Aphelion Instruction Set Architecture Reference Manual v0.3.0

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-inspired instruction set architecture. It operates on 64-bit data and uses 32-bit wide instructions. Aphelion sits in a middle ground between CISC and RISC instruction sets, balancing performance with complexity.

2 Registers

Aphelion defines sixteen 64-bit registers.

Name	Code	Description
rz	0	always 0, writes silently fail
r[a-k]	1-B	general purpose
pc	C	program counter
sp	D	stack pointer
fp	E	frame pointer
st	F	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.

2.2 RZ - Zero Register

The zero register `rz` always holds the value `0`. `rz` ignores all write operations.

2.3 PC - Program Counter

The program counter `pc` points to the current instruction being executed. It increments to the next instruction every load cycle, unless it is modified through direct means or the use of control flow instructions.

2.4 ST - Status Register

The status register contains bit flags and information about the processor state. Most flags are set by the `cmp a, b` comparison instruction, with the exception of `C, B, CU, BU`, which are ALSO set by `add` and `sub`. Modifying the status register in user mode is illegal and will trigger an `Invalid Instruction` interrupt.

`st` is laid out like so:

63 .. 31	30 .. 8	7	6	5	4	3	2	1	0
CI	[unused]	M	LU	L	E	B	C	Z	S

where:

Key	Name	Branch Relevant?	Description
S	SIGN	NO	i64(a) < 0
Z	ZERO	YES	a == 0
C	CARRY	NO	a + b + (i64)C > I64_MAX
B	BORROW	NO	a - b - (i64)B < I64_MIN
E	EQUAL	YES	a == b
L	LESS	YES	a < b
LU	LESS_UNSIGNED	YES	(u64)a < (u64)b

M	MODE	NO	reserved (processor mode)
CI	CURRENT_INST	NO	copy of the current instruction's machine code

2.5 SP, FP - Stack & Frame Pointer

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

Like all registers, `fp` and `sp` are initialized to `0`. Aphelion grows the stack downwards, so the registers should be explicitly set before any stack operations happen (`jal` and other instructions also use the stack).

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	00 imm8 0x01	B	trigger interrupt imm8 (see `Interrupts`)
usr	01 —— 0x01	B	enter user mode
fbf	02 —— 0x01	B	invert/flip branch-relevant flags (see `st`)

3.2 Input & Output

Mnemonic	**Encoding**	**Format**	**Description**
out rd/imm8, rs			Assembler alias for outr, outi
in rd/imm8, rs			Assembler alias for inr, ini
outr rd, rs	rd rs ——— 0x02	M	output data in rs to port rd
outi imm16, rs	rd -- imm16 0x03	F	output data in rs to port imm8
inr rs, rd	rs rd ——— 0x04	M	read data from port rs to rd
ini imm16, rd	rs -- imm16 0x05	F	read data from port imm8 to rd

3.3 Control Flow

Mnemonic	**Encoding**	**Format**	**Description**
jal rs, imm16	-- rs imm16 0x06	M	push (pc + 4) , pc ← rs + 4 * i64(imm16)
jalr rs, imm16, rd	rd rs imm16 0x07	M	rd ← (pc + 4) , pc ← rs + 4 * i64(imm16)
ret	-- -- ——— 0x08	M	pc ← mem[sp], sp ← sp - 4
retr rs	-- rs ——— 0x09	M	pc ← rs
b(cc) imm20	cc imm20 0x0a	B	pc ← 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`

Branch conditions like “greater than” are not available at the instruction level, but can be implemented with the **`fbf`** instruction or swapping **`cmp`** arguments, if possible and applicable.

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

Mnemonic	**Encoding**	**Format**	**Description**
`mov rd ← rs`			`rd ← rs, expands to 'add rd, rs, rz'`
`li rd, imm64`			`rd ← imm64, expands to li-family as needed`
`lli rd, imm16`	`rd 0x0 imm16 0x10`	`F`	`rd[15..0] ← imm16`
`llis rd, imm16`	`rd 0x1 imm16 0x10`	`F`	`rd ← i64(imm16)`
`lui rd, imm16`	`rd 0x2 imm16 0x10`	`F`	`rd[31..16] ← imm16`
`luis rd, imm16`	`rd 0x3 imm16 0x10`	`F`	`rd ← i64(imm16) << 16`
`lti rd, imm16`	`rd 0x4 imm16 0x10`	`F`	`rd[47..32] ← imm16`
`ltis rd, imm16`	`rd 0x5 imm16 0x10`	`F`	`rd ← i64(imm16) << 32`
`ltui rd, imm16`	`rd 0x6 imm16 0x10`	`F`	`rd[63..48] ← imm16`
`ltuis rd, imm16`	`rd 0x7 imm16 0x10`	`F`	`rd ← i64(imm16) << 48`
`lw rd, [rs + imm16]`	`rd rs imm16 0x11`	`M`	`rd ← (u64) mem[rs + imm16]`
`lh rd, [rs + imm16]`	`rd rs imm16 0x12`	`M`	`rd[31..0] ← (u32) mem[rs + imm16]`
`lhs rd, [rs + imm16]`	`rd rs imm16 0x13`	`M`	`rd ← (i32) mem[rs + imm16]`
`lq rd, [rs + imm16]`	`rd rs imm16 0x14`	`M`	`rd[15..0] ← (u16) mem[rs + imm16]`
`lqs rd, [rs + imm16]`	`rd rs imm16 0x15`	`M`	`rd ← (i16) mem[rs + imm16]`
`lb rd, [rs + imm16]`	`rd rs imm16 0x16`	`M`	`rd[7..0] ← (u8) mem[rs + imm16]`
`lbs rd, [rs + imm16]`	`rd rs imm16 0x17`	`M`	`rd ← (i8) mem[rs + imm16]`
`sw rd, [rs + imm16]`	`rd rs imm16 0x18`	`M`	`mem[rs + imm16] ← u64(rd)`
`sh rd, [rs + imm16]`	`rd rs imm16 0x19`	`M`	`mem[rs + imm16] ← u32(rd)`
`sq rd, [rs + imm16]`	`rd rs imm16 0x1a`	`M`	`mem[rs + imm16] ← u16(rd)`
`sb rd, [rs + imm16]`	`rd rs imm16 0x1b`	`M`	`mem[rs + imm16] ← u8(rd)`
`bswp rd, rs`	`rd rs ——— 0x1c`	`M`	`rd ← byteswap(rs), swap endianness`
`xch r1, r2`	`r1 r2 ——— 0x1d`	`M`	`(r1, r2) ← (r2, r1)`

