
n64docs Documentation

Release latest

Dillon Beliveau

Aug 30, 2023

Contents

1	Assumed Prior Knowledge	3
1.1	Overview	3
1.2	Memory Map	4
1.3	Boot Process	6
1.4	TLB	7
1.5	CPU (detailed)	7
1.6	Interrupts	12
1.7	Exceptions	13
1.8	MIPS Interface	15
1.9	Video Interface	17
1.10	Audio Interface	19
1.11	Peripheral Interface	20
1.12	RDRAM Interface	20
1.13	Serial Interface	20
1.14	Issues Log	20

It is my hope that this document will make Nintendo 64 emulation a more approachable and accessible project for as many people as possible.

I aim to document the sharp edges I hurt myself on along the way, as well as collect as much of the information scattered around as possible into one document. Currently it is *VERY MUCH* a work in progress, and nowhere near complete.

This guide is intended to be used by emulator developers, but homebrew authors might find the information presented here useful as well.

I'm by no means the leading expert on the N64. If you have any corrections for me, please feel free to open an issue or pull request on my [GitHub repo](#).

Assumed Prior Knowledge

While my goal is to be as clear and accessible as possible, I expect readers to be somewhat familiar with emulation already. If you've written an emulator for a system on the level of the original GameBoy or NES, you should be fine.

1.1 Overview

The Nintendo 64 is an early 3D console with some interesting quirks. Let's get right into it.

The system is often thought of as having two main components - the CPU, and the Reality Coprocessor (RCP), with the RCP again being divided into two components, the Reality Signal Processor (RSP) and the Reality Display Processor (RDP.)

The system has 4MiB of RDRAM, expandable with an add-on (Expansion Pak) to 8MiB. The system is designed in such a way that all 3 processors (CPU, RSP, RDP) can access the same memory. Thus, this RAM acts as both normal system RAM and VRAM at the same time.

Interestingly, each byte in RDRAM is actually 9 bits. In other computer systems with RDRAM, this 9th bit is normally used as a parity bit for data integrity checking, but it has been repurposed in the N64 as extra storage for the RDP. The RDP uses it for depth buffering and anti-aliasing, usually.

The 9th bits are usually implemented in emulators as an entirely separate structure from the main RDRAM array.

All memory is accessed using big-endian.

1.1.1 CPU Overview

The CPU is a fairly standard 64 bit MIPS r4300i chip.

It has 32 *64-bit* registers. The first register, r0 or \$zero, is hardwired to a value of 0, at all times. There are also special-purpose registers for things like multiplication.

It has a 64 bit program counter and can be configured to use 32 bit or 64 bit addresses to access memory, though in practice the vast majority of software uses 32 bit addressing. It can load and store 8, 16, 32, and 64 bit values. All memory accesses must be aligned, and loads/stores with unaligned addresses will throw exceptions.

All memory accesses, including instruction fetches, use virtual memory. There are segments of the address space that use fixed translation, and segments that are configurable.

MIPS has the concept of “coprocessors” built directly into the instruction set. Each one is numbered, and there are special opcodes for interacting with them, including MTCx/MFCx (move to/from coprocessor x), which are used to move data between the main CPU registers and the coprocessor’s registers.

Coprocessor zero, or COP0, is the “system control coprocessor.” It is through COP0 that virtual memory is set up, exceptions and interrupts are controlled, the CPU status is accessed, among other things. There are 32 32-bit registers, most of which have special uses.

Coprocessor one, or COP1, is a floating-point unit, or FPU. It supports a variety of floating point operations on its 32 floating-point registers.

1.1.2 Reality Coprocessor Overview

As mentioned above, the RCP is actually composed of two separate components, the RSP and the RDP.

Reality Signal Processor Overview

The RSP is a secondary CPU mostly used for 3D and audio calculations. While it runs at 2/3rds the speed of the main CPU, it has powerful *Single Instruction Multiple Data* (SIMD) instructions that make it extremely suited for the type of math required by 3D graphics. These instructions perform the same operation on multiple values at once.

It is effectively a stripped-down version of the MIPS chip used for the main CPU, missing some instructions and running at a slightly slower speed.

It has 32 32-bit registers. Note that this is different than the main CPU, which has 64-bit registers! As in the CPU, the first register, r0 or \$zero, is hardwired to 0 at all times.

