Teaching Operating Systems: A Ground Up Approach

Maximillian Sonderegger

College of Information and Computer Sciences

msonderegger@umass.edu

5/21/23

Advisor: Tim Richards

Second Committee Member: Joe Chiu

Research Type: Thesis

1. Introduction

Operating systems are a fundamental part of modern computing but current courses are not properly preparing students to understand and work on them. As computers have become more ubiquitous and advances in virtualization have continued to abstract users further and further from the hardware they are using, operating system courses have followed suit, focusing on the concepts that are crucial to modern operating systems without showing how they are implemented in practice.

While understanding concepts like virtual memory, concurrency, and file IO is important, operating system design is inseparable from hardware interfaces, so it is equally important to understand how operating system designers work with hardware to achieve these goals. By teaching concepts without practical implementation students are left confused and unprepared for operating system development, because while they have learned what an operating system does, they have not learned how it does it, which is the ostensible goal of an operating system class.

My thesis aims to bridge this gap by creating a simple, modular, and extensible operating system based on the Xv6 educational operating system designed by MIT for the RISCV architecture. In conjunction with this I will be creating a course and preparing reference materials that will help students understand the RISCV architecture and instruction set. Together these two resources will let students build their own operating system from scratch that could run on a RISCV chip. By doing this they will achieve a deeper understanding of what it means to design an operating system and the complexities that arise from working with real hardware interfaces.

The main focus of my research will be to design the operating system, and design it in such a way that it can be easily configured. In order to do so I have chosen the RISCV architecture as the target for development. The advantages of using RISCV are its small base instruction set, its open instruction set architecture, and its simplicity (looking at you x86). To run my operating system I will be using QEMU, a popular emulation software that will allow me to run the operating system on any computer and easily use debugging tools like GDB to inspect what it is doing and determine the cause of bugs that arise in the development process. This operating system will be written in a combination of assembly and C. Understanding assembly is necessary to understanding the connection between software and hardware and any operating system course would be remiss if it did not teach students how to read and write assembly code. However, assembly is difficult to program in because it is prone to bugs and hard to understand. Therefore, the operating system will largely be written in a higher level language, namely C. C has many advantages over other similar languages like C++ and Rust because it is simple and it is easy to see how it compiles down to equivalent assembly. It is also very common and potential students will very likely have already been exposed to it before.

In order to design this operating system so that it can be simple, understandable, and extensible, the following guidelines have been laid out. They may be subject to change as development continues.

1. Every function should have a comment. Every section in a function should have an explanation of what it's doing.
2. Functions should be as simple as possible and no longer than 60 lines of code.
3. A single file should have no more than 10 functions.
4. Directories will be used to group files with similar functions together.
5. Function names should avoid abbreviations where at all possible.
6. Braces will always be used with conditional and looping structures. If a statement is small it may be included on the same line with braces surrounding it.
7. Macro usage should be limited to #defines where at all possible. Macros must be used if there is a magic number (e.g. max number of processes, memory max, PTE VPN extraction etc.)
8. A README shall exist and be maintained through out the development process.
9. The course shall be accessible to as many people as possible. In particular it will not rely on any paid products.

Finally the operating system will be compiled through makefiles to simplify the distribution and compilation.

If time allows I plan to research ways to simplify the development environment by creating a web based interface to QEMU and GDB. GDB already provides a programing interface through the GDB Machine Interface (GDB/MI), allowing a developer to programmatically inject commands into a running GDB process. There are numerous bindings for this API in various languages, particularly Python through, pygdbmi. Additionally there is an existing tool called gdbgui that creates a web interface to pygdbmi allowing a developer to easily read values and execute gdb commands from a browser of their choice. It runs locally so it is a perfect candidate for creating a simple development environment that a student could use. I plan to research how to connect gdbgui to a running QEMU process to allow easy debugging of an emulated process.

2. Significance

Operating systems are undoubtedly a crucial part of modern computing. Without them we would be unable to multitask, access files, and connect to the network. In fact recent developments in supervisor and hypervisor technology can rightly be viewed as operating system development as well, since they provide the same kind of abstractions to operating systems themselves. These developments have enabled the rise of virtualization and cloud computing that has defined the past decade of computing developments. In short, the modern computing world would be impossible without operating systems.

Clearly there is a need for intelligent and competent operating system designers to maintain the operating systems we rely on for daily life and to develop the operating systems that will make tomorrow possible. Systems classes do exist and are part of most modern computer science curriculums, however they frequently prefer to teach operating systems concepts from a high level without allowing students to actually interact with and develop their own operating system. This is quite understandable as the importance of quick and reliable operating systems has led to extensive research and modern operating systems are indeed complex. The current Linux code base has over 26 million lines of code (Torvalds). This level of complexity is daunting to professors and students alike. It would be impossible to teach all of the Linux operating system in a single course (or even ten!) and so modern operating system classes have moved to teaching operating systems from a high level, focusing on concepts, like virtual memory, process management, file IO, without teaching how these are actually achieved in practice.

This is a significant failing. While those concepts are undoubtedly important and ought to be taught, on their own they are not enough. Operating systems are fundamentally grounded in hardware techniques and understanding how they work in concert to achieve those concepts is just as vital to understanding how an operating system works and how it is developed. Indeed for some students, hiding trap tables or page tables within a black box of scheduling algorithms and memory mapping, makes it harder to understand. This research seeks to bridge this gap by creating an easy to teach course and an easy to modify operating system and that will introduce students to operating system design principles through practice rather than theory and teach operating systems from the bottom up.

This is significant because while educational operating systems like Xv6 do exist, they are often hard to understand because have been written by operating system experts, they have few comments, and the comments present are unhelpful. This research advances the field in two crucial aspects. First and foremost it will improve the quality of instruction for students at the University of Massachusetts Amherst and beyond by providing them with a high quality education and real world experience in operating system design. This will ensure that there are competent developers who are capable of creating innovations in the operating system design world and putting it into practice in the real world.

Additionally it will provide an important starting place for further research into RISCV operating system design. RISCV is a new and still developing CPU ISA that continues to see more and more adoption in the embedded world but relatively less adoption in the traditional computing world. This is in part due to the fact that there is a lack of high quality operating systems designed for the RISCV ISA. By innovating and improving on an existing RISCV operating system this research will encourage adoption of an ISA that allows for both energy efficient and high speed computation. Additionally, by creating an easy to use development environment this research will aid the future development of RISCV operating systems as well.

3. Background

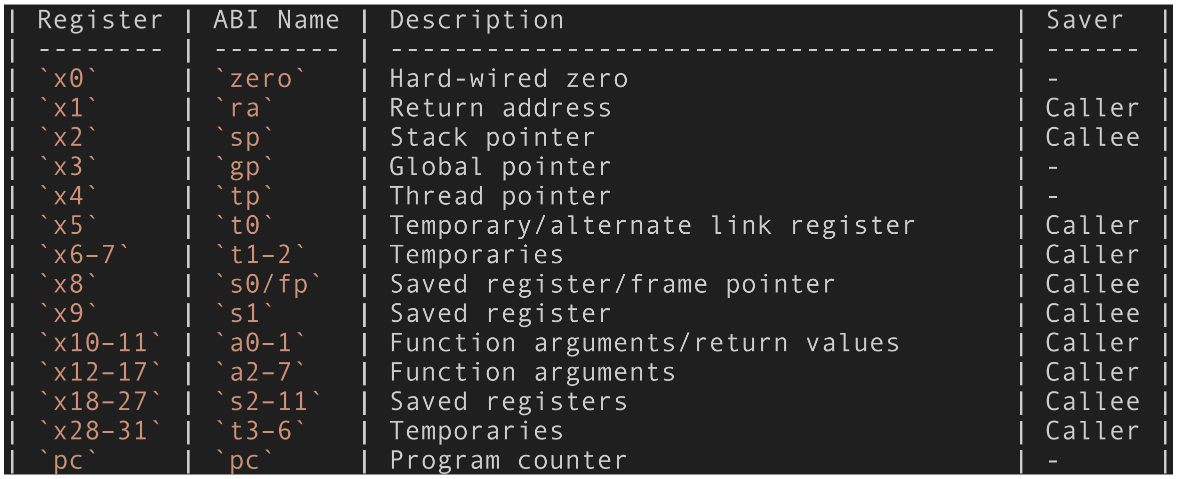
My preliminary research focused on understanding and compiling the existing RISCV literature into a comprehensible form. I began by reading the ISA documents. They are broken into two volumes for readability. The first volume contains the non privileged instructions as well as a brief intro to the model of computing that the RISCV ISA assumes. This volume contains the unprivileged instructions that an application level programmer will use most often including branches, jumps, and arithmetic operations. The second volume contains the privileged instructions necessary for operating system development, including instructions for setting the control and status registers (CSRs) which allow you to configure various operating parameters of the CPU, for example, the trap table, interrupt behavior, and page table.

The unprivileged manual lays out the standard execution model for the RISCV ISA. Conceptually it is comprised of one or more RISCV *cores* which may have one or more hardware threads or *harts*. A core is defined as a core with an independent instruction fetch unit. Interestingly the specification allows for additional, non RISCV components to be part of the system. A *harts* is similar to the conceptual model of a software thread except it represents the component itself, rather than a single running program. *A* single RISCV core may provide one or more *harts*.

The unprivileged manual also describes the execution environment interface (EEI) which exists outside the conceptual computing model and defines the resources available to the CPU, including number and kind of cores available, available privilege modes, address spaces, legal instruction behavior, and the handling of interrupts and exceptions. An EEI is then implemented by a specific execution environment implementation. This separation allows the spec to include support for virtualization techniques like emulation, hypervisors, and supervisors from the very beginning by separating the execution environment definition from the actual execution. An example of an EEI is the Linux ABI or the separately defined RISCV supervisor ABI. Some examples of different EEI implementations include, a bare metal machine, a QEMU emulation, and a hypervisor multiplexing user harts onto physical cores. This focus on abstraction and emulation is extremely interesting given the trend towards cloud computing in the past decade. However a full exploration of RISCV support for virtualization is out of the scope of this research.

