Aphelion Instruction Set Architecture Reference Manual v0.4.0

Seth Poulsen
sp@sandwichman.dev

Kayla Silk-Corke
kaylasilkc@gmail.com

Krzysztof Wolicki
der.teufel.mail@gmail.com

Contents

1 Introduction

Aphelion is a 64-bit RISC-like instruction set architecture. Aphelion aims to be a clean and featureful architecture without succumbing to paralyzing minimalism or unwieldy complexity, walking a line between CISC and RISC conventions.

2 Registers

Aphelion defines sixteen 64-bit registers. These registers are readable/writable and can be used by any instruction that uses registers, with the exception of the status register **`st`** (more on this later).

Mnemonic	**Code**	**Description**
rz	0	always 0
ra-rk	1-11	general purpose
ip	12	instruction pointer
sp	13	stack pointer
fp	14	frame pointer
st	15	status register

2.1 General Purpose Registers

Registers **`ra`** through **`rk`** can be used to store data relevant to the program. They serve no special function and are not significant to the processor in any way.

2.2 RZ - Zero Register

The zero register **`rz`** always holds the value 0. All write operations are ignored.

2.3 IP - Instruction Pointer

The instruction pointer **`ip`** holds the address of the next instruction to be executed. It is incremented after an instruction is loaded into the processor, but before that instruction is executed (an instruction loaded from **`0x00`** will see that **`ip`** is **`0x04`**). The instruction pointer can be modified directly, or through the use of dedicated control flow instructions.

The instruction pointer **`ip`** can be set to a value that is not aligned to 4 bytes, but an **`Unaligned Access`** interrupt will trigger when the next instruction is loaded.

2.4 SP, FP - Stack & Frame Pointer

Registers **`sp`** and **`fp`** are the stack pointer and the frame pointer, respectively. The stack pointer contains the memory address of the top stack entry. The frame pointer contains the base address of the current stack frame. See *Interrupts* for error states.

Like all registers, **`fp`** and **`sp`** are initialized to **`0`** upon startup. Aphelion's built-in stack instructions grow the stack downwards, so these registers should be explicitly set before any operations that involve the stack happen.

2.5 ST - Status Register

The status register contains bit flags and information about the processor state. Most flags are set by the **`cmp`** comparison instructions, with the exception of **`CB`** and **`CBU`**, which are set by **`add`** and **`sub`**. Modifying the status register directly is illegal and will trigger an **`Invalid Operation`** interrupt.

`st` is laid out like so:

63..32	31	30..8	7	6	5	4	3	2	1	0
CI	EF	[unused]	M	LU	L	E	CBU	CB	Z	S

where:

Key	Name	Description
S	SIGN	(i64)a < 0
Z	ZERO	a == 0
CB	CARRY_BORROW	a + b + (i64)C > I64_MAX || a - b - (i64)B < I64_MIN
CBU	CARRY_BORROW_UNSIGNED	a + b + (u64)C > U64_MAX || a - b - (u64)B < U64_MIN
E	EQUAL	a == b
L	LESS	a < b
LU	LESS_UNSIGNED	(u64)a < (u64)b
M	MODE	processor mode
EF	EXT_F	"Extension F - Floating Point Operations" is enabled
CI	CURRENT_INST	copy of the current instruction's machine code

3 Instruction Set

3.1 System Control

Mnemonic	Encoding	Format	Description
nop			no operation, expands to 'add rz, rz, rz'
inv			invalid opcode, expands to 'int 2'
int imm8	-- 0 imm8 0x01	F	trigger interrupt imm8 (see Interrupts)
iret	-- 1 ---- 0x01	F	return from interrupt
ires	-- 2 ---- 0x01	F	resolve interrupt
usr rd	rd 3 ---- 0x01	F	enter user mode and jump to address in rd. rd should hold a virtual address.