The RSP has no COP0 in the same way the CPU does. Instead, the COP0 registers are used to communicate with the DMA engine, RDP, and various other things. It also does not have a floating point unit available on COP1.

It, however, does have a Vector Unit (VU) available as COP2. The VU has quite a few registers which are discussed in the RSP section below. Sometimes, the normal operations of this CPU are referred to as the Scalar Unit (SU) to differentiate them from the vector unit.

All memory accesses use a physical address, there is no virtual memory involved here.

Unlike the CPU, the RSP is capable of reading and writing unaligned values. Not only will these accesses not throw exceptions, they’ll work perfectly! Again, note that memory accesses can only be to DMEM.

SP IMEM is only 0x1000 bytes in size and the bottom 12 bits of the program counter are used to address it. Because of this, the program counter can be thought of as being a simple 12 bit value.

Reality Display Processor Overview

The RDP is a rasterizer used to display images on-screen.

1.2 Memory Map

1.2.1 Virtual Memory Map

ALL memory accesses made by the main CPU, both instruction fetches and through load/store instructions, use virtual addresses. These addresses are translated to physical addresses by the Memory Management Unit (MMU) before they

are actually used to access hardware.

Address Range		Name	Description
0x00000000	0x7FFFFFFF	KUSEG	User segment. TLB mapped
0x80000000	0x9FFFFFFF	KSEG0	Kernel segment 0. Direct mapped, cached.
0xA0000000	0xBFFFFFFF	KSEG1	Kernel segment 1. Direct mapped, no cache.
0xC0000000	0xDFFFFFFF	KSSEG	Kernel supervisor segment. TLB mapped.
0xE0000000	0xFFFFFFFF	KSEG3	Kernel segment 3. TLB mapped.

Note that the cacheing mentioned above is not critical for emulation.

Virtual-to-Physical Address Translation

For the direct mapped segments (KSEG0 and KSEG1), the translation is easy. Simply subtract the start of the segment the address is in from the address itself.

For example, both virtual address 0x80001000 (in KSEG0) and virtual address 0xA0001000 (in KSEG1) would translate to the physical address 0x00001000.

For TLB-mapped segments, things are a bit more complicated. See the TLB section for more details.

1.2.2 Physical Memory Map

Address Range	Name	Description
0x00000000 0x003FFFFFFF	RDRAM	RDRAM - built in
0x00400000 0x007FFFFFFF	RDRAM	RDRAM - expansion pak (available if inserted)
0x00800000 0x03EFFFFFFF	Unused	Unused
0x03F00000 0x03FFFFFFF	RDRAM Registers	RDRAM MMIO, configures timings, etc. Irrelevant for emulation
0x04000000 0x04000FFF	SP DMEM	RSP Data Memory
0x04001000 0x04001FFF	SP IMEM	RSP Instruction Memory
0x04002000 0x0403FFFF	Unused	Unused
0x04040000 0x040FFFFF	SP Registers	Control RSP DMA engine, status, program counter
0x04100000 0x041FFFFF	DP Command Registers	Send commands to the RDP
0x04200000 0x042FFFFF	DP Span Registers	Unknown
0x04300000 0x043FFFFF	MIPS Interface (MI)	System information, interrupts.
0x04400000 0x044FFFFF	Video Interface (VI)	Screen resolution, framebuffer settings
0x04500000 0x045FFFFF	Audio Interface (AI)	Control the audio subsystem
0x04600000 0x046FFFFF	Peripheral Interface (PI)	Control the cartridge interface. Set up DMAs cart <==> RDRAM
0x04700000 0x047FFFFF	RDRAM Interface (RI)	Control RDRAM settings (timings?) Irrelevant for emulation.
0x04800000 0x048FFFFF	Serial Interface (SI)	Control PIF RAM <==> RDRAM DMA engine
0x04900000 0x04FFFFFFF	Unused	Unused
0x05000000 0x05FFFFFFF	Cartridge Domain 2 Address 1	N64DD control registers - returns open bus (or all 0xFF) when not present
0x06000000 0x07FFFFFFF	Cartridge Domain 1 Address 1	N64DD IPL ROM - returns open bus (or all 0xFF) when not present
0x08000000 0x0FFFFFFF	Cartridge Domain 2 Address 2	SRAM is mapped here
0x10000000 0x1FBFFFFFFF	Cartridge Domain 1 Address 2	ROM is mapped here
0x1FC00000 0x1FC007BF	PIF Boot Rom	First code run on boot. Baked into hardware.
0x1FC007C0 0x1FC007FF	PIF RAM	Used to communicate with PIF chip (controllers, memory cards)
0x1FC00800 0x1FCFFFFFFF	Reserved	
0x1FD00000 0x7FFFFFFF	Cartridge Domain 1 Address 3	
0x80000000 0xFFFFFFFF	Unknown	Unknown

