Project: Virtual Memory on Wally, Version 1.0.1 [1]

Assigned: Monday 4/14; Due **Monday 5/2** (midnight)

Instructor: James E. Stine, Jr.
TA: Jacob Pease / Thomas Kidd

1. Introduction

Virtual memory is a cornerstone of modern computing, providing a powerful abstraction that allows systems to manage memory efficiently, securely, and flexibly. By separating the logical memory used by applications from the actual physical memory available in hardware, virtual memory enables complex software systems to run reliably on machines with limited resources. Though largely invisible to end users, virtual memory affects virtually every aspect of everyday computing—from running multiple applications simultaneously to securing data and optimizing system performance.

The concept of virtual memory was first introduced in the 1950s and early 1960s, during the formative years of operating system development. The Atlas computer, developed at the University of Manchester in the early 1960s, is widely credited as the first system to implement virtual memory. Its designers sought to enable large programs to run on systems with limited physical memory, and the innovation of paging—dividing memory into fixed-size blocks and swapping them between disk and RAM—provided a practical solution. This approach laid the groundwork for the virtual memory systems that are standard today in all major operating systems, including Windows, macOS, and Linux.

In its earliest implementations, virtual memory was a technical solution to a pressing hardware constraint: physical memory was expensive and scarce. But as computing evolved, so too did the role of virtual memory. It became not just a way to extend memory capacity, but a mechanism for isolating processes, managing permissions, and enabling features like memory-mapped files and shared libraries. The development of hardware support for virtual memory, such as Memory Management Units (MMUs) in CPUs, further cemented its place in computer architecture.

In modern computing environments, virtual memory enables critical capabilities that users rely on every day. When you open multiple applications on your computer—editing a document, streaming a video, or browsing the internet—each program operates as if it has access to its own dedicated memory. Behind the scenes, the operating system uses virtual memory to map these logical address spaces to physical memory dynamically. If physical RAM becomes full, the system can temporarily move inactive pages to disk storage, a process known as swapping or paging out. This transparent memory management allows for smooth multitasking and better resource utilization without user intervention.

Virtual memory also underpins system security. Because each process is confined to its own virtual address space, it cannot read or overwrite the memory of another process—preventing a wide range of bugs and malicious exploits. Techniques like address space layout randomization (ASLR) further enhance security by randomizing memory locations, making it harder for attackers to predict and exploit vulnerabilities.

In summary, virtual memory is not merely a behind-the-scenes implementation detail—it is a foundational technology that has evolved over decades to support the demands of modern software and hardware. Its historical roots reflect the ingenuity of early computing pioneers, and its ongoing role ensures that systems remain responsive, secure, and capable of handling the diverse workloads of today's digital world.

This project's objective is a comprehensive experience related to computer architecture. For this project, you will use all your skills that you learned throughout the semester. It is our hope that this project will also allow you to see that how important memory and processor performance is to a system. For full credit, you should achieve the following in this project.

The objectives of this comprehensive project are the following:

[1] Note that this document is updated periodically to provide you with the most reliable information to complete the project. Although previous versions of this document are basically correct, frequent revisions are made in order to help you save time and **not** to hinder your performance. Therefore, make sure you check back periodically for updates. I will add revision numbers to help you identify which version of this document you have.

1. **Implement Virtual-to-Physical Memory Translation Using Page Tables**
   Develop an RV32 program that sets up virtual memory mapping using RISC-V's Sv32 scheme. Specifically, map two virtual addresses (e.g., `0x80000000` and `0x90000000`) to the same physical page (`0x80000000`) using software page table setup. This demonstrates aliasing and shared memory behavior.

2. **Configure and Enable Virtual Memory in Supervisor Mode**
   Modify the Wally RISC-V processor to support entering supervisor mode, configure the `satp` register with the base address of the page table, and enable address translation. Verify that the program can correctly execute code from a remapped virtual address, ensuring the translation mechanism functions as intended.

3. **Validate Functionality Through Simulation and Logging Tools**
   Use the RISC-V Sail model or Spike to simulate the program execution and confirm virtual memory behavior. Analyze log files to ensure correct address translation and that the jump to remapped virtual address space executes as expected, demonstrating a practical understanding of demand paging and memory protection.

2. How Virtual Memory Works

At the heart of virtual memory lies the mechanism of address translation. When a program accesses memory, it uses virtual addresses, which are not tied directly to the actual physical memory addresses in RAM. Instead, the system uses a Memory Management Unit (MMU) to translate these virtual addresses into physical addresses at runtime. This translation process is made possible through a data structure known as a page table, which keeps track of the mapping between virtual pages and physical pages.

