# Aphelion Instruction Set Architecture Specification Version 6

**Seth Poulsen**
sp@sandwichman.dev

**Kayla Silk-Corke**
kaylasilkc@gmail.com

**Krzysztof Wolicki**
der.teufel.mail@gmail.com

**Nick Borş**
nick@nickbors.cc

Typeface **Io Serif** created by Echo Heo for Orbit Systems documentation.

Typeface **IBM Plex Mono** used in monospace/code-snippet contexts.

──────

*Revision 1: September 17, 2025*

*Revision 2: September 25, 2025*

*Revision 3: December 23, 2025*

*Revision 4: January 22, 2026*

# Contents

# 1. Introduction

Aphelion is a little-endian, 64-bit RISC-like architecture focused on efficient, dense, data-driven execution.

# 2. General Purpose Registers

Aphelion has 32 general purpose registers (GPRs), each 64 bits wide.

| #     | Name    | Preserved across calls? | Purpose                    |
|-------|---------|-------------------------|----------------------------|
| 0     | `zr`    |                         | Hardwired to zero          |
| 1..6  | `a0..a5`| No                      | Function arguments/returns |
| 7..20 | `l0..l13`| Yes                    | Local variables            |
| 21..26| `t0..t5`| No                      | Temporary variables        |
| 27    | `tp`    |                         | Thread pointer             |
| 28    | `fp`    | Yes                     | Frame pointer              |
| 29    | `sp`    | Yes                     | Stack pointer              |
| 30    | `lp`    | No                      | Link/return pointer        |
| 31    | `ip`    |                         | Instruction pointer        |

Zero register `zr` is always zero and is unaffected by writes. Instruction pointer `ip` is unaffected by direct writes, but can be modified through control flow instructions.

Link pointer `lp`, stack pointer `sp`, frame pointer `fp`, and thread pointer `tp` are only considered special on the ABI level (see Aphelion ABI Specification).

Instruction pointer `ip` can only be modified through explicit control-flow instructions. Any writes to `ip` as if it were a standard GPR are ignored. `ip`'s main purpose as a GPR, and not a control register, is to facilitate simple position-independent code.

# 3. Control Registers

| # | Name | Description |
|---|------|-------------|
| 0..15 | **int0..int15** | Interrupt handler pointers. Bits 0..1 of addresses stored here are hardwired to 0. |
| 16 | **intip** | Interrupt instruction pointer. When an interrupt is triggered, this is set to the value of **ip** at the time of the trigger. Used by the **iret** instruction to return from an interrupt handler or to jump into user mode. Bits 0..1 of addresses stored here are hardwired to 0. |
| 17 | **intval** | Any value or address relevant to an interrupt. When an **BUS\***, **ACCESS\***, **UALIGN\***, or **VATFAIL** interrupt trigger, the target address of the memory access that caused it will be stored here. |
| 18 | **intpte** | On an **ACCESS\*** interrupt when using virtual addressing, this contains the most recent page table entry read by the processor. |
| 19 | **intcause** | The cause of the most recent interrupt. |
| 20 | **kptp** | Page table pointer for virtual addressing in kernel mode. Bits 0..11 of addresses stored here are hardwired to 0. |
| 21 | **uptp** | Page table pointer for virtual addressing in user mode. Bits 0..11 of addresses stored here are hardwired to 0. |
| 22 | **stat** | Status register. Described in full below. |
| 23 | **intstat** | Interrupt status register. When an interrupt is triggered, **stat** is copied into **intstat**. The **iret** instruction copies **intstat** back into **stat**. |

### 3.0.1. STAT - Status Register

The control register **stat** is a set of bit fields that indicate and control the processor's state:

| Bits | Name | Description |
|------|------|-------------|
| 0 | E | External Interrupts are enabled. |
| 1 | U | User mode is enabled. |
| 2 | V | Virtual address translation is enabled. |

Undefined bits are currently reserved for future use.

# 4. Interrupts

Interrupts are signals that can disrupt the normal flow of computation. Interrupts can be internal (caused by the processor) or external (caused by the system/environment).

Internal interrupts act only on the LP that triggered them. They may be triggered through the execution of special instructions, such as **syscall** and **breakpt**, but are most often generated through invalid execution of some kind, such as an unaligned memory access, virtual address translation failure, access violation, or execution of an invalid operation.

Internal interrupts generated during the execution of an instruction will cause the instruction to stop executing and prevent further side effects of the instruction, such as register write-backs or memory accesses.

External interrupts are generated by the surrounding system, usually by IO devices. They can be deferred and may only trigger during the execution of an instruction.

When an interrupt triggers, all previously executed instructions must complete in order, including incomplete or re-ordered memory operations. If re-ordered memory operations cause an interrupt of their own, the interrupt that triggers earliest in program order takes priority and replaces the current interrupt trigger.

## 4.1. Handling Interrupts

To handle an interrupt:
1. Register **ip** is saved to control register **intip** and control register **stat** is saved to **intstat**.
2. Register **ip** is set to the value of a handler control register (**int0..int15**) based on the interrupt cause.
3. The processor is put into kernel mode with external interrupts disabled;
4. The processor's lock state is unlocked (see Section 5.2).

## 4.2. Interrupt Causes

| # | Name | Description |
|---|------|-------------|
| 0 | **EXTERNL** | External (IO) interrupt. |
| 1 | **BREAKPT** | Debugger breakpoint. |
| 2 | **SYSCALL** | System call. |
| 3 | **INVALID** | Invalid operation has been loaded and executed. This is either an operation that does not exist, exists but with invalid arguments, or exists but is not permitted in the current processor mode. |
| 4 | **BUSR** | Bus fault while reading from memory. |
| 5 | **BUSW** | Bus fault while writing to memory. |
| 6 | **BUSX** | Bus fault while fetching code from memory. |
| 7 | **ACCESSR** | Access violation while reading from memory. |
| 8 | **ACCESSW** | Access violation while writing to memory. |
| 9 | **ACCESSX** | Access violation while fetching code from memory. |
| 10 | **UALIGNR** | Unaligned access while reading from memory. |
| 11 | **UALIGNW** | Unaligned access while writing to memory. |
| 12 | **UALIGNX** | Unaligned access while fetching code from memory. |
| 13 | **VATFAIL** | Virtual address translation failed due to a misconfigured/invalid page table. Access violations during address translation are converted into this interrupt code. |
| 14..15 | | Reserved. |

# 5. Memory Model

## 5.1. Weak Consistency

Aphelion follows a **weak consistency** memory model. For any two memory accesses X and Y, where X is before Y in program order, X may be reordered after Y in the global memory order if:
• There is no memory location overlap between X and Y;
• There is no corresponding fence instruction between X and Y;

In the weak model, fence instructions provide synchronization. Memory operations cannot be reordered before or after a corresponding `fence` instruction.

## 5.2. LL/SC and Atomics

Aphelion provides special load-lock (LL) and store-conditional (SC) instructions that can be used to make arbitrary computation atomic.

Each LP has a "lock state," which may be either locked or unlocked. The lock state additionally comprises a memory location and a width. In the syntax of the Rust programming language, it may be represented like so:

```rust
struct Lock {
    locked: bool,
    width: u8,
    address: u64,
}
```

Loading with an LL instruction updates the LP's lock state with the location and width of that load and "locks" it. If any LP stores to the range of memory covered by another LP's lock state (including its own), that lock state becomes unlocked. Additionally, this LP's lock state will become unlocked if any of these events happen on this LP:
• An interrupt occurs;
• An `iret` instruction executes (returning from an interrupt or entering user mode);
• A cache management instruction executes;

Storing with an SC instruction succeeds if and only if the LP's lock state is locked and the location and width of the store correspond to the location and width of the lock state.

Implementations are permitted use cache state as a simple and performant heuristic for modification, e.g. an external store to a memory location may cause locked locations in the same cache block to unlock.

Note that LL and SC instructions are *not* fences and adhere to the same weak consistency rules as traditional loads and stores.

## 5.3. Caches and Coherency

Aphelion uses a split cache architecture, using separated **data cache** (d-cache) for data loads/stores and **instruction cache** (i-cache) for instruction fetches. The caches function according to these rules:

- Each LP's d-cache is required to be fully coherent with other d-caches in the system. When a store from any LP executes in the global memory order, its effects must be visible to all LPs and external memory. External stores (e.g. from memory mapped devices) are not required to be immediately visible in d-cache. D-cache may need to be manually invalidated for external modifications to be visible.
- I-cache is not required to be automatically coherent with d-cache or main memory. When a block is not present in i-cache, it may be loaded either from main memory or from a present d-cache block. This means that external stores may not be seen by i-cache if the corresponding d-cache is not also invalidated.
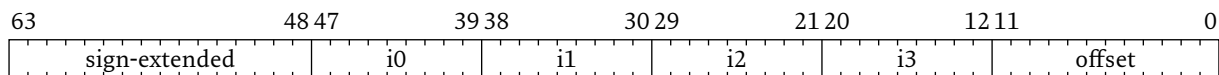
Cache blocks are 64 bytes in width.

