC-V ISA does not use flags to determine conditional operations. Instead the condition that is used to determine whether a branch is taken or not is included in the branch operation itself, with instructions like BEQ (branch

if equal) or BGT (brach if greater than). These are accompanied by comparison operations, which evalutate similar conditions, and produce either a 1 or a 0 in the target register. JAL (jump and link) branches to a specified location while storing a return address, which allows the implementation of subroutines and functions.

Priviledge levels

The previously described operations can be performed in any priviledge mode. RISC-V can support up to three priviledge modes, being user, supervisor, and machine mode, although this report will only be considering user mode and machine mode. The zicsr extension adds operations that allow a hart in machine mode to read and modify control and status registers (CSRs), such as the CSRRW, which reads a csr to a register and writes a register to a csr, although either the source or target register can be set to zero to disable either the reading or the writing. Also included is the CSRRS and CSRRC, which is similar to CSRRW but sets or clears bits of the CSRs.

2.3.3 Optional Extensions

2.3.4 Traps

In RISC-V, a trap refers to anything where the execution on the hart is handed to the trap handler. There are two categories of traps, synchronous and asynchronous

Asynchronous/Interrupts

An asynchronous trap is a break in execution caused by external factors, and can be referred to as an interrupt. There are three types of interrupt which are software, timer, and external. These are controlled by two units, the core local interruptor (CLINT) and the platform level interrupt controller (PLIC). The CLINT handles the software and timer interrupts, and the PLIC handles the external interrupts. A software interrupt is triggered when a hart sets its interrupt bit high. This is used generally for commincation between harts, and since this project will not involve more that one hart, this functionality will not be used. A timer interrupt is taken when the mtime CSR is greater than the mtimecmp CSR, so is used to generate an interrupt after a given amount of time. External interrupts are any other caused by the PLIC, which can include seperate timer interrupts, or IO sourced interrupts.

Synchronous

A synchronous trap occurs in response to the execution of an instruction, which occurs with the clock, hence syncrounous. These come from two sources, errors and enviroment calls. Errors included memory misallignments, access faults and illegal instructions. This allows for these errors to handled, either to be inform the process of the error, or to remove the process. Enviroment calls are how processes make calls to the kernel to perform actions that are above the processes access level, generally to perform IO operations, or to interact with other processes.

Chapter 3

Design and Implementation

3.1 Toolchain

3.2 Board Configuration

On the Hifive1, there are several important configuration options that affect general operation of the board. The most notable of these are the clock settings, as these indicate the frequency of the processor, input and output frequencies, and timer interrupts.

3.2.1 Clock settings

The Hifive1 has 3 clock regions, a high frequency clock, a low frequency clock, and a clock used to drive the JTAG connection. The JTAG driver is constant and only used for debugging through JTAG, so is not relavent here.

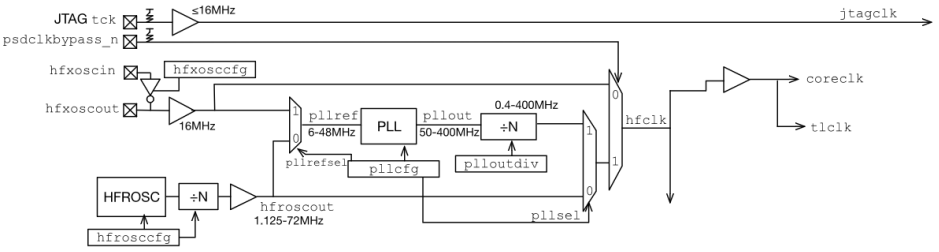

Figure 3.1: The high frequency clock generation scheme, specifying how the high frequency clock is driven and configured, taken from the Sifive FE310-G002 Manual[1]

The high frequency clock controls the processor frequency, and the baud rate of input and output is derived from it. The high frequency clock can be driven from two sources, an internally trimmable high frequency ring oscillator and an external high frequency crystal oscillator. The ring oscillator can produce frequencies ranging form 1 MHz to 75 MHz, whereas the crytal will produce a constant frequency of 16 MHz. Both of these clock sources may be used 'as is', or can be modified using a PLL and divider, giving a available range of 48 MHz to 384 MHz.