Memory is divided into fixed-size blocks called pages (typically 4KB in size), and physical memory is divided into equally sized units called pages. The operating system maintains a page table for each process, where each entry specifies whether a page is currently loaded into RAM and, if so, which page it occupies. If a page is not present in RAM, a page fault occurs, prompting the operating system to fetch the page from secondary storage, such as a hard disk or SSD, and load it into memory—possibly replacing an existing page using a page replacement policy like Least Recently Used (LRU).

Virtual memory systems also implement protection bits for each page table entry, ensuring that access permissions (read, write, execute) are enforced at the hardware level. This ensures that faulty or malicious code cannot corrupt other processes or the operating system kernel. Additionally, virtual memory enables memory sharing between processes, such as shared libraries, without duplicating content in physical memory—thereby conserving memory and reducing overhead.

To speed up the translation process, modern CPUs use a small, fast cache called the Translation Lookaside Buffer (TLB). The TLB stores recent address translations, allowing the MMU to bypass the full page table lookup for frequently accessed pages. This significantly enhances performance, especially for applications with high locality of reference.

Thus, virtual memory operates as a sophisticated collaboration between the hardware (MMU and TLB) and the operating system (page table management, swapping, and protection). This design provides the illusion of a large, contiguous memory space to each process while allowing the system to manage physical resources efficiently and securely.

3. Concepts in Virtual Memory

3.1 PMP and PMA: Memory Protection

Most processors require an address decoder to access the appropriate bank of memory or I/O peripheral specified by a physical address. RISC-V uses a PMA checker to perform this address decoding. The PMA specifies and enforces the memory map and its attributes and is often hardwired at design time.

RISC-V specifies that a core must contain a platform-dependent PMA that generates select signals for each memory bank or I/O peripheral and produces access faults if the memory access is unsuccessful. An unsuccessful access may be caused by a physical address located in a vacant portion of the memory map or an unsupported operation (e.g., an access to a peripheral that is incorrectly sized).

For security and to detect pointer errors in programs, it is helpful to limit which physical addresses software may access. An optional RISC-V physical memory protection (PMP) checker enforces protected address ranges concurrently with the PMA checker [1]. By default, the PMP grants machine-mode access to all parts of physical memory and denies supervisor- and user- mode access to all of physical memory; the PMP is then configured in software to grant access to certain portions in S- or U-mode or to revoke access to certain portions in M-mode. For example, the OpenSBI M-mode firmware configures PMP so that S- and U-mode software can access all memory except a region used by the firmware. Even if malicious software compromises the operating system, the PMP prevents it from corrupting the firmware.

3.2 Paging

Paging is a lazy loading strategy that loads a page into memory only when it is accessed. At program startup, only essential pages (like code entry points) are loaded; others are fetched when needed. This minimizes memory usage and improves startup performance.

When a process accesses a page that is not present in physical memory, the CPU triggers a page fault. The operating system then:

1. Suspends the process.
2. Locates the page on disk (e.g., from a swap file).
3. Loads the page into a free page.
4. Updates the page table and resumes execution.

Paging is critical in modern systems because it enables large applications to run on systems with limited RAM. It also provides this "magic" to convert the address automatically.

3.3 Multi-Level Paging and Cache Interaction

To handle large address spaces (e.g., 64-bit), systems implement multi-level paging. Instead of using a single flat page table, the table is broken into a hierarchy, often 2 to 4 levels deep. This structure significantly reduces memory overhead by allocating page tables only when needed. For this lab, you will use kilopage as indicated in Chapter 11.2 of [1]. Specifically, we are using `Sv32` type of virtual memory which Wally supports and we will only deal with kilopages here. If interested, students can investigate morere the way to handle megapages vs. kilopages as indicated in Chapter 11 of our textbook.

In conjunction with virtual memory, modern CPUs use a hierarchy of caches (e.g., L1, L2, and L3)—to accelerate memory access. These caches store recently accessed data and address translations (via the Translation Lookaside Buffer or the TLB), as discussed in class.

4. Project Assignment: Exploring Virtual Memory Using Wally

This project introduces students to the core mechanisms of virtual memory in the RISC-V architecture using the RV32 instruction set and the Wally (`https://github.com/openhwgroup/cvw` processor core, a pedagogically focused, open-source RISC-V implementation. The objective is to deepen students' understanding of address translation, privilege levels, and page table setup by implementing and debugging a minimal virtual memory environment directly on the Wally platform. Students will work at the software-hardware boundary—configuring virtual memory support in software while targeting the Wally core for cycle-accurate simulation and architectural tracing.

