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

Throughout an undergraduate computer science degree, many operating system concepts are covered however many are not demonstrated practically to much depth. This project uses a bare metal system to implement a simple operating system from scratch to explore these concepts, and discussing how the use of a RISC-V system differs from an ARM system. The Sifive Hifive1 Rev B was used to successfully demonstrate a simple operating system that could schedule processes and interact with a user. Future work will attempt to implement different concepts and more interactive input and output, and implement a dynamic memory system to compare its performance in a limited system.

Supervisor: Dr James Garside

Contents

1 Introduction **5**
- 1.1 Aims . . . 5
- 1.2 Motivation and Challenges . . . 6
- 1.3 Report Outline . . . 6

2 Background **8**
- 2.1 Operating System Fundamentals . . . 8
  - 2.1.1 Process Management . . . 8
  - 2.1.2 Memory Management . . . 9
  - 2.1.3 IO Management . . . 10
- 2.2 RISC-V . . . 10
  - 2.2.1 Base ISA . . . 11
  - 2.2.2 Registers . . . 12
  - 2.2.3 Control Flow . . . 14
  - 2.2.4 Traps . . . 15
- 2.3 Information on board . . . 16

3 Design and Implementation **17**
- 3.1 Code Style and Structure . . . 17
  - 3.1.1 Structure . . . 17
  - 3.1.2 Toolchain . . . 18
- 3.2 Processes . . . 20
  - 3.2.1 Process Structure . . . 20
  - 3.2.2 States . . . 22
- 3.3 Board Boot and Configuration . . . 23
  - 3.3.1 Clock settings . . . 24
- 3.4 Traps . . . 26
  - 3.4.1 Configuration . . . 26

3.4.2 Asynchronous Traps . . . 26
3.4.3 Synchronous Traps . . . 27
3.5 Scheduler . . . 27
3.5.1 Creating a Process . . . 27
3.5.2 Running a process . . . 28
3.5.3 Preemption and Scheduling . . . 28
3.6 Memory Management . . . 30
3.6.1 Address Spaces . . . 30
3.6.2 Address Space Layout . . . 31
3.6.3 PMP Use . . . 32
3.7 IO . . . 33
3.7.1 LEDs . . . 33
3.7.2 Text input and output . . . 33

4 Evaluation **36**
4.1 Challenges . . . 36
4.2 Disassembly . . . 36
4.3 Breakpoints and Inspection . . . 37

5 Conclusions and further work **39**
5.1 Summary of Work . . . 39
5.2 Future Work . . . 40

Bibliography **41**

List of Figures

2.1 RISC-V Registers 13
2.2 ARM branching code 14
2.3 RISC-V branching code 14
2.4 Sifive Hifive1 RevB 16

3.1 Process Structure 20
3.2 State Diagram 22
3.3 High Frequency Clock Diagram 24
3.4 Low Frequency Clock Diagram 25
3.5 Layout of memory 30
3.6 Process Address Space 31

Abbreviations

ABI application binary interface

CLINT core local interrupter

CSRs control and status registers

DTIM data tightly integrated memory

GPIO general purpose input/output controller

IO Input/Output

ISA Instruction Set Architecture

OTP one time programmable memory

pc program counter

PLIC platform level interrupt controller

PMP physical memory protection

PWM pulse width modulator

RISC reduced instruction set computer

SPI serial peripheral interface

UART universal asynchronous receiver/transmitter

Chapter 1

Introduction

There are many low level concepts that are discussed over the course of a standard undergraduate degree. This project sets out to explore some these concepts through basic implementation and experimentation, using simple hardware to allow full comprehension of a 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 execution 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 the main competing reduced instruction set computer (RISC) architecture to RISC-V, ARM.

1.2 Motivation and Challenges

The basis of this project is to facilitate experimentation with operating system concepts; while in a university level course such concepts are explained, it is often not possible to showcase these concepts in a practical environment, limited often to simulations. By using a bare metal system (directly on hardware with no other existing software), this project will allow full access to a machine with no other system software, allowing complete control over the environment, 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, 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 (the tools used to develop and debug code) are important as they will have to be configured for the specific system, which may be more challenging as the user base of RISC-V is more limited, so the tools required will be both less documented and available. The loading of code onto the machine is also important to consider, as it is easily possible that faulty or buggy builds could cause serious malfunction in the machine, 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 resources in comparison, resulting in a need for greater optimization and careful planning, so the requirements of the system do not exceed the resources of the machine. Another limitation is the machine's IO capabilities which not only affect the end result of the systems IO, but also increases the difficulty of development. Some features such as time based interrupts can be difficult to develop without physical feedback from the machine. This means that evaluating the project for correctness may require more manual checks than standard software design, as the lack of IO and asynchronous units reduce the control needed for standard testing environments, in addition to the lack of integration with modern testing software.

1.3 Report Outline

- Introduction: This chapter gives the basic overview of the planned project and define the projects aims and evaluation strategies

- 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 describe how the project was tested for completeness and correctness
- Conclusion: This chapter will give a summary of how the aims of the project were met and discuss possible future work

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 Sifive Hifive1 RevB, which is the board that I decided to use. 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 system's 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; process management, memory management and IO management[1].

2.1.1 Process Management

A process is a sequence of instructions that modify the state of the processor core it is run on. The goal of the operating system is to allow multiple processes to run at once. In a system with multiple cores, this would be possible to do explicitly, as the different cores could simply execute the different processes. Since the desired number of processes is almost always higher than the number of available cores (in the world of microprocessors there is rarely more than one core). Instead, processes must be run individually and interleaved to give the illusion of multi-

tasking, provided that this is done at a high enough frequency. There are several methods and approaches to this task, as described below[1].