3.2 Input & Output

Mnemonic	Encoding	Format	Description
out rd/imm16, rs			Assembler alias for outr, outi
in rd, rs/imm16			Assembler alias for inr, ini
outr rd, rs	rd rs ----- 0x02	M	output data in rs to port rd
outi imm16, rs	-- rs imm16 0x03	M	output data in rs to port imm16
inr rd, rs	rd rs ----- 0x04	M	read data from port rs to rd
ini rd, imm16	rd -- imm16 0x05	M	read data from port imm16 to rd

3.3 Control Flow

Mnemonic	Encoding	Format	Description
call rs, label			call function, expands to 'li rs, label; jal rs, 0'
callr rs, label, rd			call function, expands to 'li rs, label; jalr rs, 0, rd'
jal rs, imm16	-- rs imm16 0x06	M	push ip, ip ← rs + 4 * (i64)imm16
jalr rs, imm16, rd	rd rs imm16 0x07	M	rd ← ip, ip ← rs + 4 * (i64)imm16
ret	-- -- ----- 0x08	M	pop ip

retr rs	-- rs ——— 0x09	M	ip ← rs
b(cc) imm20	cc imm20 0x0a	B	ip ← pc + 4*(i64)imm20, branch on condition (see Branch Conditions below)

3.3.1 Branch Conditions

Mnemonic	**Code**	**With** cmpr A, B	**Condition**
bra	0x0	always	(true)
beq	0x1	A == B	EQUAL
bez	0x2	A == 0	ZERO
blt	0x3	A < B	LESS
ble	0x4	A ⩽ B	LESS || EQUAL
bltu	0x5	(u64)A < (u64)B	LESS_UNSIGNED
bleu	0x6	(u64)A ⩽ (u64)B	LESS_UNSIGNED || EQUAL
bne	0x9	!(A == B)	!EQUAL
bnz	0xA	!(A == 0)	!ZERO
bge	0xB	A ⩾ B	!LESS
bgt	0xC	A > B	!(LESS || EQUAL)
bgeu	0xD	(u64)A ⩾ (u64)B	!(LESS_UNSIGNED)
bgtu	0xE	(u64)A > (u64)B	!(LESS_UNSIGNED || EQUAL)

3.4 Stack Operations

Mnemonic	**Encoding**	**Format**	**Description**
push rs	-- rs ——— 0x0b	M	sp ← sp - 8, mem[sp] ← rs
pop rd	rd -- ——— 0x0c	M	rd ← mem[sp], sp ← sp + 8
enter	-- ————— 0x0d	B	push fp, fp = sp; enter stack frame
leave	-- ————— 0x0e	B	sp = fp, pop fp; leave stack frame

3.5 Data Flow

Note: Parameters denoted with parenthesis are optional in the assembly syntax.

Mnemonic	**Encoding**	**For-mat**	**Description**
mov rd, rs			rd ← rs, expands to 'or rd, rs, rz'
li rd, imm			rd ← imm64, expands to li-family as needed
lli rd, imm	rd 0 imm 0x10	F	rd[15..0] ← imm
llis rd, imm	rd 1 imm 0x10	F	rd ← (i64)imm
lui rd, imm	rd 2 imm 0x10	F	rd[31..16] ← imm
luis rd, imm	rd 3 imm 0x10	F	rd ← (i64)imm << 16
lti rd, imm	rd 4 imm 0x10	F	rd[47..32] ← imm
ltis rd, imm	rd 5 imm 0x10	F	rd ← (i64)imm << 32
ltui rd, imm	rd 6 imm 0x10	F	rd[63..48] ← imm
ltuis rd, imm	rd 7 imm 0x10	F	rd ← (i64)imm << 48
lw rd, rs, off, (rn, sh)	rd rs rn sh off 0x11	E	rd ← mem[rs + (i64)off + rn << sh]
lh rd, rs, off, (rn, sh)	rd rs rn sh off 0x12	E	rd[31..0] ← mem[rs + (i64)off + rn << sh]

lhs rd, rs, off, (rn, sh)	rd rs rn sh off 0x13	E	rd ← mem[rs + (i64)off + rn << sh]
lq rd, rs, off, (rn, sh)	rd rs rn sh off 0x14	E	rd[15..0] ← mem[rs + (i64)off + rn << sh]
lqs rd, rs, off, (rn, sh)	rd rs rn sh off 0x15	E	rd ← mem[rs + (i64)off + rn << sh]
lb rd, rs, off, (rn, sh)	rd rs rn sh off 0x16	E	rd[7..0] ← mem[rs + (i64)off + rn << sh]
lbs rd, rs, off, (rn, sh)	rd rs rn sh off 0x17	E	rd ← mem[rs + (i64)off + rn << sh]
sw rs, off, (rn, sh), rd	rd rs rn sh off 0x18	E	mem[rs + off + rn << sh] ← (i64)rd
sh rs, off, (rn, sh), rd	rd rs rn sh off 0x19	E	mem[rs + off + rn << sh] ← (i32)rd
sq rs, off, (rn, sh), rd	rd rs rn sh off 0x1a	E	mem[rs + off + rn << sh] ← (i16)rd
sb rs, off, (rn, sh), rd	rd rs rn sh off 0x1b	E	mem[rs + off + rn << sh] ← (i8)rd

