Final Project Documentation: Advanced Operating System Simulator

Brysen Pfingsten, Nathaniel Savoury,
David Fields

December 8, 2025

Contents

1 Introduction

1.1 Problem Statement

- Module 1: Process Simulation

- Module 2: Advanced Memory Management

- Module 3: Process Scheduling and Context Switching

- Module 4: Interrupt Handling and Dispatcher

- Module 5: Efficiency Analysis of Concurrency

1.2 Outline

2 Key Concepts and Features

2.1 Project 1

2.2 Project 2

2.3 Project 3

Wasn't Assigned

2.4 Project 4

Wasn't Assigned

3 Module 1: Process Simulation

3.1 Problem Statement

3.2 Implementation

3.2.1 Core CPU Components and Registers

4 Overview

This document specifies a small, real 32-bit ISA based closely on the original MIPS I architecture. All instructions are 32 bits wide and follow one of three formats: R-type, I-type, or J-type. The ISA includes:

- 32 general-purpose registers (GPRs).
- Special registers: PC, HI, LO.
- Basic system control registers: Status, Cause, EPC.
- Integer arithmetic and logical operations.
- Multiply and divide.
- Shift operations.
- Load and store instructions for bytes, halfwords, and words.
- Branch and jump instructions.
- System call and exception/interrupt return.

5 Registers

5.1 General-Purpose Registers

There are 32 general-purpose registers, each 32 bits wide.

Number	Name	Role / Convention
$0	zero	Constant zero, reads as 0, writes ignored
$1	at	Assembler temporary
$2–$3	v0--v1	Function return values
$4–$7	a0--a3	Function arguments
$8–$15	t0--t7	Temporaries (caller-saved)
$16–$23	s0--s7	Saved registers (callee-saved)
$24–$25	t8--t9	Temporaries
$26–$27	k0--k1	Reserved for kernel / OS use
$28	gp	Global pointer
$29	sp	Stack pointer
$30	fp/s8	Frame pointer or extra saved register
$31	ra	Return address for calls (JAL, JALR)

5.2 Special Registers

- **PC** (Program Counter): 32-bit address of the current instruction.
- **HI**, **LO**: 32-bit registers used to hold results of multiply and divide.
- **Status**: System status register (holds interrupt enable and mode bits).
- **Cause**: Encodes reason for last exception or interrupt.
- **EPC** (Exception Program Counter): Holds the address to return to after an exception, used by ERET.

6 Instruction Formats

All instructions are 32 bits. Bit 31 is the most significant bit (MSB).

6.1 R-Type Format

31–26	25–21	20–16	15–11	10–6	5–0
opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

For all R-type instructions:
$$\text{opcode} = 0.$$

6.2 I-Type Format

31–26	25–21	20–16	15–0
opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

The 16-bit immediate is sign-extended or zero-extended depending on the instruction.

6.3 J-Type Format

31–26	25–0
opcode	target
6 bits	26 bits

The effective jump address is formed as:

$$\text{PC}_\text{next} = \big(\text{PC}_\text{current}[31{:}28] \ll 28\big) \mid (\texttt{target} \ll 2).$$

7 Instruction Encoding and Semantics

This section lists the instructions in the ISA, along with their encoding and semantic meaning. All arithmetic is on 32-bit two's complement integers unless otherwise stated. For brevity, we use the following notation:

- $\text{GPR}[i]$: contents of general-purpose register i.
- PC: program counter.
- HI, LO: special multiply/divide registers.
- $\text{Mem}[a]$: memory access at byte address a.
- $\text{sext}_n(x)$: sign extension of x from n bits to 32 bits.
- $\text{zext}_n(x)$: zero extension of x from n bits to 32 bits.

7.1 Integer Arithmetic (R-Type)

All of these have $\texttt{opcode} = 0$.