Scheduling Algorithms

The main approaches to be considered for scheduling algorithms for this project will be a batch approach and an interactive approach. The most important distinction between the two is the inclusion of preemption. In a batch system, a non-preemptive approach will be taken, where once a processes execution is begun, it will be allowed to complete before any other process runs. This creates a simple system and is used in situations where completion of tasks is not expected to be quick. The main challenge of this approach is in the estimation of process execution time, as this allows the scheduler to order the execution of processes in a way that ensure small tasks are allowed to run. In an interactive system, it is expected by the user that interactions will have quick responses. This introduces the need for preemption, where a processes' execution may be paused after a short period to allow other processes to run. By running each process in small interleaved time slices, this allows multiple processes to functionally run at the same time, which means that the processes from which a user is expecting a response will not halt other processes[1].

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 malicious 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 necessitates the need for memory abstraction. One of the simplest methods of abstraction is the use of defined 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 each process to behave individually, as long as it is provided with the location of its address space[1].

2.1.3 IO Management

The device that the project features will have severely limited IO, and will be implementing both sequential and interrupt driven IO.

Sequential IO

Sequential IO is where actions that require IO are completed without interruption or preemption, which will generally function by the user code making an environment call which will then perform the IO operation, and return control to the user code once the operation is complete. This is not always desirable, as the IO is often slow, and operations that poll for resources would completely stall the system. During this time all processes are effectively blocked as the the environment call is not being preempted, so this would appear to the user as a complete halt in execution. This type of IO can be useful in cases where only small amounts of data is being transferred; in cases where only single bytes are transferred the overhead needed for interrupt driven IO would be larger than the time taken to execute the interaction in full.

Interrupt driven IO

To avoid the blocking caused by polling, IO can be interacted with through the use of interrupts. This would be done by a process making an environment call similar to programmable IO, however in this case only the caller process would be blocked until the IO operation was complete. It would also enable some form of interrupt, which signals when the IO function is available to be used. At that point an interrupt is be raised and part of the operation is completed until the IO is busy again, at which point execution returns to the unblocked processes. This means that there is less time wasted busy waiting, although this introduces overhead in the form of interrupt setup, which if done too often could slow down execution to a similar level as polling.

2.2 RISC-V

RISC-V is a RISC Instruction Set Architecture (ISA), featuring the goal to provide an open standard instruction set[2]. The ISA uses a base integer 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 as that is what this project uses[3].

2.2.1 Base ISA

The base ISA, known as RV32I, specifies the set of instructions that all RISC-V systems implement, which includes both privileged and unprivileged 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, but 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 (branch if greater than). These are accompanied by comparison operations, which evaluate the same comparison, and produce either a 1 or a 0 in a target register. JAL (jump and link) branches to a specified location and stores the address of the instruction to allow that section to be returned to, which allows the implementation of subroutines and functions[3].

Privilege levels

In most systems, there will be some mechanism for differentiating between system level and user level code. This is to prevent user level code from modifying critical sections of the system which would cause unpredictable behavior, and to maintain security between different parts of user code. In RISC-V, this is implemented through the differentiation between the unprivileged architecture and privileged architecture and the access to each being controlled by a current privilege mode. RISC-V provides specifications for a number of different modes but each implementation may choose to only implement some of these modes. The system I use only implements the simplest two modes; machine mode and user mode. These modes can be likened to the concept of a normal user and a 'super' user, where the super user is allowed to perform actions that the normal user cannot. This means programs in user mode can only execute instructions that are from the unprivileged architecture, and can only access memory that is specified by the memory protection. A program in machine mode can use all instructions

from both the unprivileged and privileged architecture, and has access to the full memory space.

Optional Extensions

There are several optional extensions that can be implemented. This includes the 'C' extensions, which implements a reduced number of instructions from the base ISA, but in a 16 bit compressed format. The 'M' extension includes 32 bit instructions to perform multiplication and division, as well as a remainder operation. The 'A' extension includes a number of atomic memory operations, such as store conditional, swap instructions and memory operations that perform logic operations atomically.

2.2.2 Registers

The registers in RISC-V can be put into two categories; general purpose registers and the control and status registers. These are then used on two levels, the programmers level and the system level.

Programmer Level

In RISC-V, there are always 32 general purpose registers, x0 to x31. The specific implementation of RISC-V can alter the width (number of bits) of each of the registers; The Hifive1 uses the rv32i architecture, which indicates that each register is 32 bits wide and uses the full instruction set instead of the reduced set as used by rv32e. Figure 2.1 show all the user level registers, each general register may be used as the programmer wishes except for x0 which has its value hardwired to zero which facilitates the comparison instructions and allows instructions to not store their output. In addition to the general purpose registers, each program also makes use of the program counter (pc). This stores the address of the current instruction. The programmer can only read the value stored using the AUIPC (add upper immediate to pc) instruction, and can only be modified through either branch or jump instructions. The pc is an example of one of the control and status registers (CSRs) that are available to be used by the programmer. Other CSRs that may be available are the cycle, time and retired instruction CSRs, which display various data about the system; these are accessed through the RD instructions if this has been permitted by the system through the *mcounteren* CSR.

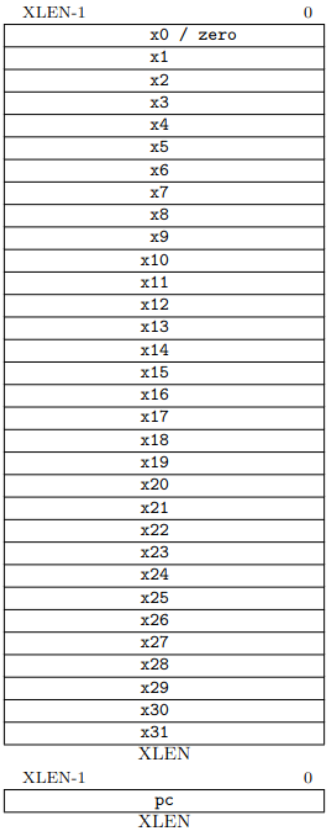

Figure 2.1: This figure shows the registers used at the programmer level, which includes the zero register, 31 general use registers and a program counter, taken from the RISC-V unprivileged architecture[3]

System Level

Above the programming level, the system utilizes the CSRs to both control and indicate the status of the system. The use of CSRs in RISC-V systems can be compared to the use of the CPSR in ARM based systems but the RISC-V CSRs are used to control and configure far more using many more than one register to do so. Important CSRs include *mstatus*, which controls many aspects of the system such as enabling traps and controlling the privilege mode, *mtvec*, which specifies how traps are executed, and the *pmpcfg* CSRs, which control memory protection. There are also memory mapped registers that are used; these include both standard registers however systems will also include many registers specific to that system. Important registers of this form include *mtime* and *mtimecmp*, which store the system time and trigger timer interrupts.