3.6 Comparisons

Mnemonic	Encoding	Format	Description
cmp r1/imm, r2/imm			Alias for cmpr, cmpi
cmpr r1, r2	r1 r2 ——— 1e	M	compare and set flags (see status register)
cmpi r1/imm, r1/imm	r1 [s] imm 1f	F	compare and set flags (see status register). imm is sign-extended. if the immediate value is first, [s] is set to 1, else 0.

3.7 Arithmetic Operations

Mnemonic	Encoding	Format	Description
add rd, r1, r2/imm16			Integer addition; alias for addr, addi
sub rd, r1, r2/imm16			Integer subtraction; alias for subr, subi
imul rd, r1, r2/imm16			Signed integer multiplication; alias for imulr, imuli
umul rd, r1, r2/imm16			Unsigned integer multiplication; alias for umulr, umuli
idiv rd, r1, r2/imm16			Signed integer division; alias for idivr, idivi
udiv rd, r1, r2/imm16			Unsigned integer division; alias for udivr, udivi
rem rd, r1, r2/imm16			Integer remainder (truncated); alias for remr, remi
mod rd, r1, r2/imm16			Integer modulus (floored); alias for modr, modi
addr rd, r1, r2	rd r1 r2 -- 0x20	R	rd ← r1 + r2
addi rd, r1, imm16	rd r1 imm16 0x21	M	rd ← r1 + (i64)imm16
subr rd, r1, r2	rd r1 r2 -- 0x22	R	rd ← r1 - r2
subi rd, r1, imm16	rd r1 imm16 0x23	M	rd ← r1 - (i64)imm16
imulr rd, r1, r2	rd r1 r2 -- 0x24	R	rd ← r1 * r2 (signed)
imuli rd, r1, imm16	rd r1 imm16 0x25	M	rd ← r1 * (i64)imm16 (signed)
idivr rd, r1, r2	rd r1 r2 -- 0x26	R	rd ← r1 / r2 (signed)
idivi rd, r1, imm16	rd r1 imm16 0x27	M	rd ← r1 / (i64)imm16 (signed)
umulr rd, r1, r2	rd r1 r2 -- 0x28	R	rd ← r1 * r2 (unsigned)

umuli rd, r1, imm16	rd r1 imm16 0x29	M	rd ← r1 * (u64)imm16 (unsigned)
udivr rd, r1, r2	rd r1 r2 -- 0x2a	R	rd ← r1 / r2 (unsigned)
udivi rd, r1, imm16	rd r1 imm16 0x2b	M	rd ← r1 / (u64)imm16 (unsigned)
remr rd, r1, r2	rd r1 r2 -- 0x2c	R	rd ← r1 % r2
remi rd, r1, imm16	rd r1 imm16 0x2d	M	rd ← r1 % (i64)imm16
modr rd, r1, r2	rd r1 r2 -- 0x2e	R	rd ← r1 % r2
modi rd, r1, imm16	rd r1 imm16 0x2f	M	rd ← r1 % (i64)imm16

3.8 Bitwise Operations

For bitwise operations, assume all immediates zero-extended unless otherwise specified.