Mnemonic	Format	Encoding	Description	Semantics
ADD	R	$\texttt{funct} = 0\text{x}20$	Add (signed)	$\text{GPR}[rd] = \text{GPR}[rs] + \text{GPR}[rt]$
ADDU	R	$\texttt{funct} = 0\text{x}21$	Add (unsigned)	Same as ADD but ignore signed overflow
SUB	R	$\texttt{funct} = 0\text{x}22$	Subtract (signed)	$\text{GPR}[rd] = \text{GPR}[rs] - \text{GPR}[rt]$
SUBU	R	$\texttt{funct} = 0\text{x}23$	Subtract (unsigned)	Same as SUB but ignore signed overflow

7.2 Multiply and Divide (R-Type)

Mnemonic	Format	Encoding	Description	Semantics
MULT	R	$\texttt{funct} = 0\text{x}18$	Signed multiply	$\{\text{HI}, \text{LO}\} = \text{sext}_{64}(\text{GPR}[rs]) \times \text{sext}_{64}(\text{GPR}[rt])$
MULTU	R	$\texttt{funct} = 0\text{x}19$	Unsigned multiply	$\{\text{HI}, \text{LO}\} = \text{zext}_{64}(\text{GPR}[rs]) \times \text{zext}_{64}(\text{GPR}[rt])$
DIV	R	$\texttt{funct} = 0\text{x}1A$	Signed divide	$\text{LO} = \text{GPR}[rs]/\text{GPR}[rt]$, $\text{HI} = \text{GPR}[rs] \bmod \text{GPR}[rt]$
DIVU	R	$\texttt{funct} = 0\text{x}1B$	Unsigned divide	$\text{LO} = \text{GPR}[rs]_u/\text{GPR}[rt]_u$, $\text{HI} = \text{GPR}[rs]_u \bmod \text{GPR}[rt]_u$
MFHI	R	$\texttt{funct} = 0\text{x}10, \texttt{rs} = \texttt{rt} = 0$	Move from HI	$\text{GPR}[rd] = \text{HI}$
MFLO	R	$\texttt{funct} = 0\text{x}12, \texttt{rs} = \texttt{rt} = 0$	Move from LO	$\text{GPR}[rd] = \text{LO}$
MTHI	R	$\texttt{funct} = 0\text{x}11, \texttt{rt} = \texttt{rd} = 0$	Move to HI	$\text{HI} = \text{GPR}[rs]$
MTLO	R	$\texttt{funct} = 0\text{x}13, \texttt{rt} = \texttt{rd} = 0$	Move to LO	$\text{LO} = \text{GPR}[rs]$

7.3 Logical and Bitwise (R-Type)

Mnemonic	Format	Encoding	Description	Semantics
AND	R	$\texttt{funct} = \text{0x24}$	Bitwise AND	$\text{GPR}[rd] = \text{GPR}[rs] \wedge \text{GPR}[rt]$
OR	R	$\texttt{funct} = \text{0x25}$	Bitwise OR	$\text{GPR}[rd] = \text{GPR}[rs] \vee \text{GPR}[rt]$
XOR	R	$\texttt{funct} = \text{0x26}$	Bitwise XOR	$\text{GPR}[rd] = \text{GPR}[rs] \oplus \text{GPR}[rt]$
NOR	R	$\texttt{funct} = \text{0x27}$	Bitwise NOR	$\text{GPR}[rd] = \neg(\text{GPR}[rs] \vee \text{GPR}[rt])$

7.4 Shift Instructions (R-Type)

Mnemonic	Format	Encoding	Description	Semantics
SLL	R	$\texttt{funct} = \text{0x00}$	Shift left logical (immediate)	$\text{GPR}[rd] = \text{GPR}[rt] \ll \texttt{shamt}$
SRL	R	$\texttt{funct} = \text{0x02}$	Shift right logical (immediate)	$\text{GPR}[rd] = \text{GPR}[rt] \gg \texttt{shamt}$ (logical)
SRA	R	$\texttt{funct} = \text{0x03}$	Shift right arithmetic (immediate)	Arithmetic right shift, preserving sign bit
SLLV	R	$\texttt{funct} = \text{0x04}$	Shift left logical (variable)	$\text{GPR}[rd] = \text{GPR}[rt] \ll (\text{GPR}[rs] \,\&\, 0x1F)$
SRLV	R	$\texttt{funct} = \text{0x06}$	Shift right logical (variable)	$\text{GPR}[rd] = \text{GPR}[rt] \gg (\text{GPR}[rs] \,\&\, 0x1F)$ (logical)
SRAV	R	$\texttt{funct} = \text{0x07}$	Shift right arithmetic (variable)	Arithmetic right shift by low 5 bits of $\text{GPR}[rs]$