## 5.4. Virtual Address Translation

Aphelion structures memory into 4 KiB ($2^{12}$B) chunks called **pages**. Using virtual address translation (VAT), the arbitrary platform-specific structure of physical memory can be reorganized into a consistent layout, and user mode programs can be isolated from kernel mode data.

When VAT is enabled, instructions that access memory will always attempt to translate addresses, even if the operation is not successful (e.g. store-conditional) or is treated as a no-op (e.g. cache management on cache-less systems).

Virtual addresses are broken into these individual fields:

| 63           | 48 47 | 39 38 | 30 29 | 21 20 | 12 11 | 0 |
|--------------|-------|-------|-------|-------|-------|---|
| sign-extended | i0    | i1    | i2    | i3    | offset |   |

A four-level page table is then used, where **i0** is the index into the first-level page table (located in physical memory at either **uptp** or **kptp**), **i1** is the index into the second-level page table, etc., and **offset** is the offset into the final page where the access occurs.

Aphelion processors may choose to implement translation cache mechanisms to prevent page table walks on every virtual access. If implemented, storing to **kptp** and **uptp** has special properties:

- Storing to **kptp** will invalidate translation cache entries associated with the previous kernel mode page table.
- Storing to **uptp** will invalidate translation cache entries associated with the previous user mode page table.

Note that kernel mode and user mode translation cache entries are invalidated separately.

Each table is a full page and contains 512 page table entries (PTEs), each 8 bytes long, broken into these bit fields:

| 63 | 12 11 | 3 2 1 0 |
|---|---|---|

| next | unused | x w v |
|---|---|---|
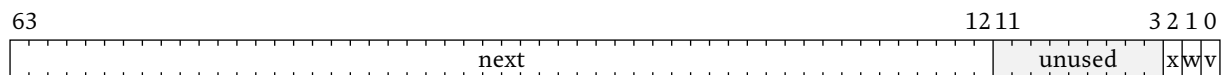
Where each bit field is defined as follows:

| Name | Description |
|---|---|
| **v** | When set, this entry is valid. |
| **w** | When set, this page is writeable. Ignored until the final table. |
| **x** | When set, this page is executable. Ignored until the final table. |
| **next** | These are the upper bits of the physical address of the next page table level or target page. |

Any unused bits are ignored by the processor and may be used for software-specific information.

Virtual address translation will trigger an **ACCESS\*** interrupt when:
- Bits 63..48 of the virtual address do not match bit 47;
- The **v** bit is not set in a page table entry used in translation;
- The **w** or **x** bits are not set in the final page table entry when writing data or fetching instructions respectively;

Virtual address translation will trigger a **VATFAIL** interrupt when the processor fails to load a page table entry.

# 6. Instructions

Instructions are always 32 bits in length. Instructions must always be loaded from addresses that are 4-byte aligned.

Bits 0..1 specify the format the instruction follows. Bits 2..8 provide a format-specific opcode. Together, the lowest byte of an instruction uniquely identify it.

| Fmt | Bits | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 31..28 | 27..23 | 22..18 | 17..13 | 12..8 | 7..2 | 1 | 0 |
| A | imm19 | | | | r1 | opcode | 0 | 0 |
| B | imm14 | | r2 | | r1 | opcode | 0 | 1 |
| C | imm9 | | r3 | r2 | r1 | opcode | 1 | 0 |

## 6.1. Opcode Mapping

Each instruction is assigned a format-specific 6-bit opcode. This 6-bit opcode is comprised of a major opcode in bits 0..2 and a minor opcode in bits 3..5. The major opcode roughly defines a "family" of behavior.

Instructions with similar behavior are designed to be similar in encoding. For example, arithmetic operations with both a register form and immediate form have identical 6-bit opcodes.

Another useful example of this is with memory loads and stores in Format C, where lower two bits of the minor opcode specify the size of the access, the high bit of the

minor opcode specifies LL/SC behavior, and the major opcode distinguishes between load and store operations.

### 6.1.1. Format A

| Minor | Major | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 000 | | | SSI | | FENCE | | | SYSCALL |
| 001 | | | | | CINVAL | | | BREAKPT |
| 010 | | | | | CFETCH | | | SPIN |
| 011 | | | | | | | | |
| 100 | | | | | | | | IRET |
| 101 | | | | | | | | LCTRL |
| 110 | | | | | | | | SCTRL |
| 111 | | | | | | | | WAIT |

### 6.1.2. Format B

| Minor | Major | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 000 | ADDI | ANDI | SI | SULTI | | | | |
| 001 | SUBI | ORI | CB | SILTI | | | | |
| 010 | MULI | NORI | REV | SULEI | | | | |
| 011 | | XORI | | SILEI | | | | |
| 100 | UDIVI | CLZ | | SEQI | JLR | | | |
| 101 | IDIVI | CTZ | | | JL | | | |
| 110 | UREMI | CSB | | | BZ | | | |
| 111 | IREMI | | | | BN | | | |

### 6.1.3. Format C

| Minor | Major | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 000 | ADD | AND | USR | SULT | LW | SW | | |
| 001 | SUB | OR | ISR | SILT | LH | SH | | |
| 010 | MUL | NOR | ROR | SULE | LQ | SQ | | |
| 011 | | XOR | ROL | SILE | LB | SB | | |
| 100 | UDIV | EXT | SL | SEQ | LLW | SCW | | |
| 101 | IDIV | DEP | | | LLH | SCH | | |
| 110 | UREM | UMULH | | | LLQ | SCQ | | |
| 111 | IREM | IMULH | | | LLB | SCB | | |

The following instruction implementations are written in pseudocode resembling the syntax of the Rust programming language.

## 6.2. Memory Loads

### 6.2.1. LW - Load Word
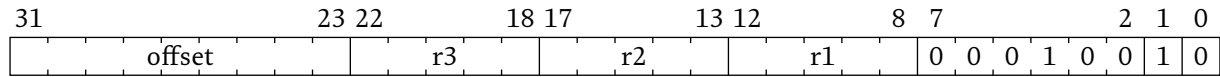
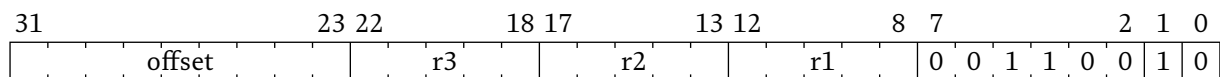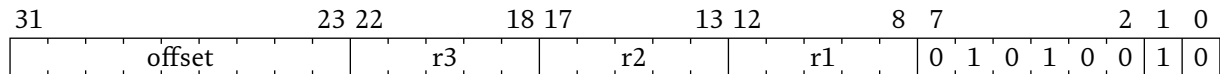| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| offset | r3 | r2 | r1 | 0 0 0 1 0 0 | 1 0 |

Load a 64-bit value into **r1** from the address given by the sum of **r2, r3**, and zero-extended **offset** shifted left by 3.

```
lw r1, [r2 + r3 + 4088] ; offset argument is divided by 8
lw r1, [r2 + 4088]      ; r3 = zr
lw r1, [r2 + r3]        ; offset = 0
lw r1, [r2]             ; r3 = zr; offset = 0
```

```
let addr = gpr[r2] + gpr[r3] + zero_extend(offset) << 3;
gpr[r1] = mem_load64(addr);
```

### 6.2.2. LH - Load Half-word

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| offset | r3 | r2 | r1 | 0 0 1 1 0 0 | 1 0 |

Load a zero-extended 32-bit value into **r1** from the address given by the sum of **r2, r3**, and zero-extended **offset** shifted left by 2.

```
lh r1, [r2 + r3 + 2044] ; offset argument is divided by 4
lh r1, [r2 + 2044]      ; r3 = zr
lh r1, [r2 + r3]        ; offset = 0
lh r1, [r2]             ; r3 = zr; offset = 0
```

```
let addr = gpr[r2] + gpr[r3] + zero_extend(offset) << 2;
gpr[r1] = zero_extend(mem_load_32(addr));
```