Mnemonic	**Encoding**	**Format**	**Description**
and rd, r1, r2/imm16			Bitwise AND, alias for andr, andi
or rd, r1, r2/imm16			Bitwise OR, alias for orr, ori
nor rd, r1, r2/imm16			Bitwise NOR, alias for norr, nori
not rd, rs			Bitwise NOT, expand to 'nor rd, rs, rz'
xor rd, r1, r2/imm16			Bitwise XOR, alias for xorr, xori
shl rd, r1, r2/imm16			Shift left, alias for shlr, shli
asr rd, r1, r2/imm16			Arithmetic shift right, alias for asrr, asri
lsr rd, r1, r2/imm16			Logical shift right, alias for asrr, asri
bit rd, r1, r2/imm16			Extract single bit, alias for bitr, biti.
andr rd, r1, r2	rd r1 r2 -- 0x30	R	rd ← r1 & r2
andi rd, r1, imm16	rd r1 imm16 0x31	M	rd ← r1 & (u64)imm16
orr rd, r1, r2	rd r1 r2 -- 0x32	R	rd ← r1 | r2
ori rd, r1, imm16	rd r1 imm16 0x33	M	rd ← r1 | (u64)imm16
norr rd, r1, r2	rd r1 r2 -- 0x34	R	rd ← !(r1 | r2)
nori rd, r1, imm16	rd r1 imm16 0x35	M	rd ← !(r1 | (u64)imm16
xorr rd, r1, r2	rd r1 r2 -- 0x36	R	rd ← r1 ^ r2
xori rd, r1, imm16	rd r1 imm16 0x37	M	rd ← r1 ^ (u64)imm16
shlr rd, r1, r2	rd r1 r2 -- 0x38	R	rd ← r1 << r2
shli rd, r1, imm16	rd r1 imm16 0x39	M	rd ← r1 << (u64)imm16
asrr rd, r1, r2	rd r1 r2 -- 0x3a	R	rd ← (i64)r1 >> r2
asri rd, r1, imm16	rd r1 imm16 0x3b	M	rd ← (i64)r1 >> (u64)imm16
lsrr rd, r1, r2	rd r1 r2 -- 0x3c	R	rd ← (u64)r1 >> r2
lsri rd, r1, imm16	rd r1 imm16 0x3d	M	rd ← (u64)r1 >> (u64)imm16
bitr rd, r1, r2	rd r1 r2 -- 0x3e	R	rd ← (r2 in 0..63) ? r1[r2] : 0
biti rd, r1, imm16	rd r1 imm16 0x3f	M	rd ← (imm16 in 0..63) ? r1[imm16] : 0
setfs rd			rd ← 'SIGN' flag, expands to 'biti rd, st, 0'
setfz rd			rd ← 'ZERO' flag, expands to 'biti rd, st, 1'
setfcb rd			rd ← 'CARRY_BORROW' flag, expands to 'biti rd, st, 2'
setfcbu rd			rd ← 'CARRY_BORROW_UNSIGNED' flag, expands to 'biti rd, st, 3'
setfe rd			rd ← 'EQUAL' flag, expands to 'biti rd, st, 4'

`setfl   rd`			`rd ← 'LESS' flag, expands to 'biti rd, st, 5'`
`setflu  rd`			`rd ← 'LESS_UNSIGNED' flag, expands to 'biti rd, st, 6'`

3.9 Extension F - Floating-Point Operations

Aphelion Extension F - Floating-Point Operations implements hardware support for floating-point formats as specified in the IEEE 754-2008 standard. The extension implements every operation for half-precision (16-bit), single-precision (32-bit), and double-precision (64-bit) formats. To specify, the instruction name must be appended with **`.16`**, **`.32`**, or **`.64`** for half, single, or double-precision, with the instruction's **`func`** field set to **`0`**, **`1`**, and **`2`** respectively.