1.3 Boot Process

When the N64 boots, the program counter is initially set to 0xBFC00000. You'll notice this is a virtual address in the segment KSEG1 that translates to the physical address 0x1FC00000. While the virtual address 0x9FC00000 seemingly also could be used, this is in KSEG0 which is a cached segment. Because this is the very first code the n64 executes on boot, the caches will not be initialized yet, and this will not work on a real console!

The code at this address is what's called the PIF ROM. This is code baked into the console, and it is used to initialize the hardware and boot the program on the cartridge.

Emulators can execute this code, but the effects of it are so simple it can make sense to simply simulate its effects instead.

1.3.1 Simulating the PIF ROM

The PIF ROM can be fairly tricky for young emulators to run, and its effects are not complicated. If you wish to simulate it, know that it has the following side effects on the console:

Set four GPRs to initial values:

Register Number	Register Name	Set to value
11	t3	0xFFFFFFFFFA4000040
20	s4	0x0000000000000001
22	s6	0x000000000000003F
29	sp	0xFFFFFFFFFA4001FF0

All other registers are left at zero.

Set some COP0 registers to initial values:

Register Number	Register Name	Set to value
1	Random	0x0000001F
12	Status	0x34000000
15	PRId	0x00000B00
16	Config	0x0006E463

All other registers are left at zero.

The first 0x1000 bytes from the cartridge are then copied to SP DMEM. This is implemented as a copy of 0x1000 bytes from 0xB0000000 to 0xA4000000.

The program counter is then set to 0xA4000040. Note that this skips the first 0x40 bytes of the ROM, as this is where the header is stored. Also note that execution begins with the CPU executing out of SP DMEM.

The ROM now begins to execute! In practice, this is the Bootcode. A reverse-engineering and analysis of this bootcode can be found [Here](#).

1.4 TLB

TLB documentation coming soon

1.5 CPU (detailed)

1.5.1 Registers

As noted in the CPU overview, r0 is **hardwired to a value of zero**, and all registers are 64 bits.

Register Number	Register Name
0	zero
1	at
2	v0
3	v1
4	a0

Continued on next page

Table 1 – continued from previous page

Register Number	Register Name
5	a1
6	a2
7	a3
8	t0
9	t1
10	t2
11	t3
12	t4
13	t5
14	t6
15	t7
16	s0
17	s1
18	s2
19	s3
20	s4
21	s5
22	s6
23	s7
24	t8
25	t9
26	k0
27	k1
28	gp
29	sp
30	s8
31	ra

While these all have special names, for the purposes of emulation they all operate identically (except for r0 and r31)

As I’ve mentioned many times by now, r0 is hardwired to 0, and r31, or \$ra, is used by the “branch/jump and link” instructions to hold the return address.

The various other names are useful for assembly programmers who need to know what registers are conventionally used for what purposes.

1.5.2 COP0 Registers

Register Number	Register Name	Size
0	Index	32 bits
1	Random	32 bits
2	EntryLo0	32 or 64 bits, depending on mode
3	EntryLo1	32 or 64 bits, depending on mode
4	Context	32 or 64 bits, depending on mode
5	PageMask	32 bits
6	Wired	32 bits
7	7	??
8	BadVAddr	32 or 64 bits, depending on mode
9	Count	32 bits

Continued on next page

Table 2 – continued from previous page

Register Number	Register Name	Size
10	EntryHi	32 or 64 bits, depending on mode
11	Compare	32 bits
12	Status	32 bits
13	Cause	32 bits
14	EPC	32 or 64 bits, depending on mode
15	PRId	32 bits
16	Config	32 bits
17	LLAddr	32 bits
18	WatchLo	32 bits
19	WatchHi	32 bits
20	XContext	64 bits
21	21	??
22	22	??
23	23	??
24	24	??
25	25	??
26	Parity Error	32 bits
27	Cache Error	32 bits
28	TagLo	32 bits
29	TagHi	32 bits
30	ErrorEPC	32 or 64 bits, depending on mode
31	31	??