### 6.2.3. LQ - Load Quarter-word

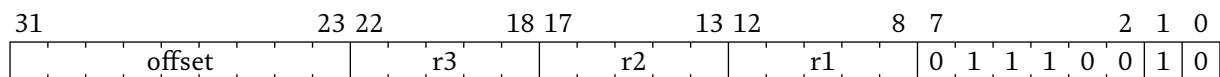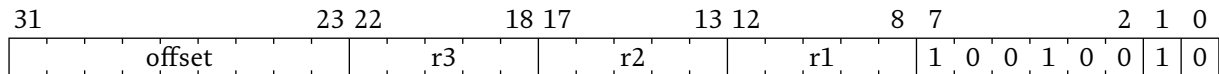| 31 | | 23 | 22 | | 18 | 17 | | 13 | 12 | | 8 | 7 | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | offset | | | r3 | | | r2 | | | r1 | | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |

Load a zero-extended 16-bit value into **r1** from the address given by the sum of **r2**, **r3**, and zero-extended **offset** shifted left by 1.

```
lq r1, [r2 + r3 + 1022] ; offset argument is divided by 2
lq r1, [r2 + 2044]      ; r3 = zr
lq r1, [r2 + r3]        ; offset = 0
lq r1, [r2]             ; r3 = zr; offset = 0
```

```
let addr = gpr[r2] + gpr[r3] + zero_extend(offset) << 1;
gpr[r1] = zero_extend(mem_load_16(addr));
```

### 6.2.4. LB - Load Byte

| 31 | | 23 | 22 | | 18 | 17 | | 13 | 12 | | 8 | 7 | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | offset | | | r3 | | | r2 | | | r1 | | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

Load a zero-extended 8-bit value into **r1** from the address given by the sum of **r2**, **r3**, and zero-extended **offset**.

```
lq r1, [r2 + r3 + 511]
lq r1, [r2 + 511]       ; r3 = zr
lq r1, [r2 + r3]        ; offset = 0
lq r1, [r2]             ; r3 = zr; offset = 0
```

```
let addr = gpr[r2] + gpr[r3] + zero_extend(offset);
gpr[r1] = zero_extend(mem_load_8(addr));
```

### 6.2.5. LLW - Load-Lock Word

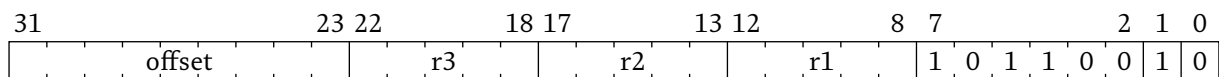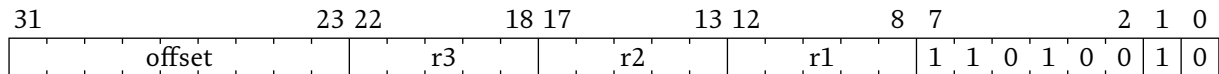| 31 | | 23 | 22 | | 18 | 17 | | 13 | 12 | | 8 | 7 | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| offset | | | r3 | | | r2 | | | r1 | | | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

Load a 64-bit value into **r1** from the address given by the sum of **r2, r3**, and zero-extended **offset** shifted left by 3, and lock that memory location.

```
llw r1, [r2 + r3 + 4088] ; offset argument is divided by 8
llw r1, [r2 + 4088]      ; r3 = zr
llw r1, [r2 + r3]        ; offset = 0
llw r1, [r2]             ; r3 = zr; offset = 0
```

```
let addr = gpr[r2] + gpr[r3] + zero_extend(offset);
lock.locked = true;
lock.address = addr;
lock.width = 8;
gpr[r1] = mem_load64(addr);
```

### 6.2.6. LLH - Load-Lock Half-word

| 31 | | 23 | 22 | | 18 | 17 | | 13 | 12 | | 8 | 7 | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| offset | | | r3 | | | r2 | | | r1 | | | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |

Load a zero-extended 32-bit value into **r1** from the address given by the sum of **r2, r3**, and zero-extended **offset** shifted left by 2, and lock that memory location.

```
llh r1, [r2 + r3 + 2044] ; offset argument is divided by 4
llh r1, [r2 + 2044]      ; r3 = zr
llh r1, [r2 + r3]        ; offset = 0
llh r1, [r2]             ; r3 = zr; offset = 0
```

```
let addr = gpr[r2] + gpr[r3] + zero_extend(offset);
lock.locked = true;
lock.address = addr;
lock.width = 4;
gpr[r1] = mem_load32(addr);
```

### 6.2.7. LLQ - Load-Lock Quarter-word

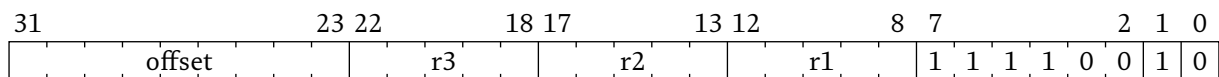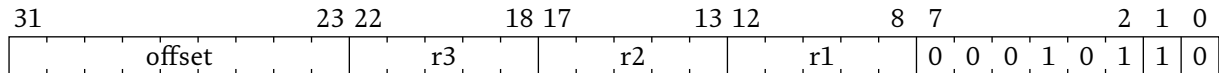| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| offset | | r3 | | r2 | | r1 | | 1 1 0 1 0 0 | | 1 | 0 |

Load a zero-extended 16-bit value into **r1** from the address given by the sum of **r2**, **r3**, and zero-extended **offset** shifted left by 1, and lock that memory location.

```
llq r1, [r2 + r3 + 1022] ; offset argument is divided by 2
llq r1, [r2 + 1022]      ; r3 = zr
llq r1, [r2 + r3]        ; offset = 0
llq r1, [r2]             ; r3 = zr; offset = 0
```

```
let addr = gpr[r2] + gpr[r3] + zero_extend(offset);
lock.locked = true;
lock.address = addr;
lock.width = 2;
gpr[r1] = mem_load16(addr);
```

### 6.2.8. LLB - Load-Lock Byte

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| offset | | r3 | | r2 | | r1 | | 1 1 1 1 0 0 | | 1 | 0 |

Load a zero-extended 8-bit value into **r1** from the address given by the sum of **r2**, **r3**, and zero-extended **offset**, and lock that memory location.

```
llq r1, [r2 + r3 + 511]
llq r1, [r2 + 511]       ; r3 = zr
llq r1, [r2 + r3]        ; offset = 0
llq r1, [r2]             ; r3 = zr; offset = 0
```

```
let addr = gpr[r2] + gpr[r3] + zero_extend(offset);
lock.locked = true;
lock.address = addr;
lock.width = 1;
gpr[r1] = mem_load8(addr);
```

## 6.3. Memory Stores

### 6.3.1. SW - Store Word

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| offset | r3 | r2 | r1 | 0 0 0 1 0 1 | 1 0 |

Store a 64-bit value from **r1** to the address given by the sum of **r2**, **r3**, and zero-extended **offset** shifted left by 3.

```
sw [r2 + r2 + 4088], r1 ; offset argument is divided by 8
sw [r2 + 4088], r1      ; r3 = zr
sw [r2 + r3], r1        ; offset = 0
sw [r2], r1             ; r3 = zr; offset = 0
```
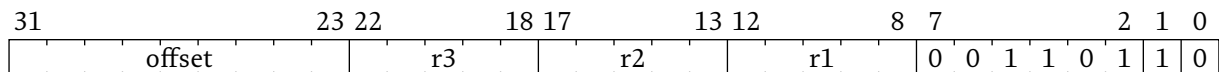
```
let addr = gpr[r2] + gpr[r3] + zero_extend(offset) << 3;
mem_store64(gpr[r1], addr);
```

### 6.3.2. SH - Store Half-word

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| offset | r3 | r2 | r1 | 0 0 1 1 0 1 | 1 0 |

Store the lower 32 bits of **r1** to the address given by the sum of **r2**, **r3**, and zero-extended **offset** shifted left by 2.

```
sh [r2 + r2 + 4088], r1 ; offset argument is divided by 4
sh [r2 + 4088], r1      ; r3 = zr
sh [r2 + r3], r1        ; offset = 0
sh [r2], r1             ; r3 = zr; offset = 0
```

```
let addr = gpr[r2] + gpr[r3] + zero_extend(offset) << 2;
mem_store32(gpr[r1] as u32, addr);
```

### 6.3.3. SQ - Store Quarter-word

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|---|---|---|---|---|
| offset | | r3 | | r2 | | r1 | | 0 1 0 1 0 1 | | 1 | 0 |

Store the lower 16 bits of **r1** to the address given by the sum of **r2, r3,** and zero-extended **offset** shifted left by 1.

```
sq [r2 + r2 + 1022], r1 ; offset argument is divided by 2
sq [r2 + 1022], r1       ; r3 = zr
sq [r2 + r3], r1         ; offset = 0
sq [r2], r1              ; r3 = zr; offset = 0
```

```
let addr = gpr[r2] + gpr[r3] + zero_extend(offset) << 1;
mem_store16(gpr[r1] as u16, addr);
```