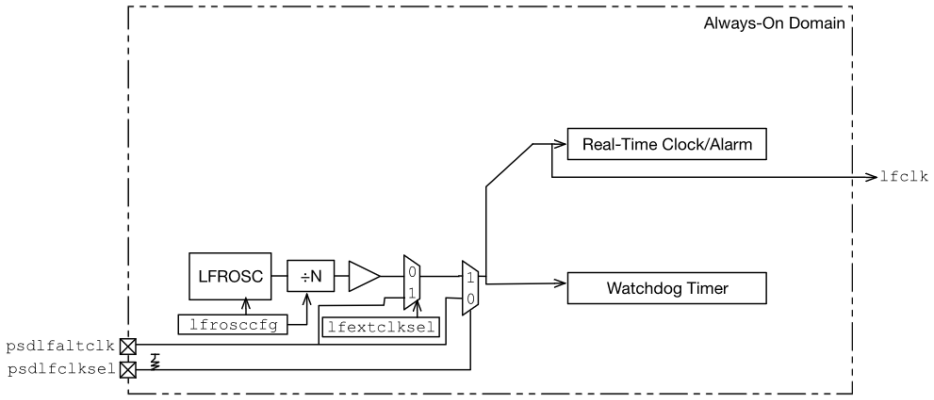

Figure 3.2: The low frequency clock generation scheme, specifying how the low frequency clock is driven and configured[1]

The low frequency clock is part of the Hifive1 'always on block' and controls the watchdog timer, which can be used to cause a reset on malfunction, and both

the realtime clock and the machine timer, both of which are used to generate timed interrupts. Similar to the high frequency clock can be driven from a ring oscillator or from an external clock, which in the Hifive1 is a crystal oscillator. The low frequency ring oscillator functions at 1.5 KHz to 230 KHz using a frequency divider, and the implemented external clock runs at a contant 32.768 KHz, with no option to divide the frequency.
For both clock domains, the crystal oscillator was chosen. The ring oscillator gives the option to operate at a higher frequency, which would result in a higher number of operations per second. While in a practical operating system this would be desirable, since this system is not intended for practical use, a constant frequency was more desirable as it would give more predictable results, and makes IO operations more reliable. For the low frequency clock, a high frequency would be beneficial for a real time system as the higher frequency would allow for more precise timing of interrupts and other functions, however for an interactive system this precision is not required, and so similar to the high frequency domain the constant frequency of a crystal oscillator was selected.

3.3 Processes

3.3.1 Process Structure

Due to the extremely limited amount of memory available on the Hifive1, the data required to store information on each process has to be structured carefully, else the amount of memory required to store process information would begin to limit the amount of memory available to the processes themselves. In the current implementation, 35 bytes are used to store process information.
One byte each is allocated for a process id, process parent id, process status, and size of a processes text section. For process id and process parent id, only one byte is needed as the Hifive1 does not have the memory to support a large of processes, so a theorectical cap of 256 processes is acceptable. Process status can only take 4 values, so only 2 bits of the byte are used, and the text size byte stores the size as a power of 2, where a process has 2^n words, where n is the value stored. While some of these values only require bits, they are stored as bytes to keep the structure word aligned.
One word each is allocated for the processes program counter, text section pointer and address space pointer. Since memory addresses are word length, these cannot be reduced.

The vast majority of the process structure is used storing the 31 general purpose registers. This is required to retain the state of each process in between scheduling. The only option to reduce this would be to limit the amount of registers available to use. Only 31 must be stored as the x0 register is hardwired to zero.
In other systems, information like the processes stack pointer may be stored, however standard RISC-V calling convention specifies x2 to be used as the stack pointer, so seperate storing of this infomation is not required, and allows a process to handle its address space on its own, however on process creation x2 and x3 are initialised as the stack pointer and the global pointer, where the stack pointer points to the bottom of the process address space and the global pointer at the top. This is done to reduce the overhead of processes that use the standard calling convention.

3.3.2 Scheduler

To implement an interactive system, a scheduler will be designed to allow a process run for a limited time, before halting the process, and running another ready process. The time allowed for a process to run is refered to as its quantum. Processes will be scheduled to be ran in a round robin fashion, to allow a fair runtime to each process while maintaining a low overhead. This is implemented using a circular queue, where new and halted processes are added to the end of the queue, and processes to run are taken from the start of the queue.

3.4 Memory Management

3.5 IO

Chapter 4

Evaluation

Chapter 5

Conclusions and further work

Bibliography

[1] *SiFive FE310-G002 Manual v1p5*. Sifive Inc, 2022.

[2] H. B. Andrew S. Tanenbaum. *Modern Operating Systems 4th Edition*. Pearson, 2015.

[3] K. A. Andrew Waterman, editor. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121*. RISC-V Foundation, December 2019.