2.2.3 Control Flow

In an ARM based system, boolean flags are utilized to control the execution of branches; to execute a conditional branch, first a comparison instruction is executed to set the flags, and then the branch instruction will execute depending on the flags, as shown in figure 2.2. Figure 2.3 shows how RISC-V does not use flags, but instead requires the branch condition comparison to be part of the branch instruction.

```
cmp r0, #0          ;sets the condition flags
beq destination     ;executes branch if the zero flag is set
```

Figure 2.2: This figure shows how an ARM system would execute a branch with the condition of a register being zero

```
beq s0, zero, destination   ;compares and branches if equal
```

Figure 2.3: This figure shows how a RISC-V system would execute a branch with the condition of a register being zero

2.2.4 Traps

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

Asynchronous Traps

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; software, timer, and external. These are controlled by two units, the core local interrupter (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 generally used for communication between harts, and since this project will not involve more that one hart, this functionality can be used as I wish. A timer interrupt is taken when the *mtime* register is greater than the *mtimecmp* register, so is used to generate an interrupt after a given time. External interrupts are any other caused by the PLIC, which can include separate timer interrupts, or IO sourced interrupts[4].

Synchronous Traps

A synchronous trap occurs in response to the execution of an instruction, which occurs with the execution of an instruction, hence 'synchronous'. These come from two sources; errors and environment calls. Errors include load/store misalignment, access faults and illegal instructions. This allows these errors to be handled, either to allow the process to catch the error, or to remove the process. Environment 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.

2.3 Information on board

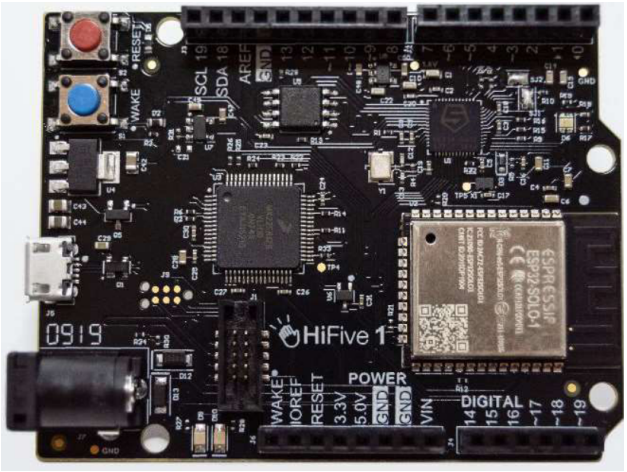

Figure 2.4: This figure shows the Sifive Hifive1 RevB development board from the Hifive1 Rev B Schematics[5]

This project will be done on the Sifive Hifive1 Rev B. This is the second iteration of the Hifive1 board, which uses the SiFive Freedom E310-G002 chip featuring a RV32IMAC core, a 16 KiB instruction cache, 16 KiB data RAM, and 32 MiB of Flash memory, which acts as ROM. It has Arduino compatible pins, a set of LEDs, a Bluetooth/WiFi chip, and a USB interface[4].

Chapter 3

Design and Implementation

This chapter will describe the major design decisions made before and during the development of the project, as well as important details about how the project was designed and executed.

3.1 Code Style and Structure

Since the code of this project is done in RISC-V assembly, it was very important to establish and maintain a strict code style and to use standard practices. This is because assembly allows minor mistakes to cause malfunction, and often is difficult to parse once written making debugging far more difficult than standard programming. To do this, I used the RISC-V standard calling convention, as specified in the RISC-V application binary interface (ABI). This indicates how each register should be used in regards to argument passing and returning, temporary and saved values, and states the use of full descending stacks[6].

3.1.1 Structure

There are three main sections of code; Kernel code, user code, and read only data. The kernel consists of the boot up script to initialize the system, and the trap handler which handles the functionality of the operating system. The user code section contains any application programs to be ran either on boot or by another application. The read only data contains data that is mostly used for debugging, such as error strings so exceptions can display more verbose descriptions. Also included are various macros. In terms of loading code onto the board, all these

sections are part of the text section of the code, which indicates to the linker where in memory they will be placed, as the text section refers to the section of a program that contains the immutable code, in comparison to the data section which is the section where data might be stored and manipulated.

3.1.2 Toolchain

To develop code for the Hifive1, it was required to develop an effective method of compiling and loading code onto the board, as well as an effective debugger. Initially I used the Sifive's Freedom E SDK with PlatformIO to perform basic prototypes and experiments with the board. However these tools did not provide a sufficient level of control over what was compiled and loaded, as the SDK interacts with many parts of system that were needed for the project, as well as altering the configuration of the board. This meant that the SDK was effective for developing simple applications with the board but did not allow for low level control of operations like interrupts or privilege levels. Inspection of the source code of these led to useful insights.
To replace these tools, I used as and ld from the riscv-gnu-toolchain to assemble and link my code into an elf file that could be loaded onto the board. Also used from that collection was objdump, which was used as a disassembler. The assembler's target architecture was rv32ima_zicsr_zifencei, to include all the extensions available on the Hifive1, except for the compressed extension. This was done to prevent the assembler generating a mix of 32 and 16 bit operations, as this could cause operations to not be word aligned, and allows jump tables to be implemented without inspecting what instructions are used. Two linker configurations were used throughout the project, the first which put both the text section and the data section into RAM, whereas the second put text and read-only data in ROM, and left the RAM free. The first configuration was used earlier, as it allowed for safe experimentation with the board while maintaining Sifive's double-tap bootloader. The double-tap bootloader allows normal operation on a regular reset but allows the board to be loaded into a safe mode when the reset is 'double tapped', which prevents execution of code not part of the bootloader, to prevent faulty code from being ran so that the board is kept in a healthy state to allow the faulty section to be replaced, as the faulty code could alter settings that would interrupt the Flashing of new code. While this was a useful feature, it limited development, as it also performed several unwanted functions, such as changing the clock configuration, and the uart configuration, which was not acceptable. The second linker config replaced the bootloader in the flash memory with the project code. This

was also necessary as it was not practical to store code in RAM, as it was limited in space, so the RAM was reserved for program data. No data is stored in memory when flashed, as that data would not be restored on reset so anything dependent on that data would not load.
To load, run and debug code on the board, I used openocd and gdb. Openocd was used to create an interface with the board, and to specify how the board should be initialized and loaded, and gdb was used to target the openocd interface, which allowed it to load elf files onto the board, and to run/debug code as normal, with some limits such as a limit to the number of breakpoints.