### 6.3.4. SB - Store Byte

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|---|---|---|---|---|
| offset | | r3 | | r2 | | r1 | | 0 1 1 1 0 1 | | 1 | 0 |

Store the lower 8 bits of **r1** to the address given by the sum of **r2, r3,** and zero-extended **offset**.

```
sb [r2 + r2 + 511], r1
sb [r2 + 511], r1       ; r3 = zr
sb [r2 + r3], r1        ; offset = 0
sb [r2], r1             ; r3 = zr; offset = 0
```

```
let addr = gpr[r2] + gpr[r3] + zero_extend(offset);
mem_store8(gpr[r1] as u8, addr);
```

### 6.3.5. SCW - Store-Conditional Word

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| offset | r3 | r2 | r1 | 1 0 0 1 0 1 | 1 0 |
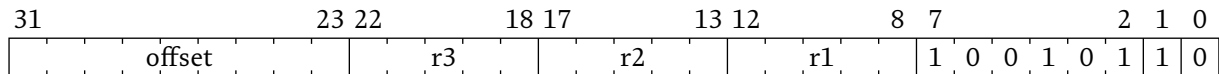
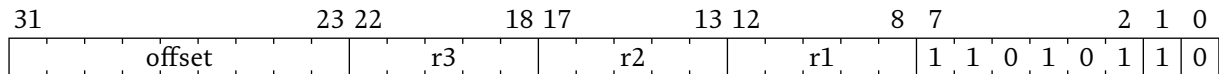Store **r1** to the address given by the sum of **r3** and zero-extended **offset** shifted left by 3, if the memory location is locked. Set **r2** to 1 if the store was successful, or 0 otherwise.

```
scw r2, [r3 + 4088], r1 ; offset argument is divided by 8
scw r2, [r3], r1        ; offset = 0
```

```
let addr = gpr[r3] + zero_extend(offset) << 3;
if lock.locked && lock.addr == addr && lock.width == 8 {
  mem_store64(gpr[r1], addr);
  gpr[r2] = 1;
} else {
  gpr[r2] = 0;
}
```

### 6.3.6. SCH - Store-Conditional Half-word
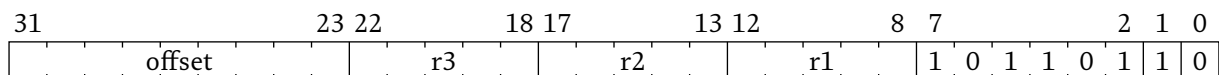
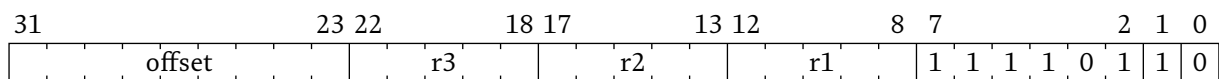| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| offset | r3 | r2 | r1 | 1 0 1 1 0 1 | 1 0 |

Store the lower 32 bits of **r1** to the address given by the sum of **r3** and zero-extended **offset** shifted left by 2, if the memory location is locked. Set **r2** to 1 if the store was successful, or 0 otherwise.

```
sch r2, [r3 + 2044], r1 ; offset argument is divided by 4
sch r2, [r3], r1        ; offset = 0
```

```
let addr = gpr[r3] + zero_extend(offset) << 2;
if lock.locked && lock.addr == addr && lock.width == 4 {
  mem_store32(gpr[r1] as u32, addr);
  gpr[r2] = 1;
  lock.locked = false;
} else {
  gpr[r2] = 0;
}
```
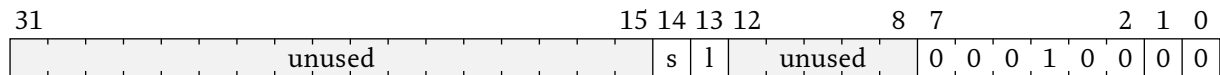
### 6.3.7. SCQ - Store-Conditional Quarter-word

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|---|---|---|---|---|
| offset | | r3 | | r2 | | r1 | | 1 1 0 1 0 1 | | 1 | 0 |

Store the lower 16 bits of **r1** to the address given by the sum of **r3** and zero-extended **offset** shifted left by 1, if the memory location is locked. Set **r2** to 1 if the store was successful, or 0 otherwise.

```
scq r2, [r3 + 1022], r1 ; offset argument is divided by 2
scq r2, [r3], r1        ; offset = 0
```

```
let mut addr = gpr[r3] + zero_extend(offset);
if lock.locked && lock.addr == addr && lock.width == 2 {
  mem_store16(gpr[r1] as u16, addr);
  gpr[r2] = 1;
  lock.locked = false;
} else {
  gpr[r2] = 0;
}
```

### 6.3.8. SCB - Store-Conditional Byte

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|---|---|---|---|---|
| offset | | r3 | | r2 | | r1 | | 1 1 1 1 0 1 | | 1 | 0 |

Store the lower 8 bits of **r1** to the address given by the sum of **r3** and zero-extended **offset**, if the memory location is locked. Set **r2** to 1 if the store was successful, or 0 otherwise.

```
scb r2, [r3 + 1022], r1 ; offset argument is divided by 2
scb r2, [r3], r1        ; offset = 0
```

```
let mut addr = gpr[r3] + zero_extend(offset);
if lock.locked && lock.addr == addr && lock.width == 1 {
  mem_store8(gpr[r1] as u8, addr);
  gpr[r2] = 1;
  lock.locked = false;
} else {
  gpr[r2] = 0;
}
```

## 6.4. Memory Effects

### 6.4.1. FENCE - Memory Fence

| 31 | 15 | 14 | 13 | 12 | 8 | 7 | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| unused | | s | l | unused | | 0 | 0 | 0 | 1 | 0 | 0 | | | 0 | 0 |

Before the next instruction executes, ensure that previous memory operations on this LP have completed in the global memory order and prevent memory operations further in program order from completing. If bit 0 of **imm19** (notated as **l**) is set, fence against load operations. If bit 1 of **imm19** (notated as **s**) is set, fence against store operations.

```
fence   ; s = 1; l = 1
fence.s ; s = 1; l = 0
fence.l ; s = 0; l = 1
```

```
if l == 1 {
    ensure_loads_complete();
}
if s == 1 {
    ensure_stores_complete();
}
```

### 6.4.2. CINVAL - Invalidate Cache

| 31 | 17 16 | 15 14 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| unused | m | i d | r1 | 0 0 1 1 0 0 | 0 0 |

Invalidate a region of cache.

If bit 0 of **imm19** (notated as **d**) is set, invalidate data cache.
If bit 1 of **imm19** (notated as **i**) is set, invalidate instruction cache.
If bits 2..3 of **imm19** (notated as **m**) are equal to:
• 0, invalidate only the cache block containing the address in **r1**.
• 1, invalidate all cache blocks associated with the page containing the address in **r1**.
• 2, invalidate the entire cache on this LP. The value of **r1** is unused.
• 3, trigger an **INVALID** interrupt.

This operation unlocks this LP's lock state.

```
cinval.block   r1 ; m = 0; i = 1; d = 1
cinval.page    r1 ; m = 1; i = 1; d = 1
cinval.all        ; m = 2; i = 1; d = 1; r1 = zr
cinval.i.block r1 ; m = 0; i = 1; d = 0
cinval.i.page  r1 ; m = 1; i = 1; d = 0
cinval.i.all      ; m = 2; i = 1; d = 0; r1 = zr
cinval.d.block r1 ; m = 0; i = 0; d = 1
cinval.d.page  r1 ; m = 1; i = 0; d = 1
cinval.d.all      ; m = 2; i = 0; d = 1; r1 = zr
```

```
let addr = gpr[r1];
lock.locked = false;
match m {
  0 => invalidate_block(addr, d, i),
  1 => invalidate_page(addr, d, i),
  2 => invalidate_all(d, i),
  3 => trigger_interrupt(INVALID),
}
```

### 6.4.3. CFETCH - Fetch Cache

| 31 | | 16 | 15 | 14 | 13 | 12 | | 8 | 7 | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | unused | | i | s | l | | r1 | | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | |

Pre-fetch a cache block for a memory operation in the near future. Note that this does not guarantee the cache block is fresh from memory, as it will not invalidate and reload blocks that are already present in cache.

If bit 0 of **imm19** (notated as **l**) is set, fetch the cache block containing the address in **r1** for a data load operation.