4.1 Assignment Description

You should write a RV32 program that executes the following tasks:

- Write an RV32 program that sets up virtual kilopages at `0x80000000` at `0x90000000` both mapping to the same physical page at `0x80000000` (a sample stubbed program is given to you).
- Enter **supervisor mode** by configuring RISC-V CSRs including `mstatus`, `satp`, and `mepc`.
- Enable virtual memory using the `satp` register to activate the page-based translation and protection mechanism.
- Perform a jump to the virtual address `0x90000000`, where instructions are fetched and executed from the aliased physical page, confirming the correctness of virtual memory translation.
- Simulate with Sail, debug, and confirm the log shows this behavior.
- You should also run the program in HDL simulation with Wally - prove that the hardware is correctly handling the conversion of the address.
- Extra Credit: Write page table entries to define an `Sv32` page system with a megapage at virtual address `0x80000000` corresponding to a physical megapage at the same address that is executable, writable, and readable in user mode. The system also has a kilopage at virtual address `0x80800000` that maps to a physical page at address `0x1000` that is only readable in supervisor mode. Suppose satp contains `0x80080500`, and the level 0 page table for the kilopage is at `0x80501000`.

4.2 Target Architecture and Tools

This project is implemented and simulated using the Wally RV32 core, which supports the rv32gc instruction set and provides support for privilege mode transitions and page-based virtual memory (Sv32). You will use the `riscv_sim_rv32d` simulator derived from the Sail specification to validate program execution and inspect translation behavior:

```
riscv_sim_rv32d --pmp-count=16 11p6.elf > log
```

The simulator, configured with 16 PMP (Physical Memory Protection) regions, produces detailed logs of register changes, address translations, privilege transitions, and instruction fetches. These logs allow students to confirm:

- Successful activation of virtual memory (via correct `satp` configuration).
- That both virtual addresses `0x80000000` and `0x90000000` resolve to the same physical address.
- That execution flow properly transfers to code located at `0x90000000`.

4.3 Sail Simulation

Computer architectures, like any other engineering implementation, have to be verified. This is difficult as there are many things going on when a program is run. They could be something on the datapath, memory timing, or even interacting with humans. The RISC-V Community coped with these items by creating a language that helps with verification, called Sail.

Again, the official formal and executable specification of the RISC-V architecture [RISCV-SAIL] is written in a language called Sail. Indeed, if the English specification ever is ambiguous or deviates from the Sail specification, Sail takes priority. Sail is a pseudocode-like language for describing instruction set architectures [SAIL] that serves several purposes as explained in Chapter 3 of our textbook [1]:

- Document the behavior of each instruction in an ISA in a precise and readable way
- Automatically create a simulator that exactly matches the specification

- Generate definitions for theorem provers aiming to prove properties about an ISA or implementation.

As seen later, we will be using the `riscv_sim_rv32d` simulator. Sail produces these RISC-V simulators called `riscv_sim_RV32` and `riscv_sim_64`. The simulators are compiled in C based on the Sail specification, so they are also known as `sail_cSim`. As explained in the textbook [1], the term "Sail" is utilized colloquially for these simulators and is automatically produced from the Sail language description of RISC-V.

Now, Let us look at a specific exampling using this simulator and compare to what is happening with our hardware. Check out the `sumtest` example in the repository. You should run this program with both the spike signature and the Sail simulator. Do you get the same result? To run this with Sail simulator, type the following after compiling the program:

```
riscv_sim_rv32d sumtest -T sumtest.sig
```

Note: before you go forward, you should understand HTIF works and its operation.

4.4 Learning Outcomes

This hands-on project allows students to:

- Construct page tables using the Sv32 format and apply correct alignment and permission bits.
- Gain experience with RISC-V CSRs and supervisor-level features of the Wally core.
- Debug virtual memory issues using Sail logs and match high-level program behavior with low-level hardware trace data.

By building directly on the Wally platform, students gain practical experience with virtual memory systems in a RISC-V SoC context, reinforcing system-level concepts through simulation, architectural inspection, and real-world debugging practices.

4.5 What you need to do

As explained in class, the use of virtual memory is key to any computer system. You will need to do a couple of things, but you will need Wally. The first thing you need to do is make sure you have a fresh pull of the `cvw` repository. You will use this later in the project, but will need this to see what is happening with the hardware.

Second, you will need to use the `CSRs` to set up the virtual memory. Before, you begin, consult the documentation in the Chapter 11 of the textbook [1]. You can also inspect the privileged instruction set manual which is also available here: `https://github.com/riscv/riscv-isa-manual`. You will

5. Testing and Validation Through Simulation

To ensure upir virtual memory implementation functions correctly, you will use the **Sail RISC-V simulator** in conjunction with the **Wally** RV32 core. Sail provides a formally-specified reference model that produces detailed execution logs, allowing students to trace the behavior of every instruction, CSR modification, and memory translation step.

5.1 Simulation Workflow

The following command is used to simulate the student-generated ELF file with virtual memory enabled:

```
riscv_sim_rv32d --pmp-count=16 project.elf > log
```

This simulation executes the compiled program and redirects all execution output—including instruction traces, privilege transitions, and memory access information—into the file `log`.