Mnemonic	**Encoding**	**Format**	**Description**
`fcmp  r1, r2`	`-- r1 r2 [p] -- 0x40`	E	`rd ← compare r1 and r2`
`fto   rd, rs`	`rd rs -- [p] -- 0x41`	E	`rd ← (f[]) rs`
`ffrom rd, rs`	`rd rs -- [p] -- 0x42`	E	`rd ← (i64) rs`
`fneg  rd, rs`	`rd rs -- [p] -- 0x43`	E	`rd ← -rs`
`fabs  rd, rs`	`rd rs -- [p] -- 0x44`	E	`rd ← |rs|`
`fadd  rd, r1, r2`	`rd r1 r2 [p] -- 0x45`	E	`rd ← r1 + r2`
`fsub  rd, r1, r2`	`rd r1 r2 [p] -- 0x46`	E	`rd ← r1 - r2`
`fmul  rd, r1, r2`	`rd r1 r2 [p] -- 0x47`	E	`rd ← r1 * r2`
`fdiv  rd, r1, r2`	`rd r1 r2 [p] -- 0x48`	E	`rd ← r1 / r2`
`fma   rd, r1, r2`	`rd r1 r2 [p] -- 0x49`	E	`rd ← rd + (r1 * r2)`
`fsqrt rd, r1`	`rd r1 -- [p] -- 0x4a`	E	`rd ← squareroot(r1)`
`fmin  rd, r1, r2`	`rd r1 r2 [p] -- 0x4b`	E	`rd ← min(r1, r2)`
`fmax  rd, r1, r2`	`rd r1 r2 [p] -- 0x4c`	E	`rd ← max(r1, r2)`
`fsat  rd, r1`	`rd r1 -- [p] -- 0x4d`	E	`rd ← smallest integer greater than or equal to r1 (basically ceil)`
`fcnv  rd, r1`	`rd r1 -- [p] -- 0x4e`	E	`rd ← cast(r1); convert between precisions`
`fnan  rd, r1`	`rd r1 -- [p] -- 0x4f`	E	`rd ← isnan(r1);`

The instruction **`fcnv`** is a special case. **`fcnv`** takes two precision tags, the first tag occupying the lower two bits of **`func`** and the second occupying the higher two bits of **`func`**. The first tag specifies the format being converted to, and the second tag specifies the format being converted from. For example, the instruction **`fcnv.64.32 rb, ra`** would convert a single-precision value in **`ra`** to the nearest double-precision value and store it in **`rb`**. Conversions where the source precision and destination precision are equal are invalid instructions.

Floating point conversions using **`ffrom`** will output **`0x7FFFFFFFFFFFFFFF`** if given a value that is NaN, greater than 9,223,372,036,854,775,807 or less than −9,223,372,036,854,775,808.

3.10 Instruction Encoding

Each instruction follows an encoding format, which separates the instruction's 32 bits into disctinct fields. The way these fields are filled out are specified in the **Encoding** column of the previous tables.

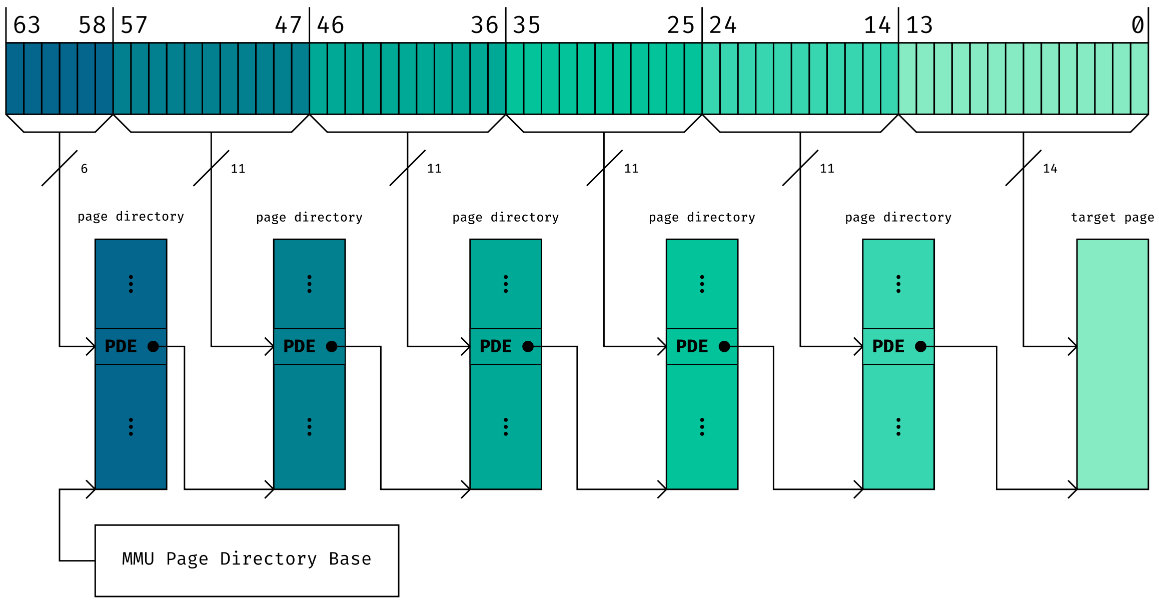
	31 .. 28	27 .. 24	23 .. 20	19 .. 16	15 .. 8	7 .. 0
E	rde	rs1	rs2	func	imm	opcode
R	rde	rs1	rs2		imm	opcode
M	rde	rs1			imm	opcode
F	rde	func			imm	opcode
B	func				imm	opcode