If bit 1 of **imm19** (notated as **s**) is set, fetch the cache block containing the address in **r1** for a data store operation.

If bit 2 of **imm19** (notated as **i**) is set, fetch the cache block containing the address in **r1** for an instruction fetch operation.

This operation unlocks this LP's lock state.

```
cfetch.l   r1 ; l = 1, s = 0, i = 0
cfetch.s   r1 ; l = 0, s = 1, i = 0
cfetch.i   r1 ; l = 0, s = 0, i = 1
cfetch.ls  r1 ; l = 1, s = 1, i = 0
cfetch.li  r1 ; l = 1, s = 0, i = 1
cfetch.si  r1 ; l = 0, s = 1, i = 1
cfetch.lsi r1 ; l = 1, s = 1, i = 1
```

```
let addr = gpr[r1];
lock.locked = false;
fetch_block(addr, l, s, i);
```

## 6.5. Arithmetic

### 6.5.1. SSI - Set Shifted Immediate

| 31 | 16 15 14 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|
| val | sh | c | r1 | 0 0 0 0 1 0 0 0 |

Set 16 bits of **r1** to the upper 16 bits of **imm19** (notated as **val**), starting from the quarter-word indexed by bits 1..2 of **imm19** (notated as **sh**), with bit 0 of **imm19** (notated as **c**) indicating whether to set the lower unmodified bits in **r1** to 0 and to sign-extend the upper unmodified bits in **r1**.

```
ssi   r1, 0xFFFF, 0   ; c = 0; sh = 0
ssi   r1, 0xFFFF, 16  ; c = 0; sh = 1
ssi   r1, 0xFFFF, 32  ; c = 0; sh = 2
ssi   r1, 0xFFFF, 48  ; c = 0; sh = 3
ssi.c r1, 0xFFFF, 0   ; c = 1; sh = 0
ssi.c r1, 0xFFFF, 16  ; c = 1; sh = 1
ssi.c r1, 0xFFFF, 32  ; c = 1; sh = 2
ssi.c r1, 0xFFFF, 48  ; c = 1; sh = 3
```

```
let shift = sh << 4;
if c == 1 {
  gpr[r1] = (val as i64 << 48) >> (64 - shift);
} else {
  let mask = ~(0xFFFF << shift);
  gpr[r1] = (gpr[r1] & mask) | (val << shift);
}
```

### 6.5.2. ADD - Integer Add

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| imm9 | r3 | r2 | r1 | 0 0 0 0 0 0 1 0 |

Add **r2** with the sum of **r3** and zero-extended **imm9** and store the result in **r1**.

```
add r1, r2, r3      ; imm9 = 0
add r1, r2, r3, 511
```

```
gpr[r1] = gpr[r2] + (gpr[r3] + zero_extend(imm9));
```

### 6.5.3. SUB - Integer Subtract

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| imm9 | r3 | r2 | r1 | 0 0 1 0 0 0 | 1 0 |
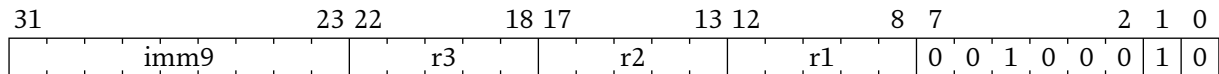
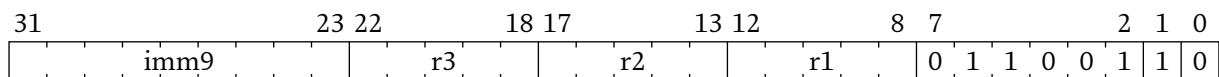Subtract **r3** from the sum of **r3** and zero-extended **imm9** and store the result in **r1**.

```
sub r1, r2, r3      ; imm9 = 0
sub r1, r2, r3, 511
```

```
gpr[r1] = gpr[r2] - (gpr[r3] + zero_extend(imm9));
```

### 6.5.4. MUL - Integer Multiply

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| imm9 | r3 | r2 | r1 | 0 1 0 0 0 0 | 1 0 |
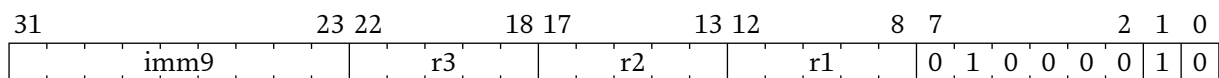
Multiply **r2** with the sum of **r3** and sign-extended **imm9** and place the lower 64 bits of the result into **r1**.

```
mul r1, r2, r3      ; imm9 = 0
mul r1, r2, r3, 255
mul r1, r2, r3, -256
```

```
gpr[r1] = gpr[r2] * (gpr[r3] + sign_extend(imm9));
```

### 6.5.5. UMULH - High Bits of Unsigned Integer Multiply

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| imm9 | r3 | r2 | r1 | 1 1 0 0 0 1 | 1 0 |
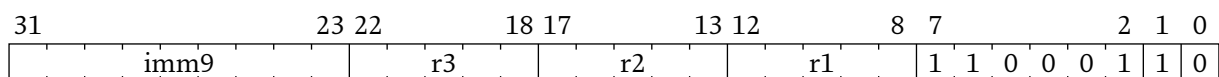
Multiply **r2** with the sum of **r3** and zero-extended **imm9** as 128-bit unsigned integers and place the upper 64 bits of the result into **r1**.

```
umulh r1, r2, r3      ; imm9 = 0
umuhl r1, r2, r3, 511
```

```
gpr[r1] = ((gpr[r2] as u128 *
         (gpr[r3] + zero_extend(imm9)) as u128) >> 64) as u64;
```

### 6.5.6. IMULH - High Bits of Signed Integer Multiply

| 31 | | 23 | 22 | | 18 | 17 | | 13 | 12 | | 8 | 7 | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | imm9 | | | r3 | | | r2 | | | r1 | | 1 | 1 | 1 | 0 | 0 | 1 | | 1 | 0 |
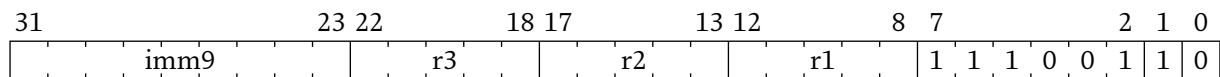
Multiply **r2** with the sum of **r3** and sign-extended **imm9** as 128-bit signed integers and place the upper 64 bits of the result into **r1**.

```
imulh r1, r2, r3       ; imm9 = 0
imulh r1, r2, r3, 511
```

```
gpr[r1] = ((gpr[r2] as i128 *
          (gpr[r3] + sign_extend(imm9)) as i128) >> 64) as u64;
```

### 6.5.7. UDIV - Unsigned Integer Divide

| 31 | | 23 | 22 | | 18 | 17 | | 13 | 12 | | 8 | 7 | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | imm9 | | | r3 | | | r2 | | | r1 | | 1 | 0 | 0 | 0 | 0 | 0 | | 1 | 0 |
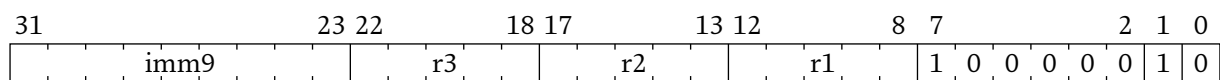
Divide **r2** by the sum of **r3** and zero-extended **imm9** as unsigned integers and store the result in **r1**. If the divisor is 0, all bits of **r1** are set to 1.

```
udiv r1, r2, r3       ; imm9 = 0
udiv r1, r2, r3, 511
```

```
let rhs = gpr[r3] + zero_extend(imm9);
gpr[r1] = if rhs != 0 {
    gpr[r2] as u64 / rhs as u64
} else {
    0xFFFFFFFFFFFFFFFF
};
```

### 6.5.8. IDIV - Signed Integer Divide

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| imm9 | | r3 | | r2 | | r1 | | 1 0 1 0 0 0 | | 1 | 0 |
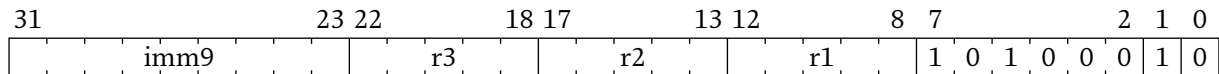
Divide **r2** by the sum of **r3** and sign-extended **imm9** as signed integers and store the result in **r1**. If the divisor is 0, all bits of **r1** are set to 1.

```
idiv r1, r2, r3       ; imm9 = 0
idiv r1, r2, r3, 255
idiv r1, r2, r3, -256
```

```
let rhs = gpr[r3] + sign_extend(imm9);
gpr[r1] = if rhs != 0 {
    gpr[r2] as i64 / rhs as i64
} else {
    0xFFFFFFFFFFFFFFFF
};
```