5.2 Key Observables in the Log File

To verify correct implementation, students should examine the log file for the following evidence:

- **SATP Register Setup**: Look for the write to the `satp` CSR indicating the enabling of virtual memory (Sv32 mode) and the root page table physical address. This confirms that paging has been activated.
- **Privilege Transition**: Ensure that the mode changes from machine mode (M-mode) to supervisor mode (S-mode). This is required for the system to honor the page table mappings.
- **Address Translation**: Look for translation of virtual addresses `0x80000000` and `0x90000000` both resolving to the same physical address (e.g., `0x80000000`). The log should show that instruction fetches from `0x90000000` are being served by the aliased physical page.
- **Instruction Execution**: Confirm that valid instructions are being fetched and executed from `0x90000000`, demonstrating that the program successfully jumped to that virtual address and continued executing code that is shared with `0x80000000`.

5.3 Recommended Debugging Strategy

Students encountering unexpected behavior are encouraged to use the following steps:

1. **Double-check Page Table Entries (PTEs)**: Ensure that the entries are properly aligned (4KiB page boundaries), have the correct permissions (valid, readable, executable), and map to the correct physical page.
2. **Verify CSR Setup**: Confirm that `satp` contains the correct mode (Sv32) and physical address of the page table base. Ensure `mstatus` has the proper bits set to allow S-mode execution.
3. **Insert Instrumentation**: Use simple `addi` and `csrr` instructions at key stages (e.g., before and after enabling paging) to log CSR states and register contents. You can check this with the Sail simulator
4. **Trace Jump Behavior**: Use log messages to verify that the jump to `0x90000000` actually occurs and that subsequent instruction fetches are correctly redirected.

5.4 What Success Looks Like

A correct implementation will produce a log where:

- The `satp` register is set with a valid page table address and `Sv32` mode.
- The system enters supervisor mode and remains there for the remainder of execution.
- Both `0x80000000` and `0x90000000` are translated to the same physical page.
- Executable instructions are fetched and run from the `0x90000000` virtual address.

This simulation-based approach allows anyone to reason about the correctness of their virtual memory implementation, debug complex translation behavior, and gain confidence in understanding how low-level mechanisms drive memory management on real systems like Wally.

5.5 Checking with Wally

You should also check that your virtual memory is working. You can do this with `wsim` command. This can be run with your program you compiled:

```
wsim rv32gc --elf example --gui
```

Check your program and whether the virtual memory translation is working. There should be a section for the MMU and you should be familiar with the Wave window on Questa/ModelSim. To understand the Wave window, you may want to inspect the top-level diagram of Wally as found in Chapter 2 of the textbook [1].

6. Getting Started & Tips

- If you have not had me before in a class, you are pretty much not going to finish this or any assignment if you start late - guaranteed[2]! Do not believe me, ask others who have had my classes. So, start early and apportion your time every day, if possible.
- It is very important you understand the terminology of paging. Therefore, spend some time reading Chapter 11 [1] and Chapter 8 of DDCA [2] extensively.
- Make sure you have Wally cloned in your home directory and test that it works by running `regression-wally`. The procedure for performing this is:
  - Clone the Wally repository
  - `cd cvw`
  - `source setup.sh`
  - `cd sim`
  - `make`
  - If successful, type `regression-wally`

  If you have an error, try the following run the commands again:

  ```
  git submodule update addins/riscv-arch-test
  ```

  from your main `cvw` cloned directory.
- It is a good idea to inspect the HTIF implementation of `sumtest` in the repository. Try to understand how to run a program here. Try another program to understand HTIF well
- Spend time understanding the Sail simulator along with `sumtest` before trying to configure things.
- The main project subdirectory of your repository has a stubbed C file that you can use. I tried to update where you will make changes. To configure virtual memory, you will have to understand how to use the privileged registers as discussed in class and Chapter 8 of the textbook [1].
- Read this handout closely and Chapter 11 closely. Ask questions to the TA or me using slack or in person.

7. Handin

You should electronically hand in your code into Canvas as with Project. Please contact the TAs or the instructor for more help. Your code should be readable and well-documented. In addition, please turn in additional test cases that you used in a inputs/ subdirectory. If you feel the need to describe any additional aspects of your design in detail, please include these in a separate README.

8. Extra Credit

Again, there are many possibilities for extra credit, but you should only attempt this if you get the baseline project done and you have extra time.

[2] Luck favors the prepared!

References

[1] D. Harris, J. Stine, S. Harris, and R. Thompson, *RISC-V Microprocessor System-On-Chip Design*. Elsevier Science, 2025.

[2] S. Harris and D. Harris, *Digital Design and Computer Architecture, RISC-V Edition*. Elsevier Science, 2021.