COP0 TLB Registers

These registers are used to query and control the TLB. Please see the TLB section for more information.

- Index
- EntryLo0
- EntryLo1
- EntryHi
- PageMask
- Context

COP0 Random Number Registers

These registers are used to generate random values.

The Random register is read-only. The high 26 bits are unused, leaving the low 6 bits to represent a random value. This value can be read and used by software, but is mainly meant to be used by the TLBWR (TLB Write Random) instruction.

On a real CPU, the value is decremented every instruction. When the value of Random is $\leq$ the value of Wired, it is reset to 0x1F (31)

It should be fine for emulation purposes to generate a random value in the range of $\text{Wired} \leq \text{Value} \leq 31$ every time Random is read, as checking and decrementing Random every single instruction will be expensive.

- Random

Holds a random value between the value of Wired and 0x1F (31)

- Wired

Provides the lower bound for the random value held in Random.

COP0 Timing Registers

Since the N64 has no timers, these registers are the only way the system can tell how much time has passed.

- Count

This value is incremented every other cycle, and compared to the value in Compare. As noted below, fire an interrupt when Count == Compare.

The easiest way to emulate this would be to store count as a 64 bit integer, increment it once per cycle, and shift it to the right by one when read or compared.

- Compare

Fire an interrupt when Count equals this value. This interrupt sets the ip7 bit in Cause to 1.

Writes to this register clear said interrupt, and sets the ip7 bit in Cause to 0.

COP0 Cache Registers

These registers are used for the cache, which is not documented here yet.

- TagLo
- TagHi

COP0 Exception/Interrupt Registers

These registers are used for exceptions and interrupts.

- BadVAddr When a TLB exception is thrown, this register is automatically loaded with the address of the failed translation.
- Cause Contains details on the exception or interrupt that occurred. Only the low two bits of the Interrupt Pending field can be written to using MTC0, the rest are read-only and set by hardware when an exception is thrown. More information can be found in the interrupts section.

Bit	Description
0-1	Unused (always zero)
2-6	Exception code (which exception/interrupt occurred?)
7	Unused (always zero)
8-15	Interrupt Pending (which interrupts are waiting to be serviced? Used with Interrupt Mask on \$Status)
16-27	Unused (always zero)
28-29	Coprocessor error (which coprocessor threw the exception, often not used)
30	Unused (always zero)
31	Branch delay (did the exception/interrupt occur in a branch delay slot?)

- EPC
- ErrorEPC

- WatchLo
- WatchHi
- XContext
- Parity Error

The N64 does not generate a parity error, so this register is never written to by hardware.

- Cache Error

The N64 does not generate a cache error, so this register is never written to by hardware.

COP0 Other Registers

These registers don't fit cleanly into any other category.

- PRIId
- Config
- LLAddr
- Status

Bit	Description
0	ie - global interrupt enable (should interrupts be handled?)
1	exl - exception level (are we currently handling an exception?)
2	erl - error level (are we currently handling an error?)
3-4	ksu - execution mode (00 = kernel, 01 = supervisor, 10 = user)
5	ux - 64 bit addressing enabled in user mode
6	sx - 64 bit addressing enabled in supervisor mode
7	kx - 64 bit addressing enabled in kernel mode
8-15	im - interrupt mask (&'d against interrupt pending in \$Cause)
16-24	ds - diagnostic status (described below)
25	re - reverse endianness (0 = big endian, 1 = little endian)
26	fr - enables additional floating point registers (0 = 16 regs, 1 = 32 regs)
27	rp - enable low power mode. Run the CPU at 1/4th clock speed
28	cu0 - Coprocessor 0 enabled (this bit is ignored by the N64, COP0 is always enabled!)
29	cu1 - Coprocessor 1 enabled - if this bit is 0, all COP1 instructions throw exceptions
30	cu2 - Coprocessor 2 enabled (this bit is ignored by the N64, there is no COP2!)
31	cu3 - Coprocessor 3 enabled (this bit is ignored by the N64, there is no COP3!)