### 6.5.9. UREM - Unsigned Integer Remainder

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| imm9 | | r3 | | r2 | | r1 | | 1 1 0 0 0 0 | | 1 | 0 |
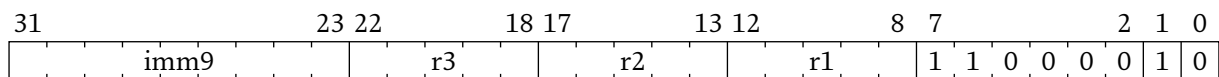
Divide **r2** by **r3** as unsigned integers and store the remainder in **r1**. If the divisor is 0, all bits of **r1** are set to 1.

```
urem r1, r2, r3       ; imm9 = 0
urem r1, r2, r3, 511
```

```
let rhs = gpr[r3] + zero_extend(imm9);
gpr[r1] = if rhs != 0 {
    gpr[r2] as u64 % rhs as u64
} else {
    0xFFFFFFFFFFFFFFFF
};
```

### 6.5.10. IREM - Signed Integer Remainder

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| imm9 | r3 | r2 | r1 | 1 1 1 0 0 0 | 1 0 |

Divide **r2** by **r3** as signed integers and store the remainder in **r1**. If the divisor is 0, all bits of **r1** are set to 1.

```
irem r1, r2, r3      ; imm9 = 0
irem r1, r2, r3, 255
irem r1, r2, r3, -256
```

```
let rhs = gpr[r3] + sign_extend(imm9);
gpr[r1] = if rhs != 0 {
    gpr[r2] as i64 % lhs as i64
} else {
    0xFFFFFFFFFFFFFFFF
};
```

### 6.5.11. ADDI - Integer Add Immediate

| 31 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|
| imm14 | r2 | r1 | 0 0 0 0 0 0 | 0 1 |

Add **r2** to zero-extended **imm14** and store the result in **r1**.

```
addi r1, r2, 16383
```

```
gpr[r1] = gpr[r2] + zero_extend(imm14);
```

### 6.5.12. SUBI - Integer Subtract Immediate

| 31 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|
| imm14 | r2 | r1 | 0 0 1 0 0 0 | 0 1 |

Subtract zero-extended **imm14** from **r2** and store the result in **r1**.

```
subi r1, r2, 16383
```

```
gpr[r1] = gpr[r2] - zero_extend(imm14);
```

### 6.5.13. MULI - Integer Multiply Immediate

| 31 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| imm14 | | r2 | | r1 | | 0 1 0 0 0 0 | | 0 | 1 |

Multiply **r2** with sign-extended **imm14** and store the result in **r1**.

```
muli r1, r2, 8191
muli r1, r2, -8192
```

```
gpr[r1] = gpr[r2] * sign_extend(imm14);
```

### 6.5.14. UDIVI - Unsigned Integer Divide Immediate

| 31 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| imm14 | | r2 | | r1 | | 1 0 0 0 0 0 | | 0 | 1 |

Divide **r2** as an unsigned integer by zero-extended **imm14** and store the result in **r1**. If the divisor is 0, all bits of **r1** are set to 1.

```
udivi r1, r2, 16383
```

```
gpr[r1] = if imm14 != 0 {
    gpr[r2] as u64 / zero_extend(imm14)
} else {
    0xFFFFFFFFFFFFFFFF
};
```

### 6.5.15. IDIVI - Signed Integer Divide Immediate

| 31 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| imm14 | | r2 | | r1 | | 1 0 1 0 0 0 | | 0 | 1 |

Divide **r2** as a signed integer by sign-extended **imm14** and store the result in **r1**. If the divisor is 0, all bits of **r1** are set to 1.

```
idivi r1, r2, 8191
idivi r1, r2, -8192
```

```
gpr[r1] = if imm14 != 0 {
    gpr[r2] as i64 / sign_extend(imm14)
} else {
    0xFFFFFFFFFFFFFFFF
};
```

### 6.5.16. UREMI - Unsigned Integer Remainder Immediate

| 31 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| imm14 | | r2 | | r1 | | 1 1 0 0 0 0 | | 0 | 1 |

Divide **r2** as an unsigned integer by zero-extended **imm14** and store the remainder in **r1**. If the divisor is 0, all bits of **r1** are set to 1.

```
uremi r1, r2, 16383
```

```
gpr[r1] = if imm14 != 0 {
    gpr[r2] as u64 % zero_extend(imm14)
} else {
    0xFFFFFFFFFFFFFFFF
};
```

### 6.5.17. IREMI - Signed Integer Remainder Immediate

| 31 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| imm14 | | r2 | | r1 | | 1 1 1 0 0 0 | | 0 | 1 |

Divide **r2** as an signed integer by sign-extended **imm14** and store the remainder in **r1**. If the divisor is 0, all bits of **r1** are set to 1.

```
iremi r1, r2, 8191
iremi r1, r2, -8192
```

```
gpr[r1] = if imm14 != 0 {
    gpr[r2] as i64 % sign_extend(imm14)
} else {
    0xFFFFFFFFFFFFFFFF
};
```

## 6.6. Bitwise Logic
### 6.6.1. AND - Logical And

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| imm9 | | r3 | | r2 | | r1 | | 0 0 0 0 0 1 | | 1 | 0 |

'AND' **r2** with the 'OR' of **r3** and zero-extended **imm9** and store the result in **r1**.

```
and r1, r2, r3        ; imm9 = 0
and r1, r2, r3, 511
```

```
gpr[r1] = gpr[r2] & (gpr[r3] | zero_extend(imm9));
```

### 6.6.2. OR - Logical Or

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| imm9 | r3 | r2 | r1 | 0 0 1 0 0 1 | 1 0 |
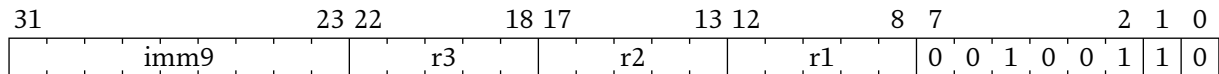
'OR' **r2** with the 'OR' of **r3** and zero-extended **imm9** and store the result in **r1**.

```
or r1, r2, r3      ; imm9 = 0
or r1, r2, r3, 511
```

```
gpr[r1] = gpr[r2] | (gpr[r3] | zero_extend(imm9));
```

### 6.6.3. NOR - Logical Nor

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| imm9 | r3 | r2 | r1 | 0 1 0 0 0 1 | 1 0 |
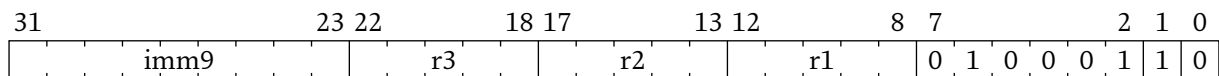
'NOR' **r2** with the 'OR' of **r3** and zero-extended **imm9** and store the result in **r1**.

```
nor r1, r2, r3     ; imm9 = 0
nor r1, r2, r3, 511
```

```
gpr[r1] = ~(gpr[r2] | (gpr[r3] | zero_extend(imm9)));
```

### 6.6.4. XOR - Logical Exclusive Or

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| imm9 | r3 | r2 | r1 | 0 1 1 0 0 1 | 1 0 |

'XOR' **r2** with the 'OR' of **r3** and zero-extended **imm9** and store the result in **r1**.

```
xor r1, r2, r3     ; imm9 = 0
xor r1, r2, r3, 511
```

```
gpr[r1] = gpr[r2] ^ (gpr[r3] | zero_extend(imm9));
```

### 6.6.5. ANDI - Logical And Immediate

| 31 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|
| imm14 | r2 | r1 | 0 0 0 0 0 1 | 0 1 |
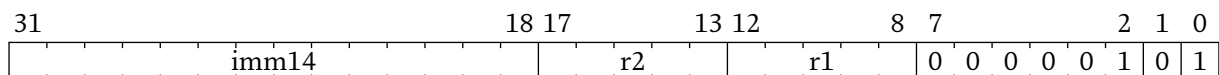
'AND' **r2** with zero-extended **imm14** and store the result in **r1**.

```
andi r1, r2, 16383
```

```
gpr[r1] = gpr[r2] & zero_extend(imm14);
```

### 6.6.6. ORI - Logical Or Immediate

| 31 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|----|-------|-------|-----|-------|
| imm14 | r2 | r1 | 0 0 1 0 0 1 | 0 1 |

'OR' **r2** with zero-extended **imm14** and store the result in **r1**.

```
ori r1, r2, 16383
```

```
gpr[r1] = gpr[r2] | zero_extend(imm14);
```