So far I have been describing "the" RISCV ISA. In reality it is actually a family of related ISAs that define differing numbers and widths of registers. RV32I means RISCV with 32 general purpose registers and a program counter register, each 32 bits wide and a 64 bit addressable memory space. Similarly RV64I and RV128I define 32 registers and a program counter with widths of 64 and 128 and 64 bit and 128 bit addressable memory spaces respectively. RV128I is not fully specified yet but is intended to future proof the ISA if address spaces larger than 64 bit are ever necessary. In addition, RV32E defines an ISA with 32 bit wide registers, similar to RV32I, but with only half as many registers for a total of 15. Xv6 is written for RV64I so I will be focusing my research on RV64I. The additional complexity involved with a larger address space can be resolved by simply ignoring the upper bits of pointers effectively shrinking the memory available to the system, as is done in Xv6 and many other major operating systems. Although there are in fact multiple RISCV ISAs, they are all largely the same, with the RV64I and RV128I supporting all of the opcodes of RV32I with a few additional versions of existing instructions to enable efficient processing of 32bit, 64 bit and 128 bit values in the case of RV128I. These are prefixed with W for 32 bit (word), D for 64 bit (double), and Q for 128 bit (quad). For convenience sake, in the rest of this paper I will refer to the collection of RISCV ISAs simply as the RISCV ISA.

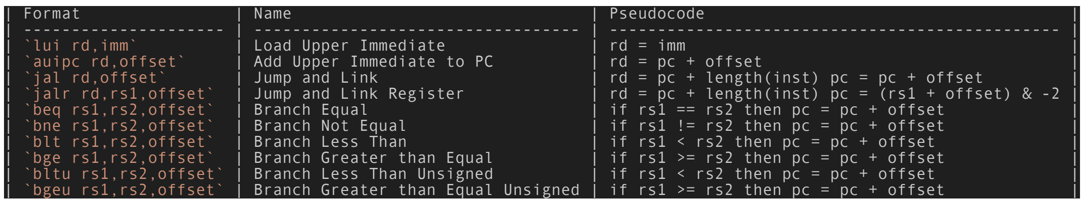
The standard calling convention, shown below defines specific registers for certain common uses, however it is not enforced by any part of the ISA. An interesting aspect of this ISA is that because there is no dedicated stack pointer register, there is also no dedicated stack operation instructions like push and pop, commonly found in modern CISC ISAs. Instead a developer can combine a subtraction instruction to manually decrement the stack pointer and then save variables as necessary. While this may seem like a more complicated and error prone way of achieving the same thing, in practice assembly is almost always written by compilers, which have no problem keeping track of the location of variables. It also aligns with the RISC philosophy of providing only the minimally necessary tools and letting the compiler and the hardware optimize the rest. Another interesting quirk is the x0 register which always holds all 0s. As any programmer will know, programs frequently need to use the value 0 so having a constant source of 0 increases the efficiency of a program (Riasanovsky 5), rather than constantly having to load it into a register. One final note on the general calling convention is the alternate link register. The RISCV ISA provides this register to allow compilers to optimize tail calling functions, or calling functions at the end of call stack, where setting up a full stack may not be fully necessary. The RISCV ISA does not explicitly describe how this should be done but does provide the necessary architecture to do so.



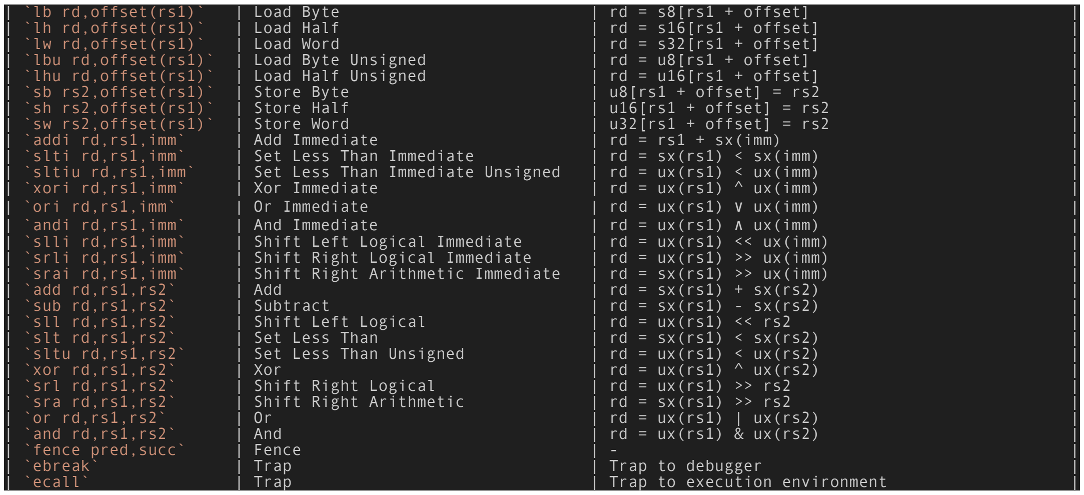
Register	ABI Name	Description	Saver
`x0`	`zero`	Hard-wired zero	-
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	-
`x4`	`tp`	Thread pointer	-
`x5`	`t0`	Temporary/alternate link register	Caller
`x6-7`	`t1-2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-11`	`a0-1`	Function arguments/return values	Caller
`x12-17`	`a2-7`	Function arguments	Caller
`x18-27`	`s2-11`	Saved registers	Callee
`x28-31`	`t3-6`	Temporaries	Caller
`pc`	`pc`	Program counter	-

One of the stated goals of the RISCV ISA is to minimize the instruction set and hardware required in the specification. This gives hardware designers more freedom to implement things as they see fit. This has two main advantages. First, optimizations are easier to design and include as they have very little behaviors to conform to. Second, extremely simple designs, aiming for low power or cheap manufacturing cost, for example in embedded environments, are free to leave out large portions of the ISA. This allows them to customize their necessary chip design to a very high degree of specificity. In fact it is even possible for a fully compliant chip to have no integer multiply command. This reduces the amount of complexity and therefore silicon needed to implement a fully compliant chip, speeding up development and production, while keeping costs and energy consumption low. The full base instruction set is listed below along with a brief description of what it does.

Regardless of the size of the registers (and thus the size of the architecture), all instructions are 32 bits wide. Most instructions take 3 arguments, a destination register, where the result will be stored, and up to two source registers that hold the arguments for the specified operation. These three registers need not be unique. Some instructions operate on immediate values, values stored within the 32 bits of the instruction itself. The width of the immediate varies. The RISCV ISA is a load/store architecture so it does not provide instructions that operate directly on memory other than a load and store command. The ISA mandates that every value on stored in a register MUST be sign extendedEvery instruction is assigned a one of several different instruction types based on the number and type of arguments it requires. Each type defines the layout of the bits in each instruction. A notable design feature of the instruction types is that any instructions that operate on immediate values always encode the most significant bit of the immediate value in the most significant bit of the instruction and the location of the register identifiers is consistent throughout all instruction types. This allows for sign extensions and register allocation to be done in parallel to the decoding of the instructions. This simplifies the design of the hardware and speeds up execution. Another notable quirk of the ISA is that all immediates are stored in sign extended two's complement format. They are all sign extended to 32 bits before being operated on. For example the ANDI instruction takes a 12 bit immediate and ANDs it with the source register. ANDI rs1, rs2, 0x800 would be sign extended and store rs2 && 0xFFFF_F800 in rs1. This may create some difficulty operating on unsigned integers, however the instruction manual observes that "In practice, most immediates are either small or require all [32] bits" (Waterman and Asanovic 16), so this is not common and compilers can easily address this if necessary. In fact the ISA provides documentation for a frequently implemented compiler pseudo operation that opaquely handles the complexities of loading immediates. The full RV32I base instruction set is shown below at only 40 instructions.



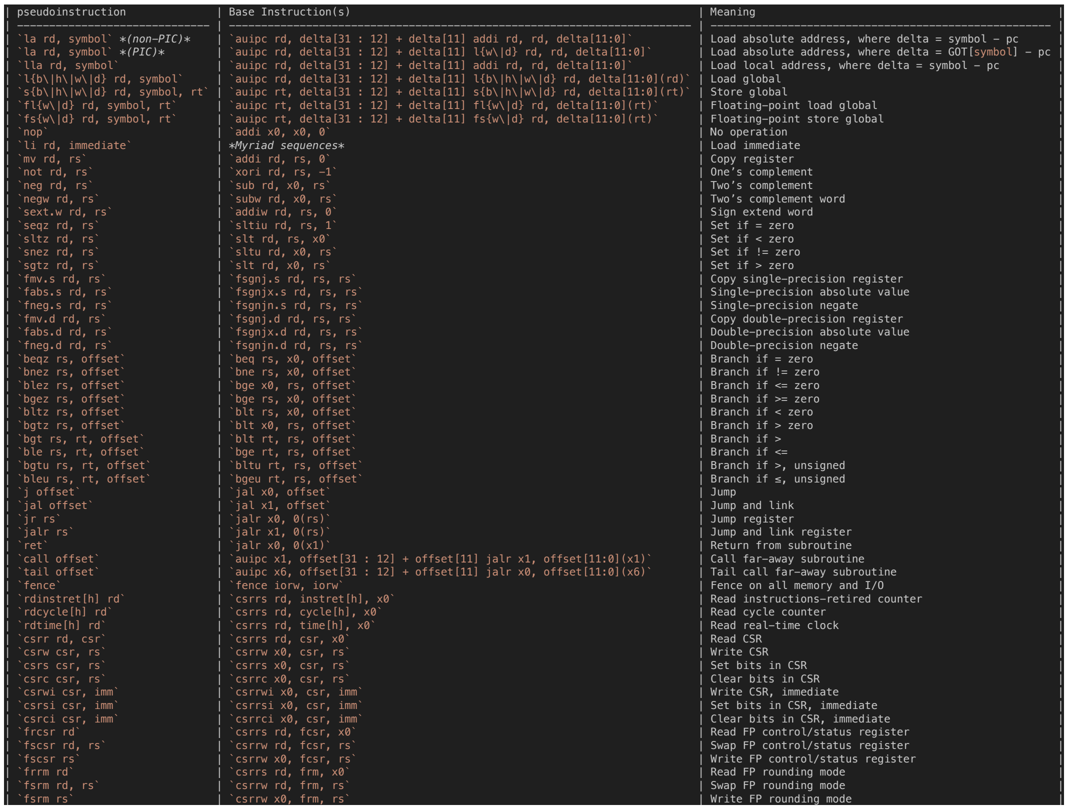
```
| Format                | Name                              | Pseudocode                                     |
| --------------------- | --------------------------------- | ---------------------------------------------- |
| `lui rd,imm`          | Load Upper Immediate              | rd = imm                                       |
| `auipc rd,offset`     | Add Upper Immediate to PC         | rd = pc + offset                               |
| `jal rd,offset`       | Jump and Link                     | rd = pc + length(inst) pc = pc + offset        |
| `jalr rd,rs1,offset`  | Jump and Link Register            | rd = pc + length(inst) pc = (rs1 + offset) & -2 |
| `beq rs1,rs2,offset`  | Branch Equal                      | if rs1 == rs2 then pc = pc + offset            |
| `bne rs1,rs2,offset`  | Branch Not Equal                  | if rs1 != rs2 then pc = pc + offset            |
| `blt rs1,rs2,offset`  | Branch Less Than                  | if rs1 < rs2 then pc = pc + offset             |
| `bge rs1,rs2,offset`  | Branch Greater than Equal         | if rs1 >= rs2 then pc = pc + offset            |
| `bltu rs1,rs2,offset` | Branch Less Than Unsigned         | if rs1 < rs2 then pc = pc + offset             |
| `bgeu rs1,rs2,offset` | Branch Greater than Equal Unsigned | if rs1 >= rs2 then pc = pc + offset           |
```