4 Memory

Aphelion uses a full 64-bit address space internally. Maximum addressable space is not defined by the ISA, but by the memory limitations of the system hardware. Memory is separated into 16 KiB (2^{14} B) pages. Like all other registers, the instruction pointer **`ip`** is initialized to **`0`**. Consequently, address **`0`** is where code execution begins on system startup.

4.1 Address Translation

Aphelion provides user-mode virtual memory in the form of a five-level page table:



Page directories (PDs) are a single page long (exluding the top-level directory) and must be aligned with page boundaries. Each PD is an array of 2048 page directory entries (PDEs). PDEs contain information about the next level directory/page as well as permissions. PDEs are laid out like so:

63 .. 14	13 .. 5	4	3	2	1	0
NX	[reserved]	X	W	R	O	V

where:

Key	Name	Description
NX	Next	These are the upper 50 bits of the next sub-directory/page.
V	Valid	If set (1), this entry is valid and can be used for further translation. If not set (0), all sub-pages are invalid/unmapped and should trigger an access violation if used.
O	Override	If set, permissions in this entry are authoritative for all valid sub-pages. If not set, permissions are left to sub-directories. This bit is ignored for the lowest order page directories, which have no sub-directories. Authoritative permissions can be reset by another sub-entry with Override set.
R	Read	If set && Override is set, all valid sub-pages are readable. If unset && Override is set, all valid sub-pages are not readable.
W	Write	If set && Override is set, all valid sub-pages are writable. If unset && Override is set, all valid sub-pages are not readable.
X	Execute	If set && Override is set, all valid sub-pages are executable. If unset && Override is set, all valid sub-pages are not executable.

5 Interrupts

The Interrupt Vector Table (IVT) has 256 entries. Each entry is a function address, making the full table 2048 bytes wide. The location of an interrupt's IVT entry is defined as **`IVT_BASE_ADDRESS+(int*8)`**, where **`int`** is the interrupt number. When an interrupt is triggered, it will exit to kernel mode (if not in kernel mode already) and run code at the address defined at the relevant IVT entry. Interrupt handlers should be returned from with the **`iret`** instruction or marked as resolved with the **`ires`** instruction.

The IVT's location is initialized to **`0`**, so should be initialized somewhere else as soon as possible after startup. For information about setting the location of the IVT, see *reserved ports*.

Aphelion's reserved interrupts are as follows:

Code	Name	Description
0×00	Divide By Zero	Triggers when the second argument of a div, mod, or rem instruction is zero.
0×01	Breakpoint	Reserved for debugger breakpoints.
0×02	Invalid Operation	Triggers when some kind of restricted or invalid operation occurs. This includes unrecognized opcode, unrecognized secondaryfunction values, or when a restricted instruction is encountered / modification of a restricted register is attempted in user mode.
0×03	Stack Underflow	Triggers when sp > fp, which means a stack underflow has occurred.
0×04	Unaligned Access	Memory has been accessed across type width boundaries.
0×05	Access Violation	Memory has been accessed in an invalid way: In kernel mode, this triggers due to accesses outside physical memory bounds. In user mode, this triggers when unmapped/invalid memory is accessed or when virtual memory permissions do not allow the access.
0×06	Interrupt Overflow	Interrupt controller has experienced an interrupt queue overflow, meaning too many interrupts have triggered in a certain time.

The Interrupt Controller has an internal FIFO 32-item queue for pending interrupt signals. If an interrupt triggers when a handler has not yet returned or resolved, it is pushed to the queue and will trigger immediately after the current interrupt handler returns or resolves. If this queue overflows, the queue will reset and an **Interrupt Overflow** interrupt will be pushed onto it, so that it will trigger immediately after the current interrupt handler is complete.