3.2 Processes

3.2.1 Process Structure

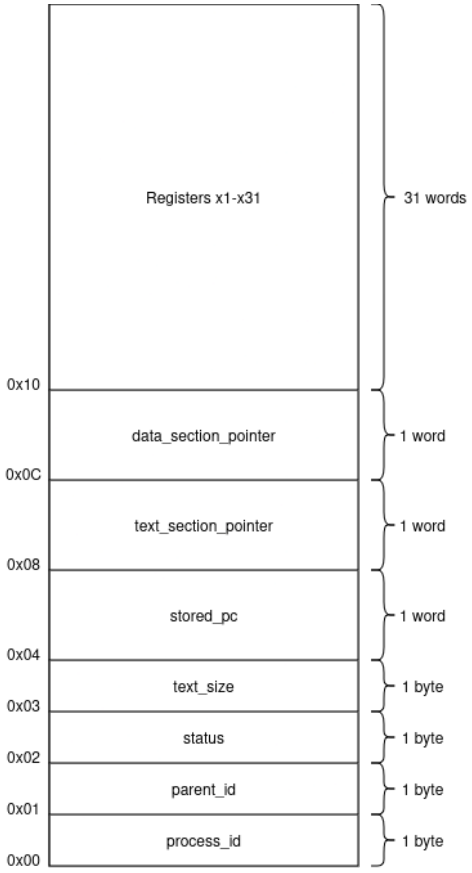

Figure 3.1: This figure shows the structure of process data stored, with the relative offset for each element and the size of each section

Due to the limited memory available on the Hifive1, the data required to store information on each process has to be structured carefully, else the memory required to store process information would begin to memory available to the processes themselves. In the current implementation, 35 words 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 theoretical 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. The minimum number of bits needed to store these is 13, however this would need to be padded to either 32 bits, to ensure that the structure remained word aligned, as an attempt to use the load word instruction on a non word aligned address causes a load address misaligned error. Instead each entry is stored as a byte, as this is the smallest size load that RISC-V allows, which avoids the need to use bit masks to retrieve these entries.

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 separate storing of this information is not required, and allows a process to handle its address space on its own, however on process creation x2 and x3 are initialized 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.

3.2.2 States

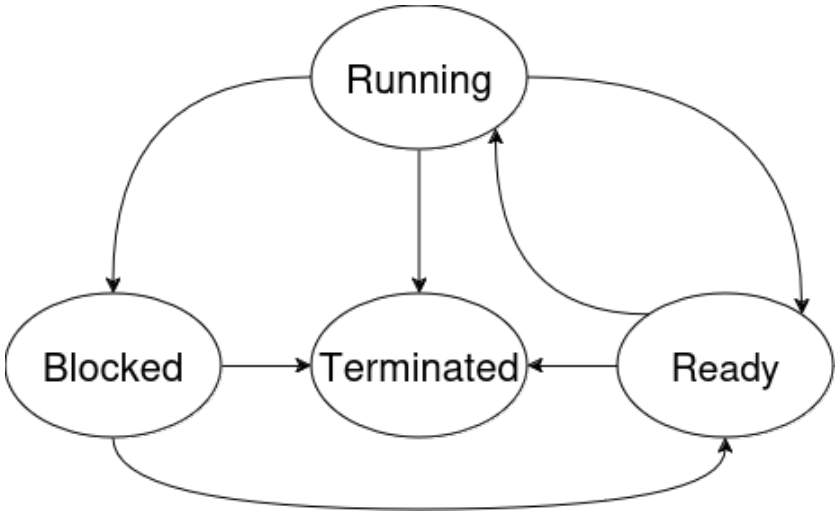

Figure 3.2: State machine showing the four states a process may be in, and the possible transitions

Each process has a state, which indicates how the process should be scheduled. In our model, there are four states that a process can be in; ready, running, blocked, and terminated. Running indicates that the process is currently being executed. A running process can transition to ready if the scheduler preempts the process, or if the process yields to the processor with an environment call. A process will transition to blocked when it makes an interrupt driven IO call. The blocked process may then transition to ready when the interrupt has been handled. The transition between ready and running is handled by the scheduler, whereas the status of blocked processes are dealt with by each IO handler. The state value stored in each process entry is used as metadata only except for the terminated state, as instead the scheduler and each IO handler tracks state by how each process is stored, for instance any process in a queue to be ran is in the ready state, the current process is in the running state, and any processes not in the queue and not running must be blocked. The terminated state indicates to the scheduler that the process should not be re-queued.

3.3 Board Boot and Configuration

On power on the Hifive1 will begin execution at the reset vector of 0x1004, which in this implementation is not configurable and will jump to the Mask ROM at 0x1_0000, in which similarly cannot be configured in the Hifive1's implementation of the E31, which is the general architecture that the Hifive1's chips is implemented from. The mask ROM will immediately jump to the one time programmable memory (OTP) memory. This is configurable, but will not be done in this project. This is because the OTP memory can only be edited by code executing on the board, and only one bit at a time. Since the OTP memory is executed as part of the boot sequence, if it is programmed in a way that does not jump to code in flash or memory it will prevent the board from executing code, which also means that the malfunctioning OTP code cannot be fixed, leaving the board in an unrecoverable state. This danger is obviously unacceptable for this project, so the OTP memory will be left to its default, which will jump to 0x2000_0000, which is the beginning of flash memory.
From that point, execution of the system setup begins. 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. This is the second thing configured during boot, after zeroing the registers.