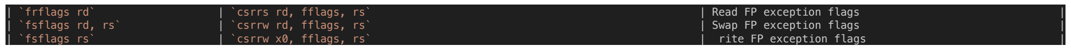
`lb rd,offset(rs1)`	Load Byte	rd = s8[rs1 + offset]
`lh rd,offset(rs1)`	Load Half	rd = s16[rs1 + offset]
`lw rd,offset(rs1)`	Load Word	rd = s32[rs1 + offset]
`lbu rd,offset(rs1)`	Load Byte Unsigned	rd = u8[rs1 + offset]
`lhu rd,offset(rs1)`	Load Half Unsigned	rd = u16[rs1 + offset]
`sb rs2,offset(rs1)`	Store Byte	u8[rs1 + offset] = rs2
`sh rs2,offset(rs1)`	Store Half	u16[rs1 + offset] = rs2
`sw rs2,offset(rs1)`	Store Word	u32[rs1 + offset] = rs2
`addi rd,rs1,imm`	Add Immediate	rd = rs1 + sx(imm)
`slti rd,rs1,imm`	Set Less Than Immediate	rd = sx(rs1) < sx(imm)
`sltiu rd,rs1,imm`	Set Less Than Immediate Unsigned	rd = ux(rs1) < ux(imm)
`xori rd,rs1,imm`	Xor Immediate	rd = ux(rs1) ^ ux(imm)
`ori rd,rs1,imm`	Or Immediate	rd = ux(rs1) ∨ ux(imm)
`andi rd,rs1,imm`	And Immediate	rd = ux(rs1) ∧ ux(imm)
`slli rd,rs1,imm`	Shift Left Logical Immediate	rd = ux(rs1) << ux(imm)
`srli rd,rs1,imm`	Shift Right Logical Immediate	rd = ux(rs1) >> ux(imm)
`srai rd,rs1,imm`	Shift Right Arithmetic Immediate	rd = sx(rs1) >> ux(imm)
`add rd,rs1,rs2`	Add	rd = sx(rs1) + sx(rs2)
`sub rd,rs1,rs2`	Subtract	rd = sx(rs1) - sx(rs2)
`sll rd,rs1,rs2`	Shift Left Logical	rd = ux(rs1) << rs2
`slt rd,rs1,rs2`	Set Less Than	rd = sx(rs1) < sx(rs2)
`sltu rd,rs1,rs2`	Set Less Than Unsigned	rd = ux(rs1) < ux(rs2)
`xor rd,rs1,rs2`	Xor	rd = ux(rs1) ^ ux(rs2)
`srl rd,rs1,rs2`	Shift Right Logical	rd = ux(rs1) >> rs2
`sra rd,rs1,rs2`	Shift Right Arithmetic	rd = sx(rs1) >> rs2
`or rd,rs1,rs2`	Or	rd = ux(rs1) | ux(rs2)
`and rd,rs1,rs2`	And	rd = ux(rs1) & ux(rs2)
`fence pred,succ`	Fence	-
`ebreak`	Trap	Trap to debugger
`ecall`	Trap	Trap to execution environment

Volume I goes on to define 8 more extensions for handling floating point operations with multiple levels of precision, atomicity, memory consistency constraints and additional arithmetic operations (notably multiply and divide) as well as 9 more unfinished extensions for features such as bit manipulation, decimal floating points, additional support for dynamically translated (programming) languages, vectorized operations and more.

Finally Volume I provides a helpful list of commonly implemented compiler pseudo instructions that provide helpful mnemonics or simplifications to make the RISCV ISA a little less RISC-y. The list of pseudo instructions is provided below.



pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol` *(non-PIC)*	`auipc rd, delta[31 : 12] + delta[11] addi rd, rd, delta[11:0]`	Load absolute address, where delta = symbol − pc
`la rd, symbol` *(PIC)*	`auipc rd, delta[31 : 12] + delta[11] l{w|d} rd, rd, delta[11:0]`	Load absolute address, where delta = GOT[symbol] − pc
`lla rd, symbol`	`auipc rd, delta[31 : 12] + delta[11] addi rd, rd, delta[11:0]`	Load local address, where delta = symbol − pc
`l{b|h|w|d} rd, symbol`	`auipc rd, delta[31 : 12] + delta[11] l{b|h|w|d} rd, delta[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, delta[31 : 12] + delta[11] s{b|h|w|d} rd, delta[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, delta[31 : 12] + delta[11] fl{w|d} rd, delta[11:0](rt)`	Floating−point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, delta[31 : 12] + delta[11] fs{w|d} rd, delta[11:0](rt)`	Floating−point store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, −1`	One’s complement
`neg rd, rs`	`sub rd, x0, rs`	Two’s complement
`negw rd, rs`	`subw rd, x0, rs`	Two’s complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if != zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single−precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single−precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single−precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double−precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double−precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double−precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if != zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if <= zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if >= zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if <=
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, 0(rs)`	Jump register
`jalr rs`	`jalr x1, 0(rs)`	Jump and link register
`ret`	`jalr x0, 0(x1)`	Return from subroutine
`call offset`	`auipc x1, offset[31 : 12] + offset[11] jalr x1, offset[11:0](x1)`	Call far−away subroutine
`tail offset`	`auipc x6, offset[31 : 12] + offset[11] jalr x0, offset[11:0](x6)`	Tail call far−away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Read instructions−retired counter
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Read cycle counter
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Read real−time clock
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate
`frcsr rd`	`csrrs rd, fcsr, x0`	Read FP control/status register
`fscsr rd, rs`	`csrrw rd, fcsr, rs`	Swap FP control/status register
`fscsr rs`	`csrrw x0, fcsr, rs`	Write FP control/status register
`frrm rd`	`csrrs rd, frm, x0`	Read FP rounding mode
`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap FP rounding mode
`fsrm rs`	`csrrw x0, frm, rs`	Write FP rounding mode

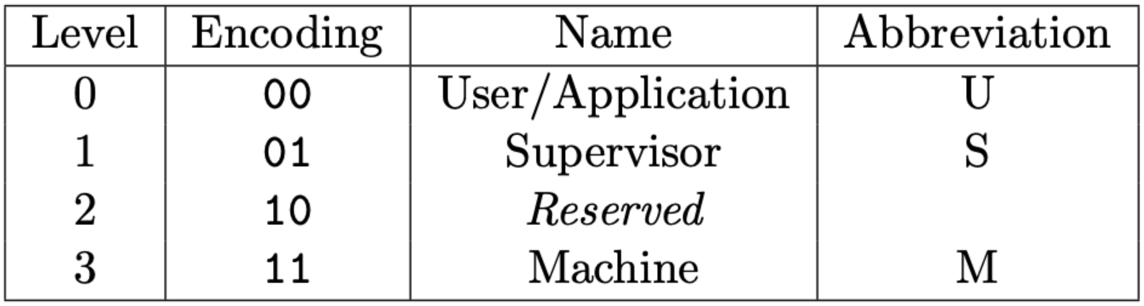


```
| `frflags rd`              | `csrrs rd, fflags, rs`                                   | Read FP exception flags                                    |
| `fsflags rd, rs`          | `csrrw rd, fflags, rs`                                   | Swap FP exception flags                                    |
| `fsflags rs`              | `csrrw x0, fflags, rs`                                   |  rite FP exception flags                                   |
```

Volume II defines the privileged architecture which adds several registers and the concept of privilege modes to each hart. These extra registers are the control and status registers (CSRs). Privileged instructions allow the programmer to read and write to them. Attempting to execute a privileged instruction without the proper privilege mode will result in an exception. Note that some of these registers are actually accessible from an unprivileged state and thus the Zicsr extension, which is necessary for accessing the CSRs and enabling the privileged architecture, is actually defined in the unprivileged architecture manual.

The CSRs can be thought of as existing in a special 12 bit encoding space for up to 4,096 CSRs. The address of a given CSR encodes its accessibility like so. If the top two bits are 00, 01, or 10 then it is read/write. Otherwise (the top two bits are 11) it is read only. The next two bits specify the lowest privilege level that can access the CSR. If a hart attempts to access a CSR that its current privilege level doesn't allow it raises an exception and traps to an error routine.

The privilege levels specified in the RISCV ISA are shown below:



Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

Supervisor mode is intended to allow for a more streamlined implementation of hypervisors by separating the supervisor from its execution environment. This allows for efficiently running multiple different *OSes* on the same chip. Alternatively it also allows for an easy way to multiplex a conforming RISCV OS onto a chip along with other OSes. All implementations of the RISCV ISA must implement machine mode as it is the only way to configure the hardware. While user level programs will run at the user level the lowest permission level. Interestingly, while many other ISAs have only two modes and thus OSes are forced to run at the machine mode equivalent, in RISCV the intended privilege level for OSes is the supervisor mode. As mentioned above, this level of abstraction is designed to make RISCV compatible with todays modern computing environment where operating systems are typically run inside of other programs. Virtualization is fully supported by the RISCV ISA as can be further seen by the full listing of CSRs defined by the RISCV ISA. A hart typically runs in user mode for the majority of the time. In order to access higher privilege levels, the hart must switch to a trap handler which will elevate the hart's permissions, execute some code, and then resume execution right after the trap instruction back in user mode. These traps can be assigned to take the user code to different privilege levels.

```
# Unprivileged Floating-Point CSRs
| Number  | Privilege | Name              | Description                                                |
| ------- | --------- | ------------------| ---------------------------------------------------------- |
| `0x001` | URW       | `fflags`          | Floating-Point Accrued Exceptions.                         |
| `0x002` | URW       | `frm`             | Floating-Point Dynamic Rounding Mode.                      |
| `0x003` | URW       | `fcsr`            | Floating-Point Control and Status Register (frm + fflags). |

# Unprivileged Counter/Timers
| Number  | Privilege | Name              | Description                                                |
| ------- | --------- | ----------------- | ---------------------------------------------------------- |
| `0xC00` | URO       | `cycle`           | Cycle counter for RDCYCLE instruction.                     |
| `0xC01` | URO       | `time`            | Timer for RDTIME instruction.                              |
| `0xC02` | URO       | `instret`         | Instructions-retired counter for RDINSTRET instruction.    |
| `0xC03` | URO       | `hpmcounter3`     | Performance-monitoring counter.                            |
| `0xC04` | URO       | `hpmcounter4`     | Performance-monitoring counter.                            |
| ...     | ...       | ...               | ...                                                        |
| `0xC1F` | URO       | `hpmcounter31`    | Performance-monitoring counter.                            |
| `0xC80` | URO       | `cycleh`          | Upper 32 bits of cycle, RV32 only.                         |
| `0xC81` | URO       | `timeh`           | Upper 32 bits of time, RV32 only.                          |
| `0xC82` | URO       | `instreth`        | Upper 32 bits of instret, RV32 only.                       |
| `0xC83` | URO       | `hpmcounter3h`    | Upper 32 bits of hpmcounter3, RV32 only.                   |
| `0xC84` | URO       | `hpmcounter4h`    | Upper 32 bits of hpmcounter4, RV32 only.                   |
| ...     | ...       | ...               | ...                                                        |
| `0xC9F` | URO       | `hpmcounter31h`   | Upper 32 bits of hpmcounter31, RV32 only.                  |

# Supervisor Trap Setup
| Number  | Privilege | Name              | Description                                                |
| ------- | --------- | ----------------- | ---------------------------------------------------------- |
| `0x100` | SRW       | `sstatus`         | Supervisor status register.                                |
| `0x104` | SRW       | `sie`             | Supervisor interrupt-enable register.                      |
| `0x105` | SRW       | `stvec`           | Supervisor trap handler base address.                      |
| `0x106` | SRW       | `scounteren`      | Supervisor counter enable.                                 |

# Supervisor Configuration
| Number  | Privilege | Name              | Description                                                |
| ------- | --------- | ----------------- | ---------------------------------------------------------- |
| `0x10A` | SRW       | `senvcfg`         | Supervisor environment configuration register.             |

# Supervisor Trap Handling
| Number  | Privilege | Name              | Description                                                |
| ------- | --------- | ----------------- | ---------------------------------------------------------- |
| `0x140` | SRW       | `sscratch`        | Scratch register for supervisor trap handlers.             |
| `0x141` | SRW       | `sepc`            | Supervisor exception program counter.                      |
| `0x142` | SRW       | `scause`          | Supervisor trap cause.                                     |
| `0x143` | SRW       | `stval`           | Supervisor bad address or instruction.                     |
| `0x144` | SRW       | `sip`             | Supervisor interrupt pending.                              |

# Supervisor Protection and Translation
| Number  | Privilege | Name              | Description                                                |
| ------- | --------- | ----------------- | ---------------------------------------------------------- |
| `0x180` | SRW       | `satp`            | Supervisor address translation and protection.             |

# Debug/Trace Registers
| Number  | Privilege | Name              | Description                                                |
| ------- | --------- | ----------------- | ---------------------------------------------------------- |
| `0x5A8` | SRW       | `scontext`        | Supervisor-mode context register.                          |

# Hypervisor Trap Setup
| Number  | Privilege | Name              | Description                                                |
| ------- | --------- | ----------------- | ---------------------------------------------------------- |
| `0x600` | HRW       | `hstatus`         | Hypervisor status register.                                |
| `0x602` | HRW       | `hedeleg`         | Hypervisor exception delegation register.                  |
| `0x603` | HRW       | `hideleg`         | Hypervisor interrupt delegation register.                  |
| `0x604` | HRW       | `hie`             | Hypervisor interrupt-enable register.                      |
| `0x606` | HRW       | `hcounteren`      | Hypervisor counter enable.                                 |
| `0x607` | HRW       | `hgeie`           | Hypervisor guest external interrupt-enable register.       |

# Hypervisor Trap Handling
| Number  | Privilege | Name              | Description                                                |
| ------- | --------- | ----------------- | ---------------------------------------------------------- |
| `0x643` | HRW       | `htval`           | Hypervisor bad guest physical address.                     |
| `0x644` | HRW       | `hip`             | Hypervisor interrupt pending.                              |
| `0x645` | HRW       | `hvip`           | Hypervisor virtual interrupt pending.                         |
| `0x64A` | HRW       | `htinst`         | Hypervisor trap instruction (transformed).                    |
| `0xE12` | HRO       | `hgeip`          | Hypervisor guest external interrupt pending.                  |

# Hypervisor Configuration
| Number  | Privilege | Name             | Description                                                   |
| ------- | --------- | ---------------- | ------------------------------------------------------------- |
| `0x60A` | HRW       | `henvcfg`        | Hypervisor environment configuration register.                |
| `0x61A` | HRW       | `henvcfgh`       | Additional hypervisor env. conf. register, RV32 only.         |

# Hypervisor Protection and Translation
| Number  | Privilege | Name             | Description                                                   |
| ------- | --------- | ---------------- | ------------------------------------------------------------- |
| `0x680` | HRW       | `hgatp`          | Hypervisor guest address translation and protection.          |

# Debug/Trace Registers
| Number  | Privilege | Name             | Description                                                   |
| ------- | --------- | ---------------- | ------------------------------------------------------------- |
| `0x6A8` | HRW       | `hcontext`       | Hypervisor-mode context register.                             |

# Hypervisor Counter/Timer Virtualization Registers
| Number  | Privilege | Name             | Description                                                   |
| ------- | --------- | ---------------- | ------------------------------------------------------------- |
| `0x605` | HRW       | `htimedelta`     | Delta for VS/VU-mode timer.                                   |
| `0x615` | HRW       | `htimedeltah`    | Upper 32 bits of htimedelta, HSXLEN=32 only.                  |

# Virtual Supervisor Registers
| Number  | Privilege | Name             | Description                                                   |
| ------- | --------- | ---------------- | ------------------------------------------------------------- |
| `0x200` | HRW       | `vsstatus`       | Virtual supervisor status register.
| `0x204` | HRW       | `vsie`           | Virtual supervisor interrupt-enable register.
| `0x205` | HRW       | `vstvec`         | Virtual supervisor trap handler base address.
| `0x240` | HRW       | `vsscratch`      | Virtual supervisor scratch register.
| `0x241` | HRW       | `vsepc`          | Virtual supervisor exception program counter.
| `0x242` | HRW       | `vscause`        | Virtual supervisor trap cause.
| `0x243` | HRW       | `vstval`         | Virtual supervisor bad address or instruction.
| `0x244` | HRW       | `vsip`           | Virtual supervisor interrupt pending.
| `0x280` | HRW       | `vsatp`          | Virtual supervisor address translation and protection.

# Machine Information Registers
| Number  | Privilege | Name             | Description                                                   |
| ------- | --------- | ---------------- | ------------------------------------------------------------- |
| `0xF11` | MRO       | `mvendorid`      | Vendor ID.
| `0xF12` | MRO       | `marchid`        | Architecture ID.
| `0xF13` | MRO       | `mimpid`         | Implementation ID.
| `0xF14` | MRO       | `mhartid`        | Hardware thread ID.
| `0xF15` | MRO       | `mconfigptr`     | Pointer to configuration data structure.

# Machine Trap Setup
| Number  | Privilege | Name             | Description                                                   |
| ------- | --------- | ---------------- | ------------------------------------------------------------- |
| `0x300` | MRW       | `mstatus`        | Machine status register.
| `0x301` | MRW       | `misa`           | ISA and extensions.
| `0x302` | MRW       | `medeleg`        | Machine exception delegation register.
| `0x303` | MRW       | `mideleg`        | Machine interrupt delegation register.
| `0x304` | MRW       | `mie`            | Machine interrupt-enable register.
| `0x305` | MRW       | `mtvec`          | Machine trap-handler base address.
| `0x306` | MRW       | `mcounteren`     | Machine counter enable.
| `0x310` | MRW       | `mstatush`       | Additional machine status register, RV32 only.

# Machine Trap Handling
| Number  | Privilege | Name             | Description                                                   |
| ------- | --------- | ---------------- | ------------------------------------------------------------- |
| `0x340` | MRW       | `mscratch`       | Scratch register for machine trap handlers.                   |
| `0x341` | MRW       | `mepc`           | Machine exception program counter.                            |
| `0x342` | MRW       | `mcause`         | Machine trap cause.                                           |
| `0x343` | MRW       | `mtval`          | Machine bad address or instruction.                           |
| `0x34A` | MRW       | `mip`            | Machine interrupt pending.                                    |
| `0x344` | MRW       | `mtinst`         | Machine trap instruction (transformed).                       |
| `0x34B` | MRW       | `mtval2`         | Machine bad guest physical address.                           |

# Machine Configuration
| Number  | Privilege | Name             | Description                                                   |
| ------- | --------- | ---------------- | ------------------------------------------------------------- |
| `0x30A` | MRW       | `menvcfg`        | Machine environment configuration register.                   |
| `0x31A` | MRW       | `menvcfgh`       | Additional machine env. conf. register, RV32 only.            |
| `0x747` | MRW       | `mseccfg`        | Machine security configuration register.                      |
| `0x757` | MRW       | `mseccfgh`       | Additional machine security conf. register, RV32 only.        |

# Machine Memory Protection
| Number  | Privilege | Name             | Description                                                   |
| ------- | --------- | ---------------- | ------------------------------------------------------------ |
| `0x3A0` | MRW       | `pmpcfg0`        | Physical memory protection configuration.                    |
| `0x3A1` | MRW       | `pmpcfg1`        | Physical memory protection configuration, RV32 only.         |
| `0x3A2` | MRW       | `pmpcfg2`        | Physical memory protection configuration.                    |
| `0x3A3` | MRW       | `pmpcfg3`        | Physical memory protection configuration, RV32 only.         |
| ...     | ...       | ...              | ...                                                          |
| `0x3AE` | MRW       | `pmpcfg14`       | Physical memory protection configuration.                    |
| `0x3AF` | MRW       | `pmpcfg15`       | Physical memory protection configuration, RV32 only.         |
| `0x3B0` | MRW       | `pmpaddr0`       | Physical memory protection address register.                 |
| `0x3B1` | MRW       | `pmpaddr1`       | Physical memory protection address register.                 |
| ...     | ...       | ...              | ...                                                          |
| `0x3EF` | MRW       | `pmpaddr63`      | Physical memory protection address register.                 |

# Machine Counter/Timers
| Number  | Privilege | Name             | Description                                                  |
| ------- | --------- | ---------------- | ------------------------------------------------------------ |
| `0xB00` | MRW       | `mcycle`         | Machine cycle counter.                                       |
| `0xB02` | MRW       | `minstret`       | Machine instructions-retired counter.                        |
| `0xB03` | MRW       | `mhpmcounter3`   | Machine performance-monitoring counter.                      |
| `0xB04` | MRW       | `mhpmcounter4`   | Machine performance-monitoring counter.                      |
| ...     | ...       | ...              | ...                                                          |
| `0xB1F` | MRW       | `mhpmcounter31`  | Machine performance-monitoring counter.                      |
| `0xB80` | MRW       | `mcycleh`        | Upper 32 bits of mcycle, RV32 only.                          |
| `0xB82` | MRW       | `minstreth`      | Upper 32 bits of minstret, RV32 only.                        |
| `0xB83` | MRW       | `mhpmcounter3h`  | Upper 32 bits of mhpmcounter3, RV32 only.                    |
| `0xB84` | MRW       | `mhpmcounter4h`  | Upper 32 bits of mhpmcounter4, RV32 only.                    |
| ...     | ...       | ...              | ...                                                          |
| `0xB9F` | MRW       | `mhpmcounter31h` | Upper 32 bits of mhpmcounter31, RV32 only.                   |

# Machine Counter Setup
| Number  | Privilege | Name            | Description                                                   |
| ------- | --------- | --------------- | ------------------------------------------------------------- |
| `0x320` | MRW       | `mcountinhibit` | Machine counter-inhibit register.                             |
| `0x323` | MRW       | `mhpmevent3`    | Machine performance-monitoring event selector.                |
| `0x324` | MRW       | `mhpmevent4`    | Machine performance-monitoring event selector.                |
| ...     | ...       | ...             | ...                                                           |
| `0x33F` | MRW       | `mhpmevent31`   | Machine performance-monitoring event selector.                |

# Debug/Trace Registers (shared with Debug Mode)
| Number  | Privilege | Name            | Description                                                   |
| ------- | --------- | --------------- | ------------------------------------------------------------- |
| `0x7A0` | MRW       | `tselect`       | Debug/Trace trigger register select.                          |
| `0x7A1` | MRW       | `tdata1`        | First Debug/Trace trigger data register.                      |
| `0x7A2` | MRW       | `tdata2`        | Second Debug/Trace trigger data register.                     |
| `0x7A3` | MRW       | `tdata3`        | Third Debug/Trace trigger data register.                      |
| `0x7A8` | MRW       | `mcontext`      | Machine-mode context register.                                |

# Debug Mode Registers
| Number  | Privilege | Name            | Description                                                   |
| ------- | --------- | --------------- | ------------------------------------------------------------- |
| `0x7B0` | DRW       | `dcsr`          | Debug control and status register.                            |
| `0x7B1` | DRW       | `dpc`           | Debug PC.                                                     |
| `0x7B2` | DRW       | `dscratch0`     | Debug scratch register 0.                                     |
| `0x7B3` | DRW       | `dscratch1`     | Debug scratch register 1.                                     |
```