### 6.6.7. NORI - Logical Nor Immediate

| 31 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|----|-------|-------|-----|-------|
| imm14 | r2 | r1 | 0 1 0 0 0 1 | 0 1 |

'NOR' **r2** with zero-extended **imm14** and store the result in **r1**.

```
nori r1, r2, 16383
```

```
gpr[r1] = ~(gpr[r2] | zero_extend(imm14));
```

### 6.6.8. XORI - Logical Exclusive Or Immediate

| 31 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|----|-------|-------|-----|-------|
| imm14 | r2 | r1 | 0 1 1 0 0 1 | 0 1 |

'XOR' **r2** with zero-extended **imm14** and store the result in **r1**.

```
xori r1, r2, 16383
```

```
gpr[r1] = gpr[r2] ^ zero_extend(imm14);
```

### 6.6.9. SL - Shift Left

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| imm9 | | r3 | | r2 | | r1 | | 1 | 0 | 0 | 0 | 1 | 0 | | 1 | 0 | |

Shift **r2** left by the sum (modulo 64) of zero-extended **imm9** and **r3**, and store the result in **r1**.

```
sl r1, r2, r3, 63
sl r1, r2, 63      ; r3 = zr
sl r1, r2, r3      ; imm9 = 0
```

```
let shamt = (gpr[r3] + zero_extend(imm9)) & 0b111111;
gpr[r1] = gpr[r2] << shamt;
```

### 6.6.10. USR - Unsigned Shift Right

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| imm9 | | r3 | | r2 | | r1 | | 0 | 1 | 0 | 0 | 1 | 0 | | 1 | 0 | |

Shift **r2** right by the sum (modulo 64) of zero-extended **imm9** and **r3,** and store the result in **r1**. "Empty" bits at the most-significant end are set to zero.

```
usr r1, r2, r3, 63
usr r1, r2, 63      ; r3 = zr
usr r1, r2, r3      ; imm9 = 0
```

```
let shamt = (gpr[r3] + zero_extend(imm9)) & 0b111111;
gpr[r1] = gpr[r2] as u64 >> shamt;
```

### 6.6.11. ISR - Signed Shift Right

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| imm9 | | r3 | | r2 | | r1 | | 0 | 1 | 1 | 0 | 1 | 0 | | 1 | 0 | |

Shift **r2** right by the sum (modulo 64) of zero-extended **imm9** and **r3,** and store the result in **r1**. "Empty" bits at the most-significant end are copied from bit 63 of **r2.**

```
isr r1, r2, r3, 63
isr r1, r2, 63      ; r3 = zr
isr r1, r2, r3      ; imm9 = 0
```

```
let shamt = (gpr[r3] + zero_extend(imm9)) & 0b111111;
gpr[r1] = gpr[r2] as i64 >> shamt;
```

### 6.6.12. SI - Shift Immediate

| 31 30 29 | | 24 23 | | 18 17 | | 13 12 | | 8 7 | | | 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | i | rsh | | lsh | | r2 | | r1 | | 0 0 0 0 1 0 | 0 1 |

Shift **r2** left by bits 0..5 of **imm14** (notated as **lsh**), then shift right by bits 6..11 of **imm14** (notated as **rsh**). If bit 12 of **imm14** (notated as **i**) is set, the right shift is signed, otherwise unsigned.

```
si.u r1, r2, 63, 63 ; lsh = 1st, rhs = 2nd, i = 0
si.i r1, r2, 63, 63 ; lsh = 1st, rhs = 2nd, i = 1
```

```
gpr[r1] = if i == 0 {
   (gpr[r2] << lsh) as u64 >> rsh
} else {
   (gpr[r2] << lsh) as i64 >> rsh
};
```

### 6.6.13. CB - Clear Bits

| 31 30 29 | | 24 23 | | 18 17 | | 13 12 | | 8 7 | | | 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | rsh | | lsh | | r2 | | r1 | | 0 0 1 0 1 0 | 0 1 |

Shift an all-bits-set mask by left by bits 0..5 of **imm14** (notated as **lsh**), then shift right (unsigned) by bits 6..11 of **imm14** (notated as **rsh**). Clear bits in **r2** corresponding to set bits of the mask and store the result in **r1**.

```
cb r1, r2, 63, 63 ; lsh = 1st, rhs = 2nd
```

```
gpr[r1] = gpr[r2] &
   ~((0xFFFFFFFFFFFFFFFF as u64 << lsh) as u64 >> rhs);
```

### 6.6.14. ROR - Rotate Right

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| imm9 | | r3 | | r2 | | r1 | | 0 0 0 0 1 0 | | 1 | 0 |

Rotate **r2** right according to the sum of zero-extended **imm9** and **r3** as an unsigned integer and store the result in **r1**.

```
ror r1, r2, r3, 63
ror r1, r2, 63      ; r3 = zr
ror r1, r2, r3      ; imm9 = 0
```

```
let shamt = (gpr[r3] + zero_extend(imm9)) & 0b111111;
gpr[r1] = (gpr[r2] as u64 >> shamt)
        | (gpr[r2] as u64 << (64 - shamt));
```

### 6.6.15. ROL - Rotate Left

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| imm9 | | r3 | | r2 | | r1 | | 0 0 1 0 1 0 | | 1 | 0 |

Rotate **r2** left according to the sum of zero-extended **imm9** and **r3** as an unsigned integer and store the result in **r1**.

```
rol r1, r2, r3, 63
rol r1, r2, 63      ; r3 = zr
rol r1, r2, r3      ; imm9 = 0
```

```
let shamt = (gpr[r3] + zero_extend(imm9)) & 0b111111;
gpr[r1] = (gpr[r2] as u64 << shamt)
        | (gpr[r2] as u64 >> (64 - shamt));
```

### 6.6.16. REV - Reverse Bits

| 31 | 24 23 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| unused | set | r2 | r1 | 0 1 0 0 1 0 | 0 1 |

Reverse bits of **r2** based on bits 0..5 of **imm19** (notated as **set**) and store the result to **r1** and store the result in **r1**.

```
rev     r1, l1, 0b111111
rev.h   r1, r2 ; set = 0b100000
rev.q   r1, r2 ; set = 0b110000
rev.b   r1, r2 ; set = 0b111000
rev.bit r1, r2 ; set = 0b111111
```

```
let mut value = gpr[r2];
if set & 0b100000 != 0 {
  value = (0xFFFFFFFF00000000 & value) >> 32
        | (0x00000000FFFFFFFF & value) << 32;
}
if set & 0b10000 != 0 {
  value = (0xFFFF0000FFFF0000 & value) >> 16
        | (0x0000FFFF0000FFFF & value) << 16;
}
if set & 0b1000 != 0 {
  value = (0xFF00FF00FF00FF00 & value) >> 8
        | (0x00FF00FF00FF00FF & value) << 8;
}
if set & 0b100 != 0 {
  value = (0xF0F0F0F0F0F0F0F0 & value) >> 4
        | (0x0F0F0F0F0F0F0F0F & value) << 4;
}
if set & 0b10 != 0 {
  value = (0xCCCCCCCCCCCCCCCC & value) >> 2
        | (0x3333333333333333 & value) << 2;
}
if set & 0b1 != 0  {
  value = (0xAAAAAAAAAAAAAAAA & value) >> 1
        | (0x5555555555555555 & value) << 1;
}
gpr[r1] = value;
```

### 6.6.17. CSB - Count Set Bits

| 31 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|
| unused | r2 | r1 | 1 1 0 0 0 1 | 0 1 |

Count the number of set bits in **r2** and store the count in **r1**.

```
csb r1, r2
```

```
let mut count = 0;
for i in 0..64 {
  if (1 << i) & gpr[r2] != 0 {
    count += 1;
  }
}
gpr[r1] = count;
```

### 6.6.18. CLZ - Count Leading Zeroes

| 31 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|
| unused | r2 | r1 | 1 0 0 0 0 1 | 0 1 |

Count the number of leading (most-significant) bits in **r2** that are set to 0 and store the count in **r1**.

```
clz r1, r2
```

```
let mut count = 0;
// 63, 62, 61, etc.
for i in (0..64).rev() {
  if (1 << i) & gpr[r2] != 0 {
    break;
  }
  count += 1;
}
gpr[r1] = count;
```

### 6.6.19. CTZ - Count Trailing Zeroes

| 31 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|
| unused | r2 | r1 | 1 0 1 0 0 1 | 0 1 |