7.5 Immediate Arithmetic and Logical (I-Type)

Mnemonic	Format	Opcode	Description	Semantics
ADDI	I	0x08	Add immediate (signed)	$\text{GPR}[rt] = \text{GPR}[rs] + \text{sext}_{16}(\texttt{imm})$
ADDIU	I	0x09	Add immediate (unsigned)	Same as $\texttt{ADDI}$ but ignore signed overflow
ANDI	I	0x0C	And immediate	$\text{GPR}[rt] = \text{GPR}[rs] \wedge \text{zext}_{16}(\texttt{imm})$
ORI	I	0x0D	Or immediate	$\text{GPR}[rt] = \text{GPR}[rs] \vee \text{zext}_{16}(\texttt{imm})$
XORI	I	0x0E	Xor immediate	$\text{GPR}[rt] = \text{GPR}[rs] \oplus \text{zext}_{16}(\texttt{imm})$
SLTI	I	0x0A	Set less than immediate (signed)	$\text{GPR}[rt] = (\text{GPR}[rs] < \text{sext}_{16}(\texttt{imm}))?1:0$
SLTIU	I	0x0B	Set less than immediate (unsigned)	Unsigned comparison version of $\texttt{SLTI}$
LUI	I	0x0F	Load upper immediate	$\text{GPR}[rt] = \texttt{imm} \ll 16$

7.6 Load and Store (I-Type)

Effective address:

$$\text{EA} = \text{GPR}[rs] + \text{sext}_{16}(\texttt{imm}).$$

Memory is typically treated as byte-addressed, little-endian.

Mnemonic	Format	Opcode	Description	Semantics
LW	I	0x23	Load word	$\text{GPR}[rt] = \text{Mem32}[\text{EA}]$
SW	I	0x2B	Store word	$\text{Mem32}[\text{EA}] = \text{GPR}[rt]$
LB	I	0x20	Load byte (signed)	$\text{GPR}[rt] = \text{sext}_8(\text{Mem8}[\text{EA}])$
LBU	I	0x24	Load byte (unsigned)	$\text{GPR}[rt] = \text{zext}_8(\text{Mem8}[\text{EA}])$
LH	I	0x21	Load halfword (signed)	$\text{GPR}[rt] = \text{sext}_{16}(\text{Mem16}[\text{EA}])$
LHU	I	0x25	Load halfword (unsigned)	$\text{GPR}[rt] = \text{zext}_{16}(\text{Mem16}[\text{EA}])$
SB	I	0x28	Store byte	$\text{Mem8}[\text{EA}] = \text{GPR}[rt] \,\&\, 0xFF$
SH	I	0x29	Store halfword	$\text{Mem16}[\text{EA}] = \text{GPR}[rt] \,\&\, 0xFFFF$

7.7 Branches (I-Type)

The branch target address is computed relative to the address of the instruction *following* the branch. Let PC_{next} be the PC after fetching the branch (i.e., $\text{PC} + 4$). Then:

$$\text{Target} = \text{PC}_{\text{next}} + \big(\text{sext}_{16}(\texttt{imm}) \ll 2\big).$$

Mnemonic	Format	Opcode	Description	Semantics
BEQ	I	0x04	Branch if equal	If GPR$[rs]$ = GPR$[rt]$, then PC = Target
BNE	I	0x05	Branch if not equal	If GPR$[rs]$ $\neq$ GPR$[rt]$, then PC = Target

7.8 Jumps (J-Type and R-Type)

Mnemonic	Format	Opcode/Funct	Description	Semantics
J	J	`opcode = 0x02`	Jump	PC = (PC$_{\text{current}}$[31:28] $\ll$ 28) $\mid$ (`target` $\ll$ 2)
JAL	J	`opcode = 0x03`	Jump and link	GPR[31] = PC$_{\text{next}}$; then same as J
JR	R	`funct = 0x08`	Jump register	PC = GPR$[rs]$
JALR	R	`funct = 0x09`	Jump and link register	GPR$[rd]$ = PC$_{\text{next}}$; PC = GPR$[rs]$

7.9 System and Exception Instructions

For system and exception-related instructions, we describe them in prose rather than putting lists inside table cells (which can cause LaTeX errors).