There are many interesting features of the RISCV ISA revealed in this listing and a full analysis of all of them would be out of the scope of this research. As my research is focused on making a minimal OS, I will focus on the minimum necessary set of CSRs to implement an OS with all modern abstractions. To do so in a fully compliant manner I will need to use all three modes. My OS will have to boot into machine mode, as all programs do, do some initial configuration to set up traps and virtual memory structures, then jump to supervisor mode where the OS code will take over and eventually run the first user process in user mode.

The CSRs that will be necessary to support this behavior are the following:

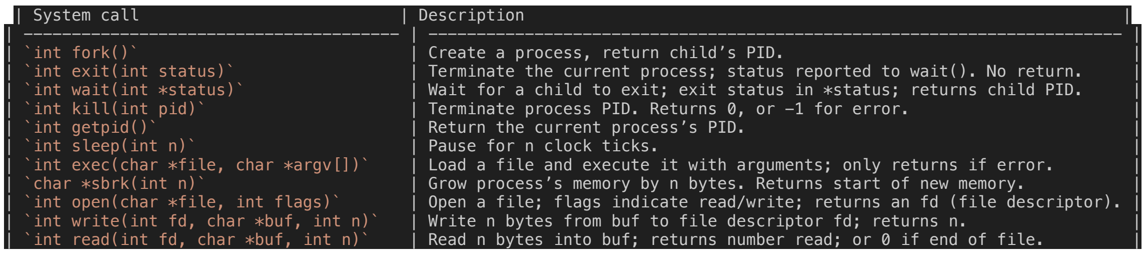
- mhartid: A machine mode, read-only register that holds a unique value used to identify a hart. The ISA specifies that while their must always be a hart with id 0, the ids need not be sequential, although they should be kept small. Xv6 uses this to figure out which CPU should run set up code that should only be executed once

- mscratch: A machine mode, read-write register that can be used to store an extra word of data. The ISA states that the register is "Typically, it is used to hold a pointer to a machine-mode hart-local context space" (Waterman, et al. 37). Xv6 uses this to store a pointer to per hart interrupt state.
- sip: A supervisor read-write register that contains information on pending supervisor level interrupts. Xv6 uses this to trap into supervisor mode after a timer interrupt (which is handled by the machine mode). In order for a supervisor interrupt to occur either:
    (a) The current privilege mode is S and the SIE (supervisor interrupt enable) bit in the sstatus register is set
    (b) Or the current privilege mode has less privilege than S-mode and bit i is set in both sip and sie
- sscratch: A supervisor mode, read-write register that can be used to store an extra word of data. Used similarly to mscratch but for code running at the supervisor level.
- satp: A supervisor mode read-write register that holds the physical page number of the root page table (its physical address divided by 4Kb).

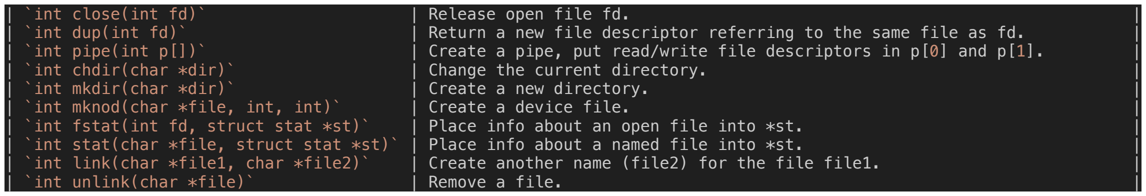
Stripping functionality from Xv6 will likely require fewer, not more special registers which means that my operating system will likely use a max of 5 special registers. It is amazing that a modern operating system is capable of operating with such a limited set of resources. A lot more research is necessary to fully understand the how traps and interrupts work in RISCV.

Writing an operating system from scratch is a monumental task and would add an additional level of difficulty to this research. Fortunately there is an existing implementation of a RISCV operating system, Xv6. The Xv6 operating system manual details the philosophy and the technical decision behind the design.

According to the Xv6 manual, "xv6, provides the basic interfaces introduced by Ken Thompson and Dennis Ritchie's Unix operating system, as well as mimicking Unix's internal design" (Cox, et al. 9). Similar to many modern operating systems, Xv6 takes the form of a kernel that running processes can call into via system calls. System calls are achieved by executing a special instruction that raises an exception. This causes the CPU to raise its privilege mode and jump to code specified on startup that handles the system call and then returns to the running process. The system calls provided by Xv6 are listed below.



System call	Description
`int fork()`	Create a process, return child's PID.
`int exit(int status)`	Terminate the current process; status reported to wait(). No return.
`int wait(int *status)`	Wait for a child to exit; exit status in *status; returns child PID.
`int kill(int pid)`	Terminate process PID. Returns 0, or -1 for error.
`int getpid()`	Return the current process's PID.
`int sleep(int n)`	Pause for n clock ticks.
`int exec(char *file, char *argv[])`	Load a file and execute it with arguments; only returns if error.
`char *sbrk(int n)`	Grow process's memory by n bytes. Returns start of new memory.
`int open(char *file, int flags)`	Open a file; flags indicate read/write; returns an fd (file descriptor).
`int write(int fd, char *buf, int n)`	Write n bytes from buf to file descriptor fd; returns n.
`int read(int fd, char *buf, int n)`	Read n bytes into buf; returns number read; or 0 if end of file.



```
| `int close(int fd)`                    | Release open file fd.                                            |
| `int dup(int fd)`                      | Return a new file descriptor referring to the same file as fd.   |
| `int pipe(int p[])`                    | Create a pipe, put read/write file descriptors in p[0] and p[1]. |
| `int chdir(char *dir)`                 | Change the current directory.                                    |
| `int mkdir(char *dir)`                 | Create a new directory.                                          |
| `int mknod(char *file, int, int)`      | Create a device file.                                            |
| `int fstat(int fd, struct stat *st)`   | Place info about an open file into *st.                          |
| `int stat(char *file, struct stat *st)` | Place info about a named file into *st.                         |
| `int link(char *file1, char *file2)`   | Create another name (file2) for the file file1.                  |
| `int unlink(char *file)`               | Remove a file.                                                   |
```

Something that will complicate my research is that Xv6 is designed as a monolithic kernel. This means that the whole OS is designed to run with supervisor privileges. This makes it easy for the developer by simplifying the transitions between different parts of the kernel. However it also leads to more complicated interfaces between them because the different parts of the kernel rely on each other in unpredictable ways. Part of my research will involve breaking the OS into pieces such that they can be enabled and disabled independently. This will naturally lead to a more modular and maintainable OS. It will also explore the feasibility of a micro kernel built with RISCV.