1.5.3 COP1 (FPU) Registers

TODO

1.5.4 Instructions

See either the official manual, or [this fantastic wiki page](#)

1.6 Interrupts

Interrupts on the N64 are a multi-layered system. The N64 hardware raises and lowers interrupts through the MIPS interface (MI from now on.) These interrupts can be masked there as well. If an interrupt makes it through this mask, an interrupt will be sent to the CPU.

There are two important registers in the MI that are used for interrupt handling. MI_INTR_REG, and MI_INTR_MASK_REG.

MI_INTR_REG is not writable by the program. Each bit is instead controlled individually by a different component's interrupt. For example, the peripheral interface (PI) will set bit 4 in MI_INTR_REG when a PI DMA completes, and hold it high until it is lowered through a write to PI_STATUS_REG.

MI_INTR_MASK_REG is writable by the program, and is used to enable and disable certain interrupts. If a programmer wishes to ignore PI interrupts, they will set bit 4 of MI_INTR_MASK_REG to 0, and to 1 if they wish to enable PI interrupts.

In hardware, this is implemented as a circuit that outputs a 1 when any two corresponding bits are both 1. Most likely with a series of AND gates tied together with a single OR gate, if I had to guess. Because of this, the interrupts are *technically* checked every cycle, but in an emulator you only need to check when either register is written. The check can be implemented like this. Remember, none of this is happening inside the CPU.

```
// Should we send an interrupt to the CPU?
bool interrupt_fired = (MI_INTR_REG & MI_INTR_MASK_REG) != 0;
```

Now, we're ready to talk about what happens inside the CPU.

The MIPS CPU inside the N64 has eight possible interrupts that can be requested. These correspond to the eight bits in the "Interrupt Pending" field in the COP0 \$Cause register.

Bit	Description
0 (bit 8 of \$Cause)	ip0 - This is writable by MTC0, and is used as a "software interrupt"
1 (bit 9 of \$Cause)	ip1 - This is writable by MTC0, and is used as a "software interrupt"
2 (bit 10 of \$Cause)	ip2 - This is connected to the MI interrupt process described above. It is set to 1 when (MI_INTR_REG & MI_INTR_MASK_REG) != 0
3 (bit 11 of \$Cause)	ip3 - This is connected to the cartridge slot. Cartridges with special hardware can trigger this interrupt. Unsure how common this is in practice.
4 (bit 12 of \$Cause)	ip4 - This is connected to the Reset button on the top of the console. When pressed, this becomes 1.
5 (bit 13 of \$Cause)	ip5 - Connected to the Indy dev kit's RDB port. Set to 1 when a value is read.
6 (bit 14 of \$Cause)	ip6 - Connected to the Indy dev kit's RDB port. Set to 1 when a value is written.
7 (bit 15 of \$Cause)	ip7 - This is connected to the \$Count/\$Compare interrupt mechanism inside COP0, described in the COP0 Timing Registers section.

As with MI_INTR_REG and MI_INTR_MASK_REG, the interrupt pending field also has a corresponding mask field. It's located at bits 8-15 of \$Status. Again, as with the MI registers, these two fields are &'d together, and an interrupt is serviced if the two have any corresponding bits both set to 1.

To stop an interrupt being serviced over and over again in an endless loop, there are additional bits checked in addition to IP & IM. The bits checked are the IE, EXL, and ERL bits in \$Status.

We only want to handle interrupts if they are enabled, we're not currently handling an exception, and we're not currently handling an error. Thus, the full condition for an interrupt being serviced is:

```
bool interrupts_pending = (status.im & cause.ip) != 0;
bool interrupts_enabled = status.ie == 1;
bool currently_handling_exception = status.exl == 1;
bool currently_handling_error = status.erl == 1;

bool should_service_interrupt = interrupts_pending
                                && interrupts_enabled
                                && !currently_handling_exception
                                && !currently_handling_error;
```

This condition is checked every cycle, but can be optimized to be only re-checked when any of these bits change.

When an interrupt is determined to be serviced, an exception is thrown, with exception code 0, meaning interrupt, and no coprocessor error. Exceptions are described in the next section.

1.7 Exceptions

Exceptions are how the N64 handles both errors in instructions and interrupts.

1.7.1 Exception Codes

There are a lot of exceptions. It's worth noting that to get games booting, you pretty much only need the interrupt exception. To get games fully working, you need the interrupt exception and the coprocessor unusable exception for COP1. Games will boot without the COP1 unusable exception, but will have mild to serious glitches.