3.6 Comparisons

Mnemonic	**Encoding**	**Format**	**Description**
`cmp r1, r2/imm16`			`Alias for cmpr, cmpi`
`cmpr r1, r2`	`-- r1 r2 -- 1e`	`R`	`compare and set flags (see st)`

cmpi r1, imm16	-- r1 imm16 1f	M	compare and set flags (see `st`). imm16 is sign-extended.

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, alias for remr, remi
mod rd, r1, r2/imm16			Integer modulus, alias for modr, modi
addr rd, r1, r2	rd r1 r2 -- 0x20	R	rd ← r1 + r2 + carry
addi rd, r1, imm16	rd r1 imm16 0x21	M	rd ← r1 + (i64)imm16 + carry
subr rd, r1, r2	rd r1 r2 -- 0x22	R	rd ← r1 - r2 - borrow
subi rd, r1, imm16	rd r1 imm16 0x23	M	rd ← r1 - (i64)imm16 - borrow
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 (floored)
remi rd, r1, imm16	rd r1 imm16 0x2d	M	rd ← r1 %% i64(imm16) (floored)
modr rd, r1, r2	rd r1 r2 -- 0x2e	R	rd ← r1 % r2 (truncated)
modi rd, r1, imm16	rd r1 imm16 0x2f	M	rd ← r1 % i64(imm16) (truncated)

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
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 << u16(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 >> u16(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 >> u16(imm16)`

3.9 Posit Operations

Aphelion uses posits instead of traditional floating-point as its rational number format. For simplicity, we use 64-bit posits with a maximum of 8 exponent bits (subject to change in future versions). For simplicity it is not completely posit-compliant, but does implement common/necessary operations.

Mnemonic	**Encoding**	**Format**	**Description**
`pto   rd, rs`	`rd rs ——— 0x40`	M	`rd ← cast(posit) rs`
`pfrom rd, rs`	`rd rs ——— 0x41`	M	`rd ← cast(i64) rs`
`pcmp  rd, rs`	`rd rs ——— 0x42`	M	`Compare and set flags`
`pneg  rd, rs`	`rd rs ——— 0x43`	M	`rd ← -rs`
`pabs  rd, r1`	`rd rs ——— 0x44`	M	`rd ← |rs|`
`padd  rd, r1, r2`	`rd r1 r2 -- 0x45`	R	`rd ← r1 + r2`
`psub  rd, r1, r2`	`rd r1 r2 -- 0x46`	R	`rd ← r1 - r2`
`pmul  rd, r1, r2`	`rd r1 r2 -- 0x47`	R	`rd ← r1 * r2`
`pdiv  rd, r1, r2`	`rd r1 r2 -- 0x48`	R	`rd ← r1 / r2`

3.10 Instruction Formats

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 .. 8	7 .. 0
R	rde	rs1	rs2	func	opcode
M	rde	rs1		imm	opcode
F	rde	func		imm	opcode
J	rde			imm	opcode
B	func			imm	opcode

4 Memory

Pointers and addresses are 64-bit. Maximum addressable space is not defined by the ISA, but by the memory limitations of the system hardware. Like all other registers, the program counter **pc** is initialized to zero. This is where execution starts.

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. Interrupts push the program counter to the stack before jumping to the interrupt handler.

The IVT's location is initialized to **`0x0`**, 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**
0x00	Divide By Zero	Triggers when the second argument of a div, mod, or rem instruction is zero.
0x01	Breakpoint	Reserved for debugger breakpoints.
0x02	Invalid Instruction	Triggers when execution of an invalid instruction/operation is attempted. This includes unrecognized opcodes, unrecognized func values (instruction variations), or when a restricted instruction is encountered / modification of a restricted register is attempted in user mode.
0x03	Stack Underflow	Triggers when sp > fp, i. e. a stack underflow has occurred.
0x04	Unaligned Access	Memory is accessed across type width boundaries e.g. a word (8 bytes) is read at an address that is not a multiple of 8.
0x05	Access Violation	Reserved for MMU use.

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. The value of a port is the most recent value written to it (by the device, not the CPU).

Aphelion, in theory, handles a maximum of 65,536 ports. Obviously, the use and availability of ports depends on individual system hardware.

The ports reserved for internal use are as follows:

Port	**Name**	**Description**
0	Interrupt Controller	Manages interrupts and the IVT.
1	Input/Output Controller	Manages I/O operation and provides I/O information.
2	Memory Management Unit	Currently reserved.
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**
0x00	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. Replies with 0x000000000001F44D if the operation is successful.

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. Replies with 0×000000000001F44D if the operation is successful.

Invalid commands are discarded.

6.3 Memory Management Unit

The Memory Management Unit (MMU) is under development and not currently available. This port should be treated as unused/disconnected.

6.4 System Timer

The System Timer provides various types of information about time and also functions as a PIT. This PIT 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. Replies with 0×000000000001F44D if the operation is successful.
0×03	Set Alarm	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.