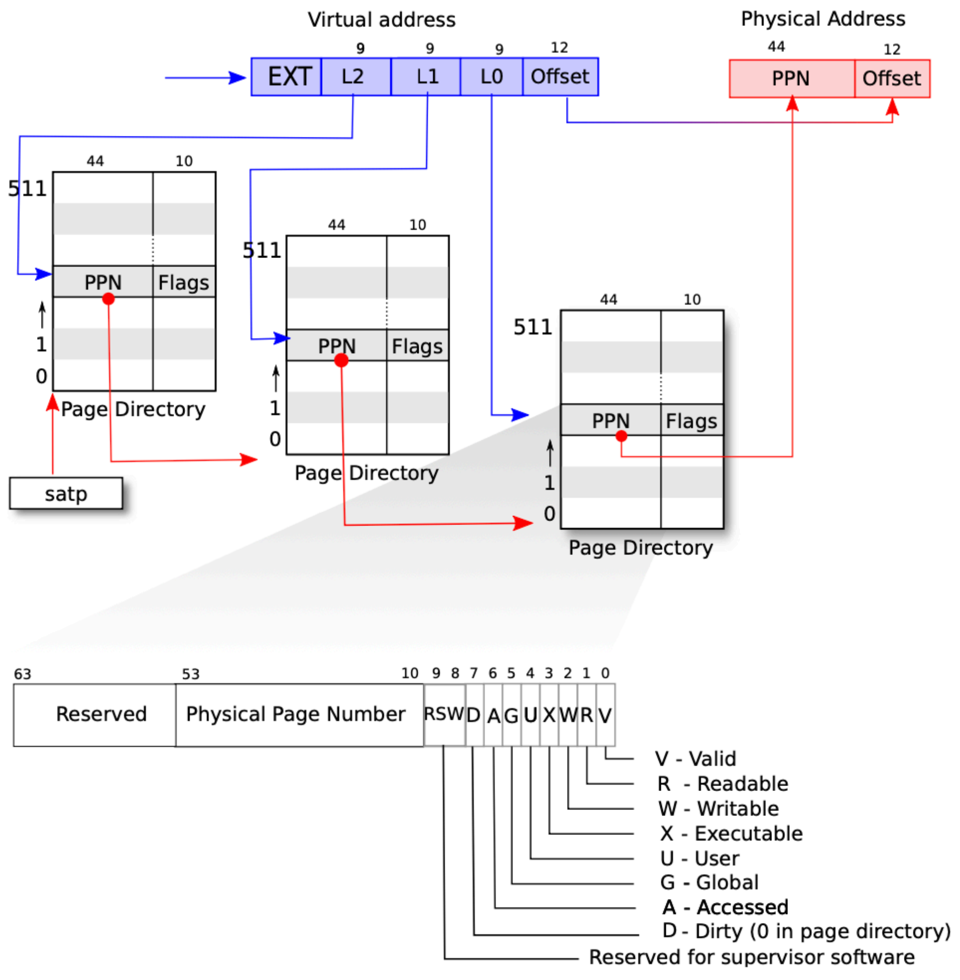
One of the first ways that a computer is able to interact with the world is by printing to the screen. In order to implement this it is necessary to understand how Xv6 interacts with the screen. This requires understanding both how it handles interrupts and how it handles IO. Xv6 uses a UART to communicate with the screen, in particular QEMU emulates a 16550 UART chip. This chip consists of a few memory mapped registers starting at 0x10000000 which is defined in the kernel/memlayout.h (Cox, et al. 49-50) file as UART0. These registers are responsible for signaling to the CPU that the serial chip is ready to handle more data and that there is data waiting to be processed.

The main code responsible for setting this up is stored in consoleinit. The code responsible for processing device interrupts and eventually console input is uartintr. The function responsible for sending output to the console is uartstart. Understanding these three function calls is necessary to understanding how a RISCV chip handles IO with attached devices, a crucial role of any successful operating system. Separating this code and simplifying it will be a primary focus of this research as it both demonstrates the role of an OS as a device manager, and emphasizes the need for system calls and traps. Furthermore it will ultimately end up being an important resource to synchronize when the operating system is parallelized which will be a good lead in to synchronizing primitives.

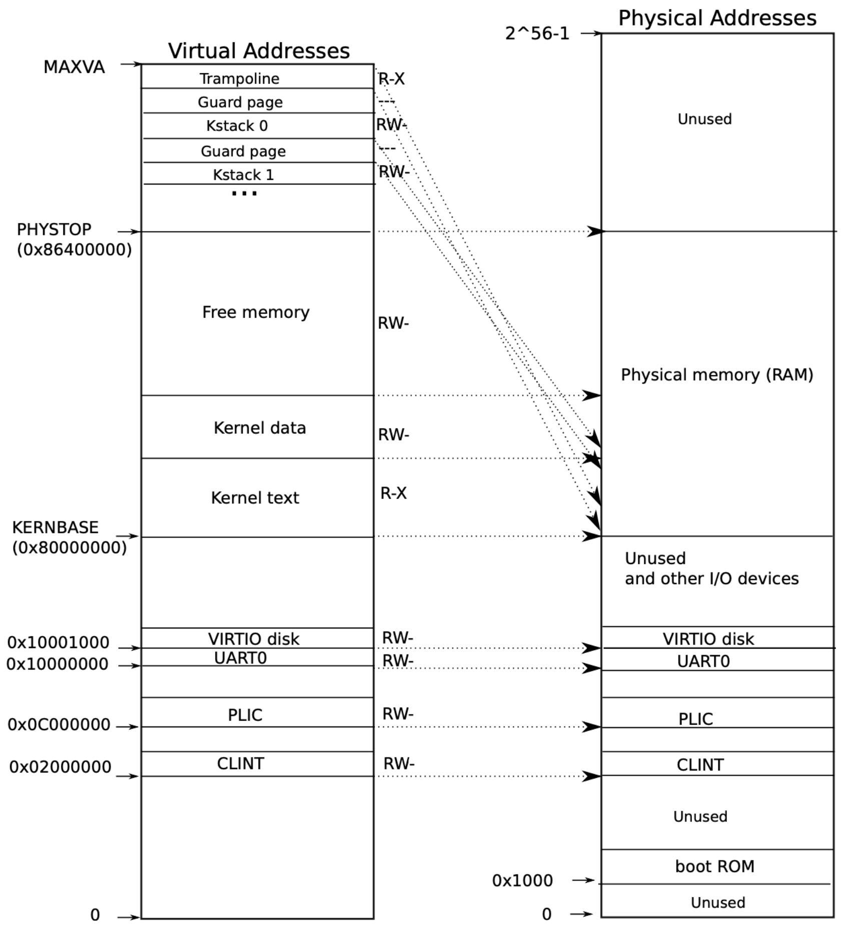
The next important service provided by an operating system is memory virtualization. In modern operating systems this is handled with a combination of hardware and software. The hardware necessary, mentioned above, is the supervisor address translation pointer, which points to the page table, originally set up in the kvminithart function. In order to understand RISCV virtual memory it is necessary to understand the RISCV paging table structure. According to the Xv6 manual:

> xv6 runs on Sv39 RISC-V, which means that only the bottom 39 bits of a 64-bit virtual address are used; the top 25 bits are not used. In this Sv39 configuration, a RISC-V page table is logically an array of 2^{27} (134,217,728) *page table entries (PTEs)*. Each PTE contains a 44-bit physical page number (PPN) and some flags. The paging hardware translates a virtual address by using the top 27 bits of the

39 bits to index into the page table to find a PTE, and making a 56-bit physical address whose top 44 bits come from the PPN in the PTE and whose bottom 12 bits are copied from the original virtual address. Figure 3.1 shows this process with a logical view of the page table as a simple array of PTEs (see Figure 3.2 for a fuller story). A page table gives the operating system control over virtual-to physical address translations at the granularity of aligned chunks of 4096 (2^{12}) bytes.



The details of the Xv6 memory layout are provided in the following graphic. Most of the details are not relevant, however it does offer a few important features that help to simplify the organization. The first and most important is that the kernel is direct mapped into the virtual address space. This simplifies the OS by removing the need to carefully check which address space pointers came from. This is especially helpful for functions like fork that use physical memory pointers. An address space for each process is created in procinit by using kvmmap to add PTE entires to the process's address space. Kvmmap ultimately calls mappages which is responsible for finding free physical pages on the kernel list and installing them into the proper



PTEs.

Understanding how Xv6 handles memory mapping will help to separate the virtual memory code from the kernel. This will allow the OS to run without virtual memory. While this may seem like a downside, allowing students to program in an unvirtualized address space will allow them to discover for themselves why it is such a bad idea. No modern operating system supports an unvirtualized memory system (for good reason) so this experience is not only helpful, but hard to get otherwise. It will also help me to understand how they implement the synchronization and virtualization of memory necessary to support multiprocessing. That will ultimately help me with the final goal of my research.

The final steps of my research will be to understand how Xv6 implements and handles multiprocessing, including address spaces, scheduling, and preemption. As with the rest of the functions provided by modern operating systems, this requires a combination of hardware and software. As covered above, the hardware provides a virtualization method through the address translation protocol, which the kernel takes advantage of by creating and managing multiple page tables structures in memory. But how does Xv6 handle switching between processes?

Like many modern operating systems it is a preemptive OS, which means that in addition to software interrupts with the yield system call, the OS also configures an interrupt timer to periodically interrupt compute bound processes. This requires specialized hardware to implement. This takes the from of the core local interruptor (CLINT). This piece of hardware relies on the memory mapped registers referred to in the ISA as mtime and mtimecmp. Note that these are not physical registers so they are not included int he list above. This allows them to have a fixed 64 bit width on all architectures. The value of mtime is described in the ISA as: “mtime must increment at constant frequency, and the platform must provide a mechanism for determining the period of an mtime tick. The mtime register will wrap around if the count overflows” (Waterman, et al. 44). This is used in conjunction with mtimecmp which holds a threshold value. If mtime is every greater than or equal to mtimecmp the machine timer interrupt becomes pending and will be handled as soon as the EEI allows.

The interrupt will only be taken if interrupts are enabled and the MTIE bit is set in the mie register, thus allowing for clock interrupts to be disabled if desired. These are machine mode registers as indicated by the prefixed ‘m’ and RISCV mandates that timer interrupts be handled in machine mode. This means that unlike most RISCV interrupts which can be delegated to the supervisor mode (where the operating system normally runs), timer interrupts must be handled by special code that runs in machine mode. Consequently, once initially configured at boot time, via timerinit, timer interrupts cannot be disabled by the OS, even during critical sections. To work around this, the Xv6 timer interrupt handler raises a software interrupt, which can be delegated and handled by the regular OS code, or ignored if necessary. The code that handles the timer interrupt is timervec, written entirely in assembly. This simplifies the process of jumping between C code by obviating the need to set up a stack and deal with the fact that machine mode code is unpaged.