Name	Code	Cop. Err	Description
Interrupt	0	Undefined	Thrown when an interrupt occurs.
TLB Modification	1	Undefined	Thrown when a TLB page marked read-only is written to
TLB Miss - Load	2	Undefined	Thrown when no valid TLB entry is found when translating an address to be used for a load (instruction fetch or data access)
TLB Miss - Store	3	Undefined	Thrown when no valid TLB entry is found when translating an address to be used for a store (data access)
Address Error - Load	4	Undefined	Thrown when data or an instruction is loaded from an unaligned address.
Address Error - Store	5	Undefined	Thrown when data is stored to an unaligned address.
Bus Error - Instruction Fetch	6	Undefined	Hardware bus error (timeouts, data corruption, invalid physical memory addresses) when fetching an instruction.
Bus Error - Load/Store	7	Undefined	Hardware bus error (timeouts, data corruption, invalid physical memory addresses) when loading or storing data.
Syscall	8	Undefined	Thrown by the SYSCALL MIPS instruction.
Breakpoint	9	Undefined	Thrown by the BREAK MIPS instruction
Reserved Instruction	10	Undefined	Thrown when an invalid instruction is executed. Details below.
Coprocessor Unusable	11	Cop. used	Thrown when a coprocessor instruction is used when that coprocessor is disabled. Note that COP0 is never disabled.
Arithmetic Overflow	12	Undefined	Thrown by arithmetic instructions when their operations overflow.
Trap	13	Undefined	Thrown by the TRAP family of MIPS instructions.
Floating Point	15	Undefined	Thrown by the FPU when an error case is hit.
Watch	23	Undefined	Thrown when a load or store matches the address specified in the COP0 \$WatchLo and \$WatchHi registers.

Reserved Instruction Exception cases:

- Undefined opcode
- Undefined SPECIAL sub-opcode
- Undefined REGIMM sub-opcode
- 64 bit operation run in 32 bit mode. Note that in kernel mode, 64 bit operations are *always valid*, regardless if KX (enable 64 bit addressing in kernel mode) is set or not.

1.7.2 Exception Handling Process

When an exception is thrown, the CPU will update some state inside COP0, and set the program counter to the address of the appropriate exception handler. This address varies depending on the type of exception, and on some state within COP0.

Here is a description on what happens, step by step.

1. If the program counter is currently inside a branch delay slot, set the branch delay bit in \$Cause (bit 31) to 1. Otherwise, set this bit to 0.
2. If the EXL bit is currently 0, set the \$EPC register in COP0 to the current PC. Then, set the EXL bit to 1. A. If we are currently in a branch delay slot, instead set EPC to the address of the *branch that we are currently in the delay slot of*, i.e. *current_pc - 4*.
3. Set the exception code bit in the COP0 \$Cause register to the code of the exception that was thrown.
4. If the coprocessor error is a defined value, i.e. for the coprocessor unusable exception, set the coprocessor error field in \$Cause to the coprocessor that caused the error. Otherwise, the value of this field is undefined behavior in hardware, so it *shouldn't* matter what you emulate this as.
5. Jump to the exception vector. A detailed description on how to find the correct exception vector is found on pages 180 through 181 of the manual, and described in less detail below. A. Note that there is no “delay slot” executed when jumping to the exception vector, execution jumps there immediately.

1.7.3 Exception Vector Locations

Note that all of these addresses are sign extended to 64 bits.

The reset and NMI exceptions always jump to 0xBF00'0000. You'll note that this is the base address of the PIF ROM - jumping here will start execution over from scratch.

The locations of the rest of the vectors depend on the BEV bit. This bit is set by the boot process to let hardware know how much of the system's initialization has happened. If BEV=1, we are early in the boot process, and exception vectors should use different code than they will later on. I personally have never run into an exception early enough in the boot process for BEV to be 1, but it's good to check it anyway, just in case.

When BEV is 0, exceptions are handled in a cached region, since it's assumed the cache has already been initialized.

- 32 bit TLB exceptions jump to 0x8000'0000 when EXL = 0, and 0x8000'0180 when EXL = 1.
- 64 bit TLB exceptions jump to 0x8000'0080 when EXL = 1, and 0x8000'0180 when EXL = 1.
- All other exceptions jump to 0x8000'0180.