1. The pc is set to 0x2000_0000
2. The system stack pointer and global pointer is set, and the other registers are zeroed
3. Both high frequency and low frequency clocks are set to 16 MHz and 32 KHz
4. The trap vector base and mode is set
5. The GPIO is set to enable the LEDs and the UART
6. The UART is configured, and the UART print queue is created
7. The external interrupt settings are configured
8. The scheduler queue and available process stack are created

9. The process table is populated with basic information including each processes address space, and each process is added to the available process stack

10. Any initial user applications are added to the queue for scheduling

11. The scheduler is started

3.3.1 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 relevant here.

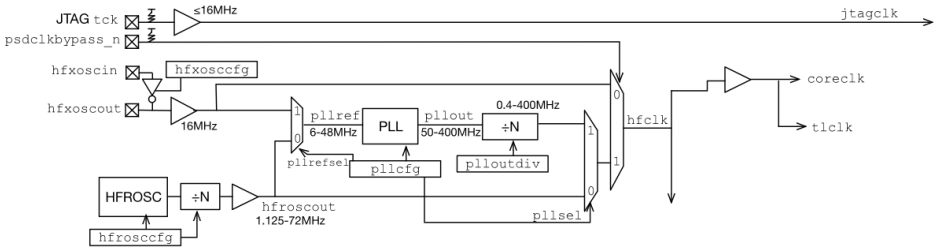

Figure 3.3: The high frequency clock generation scheme, specifying how the high frequency clock is driven and configured, taken from the Sifive FE310-G002 Manual[4]

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 crystal 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 400 MHz.

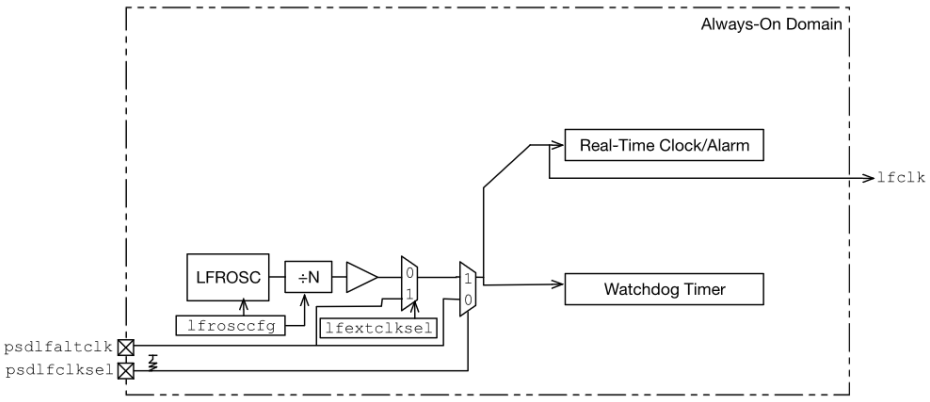

Figure 3.4: The low frequency clock generation scheme, specifying how the low frequency clock is driven and configured, taken from the Sifive FE310-G002 Manual[4]

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 real-time 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 constant 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.

3.4 Traps

3.4.1 Configuration

When an enabled trap is raised, the pc is set to the trap vector according to the trap mode. The trap vector and mode are stored in the 32 bit csr *mtvec*. In this csr, bits [1:0] are used to store the mode, where 0 is direct mode and 1 is vectored mode. Bits [31:2] represent vector base address, which is the top 30 bits of a 32 bit address. Since this is stored as 30 bits, two zeros are added to pad the value. This means that the trap handler address must be aligned to 64 bytes, which can be done simply using the .align pseudo-instruction. When in direct mode, a trap will set the pc to the base vector address, whereas in vectored mode only synchronous traps are set to the base address, and asynchronous traps set the pc to the base address added to four times the trap code. For this system the vectored mode was chosen. This was because the asynchronous traps are more frequent than synchronous traps, and by jumping to the specific trap call directly a large amount of overhead is skipped. This is relevant for functions like the preemptive scheduler, as it reduces the time to switch processes, which will allow the preempter to be more effective with short quantum's.

3.4.2 Asynchronous Traps

Due the use of vectored trap mode, instead of having one single handler for asynchronous traps, each interrupt has it's own handler. Since the Hifive1 does not implement a supervisor mode, there are only 3 available interrupt types, software, timer and external. The software interrupts are not used, as they are mainly used for communication between processes running in parallel, and the Hifive1 only has a single core, so this can be reserved for another use. The machine timer is used for the preemptive scheduler, which will be explored in detail in further sections. Finally the external interrupts, the handler will read from the CLINT claim/-complete register. This will provide the handler with the highest priority interrupt code, which will then be used in a jump table to execute the correct interrupt, and once completed the same value is written back to the claim/complete register to indicate that the interrupt has been handled. The interrupts implemented in this will be explored in further sections.

3.4.3 Synchronous Traps

The trap code for synchronous traps can be fetched from the *mcause* register. This can then be used with a jump table to handle the correct trap. Synchronous traps fall into two main categories; environment calls and exceptions. For exceptions, most function by outputting an error message, and then depending on the severity of the exception either halting execution or returning execution to the cause in the case of a non fatal error.

Environment Calls

Environment calls function using the standard calling convention, where the value in a7 is used to determine the desired environment call, and a0-6 are used as arguments like a standard function. Since the board implements user mode and machine mode, there are two separate codes for environment calls from user mode and machine mode however for most cases these can be combined.

3.5 Scheduler

This system will be implemented with an interactive scheduler. The functions that the scheduler must perform are storing all the processes that are in the ready state, running ready processes, and preempting processes that have been running for longer than the quantum, which is the amount of time given to the process. The scheduler will be ran mainly through the use of interrupts.