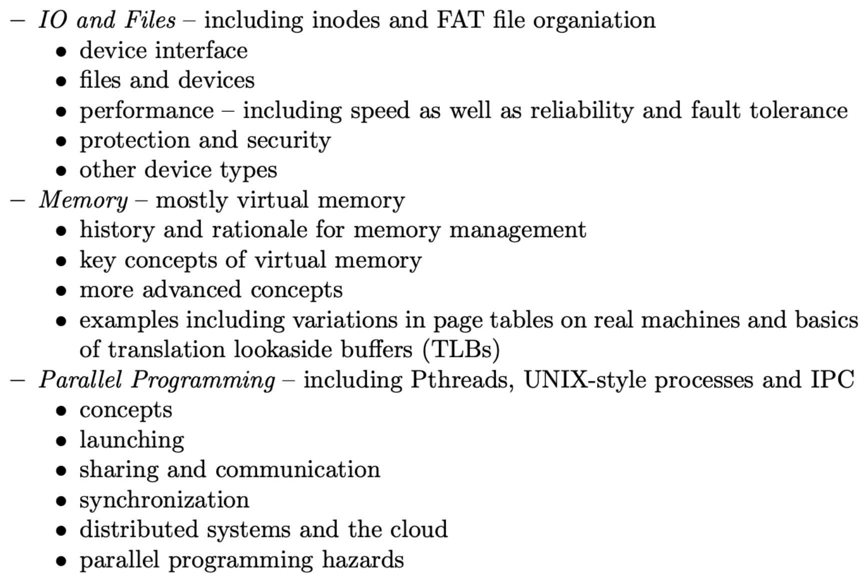
Once a thread relinquishes control the hart will eventually jump to the schedule function and from there choose a new thread to be run. The current scheduling algorithm is a simple round robin function that services each thread once before looping back around. It locks the process structure before checking if it is ready to be ran, however it does not account for the fact that if the hart running at the “front of the line” is compute bound, then all the other harts behind it will be stuck waiting for that thread to be released, instead of serving other runnable threads. This is a place for large improvements.

Scheduling thus also will involve a discussion of concurrency, which involves understanding how Xv6 handles locking using the RISCV atomic primitives. The code that handles locking in Xv6 is stored in sleeplock.c and spinlock.c. Each contains code to implement sleep locks and spin locks respectively. Underneath both is the acquire call which uses the portable C call __sync_lock_test_and_set which compiles to an amoswap (atomic swap) instruction on RISCV. The basic lock used in Xv6 is a spin lock. It simply loops on the amoswap instruction until it is able to acquire the lock. This works well for resources that are not held very long, such as the ticks variable, which counts how many hart cycles the OS has been running for. This lock need only be held for at most three instructions to load, update, and then store the new value. But what about resources that are held for longer like files and IO descriptors?

In that case a spin lock would be wasteful because any other thread waiting for that resource would have to spin while waiting for the resource to be available. Furthermore, in order to simplify locking orders, Xv6 mandates that if a hart holds any lock, it must disable all interrupts. This stems from the necessary condition that if an interrupt handler uses a lock, then the hart can never hold that lock with interrupts enabled, otherwise there is the risk of a deadlock in the handler itself, causing the hart to become unresponsive. This restriction means that a thread cannot yield the hart while acquiring or holding a lock. Clearly this is a limitation that would best to be avoid as a single thread can lock up a hart indefinitely when using a lock. To work around this Xv6 also provides sleep locks, which allow the thread to acquire and hold the lock without disabling interrupts. The sleep locks use a spin lock to protect a structure with a locked field that indicates if it is held or not. You acquire it with sleepacquire which acquires the spinlock, checks if the lock can be acquired, and sleeps until it can, releasing the spinlock while it waits, allowing other threads to attempt to acquire the lock as well. Thus Xv6s sleep locks perform very similarly to condition variables found in other operating systems.

With that I have provided a (brief) summary of how Xv6 uses the RISCV ISA to implement common operating system goals. Together these features create a recognizable modern operating system.

The next part of research focuses on the pedagogical side of my project to see what research has already been done and how I can improve on previous research in my proposed course. Machanick perfectly summarizes my understanding of operating system classes: “There are two major approaches to teaching operating systems: conceptual and detailed”. He is a professor of computer science at Rhodes University. He teaches their systems course there and is experimenting with a more hands on approach to teaching operating systems. At Rhodes his course is divided into short modules that run over 4 weeks with 5 lectures and one practical per week. Given the short length of each module, a course that goes into full detail on every operating system technique is out of the question. To compensate for this he has experimented with a course that aims to strike a midpoint between conceptual and detailed courses. To do so he uses a lot of simulation technologies including faking parts of the system with structures in RAM and simulating parts of the system with traces. His course provides a good example of a possible structure.



- *IO and Files* – including inodes and FAT file organiation
  - device interface
  - files and devices
  - performance – including speed as well as reliability and fault tolerance
  - protection and security
  - other device types
- *Memory* – mostly virtual memory
  - history and rationale for memory management
  - key concepts of virtual memory
  - more advanced concepts
  - examples including variations in page tables on real machines and basics of translation lookaside buffers (TLBs)
- *Parallel Programming* – including Pthreads, UNIX-style processes and IPC
  - concepts
  - launching
  - sharing and communication
  - synchronization
  - distributed systems and the cloud
  - parallel programming hazards

An interesting takeaway from his description of trace based simulation is his use of the Intel Pin tool to instrument a regular user level program (for example ls or cat) in order to capture its memory accesses and use them to simulate a paging algorithm.

His experience teaching the course such a is summed up at the end of his paper: “My experience of explaining concepts like multilevel page tables and TLBs in lectures is that they are very difficult concepts to grasp in the abstract. Parallel programming is another area where doing is really required to learn.” (Citation needed) This confirms my belief that systems engineering students typically learn better through hands-on activities. He also points out that a more hands on approach allows more in-depth discussions on the development of relevant technologies. The example he gives is the differing speeds of development between the Linux scheduler and the WindowsNT scheduler. That discussion was only possible because his students had an understanding of how they were actually implemented.

His conclusions after teaching the course are that it was a difficult course, in part because students typically enter a systems programming course with little or no experience with C. This adds additional complexity to an already difficult topic. He acknowledges that the course has not been taught long enough for a qualitative evaluation, however he states that, “My own experience is that students taught using this just enough abstraction approach have a better appreciation of implementation and design issues than those taught using a purely theoretical approach” (Machanick 12).

Note that I have decided that researching the possibility of Rust for systems development is out of the scope of this research so I did not complete the reading on Rust for systems development. Instead I focused on the pedagogical side of my project to see what research has been done on how computer science students actually learn.

Richards work suggests that students actually learn through mental models of computers and that those models are often extremely flawed. This of course how learning works but the underlying idea of mental models is extremely encouraging. It suggests that students will learn best through hands on activities, as their machines evolve to match the behavior they see in the machines they work with. One of the most complicated parts of these machines is often how they hold state and how they transition from one state to the next. Although they are by definition abstractions, it is clear that providing concrete examples of such machines is crucial to providing context for what a state really is. Operating systems are fundamentally just big programs that run with state and transitions between them. Thus it should be possible to construct a mental model of one. Of course modern operating systems are extremely complicated, far too large for a person to hold an entire mental model of. However, when dealing with a smaller codebase, like Xv6, it may be possible to fit the entire thing in one’s memory. Further more the interactive approach taken by Richards to develop and expand the mental machines indicates that adding complexity slowly and thoroughly explaining how it works leads the best evolution of a student’s mental machine.

Some key takeaways from this paper were the benefits the mental machine model provided. By giving the students agency in creating the model they were able to shape it in the context of their own knowledge and expectations. This allowed them to have a deeper understanding of how it worked and to design an extensible model that they can develop on their own as their education continues. This demonstrates the benefits of “hands-on” work. While my project may have a much more physical aspect, the goal is to have students build their own operating systems as the class progresses. Thus they will be progressively expanding their mental model of how their operating system works, and importantly how it doesn’t work and what changes they had to make to correct those flaws. Being able to follow the evolution of their operating system through the class is similar to the students being able to look back at their experience with previous versions of the mental machine and see how it evolved to meet new demands.

Another takeaway is the difficulties they faced. First and foremost they found that a mental model of a machine is inseparable from the machine itself. It is ultimately produced and developed based on interaction with a specific machine and thus their development is closely related. Additionally it is hard to construct a generic mental model. Different students will make different connections and so it is impossible to see what is “logical” from everyones’ perspective. While this was a difficulty for Richard’s class, this may be a strength in my proposed class, as it is explicitly designed for each student to come up with their own model of how and operating system works as they develop their own simple operating system. Of course it is also something to be aware of as I prepare materials for it because the steps and questions that follow logically to a systems engineer who can see the bigger picture may not seem at all logical to a student.

4. Methodology

My research will focus on deconstructing Xv6 in order to understand how it works and then rewriting it so that it can run with and without virtual memory, with or without concurrency, and with a custom scheduler. Xv6 was produced by MIT and based on Ken Thompsons (Cox et al.) original operating system, V6. It is written in a combination of C and RISCV assembly. It is compiled with a makefile and a custom linker script. It can be run on a RISCV CPU, or emulated on any CPU with QEMU a popular tool for virtualization.

I will therefore be doing most of programming in C. However, much of the code that handles system start up and switching requires extremely low level access to the CPU. This means I must write some sections in assembly. I plan on writing this section in VSCode because it has excellent support for assembly in many different formats. Assembly is a very low level language so it is impossible to use libraries for this section. While this may seem like a drawback, this actually aligns with goal 2 to ensure that the code is as simple as possible, as assembly sub routines are often quite difficult to understand.

Xv6 is a RISCV operating system so I will be writing in RISCV assembly. This is a natural choice for a beginner class in operating systems because it is a RISC architecture so it has a very small instruction set. In fact the entire set comes out to just over 200 instructions with every single extension enabled. In practice I will rarely have to use anything outside of the base instruction set which comes in at a reasonable 40 opcodes. This ensures that students will spend a minimum amount of time looking up opcodes, allowing them to focus on the development of their operating systems. Additionally RISCV is open source. This means that it can be used free of charge and there are many resources online to help students understand it, which aligns with goal 9. After the initial bootstrapping, it is more practical to jump to C code for ease of development and debugging. This will make it much easier for me to avoid mistakes and to track down bugs, allowing me to iterate faster.