SYSCALL

Encoded as an R-type instruction with `opcode = 0` and `funct = 0x0C`. When executed, this instruction triggers a software exception. The simulator should:

1. Save the appropriate instruction address into `EPC` (either the address of the syscall or the next instruction, depending on your chosen convention).
2. Set `Cause` to a code representing a system call exception.
3. Update `Status` to indicate kernel mode and (optionally) disable further interrupts.
4. Set `PC` to the configured exception vector address (e.g. `0x80000180`).

BREAK

Encoded as an R-type instruction with `opcode = 0` and `funct = 0x0D`. When executed, this triggers a breakpoint exception, which is handled similarly to `SYSCALL`, but with a different `Cause` code to distinguish it (e.g. for debugging or traps).

7.10 Exception Return (ERET)

In real MIPS this is encoded as a coprocessor 0 instruction. For this ISA we define:

- `opcode = 0x10` (COP0),
- `rs = 0x10`,
- bits 5–0 (funct) = 0x18,
- all other fields zero.

Decoding is implemented as a special case: "if opcode is 0x10 and `funct = 0x18`, execute `ERET`."

Semantics.

- PC $\leftarrow$ `EPC`.
- Restore user/kernel mode and interrupt enable bits in `Status` as appropriate.

8 Exception and Interrupt Model

8.1 Exception Types

Typical exception causes include:

- System call (`SYSCALL`).
- Breakpoint (`BREAK`).

- Arithmetic overflow (e.g., `ADD` with overflow).
- Invalid instruction.
- Address error on load/store.
- External interrupt (e.g., timer, I/O).

The simulator sets `Cause` to an integer code representing one of these reasons.

8.2 Exception Entry

On an exception or interrupt, the CPU performs:

1. Save the faulting instruction address or the following address into `EPC`.
2. Set `Cause` to the appropriate exception code.
3. Modify `Status` to:
   - switch to kernel mode,
   - optionally disable further interrupts.
4. Set `PC` to a fixed exception vector address, e.g. `0x80000180`.

The kernel's exception handler at that address can then inspect `Cause`, `EPC`, and general registers to decide what to do.

8.3 Exception Return

When the kernel is finished handling the exception or interrupt, it executes `ERET`, which:

- restores `PC` from `EPC`,
- restores user/kernel mode (and possibly interrupt enable) from `Status`.

8.3.1   Process Control Block

8.3.2   Fetch-Decode-Execute Cycle

9 Module 2: Advanced Memory Management

9.1 Problem Statement

9.2 Implementation

9.2.1 Hierarchical Memory System

9.2.2 Memory Table

9.2.3 Dynamic Memory Allocation and Deallocation

10 Module 3: Process Scheduling and Context Switching

10.1 Problem Statement

alla

10.2 Implementation

akda

10.2.1 Process Control Block Enhancements

lsok

10.2.2 Scheduling Algorithms

ls

Round-Robin

ls

Priority-Based Scheduling

lsf

Shortest Time Remaining

lsf

Highest Response Ratio Next

lsv

First Come First Serve

kfs

Shortest Process Next

skf

Feedback Scheduling

jsnf

10.2.3 Context Switching

snfj

10.2.4 Integration with Fetch-Decode-Execute Cycle

jnsf

11 Module 4: Interrupt Handling and Dispatcher

11.1 Problem Statement

11.2 Implementation

11.2.1 Types of Interruption

11.2.2 Interrupt Vector Table

11.2.3 Context Switching

12 Module 5: Efficiency Analysis of Concurrency

12.1 Problem Statement

12.2 Implementation

12.2.1 Performance Metrics Setup

12.2.2 Implementation of Time Tracking

12.2.3 Data Comparison

12.2.4 Performance Comparison

12.2.5 Visualization and Reporting

13   The Simulation

14 Testing and Debugging

15 Conclusion

16   Appendix A: Screenshots