3.5.1 Creating a Process

On boot, a table of process entries is allocated in memory, and parts of each are initialized, such as each processes address space and their stack and global pointers. The scheduler will then store the pointer to each entry on a stack which is used to distribute a process entry to each new process. When a new process is created, the pointer on the top of the stack is used to populate that entry with information about the new process, which consists of a pointer to its text section, the size of the text section, the initial pc (which by default is the start of its text section), and a sequentially generated process id. Once the process is initialized, it is then added to a queue in the scheduler, which is implemented as a circular queue.

3.5.2 Running a process

To make a process run, the scheduler will first dequeue from the ready processes queue. This will then be used to set the processes state. The stored pc will be used to set the *mepc* csr, which stores the value of the pc to return to when mret is called, as the mret instruction restores the previous privilege mode and sets the pc to the value in *mepc*. The physical memory protection (PMP) is then set to allow access to that processes' address space, as well as allowing read access to that process text section in the Flash memory. The state of the process in the process entry is changed to running. Then most of the registers have their values set, except for t1-4. This is because before these values are set, the scheduler sets the quantum for the process by setting the timer comparison to the current time plus the quantum, and this requires a minimum of 4 registers to do. This is done as close to the start of execution instead of before the registers have their state set, as this ensures the time allowed for each process is as close to the specified quantum as possible. Finally the last 4 registers are set and MRET is called, which begins the execution of the process. The process will then execute until it is stopped, which can happen in five different ways. The first three is where execution is handed back to the scheduler, through either yielding, termination, blocking, or preemption. The final way is that a process could cause an error which could either halt execution, return control to the application or cause it to be terminated.

3.5.3 Preemption and Scheduling

If either a yield or a termination occurred control will return to the scheduler without preemption. Each process is given a 50 millisecond quantum, after which it is preempted and control is given to the scheduler. This time includes execution of user code as well as environment calls, however a process cannot be preempted in the middle of an environment call, so if a process reaches the end of its execution time during an environment call, it will be preempted as soon as the system returns to the user code. As previously mentioned, the preemption is timed using the machine timer interrupts. The machine timer was chosen over the timer interrupt generated by the AON unit, as the latter interrupt is part of the external interrupts so there is an increased amount of overhead required for it to be processed. Using the machine timer interrupt also addresses another problem, which is maintaining the state of the process. When the preemption occurs, the first thing that must happen is that the processes registers must be saved to the process entry, which requires the global pointer. The global pointer is a constant value, as it always

points to top of the machine address space, however to use it, it must be stored into a register, which requires that registers value to be store to the machine stack. For this we use the *mscratch* csr to store the machine pointer during normal execution, which when in machine mode we swap with the current processes stack pointer. This allows us to use the machine stack while also storing the processes stack. In addition to the global pointer we also save the value in t0 and t1 to store the process entry pointer, and to pop and store the values back from the stack and the *mscratch* csr, before saving each of the remaining 26 registers. The operations of the preempter occur as follows:

1. The PMP is reset, clearing all user memory access
2. The fence.i instruction is called, which invalidates all entries in the instruction cache
3. The process sp is switched with the machine sp in the *mscratch* csr
4. Registers gp, t0, and t1 are pushed to the stack
5. The global pointer is loaded into gp, and used to load the current process entry pointer into t0
6. The 3 values pushed to the stack are each pop into t1 and then stored into the process entry, as well the the value that was swapped into the *mscratch* csr
7. All 26 remaining registers are stored to the process entry
8. The value stored in the *mepc* csr is loaded and stored
9. If the process had the running state, it is changed to ready, otherwise it remains unchanged.
10. If the status is terminated, add the process entry pointer to the free entry stack
11. Else add the pointer to the process queue

Blocking and Yielding

Scheduling ready processes and terminated processes is intuitive, however when a process is blocked, the scheduler is instructed to ignore that process. This is because it is the job of the blocking source to store its blocked processes. This reduces the complexity of the scheduler and gives more freedom as to how a process in unblocked.
When a process is yields or is blocked, the scheduler is requested to schedule a new process. One method of doing this would be to call the scheduler directly from whatever environment call or interrupt caused the yield or block. However this could create an inconsistent state, as the scheduler preempter is always called from an interrupt to user code. Because of this, when a process yields or is blocked, it simply zeros the timer compare, so that when it returns to the user code it will immediately raise an exception, allowing the scheduler to handle the process as normal.

3.6 Memory Management

3.6.1 Address Spaces

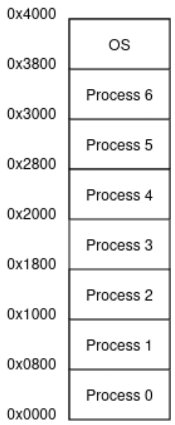

Figure 3.5: The sections of the DTIM serving as memory, with the address offsets from 0x80000000

For this project, RISC-V's PMP was used to implement and enforce address spaces. The Hifive1 has two forms of memory, a 512 MiB flash memory to serve

as ROM and an 16 KiB L1 data tightly integrated memory (DTIM) which served as RAM. Both of these sections allow for execution of code. Because of this and the limited amount of RAM, the text section of each program will be stored on and executed from the flash memory. The 16KiB of RAM is split into 8 address spaces, making each space 2KiB. This size was chosen as a balance between number of available processes and amount of memory available to each process, as well as ensuring that the space needed to store information of the processes did not take up the majority of the machine address space. This is because since each process entry requires 35 words or 140 bytes, so if split into 8 address spaces there will be 7 available processes, which would take 980 bytes. So with only 7 processes, the process entry table would already take 48% of the machine address space. This could be adjusted by keeping the machine address space at 2KiB while decreasing the address space size to each process, however it is useful to have the address spaces be in powers of 2, as will be discussed later with the PMP. If equal sized spaces were maintained but split into 16 1 KiB spaces, the 15 process entries would need 2100 bytes, which is over 200% of what is available. If we were to increase the size of the spaces to 4 KiB, that would only make 4 address spaces, with only 3 available processes, which is too small to be practical.