When BEV is 1, exceptions are handled in an *uncached* region, since it's assumed the cache has *not* been initialized yet.

- 32 bit TLB exceptions jump to 0xBF00'0200 when EXL = 0, and 0xBF00'0380 when EXL = 1.
- 64 bit TLB exceptions jump to 0xBF00'0280 when EXL = 1, and 0xBF00'0380 when EXL = 1.
- All other exceptions jump to 0xBF00'0380.

1.8 MIPS Interface

1.8.1 0x04300000 - MI_MODE_REG (Read / Write)

Sets and retrieves some values. I am uncertain of what they are used for.

More importantly, this register allows the game to lower the DP interrupt.

Writes

Bit	Explanation
0-6	Sets init length (??)
7	Clear init mode (??)
8	Set init mode (??)
9	Clear ebus test mode (??)
10	Set ebus test mode (??)
11	Lower DP Interrupt: Sets the bit in MI_INTR_REG to low
12	Clear RDRAM reg mode (??)
13	Set RDRAM reg mode (??)

Writes with “set” bits high will set the bits in the actual register to high. Writes with “set” bits low will have no effect.

Writes with “clear” bits high will clear the bits in the actual register. Writes with “clear” bits low will have no effect.

If both the “set” and “clear” bits are high, the value is set (?? I think ??)

Reads

Bit	Explanation
0-6	Gets init length - returns the value written above
7	Gets init mode - returns the value written above
8	Gets ebus test mode - returns the value written above
9	Gets RDRAM reg mode - returns the value written above

1.8.2 0x04300004 - MI_VERSION_REG (Read only)

Bit	Explanation
0-7	IO Version
8-15	RAC Version
16-23	RDP Version
24-31	RSP Version

This register should return 0x02020102 always.

1.8.3 0x04300008 - MI_INTR_REG (Read only)

Bits in this register are raised and lowered as interrupts are raised and lowered by other parts of the system.

Bit	Explanation
0	SP Interrupt - Set by the RSP when requested by a write to the SP status register, and optionally when the RSP halts.
1	SI Interrupt - Set by the serial interface, when SI DMAs to/from PIF RAM finish.
2	AI Interrupt - Set by the audio interface, when there are no more samples remaining in an audio stream
3	VI Interrupt - Set by the video interface, when V_CURRENT == V_INTR. Allows an interrupt to be raised on a given scanline.
4	PI Interrupt - Set by the peripheral interface, when a PI DMA between the cartridge and RDRAM finishes.
5	DP Interrupt - Set by the RDP, when a full sync completes.

1.8.4 0x0430000C - MI_INTR_MASK_REG (Read / Write)

This register sets up a mask. If $(\text{MI_INTR_REG} \& \text{MI_INTR_MASK_REG}) \neq 0$, then a MIPS interrupt is raised.

Writes

Bit	Explanation
0	Clear SP Mask
1	Set SP Mask
2	Clear SI Mask
3	Set SI Mask
4	Clear AI Mask
5	Set AI Mask
6	Clear VI Mask
7	Set VI Mask
8	Clear PI Mask
9	Set PI Mask
10	Clear DP Mask
11	Set DP Mask

See MI_MODE_REG for an explanation on set/clear bits.

Reads

Bit	Explanation
0	SP Mask
1	SI Mask
2	AI Mask
3	VI Mask
4	PI Mask
5	DP Mask

1.9 Video Interface

1.9.1 0x04400000 - VI_STATUS_REG/VI_CONTROL_REG

Can be called the VI_STATUS_REG, or the VI_CONTROL REG, whichever you prefer.

This register describes the format of the framebuffer in RDRAM, as well as enables and disables effects such as gamma, dithering, anti-aliasing, etc.

Bit	Explanation
0-1	Framebuffer bits-per-pixel (see below)
2	Gamma dither enable
3	Gamma enable
4	Divot enable
5	Reserved
6	Serrate
7	Reserved
8-9	Anti-alias mode (see below)
10	Unused
11	Reserved
12-15	Reserved
16-31	Unused

Enum Definitions

Framebuffer bits per pixel:

0. Blank
1. Reserved
2. RGBA 5553 “16” bits per pixel (should be able to ignore alpha channel and treat this as RGBA5551)
3. RGBA 8888 32 bits per pixel

Anti-alias mode:

0. Anti-alias and resample (always fetch extra lines)
1. Anti-alias and resample (fetch extra lines if needed)
2. Resample only (treat as all fully covered)
3. No anti-aliasing or resampling, no interpolation.