If there are interrupts waiting to be handled in the interrupt queue and a handler returns using **`iret`**, the next handler will *immediately* be executed instead of immediately returning to the code that triggered it. The instruction pointer **`ip`** and the status register **`st`** will be stored. When the interrupt queue is clear, **`ip`** and **`st`** are restored and execution resumes smoothly from the original trigger point.

If an **`ires`** instruction is used instead of an **`iret`** instruction, execution resumes after the **`ires`** instruction itself. Stored values of **`ip`** and **`st`** are discarded and the current **`ip`** and **`st`** are saved in their place. Queued handlers will use these new values when restoring the state of the machine after the queue is clear. This is useful for interrupts that must be considered "resolved" at some point but that may not return to where they were triggered (such as an exit syscall).

`iret` and **`ires`** do absolutely nothing when the interrupt queue is empty.

6 Input/Output

Aphelion uses a port-based I/O system. The ISA reserves some ports for internal system configuration, while the rest are general-purpose. Ports are 64-bit, and ports that do not use the entire 64-bits should have unused bits held low (at 0). The value of a port is the most recent value written to it by the corresponding external device.

In theory, there are a maximum of 65,536 ports. Obviously, the use and availability of ports depends on the capabilities of the implementation system's hardware.

There are a small number of internal system devices hardwired to reserved ports. These are used for system configuration, serving the same purpose that control registers and special configuration instructions would on other architectures.

Port	**Name**	**Description**
0	Interrupt Controller	Manages interrupts and the Interrupt Vector Table.
1	Input/Output Controller	Manages I/O operation and provides I/O information.
2	Memory Management Unit	Oversees memory access and address translation.
3	System Timer	Provides time information and can be used as a PIT/HPET.

6.1 Interrupt Controller

The Interrupt Controller is the internal device that manages the interrupt system and handles interrupt sources. The commands it accepts are as follows:

Code	**Name**	**Description**
0×00	Set IVT Base Address	Primes the controller for setting the IVT base address. The next data it expects to receive is the new address of the IVT, which must be word-aligned.

Invalid commands are discarded.

6.2 Input/Output Controller

The Input/Output Controller manages I/O events and routing. The commands it accepts are as follows:

Code	**Name**	**Description**
0×00	Bind Port to Interrupt	Primes the controller to bind input activity on a port to the triggering of an interrupt. After this, the controller expects to receive the port number, and then the interrupt number.

Invalid commands are discarded.

6.3 Memory Management Unit

The Memory Management Unit oversees memory access and virtual address translation. The commands it accepts are as follows:

Code	Name	Description
0×00	Flush Translation Lookaside Buffer	Reset the MMU's virtual address translation cache.
0×01	Set Page Table Base	Primes the MMU to set the base address of the page table, after which it expects an address. This will be force-aligned downward to the nearest page boundary less than or equal to the given address. This will also flush the translation lookaside buffer automatically.
0×02	Translate Address	Primes the MMU to translate an address given the current page table base, after which it expects the address. The MMU will walk the page table and reply with the translated address. If the address is unmapped, it will reply with the untranslated address given to it.

Invalid commands are discarded.

6.4 System Timer

The System Timer provides various types of information about time and also functions as a PIT/HPET. This PIT/HPET has a maximum of 8 invidual timers/alarms that can be set.

Code	Name	Description
0×00	Query Uptime Microseconds	Replies with the microseconds elapsed since execution began, as a 64-bit signed integer.
0×01	Query Global Microseconds	Replies with the current Unix Timestamp in microseconds, as a 64-bit signed integer.
0×02	Set Global Microseconds	Expects to recieve a Unix Timestamp in microseconds as a 64-bit signed integer.
0×03	Set Timer	Sets an unset timer to trigger in [x] microseconds. Expects to recieve an interval in microseconds as a 64-bit unsigned integer. Replies with 0×000000000001F44D if the operation is successful. If all timers are already set/active, the operation fails and the ST replies with 0×000000000001F44E.

Invalid commands are discarded.

7 User Mode

When user mode is active. There are a few limitations placed on the processor:

- **I/O Disabled** - I/O instructions are disabled and are treated like invalid instructions.
- **Memory Translation** - The MMU uses the aforementioned page table to translate memory accesses from a virtual address space.
- **System Control Disabled** - `iret`, `ires`, and `usr` are invalid in user mode, since all interrupt handlers execute in kernel mode.