3.6.2 Address Space Layout

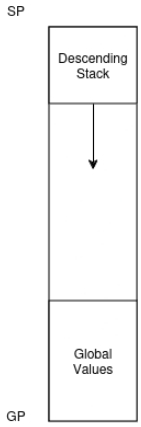

Figure 3.6: The structure of each of the seven process address spaces, featuring the stack and globals

Each 2 KiB address space has 2 pointers. The global pointer, gp, points to the bottom of the address space, and is used to store global variables for a process, so should not change unless a program chooses to use the gp register for another purpose. The stack pointer, sp, initially points to the word above the top of the address space. This is used to implement a full descending stack. Sp will always store the pointer to the last item added to the stack, so when the stack is empty it will point to the word outside of the stack.

3.6.3 PMP Use

To enforce the address spaces, RISC-V's PMP is used. PMP allows us to specify sections of code to have have restricted access to calls from user mode. This is done by setting the values in the *pmpaddr* and *pmpcfg* CSRs. It can be done in two ways, either the top of range setting which protects the region between a specified top address and either the previously referenced address, or 0x00000000. This allows a precise range however it would require storing two values in each process for its text section. The other method is to specify an address which is naturally aligned to a power of two. This means that the address is a multiple of that power of two. The value of the power of two is specified by using the trailing zero bits of the naturally aligned address, where all but the most significant zero bit is set to one. The configuration for each is stored as one byte, with 5 fields, R, W, X, A, and L. R, X, and W control the read, write and execution access in each region, A controls whether that control region is active and the method which the region is specified, and L will lock the settings and also enforce the settings to both user mode and machine mode until reset. When L is not set, code executed in machine mode will ignore the PMP settings.
These addresses are stored in the CSRs *pmpaddr0* to *pmpaddr7*, which means that the Hifive1 implements up to 8 PMP controlled regions. The configuration of each of these addresses is stored in the CSRs *pmpcfg0* and *pmpcfg1*, which each control *pmpaddr0-3* and *pmpaddr4-7*, respectively.

Since the address spaces in RAM are constant the pmpaddr value can be generated by setting address[8] low and address[7:0] high. The text section of a process has a variable size, which is stored as the value n, where the size of the section is 2^n words. There are then two cases to consider when implementing this. When $n == 0$ it represents the case when the size of the section is only one word. Since this will always result in a naturally aligned address, there is a separate value of A to represent this NA4 case, so all that is required for this case is to set A to 2,

and specify the word address. The general case where $n > 0$ works as previously specified.

3.7 IO

The Hifive1 has few forms of IO available. Most forms of IO are driven through the general purpose input/output controller (GPIO). This controls the function of each of the Arduino-compatible pins, as well as up to two alternate functions for each GPIO pin, which is used to control the universal asynchronous receiver/-transmitter (UART), serial peripheral interface (SPI), and pulse width modulator (PWM), as well as three LED's featured on the board.

3.7.1 LEDs

The simplest form of IO on the Hifive1 are the three LED's, which can be used as simple indicators, however they utilize the GPIO just as the other IO functions do. The green, red, and blue LED's use GPIO pins 19, 21, and 22 respectively. To use them, the bits 19, 21, and 22 in the GPIO output enable register, memory mapped to 0x1001_2008, must be set low, since the GPIO uses low enable. This enables the LED's to function, which can turned on and off by setting the bits in the GPIO output register similarly to how they were set for the enable register.

3.7.2 Text input and output

The main way a user may interact with a program in this system will be through the use of a terminal, which requires the need to print and receive text. To do this, we will utilize one of the two UART peripherals. This will allow us to send and receive text from a host machine that is connected via the USB port.

Host machine

When the Hifive1 is connected to a host machine, the two UART peripherals are represented as two ttyACM devices, which can be interfaced with using the two files /dev/ttyACM0 and /dev/ttyACM1. To fetch and send data, the terminal used must be configured to function at the target baud rate.

UART configuration

To enable the use of UART, the specific peripheral must be enabled through the GPIO. To do, bits 16 and 17 of the iof_en register must be set to 1, and same bits of iof_sel should be set to 0. This toggles GPIO_16 and GPIO_17 from software control to hardware control from the UART0 peripheral. After this, UART0 must be configured. The first setting to alter is the frequency divider, which is used to determine the baud rate of the peripheral from the high frequency clock. This uses the formula

$$f_{baud} = \frac{f_{hfclk}}{div + 1}$$

. The target baud rate will be 115200 Hz. Using the hfclock frequency of 16 MHz, we can deduce that the required divisor will be 138, which give us the actual baud rate of 115107 Hz, which has an error 0.08%. Other settings include setting the rx_en and tx_en to high, to enable both reading and transmitting. Other configurations include the watermark level, which will be discussed in a later section.

Transmitting

The transmission of data from user is interrupt driven. To begin a transaction, a process will make a environment call, passing it a pointer to the first character of a zero terminated string as an argument. First the calling process is blocked, and that process is queued onto a queue structure, which stores all processes that are currently blocked due to an ongoing request to transmit. The string pointer is then queued onto a second queue, with all the processes strings. It then checks if the transmission interrupts are already enabled due to other requests, and enables them if not, before returning from the environment call. The transmission interrupts function of the watermark value defined in tx_en in combination with the transmission queue. The UART has a queue that stores up to eight bytes during transmission. By setting the transmission watermark to one, it will raise an interrupt whenever the the queue has less than one element in it. When this is enabled, the initial interrupt can be used to add to this queue until it is full, and then return to the calling environment. The UART will then transmit this data, and once the queue is empty again, another interrupt will be raised and the queue will be filled again. This will continue until the end of the string is reached, at which point the interrupt will dequeue from the blocked process queue and unblock that process. It will then determine if there is another string to print, which if there is it will repeat the same process, and if there is not it will disable the transmission interrupts.

Receiving

Unlike the transmission, receiving is done using programmable IO. This is done because only one stream of serial IO is available, because although there are two UART units on the board, UART1 is reserved for use by the wireless module, so only UART0 is available for general use. If the input was interrupt based, it would mean it would allow an output to interrupt a users command line while they were typing. This could be avoided by using a more complex interaction system, or by developing a more complex terminal program on the host computer. Because of this, when a process asks to receive information, it makes an environment call which takes a pointer to a buffer in memory as an argument, and will receive characters from UART0 until it receives a zero byte, at which point control is return from the environment. In future this should also be done in a similar method to transmission.