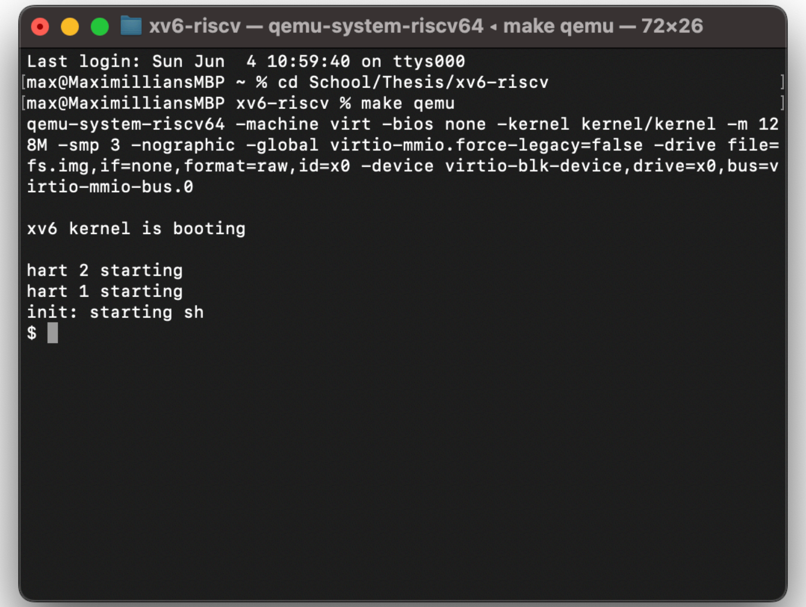
Jumping from assembly to C will require an exact knowledge of the memory layout of the program. For this I will use a linker script adapted from the one provided by Xv6. Understanding linker scripts is a good way to introduce students to the concept of a memory layout so the work I do to understand the syntax and prepare examples will be useful in preparing course materials as well. I will also be writing my C code in VSCode to simplify my development environment. Of course because I am writing an operating system, there are no library routines other than the ones I implement myself. In an effort to reduce the amount of code in the operating system I will not be re-implementing the entire C standard library, however I will be re-implementing a few common APIs (open, read, write, close) to get use to hardware interfaces and to show the students what it takes to implement your own library functions.

When it comes to designing the course and requisite material I will be working closely with Professor Richards to make up for my lack of experience in pedagogy. I plan to rely on my existing research and the wiki pages I will be create to supply my course with helpful resources to understand the complicated workings of both RISCV and Xv6. Creating the slides for this course will go hand in hand with the process of writing my thesis paper. Creating the necessary changes to Xv6 will go hand in hand with creating problems for exams and projects.

5. Preliminary Findings

My preliminary research focused on understanding RISCV and existing RISCV operating system implementations. Unfortunately RISCV is a very new standard. It's latest revision was released in 2021. Due to this I was only able to find one example of a RISCV operating system, Xv6.

Xv6 is an educational operating system developed by MIT and recently rewritten to be in RISCV because it is an open source standard and simple to work with as it is a



```
Last login: Sun Jun  4 10:59:40 on ttys000
max@MaximilliansMBP ~ % cd School/Thesis/xv6-riscv
max@MaximilliansMBP xv6-riscv % make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 12
8M -smp 3 -nographic -global virtio-mmio.force-legacy=false -drive file=
fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=v
irtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ 
```

RISC ISA. As seen above I have spent time organizing the existing RISCV documentation into a more accessible format. This was useful to me to help understand the RISCV architecture. This will also be useful when I begin making course materials.

I have also successfully compiled and ran Xv6 on my laptop, using QEMU for emulation. In Professor Richard's 377 honors section I was also able to compile additional system calls and make modifications to the scheduling algorithm. The existing scheduling algorithm is extremely simple, consisting of a single round robin loop running on both of the simulated cores. Notably, it does not perform any checks to see if a process is already being run before attempting to run it. This makes it very simple, easy to understand and easy to modify. However it can result in a very unresponsive system if there is IO being run because when the second core reenters the scheduling algorithm it will always wait for the first core to finish it's current process, and then run whatever process that was instead of jumping to another process that could be run.

6. Evaluation

This research will be broken into the following milestones:

1. Completing the instruction set listing on the RISCV wiki.
2. Completing the reference pages for the RISCV wiki, providing a short and concise summary of the RISCV specification.
3. Completing the Xv6 reference pages, providing a short and concise summary of all functions and their role in the operating system overall.
4. Reorganizing the Xv6 code into more sensible modules. Additionally making the organization of the codebase more apparent by renaming the functions to remove abbreviations and using prefix name spacing.
5. Commenting the Xv6 source code fully.
6. Creating a development environment to build Xv6, run it with QEMU, and debug it with GDB.
7. Modularizing and editing the Xv6 source code such that it is possible to build it:
    - With or without virtual memory.
    - With or without concurrency.
    - With a custom scheduler.
8. Creating a course plan with slides, exams, and projects.
9. Creating a web based development environment, possibly based on gdbgui.
10. Making substantive changes to Xv6.

These milestones will be evaluated according to the following criteria:

1. All of the RISCV instruction set listing pages included on the landing page are fully and and correctly completed. The page is largely free of typos and other grammatical mistakes.
2. There are summaries for the following features the RISCV ISA provides outside of the instruction set listing. The pages are largely free of typos and other grammatical mistakes.
    - All CSSR registers that handle interrupt masking, interrupt delegation, execution mode, trap vectors, and virtual memory. Extra CSSRs may be included.
3. Every function and file should have a page describing it’s purpose and any important notes about the functionality or behavior. The pages are largely free of typos and other grammatical mistakes.

4. This is largely a qualitative measurement so the grading will be subjective. Consideration shall be given to demonstrated effort to move files around and rewrite code to ensure that a single file has no more than 10 functions and the functions are grouped sensibly and legibly.
5. Again comment quality is a subjective measure, however checking that every function has a header comment describing it's purpose and comments explaining all non obvious sections. Consideration should be given to the accuracy and thoroughness of the comments.
6. This one is fairly simple. I should demonstrate that I have a makefile or script that is capable of setting all of these systems up to run.
7. It should be possible to build a version of Xv6 with any of those possibilities. This can be given partial credit based on how many of the possibilities are possible.
8. I will work closely with Professor Richards to propose a curriculum that will teach students how operating systems work from the ground up and incorporate the operating system I have created into several projects that involve adding functionality to the operating system. I will provide sample solutions for all projects assigned. I have very little experience teaching and designing courses so I plan to communicate heavily with Professor Richards to rapidly iterate on my proposed course materials based on his feedback. It is hard to quantify course design so this part of the project should be graded based on demonstrated effort. Communication will be extremely important for this part of the project so I should also be graded on how often I communicate to ensure that I get my work done in a timely manner.
9. Similar to 6, for this milestone I should be able to demonstrate that I have a makefile or script that is capable of setting the system up to allow for debugging.
10. This is a stretch goal and so it is relatively more vague. I should make a contribution that includes at least 3 more functions and no less than 100 lines of code. I should be able to demonstrate it working

Achieving up to milestone 5 will earn a C. This is due to the fact the work is important to the course but not necessarily very hard, as it is mostly copying and pasting into tables. Achieving milestone 7 will earn a B. This work is harder as it involves writing code and a thorough understanding of how all the pieces of the operating system fit together. Achieving up to 9 will earn an A. After revising the existing Xv6 codebase, creating the course is the next goal of this thesis. Fully completing this stage will require many hours of both programming and careful distillation of the concepts that are crucial to modern operating system development. This is where I will demonstrate the knowledge I have gained through my research. As previously stated milestone 10 is a stretch goal and will require significant work. I hope to be able to complete this and truly make a UMass specific operating system, however as it is not necessary to my goal I do not think my grade should be tied to it.

Feedback on these goals should be provided verbally via weekly meetings with Professor Richards where we review the work I have done that week, whether it is code, slides, or scripts. My ultimate product will be a cd with Git repos containing all of the content I have produced for the course including my fork of Xv6, the accompanying wiki which contains the reference pages and the slides, projects, and lesson plans necessary for my proposed course.

7. Communication

I expect to meet with Professor Richards for an hour once a week over Discord and with the full committee at least once a month. Feedback on these goals should be provided verbally via weekly meetings with Professor Richards where we review the work I have done that week, whether it is code, slides, or scripts. In these meetings he can provide direction and answer any questions I may have. I expect to commit at least 6 hours of work a week to this project not including time spent meeting with my committee members.

8. Timeline

- 09/10/23: Completing the instruction set listing on the RISCV wiki:
- 09/17/23: Completing the reference pages for the RISCV wiki, providing a short and concise summary of the RISCV specification.
- 10/01/23: Completing the Xv6 reference pages, providing a short and concise summary of all functions and their role in the operating system overall.
- 10/15/23: Reorganizing the Xv6 code into more sensible modules. Additionally making the organization of the codebase more apparent by renaming the functions to remove abbreviations and using prefix name spacing.
- 10/22/23: Commenting the Xv6 source code fully.
- 10/22/23: Creating a development environment to build Xv6, run it with QEMU, and debug it with GDB.
- 11/12/23 Modularizing and editing the Xv6 source code such that it is possible to build it:
    - With or without virtual memory.
    - With or without concurrency.
    - With a custom scheduler.
- 1Creating a course plan with slides, exams, and projects.
- 11/24/23: Creating a web based development environment, possibly based on gdbgui.
- 11/24/23: 1st Draft of thesis submitted to advisor
- 12/01/23: 2nd Draft of Thesis submitted to HPD
- 12/08/23: Oral Defense (tentative)
- 12/08/23: Final Submission to CHC

9. References

- "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, December 2019. Retrieved December 12, 2022, from https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.
- "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203", Editors Andrew Waterman, Krste Asanovic, and John Hauser, RISC-V International, December 2021. Retrieved December 12, 2022, from https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf.
- "xv6: a simple, Unix-like teaching operating system", Russ Cox, Frans Kaashoek, and Robert Morris, MIT, August 2021. Retrieved December 12, 2022, from https://pdos.csail.mit.edu/6.S081/2020/xv6/book-riscv-rev1.pdf.
- "Teaching Operating Systems: Just Enough Abstraction", Philip Machanik, Rhodes University, July 2016. Retrieved December 12, 2022, from http://dx.doi.org/10.1007/978-3-319-47680-3_10.
- "RISC-V Instruction Set Reference." *Rv8*. Retrieved 12 June 2023, from , michaeljclark.github.io/isa.
- Riasanovsky, Nick. *Understanding RISC-V Calling Convention - University of California*, inst.eecs.berkeley.edu/~cs61c/resources/RISCV_Calling_Convention.pdf. Accessed 20 June 2023.
- Paul E. Dickson, Tim Richards, and Brett A. Becker. 2022. *Experiences Implementing and Utilizing a Notional Machine in the Classroom*. In Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1 (SIGCSE 2022), Vol. 1. Association for Computing Machinery, New York, NY, USA, 850–856. https://doi.org/10.1145/3478431.3499320
- Torvalds, L., et al. *Linux Kernel Source Tree*. GitHub. https://github.com/torvalds/linux