1.9.2 0x04400004 - VI_ORIGIN_REG

Describes where in RDRAM the VI should display the framebuffer from. Bits 0 through 23 are used, bits 24 through 31 are ignored by hardware.

Bit	Description
0-23	RDRAM address of framebuffer
24-31	Unused

1.9.3 0x04400008 - VI_WIDTH_REG

TODO

1.9.4 0x0440000C - VI_INTR_REG

TODO

1.9.5 0x04400010 - VI_V_CURRENT_REG

TODO

1.9.6 0x04400014 - VI_BURST_REG

TODO

1.9.7 0x04400018 - VI_V_SYNC_REG

TODO

1.9.8 0x0440001C - VI_H_SYNC_REG

TODO

1.9.9 0x04400020 - VI_LEAP_REG

TODO

1.9.10 0x04400024 - VI_H_START_REG

TODO

1.9.11 0x04400028 - VI_V_START_REG

TODO

1.9.12 0x0440002C - VI_V_BURST_REG

TODO

1.9.13 0x04400030 - VI_X_SCALE_REG

TODO

1.9.14 0x04400034 - VI_Y_SCALE_REG

TODO

1.10 Audio Interface

TODO

1.11 Peripheral Interface

TODO

1.12 RDRAM Interface

TODO

1.13 Serial Interface

TODO

1.14 Issues Log

I'm keeping a small log of various bugs and issues I fixed, categorized under the games they fixed.

1.14.1 Namco Museum & Super Mario 64

Distorted audio - the operating system uses COP1 Unusable Exceptions as a trigger to save the FPU registers on a context switch.

If these are not thrown, the FPU registers will not be properly saved/restored, and the audio will be extremely distorted due to incorrect values remaining in the registers when the OS switches back to the audio thread.

1.14.2 Ocarina of Time

I had major graphical issues in the game when I first got it booting. The game would only draw perfectly horizontal lines across the screen, but otherwise appeared to work fine. This was solved by ensuring that LWC1, LDC1, SWC1, and SDC1 threw COP1 unusable exceptions. I missed this because these instructions have first-level opcodes unlike the other COP1 instructions.

The game would hang upon opening the pause menu. This is because the operating system uses software timers while opening this screen. These were broken because my handling of Compare interrupts was incorrect.

1.14.3 Mario Kart 64, other games using EEPROM

Game would hang upon completing a Grand Prix. This ended up being because EEPROM wasn't identified correctly in the PIF, when channel 4 was requested in a controller ID command.

1.14.4 F-Zero X

Expects the N64DD's status register at 0x05000508 to return 0xFFFFFFFF if the DD is absent. Or, I assume, the N64DD to be correctly emulated. Otherwise, it will hang indefinitely on a black screen.

The read from the status register is performed at PC value 0x800C5A84. The hang happens at PC value 0x80414CF4.

1.14.5 Paper Mario

The game would hang when Mario falls off the veranda behind the goombas' house. The cause ended up being that my DIVU instruction was broken.

DIV is best implemented with 64 bit signed integers, to guard against an INT_MIN / -1 case. When implementing DIVU, I copy-pasted my DIV implementation and made the 64 bit integers unsigned.

Paper Mario's rand_int() function performs a DIVU with 0xFFFFFFFF / x. My DIVU implementation was calculating this as 0xFFFFFFFF'FFFFFFFF / x. With a signed divide, this is fine, since both of those numbers represent negative one. With an unsigned divide, however, they give different results.

This was causing random event probabilities to be very incorrect, which, long story short, ended up causing a hang.

1.14.6 Banjo-Tooie

Uses CIC-NUS-6105's "challenge/response" process through the PIF. The game will hang if this is not implemented.

1.14.7 Banjo-Tooie / Banjo-Kazooie

Sets the dpc_status.freeze bit to 1, which causes the game's graphics to hang indefinitely. A simple solution is simply never allowing this bit to be set to 1 through writes to the DPC status register, but I'm sure there's something more complicated going on.

1.14.8 Star Wars: Rogue Squadron

The game worked fine, except it would draw a black screen instead of the title screen. The intro and even in-game worked perfectly fine.

The solution ended up being that I needed to respect the serrate bit in VI Status.