Chapter 4

Evaluation

In this section, I will discuss the capabilities of the designed operating system with comparisons to the original concepts it was implementing to evaluate its effectiveness and completeness.

4.1 Challenges

Since this project is hardware based, there are several complications when it comes to testing that do not occur in standard software. This is because the tools available to validate programs are less available, which results on a far greater reliance on manual checking as opposed to automated testing. This problem is made worse as many of the methods of checking for correct execution are reliant on standard IO, which is not available to us as the IO itself is being developed.
Another challenge is the problem of developing with interrupts that are triggered by uncontrollable events, or events that need to function at a high frequency. These include timer interrupts, and external interrupts that function asynchronously like the UART. These limit the functionality of breakpoints, as often the asynchronous behavior can continue while execution of the rest of the processor is paused, which puts the system into a state that would not occur during normal execution.

4.2 Disassembly

The first step to ensuring code is correct was to ensure that the assembled code was assembled and linked correctly. To facilitate this, on assembly the makefile was configured to disassemble the generated executable, using objdump from the

riscv-gnu-toolchain. This enabled me to ensure that the structure of each section was maintained, which is important as many parts of the system require memory alignment, for example in the jump tables used in the environment handler and the exception handler require the branch instructions to be word aligned, and that the spacing between entries is correct. The decompilation was also vital when developing the toolchain, as it allowed me to validate that the system would be loaded correctly between Flash and the DTIM.

4.3 Breakpoints and Inspection

Of the tools available to debug, one of the most powerful tools is the JTAG interface. The JTAG interface allows an external connection to access the debug CSRs of the system, which allows a host machine to control the execution and memory state of the board, through the use of breakpoints and step through execution and memory commands. To utilize this, I used GDB to load code onto the board, at which point GDB could interface via JTAG to create a similar debug environment on the host machine to the debug environment of a standard program. This allowed the use of up to eight breakpoints, the limited number due to the requirement to use the hardware breakpoints. The ability to inspect and modify memory directly was very powerful in both initial experimentation and later debugging. Since most of the system's functions are memory mapped, it allows direct inspection of the boards configuration and many IO functions, which gives a much greater insight into their function than the documentation as the information on the board is by definition fully accurate. Some examples of information is the boot state of the board, as the state of many registers is often specified as 'X', which means the actual value is determined by the implementation, so it is unknown without inspection. For debugging, memory inspection allowed me to manual verify the execution of the system. The manual verification was done in two ways; comparing data to 'golden data', and comparing observed behavior with expected behavior. For sections which need to produce a specific output I can calculate the result before execution and compare the generated result with that, as well as observing all intermediate values. An example of this is when calculating the encoding for the PMP, so the address indicates the correct NAPOT range. This can then be followed with observing the following behavior. In the case of the PMP range, I can use text sections of sizes that approach or exceed the size limit specified, and check that the process is or is not allowed to fetch instructions from that area. This method is especially important when testing asynchronous

operations like timer interrupts, as a step by step approach does not function, so it is often to require the function to produce an additional output like an LED, so that the behavior of the function can be observed in a real time environment.
At an application level, once IO had been developed it becomes possible to debug programs more simply by comparing the output of the program through prints allowing simple access to the results of a program, however this does not replace the more thorough inspection as a program could get the correct result while the underlying system was malfunctioning.

Chapter 5

Conclusions and further work

This chapter will give an overview and critique of the result of the project, and evaluation of the aims and achievements, and some concepts that could be explored for future work.

5.1 Summary of Work

This project set out to design a simple operating system. This required extensive research into the operations of the Hifive1, and the RISC-V architecture. The final result successfully featured an interactive scheduler, that was capable of managing the state of up to seven processes at a time. This success was limited by the small memory space limiting the number of processes that could actually be limited, and the limited amount of IO reduced the interactivity of the system. The memory was split into eight equal sections, each of which had access protected by the physical memory protection. This was a fixed system, as a dynamic approach with such a small memory space would risk individual processes easily choking the entire system. While this limited the number of processes, this was a good compromise as each process required a large amount of system memory to store, so by limiting the number of processes I increased the resources available to each process. In terms of IO, a simple command line was implemented using the UART to allow a host machine to interact with the board through text. This was implemented though interrupts, which allowed non-caller processes to continue execution during the transmission. However the receiver part of this still blocked all execution, to avoid confusion of the user as to which process was taking input. To improve on this a more complex handler that uses a claiming system could be

implemented. Also implemented was the the use of the LEDs, however the use of the LEDs is not made exclusive so the output of the LEDs could be ambiguous. Overall the main aims of the project have been achieved, having produced a working albeit basic operating system for a bare metal system.

5.2 Future Work

To improve the experience of the user with the board, it would be important to implement a more comprehensive IO, through the use of external components making using of the GPIO pins. This would allow more interaction with the board directly, instead of through a host machine. It would be interesting to make greater use of the atomics extension on the Hifive1 to implement more interprocess communication. In terms of scheduling it would be possible to implement alternate algorithms for the interactive scheduler than the round robin approach, for example to use a lottery system to randomly pick processes to schedule, which has the possibility to allow certain processes to be scheduled more often if more lottery tickets are allocated to it. To provide a better evaluation more performance monitoring could help provide a good comparison against other systems, and basic use of the system could be improved through the implementation of simple system software, for instance a software process manager or viewer, or a more standard command line.

Bibliography

[1] Herbert Bos Andrew S. Tanenbaum. *Modern Operating Systems 4th Edition*. Pearson, 2015.

[2] About risc-v. `https://riscv.org/about/`. Accessed: 2024-05-10.

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

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

[5] *Hifive1 Rev B Schematics*. Sifive Inc, 2021.

[6] *RISC-V ABIs Specification, Version 1.0*. RISC-V International, November 2022.