Count the number of trailing (least-significant) bits in **r2** that are set to 0 and store the count in **r1**.

```
ctz r1, r2
```

```
let mut count = 0;
for i in 0..64 {
  if (1 << i) & gpr[r2] != 0 {
    break;
  }
  count += 1;
}
gpr[r1] = count;
```

### 6.6.20. EXT - Extract Bits

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| unused | r3 | r2 | r1 | 1 0 0 0 0 1 | 1 0 |

Extract bits from **r2** according to a mask **r3** and place masked bits contiguously into **r1**. The inverse operation to **dep**.

```
ext r1, r2, r3
```

```
let mut result = 0;
let mut k = 0;
for i in 0..64 {
  // if mask bit is set
  if (1 << i) & gpr[r3] != 0 {
    // get corresponding bit from source
    let bit = (gpr[r2] >> i) & 1;
    // place source bit at next location
    result |= bit << k;
    k += 1;
  }
}
gpr[r1] = result;
```

### 6.6.21. DEP - Deposit Bits

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| unused | | r3 | | r2 | | r1 | | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

Deposit contiguous bits from **r2** according to a mask **r3** and place masked bits non-contiguously into **r1**. The inverse operation to **ext**.

```
dep r1, r2, r3
```

```
let mut result = 0;
let mut k = 0;
for i in 0..64 {
  // if mask bit is set
  if (1 << i) & gpr[r3] != 0 {
    // get next bit from source
    let bit = (gpr[r2] >> k) & 1;
    // place source bit at mask bit location
    result |= bit << i;
    k += 1;
  }
}
gpr[r1] = result;
```

## 6.7. Comparison

### 6.7.1. SEQ - Set Equal

| 31 | 23 | 22 | 18 | 17 | 13 | 12 | 8 | 7 | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| imm9 | | r3 | | r2 | | r1 | | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |

if **r2** is equal to the sum of sign-extended **imm9** and **r3**, set **r1** to 1, otherwise set **r1** to 0.

```
seq r1, r2, r3      ; imm9 = 0
seq r1, r2, r3, 511
```

```
gpr[r1] = (gpr[r2] == gpr[r3] + sign_extend(imm9)) as u64;
```

### 6.7.2. SULT - Set Unsigned Less Than

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| imm9 | r3 | r2 | r1 | 0 0 0 0 1 1 | 1 0 |
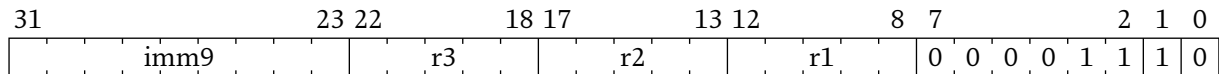
if **r2** is less than the sum of zero-extended **imm9** and **r3** as unsigned integers, set **r1** to 1, otherwise set **r1** to 0.

```
sult r1, r2, r3      ; imm9 = 0
sult r1, r2, r3, 511
```

```
gpr[r1] = (gpr[r2] as u64 < (gpr[r3] + zero_extend(imm9)) as u64)
as u64;
```

### 6.7.3. SILT - Set Signed Less Than

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| imm9 | r3 | r2 | r1 | 0 0 1 0 1 1 | 1 0 |
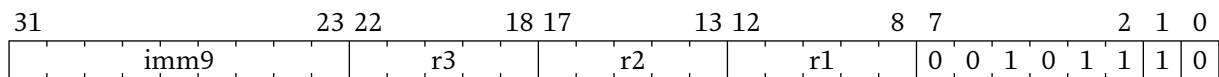
if **r2** is less than the sum of sign-extended **imm9** and **r3** as signed integers, set **r1** to 1, otherwise set **r1** to 0.

```
silt r1, r2, r3      ; imm9 = 0
sile r1, r2, r3, 255
sile r1, r2, r3, -256
```

```
gpr[r1] = (gpr[r2] as i64 < (gpr[r3] + zero_extend(imm9)) as i64)
as u64;
```

### 6.7.4. SULE - Set Unsigned Less or Equal

| 31 | 23 22 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|---|
| imm9 | r3 | r2 | r1 | 0 1 0 0 1 1 | 1 0 |

if **r2** is less than or equal to the sum of zero-extended **imm9** and **r3** as unsigned integers, set **r1** to 1, otherwise set **r1** to 0.

```
sule r1, r2, r3
sule r1, r2, r3, 511
```

```
gpr[r1] = (gpr[r2] as u64 < (gpr[r3] + zero_extend(imm9)) as u64)
as u64;
```

### 6.7.5. SILE - Set Signed Less or Equal

| 31 | | 23 | 22 | | 18 | 17 | | 13 | 12 | | 8 | 7 | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | imm9 | | | r3 | | | r2 | | | r1 | | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | | |

if **r2** is less than or equal to the sum of sign-extended **imm9** and **r3** as signed integers, set **r1** to 1, otherwise set **r1** to 0.
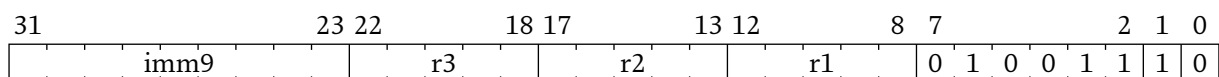
```
sile r1, r2, r3
sile r1, r2, r3, 255
sile r1, r2, r3, -255
```

```
gpr[r1] = (gpr[r2] as i64 <= (gpr[r3] + sign_extend(imm9)) as
i64) as u64;
```

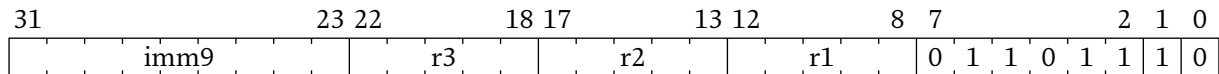### 6.7.6. SEQI - Set Equal Immediate

| 31 | | | 18 | 17 | | 13 | 12 | | 8 | 7 | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | imm14 | | | r2 | | | r1 | | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | | |
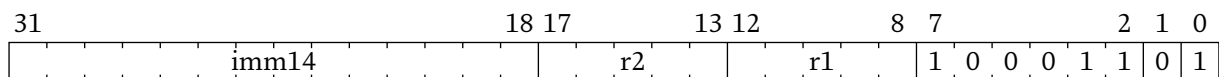
if **r2** is equal to sign-extended **imm14**, set **r1** to 1, otherwise set **r1** to 0.

```
seqi r1, r2, 8191
seqi r1, r2, -8192
```

```
gpr[r1] = (gpr[r2] as i64 == sign_extend(imm14)) as u64;
```

### 6.7.7. SULTI - Set Unsigned Less Than Immediate

| 31 | | | 18 | 17 | | 13 | 12 | | 8 | 7 | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | imm14 | | | r2 | | | r1 | | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | | |

if **r2** as an unsigned integer is less than zero-extended **imm14**, set **r1** to 1, otherwise set **r1** to 0.

```
sulti r1, r2, 16383
```

```
gpr[r1] = (gpr[r2] as u64 < zero_extend(imm14)) as u64;
```

### 6.7.8. SILTI - Set Signed Less Than Immediate
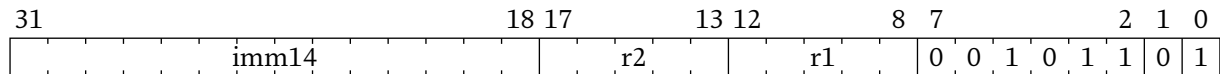
| 31 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|
| imm14 | r2 | r1 | 0 0 1 0 1 1 | 0 1 |

if **r2** as a signed integer is less than or equal to sign-extended **imm14**, set **r1** to 1, otherwise set **r1** to 0.

```
silti r1, r2, 8191
silti r1, r2, -8192
```

```
gpr[r1] = (gpr[r2] as i64 < sign_extend(imm14)) as u64;
```

### 6.7.9. SULEI - Set Unsigned Less or Equal Immediate

| 31 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|
| imm14 | r2 | r1 | 0 1 0 0 1 1 | 0 1 |

if **r2** as an unsigned integer is less than or equal to zero-extended **imm14**, set **r1** to 1, otherwise set **r1** to 0.

```
sulei r1, r2, 16383
```

```
gpr[r1] = (gpr[r2] as u64 <= zero_extend(imm14)) as u64;
```

### 6.7.10. SILEI - Set Signed Less or Equal Immediate

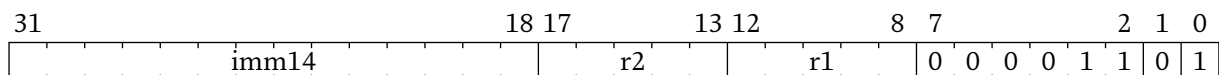| 31 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|
| imm14 | r2 | r1 | 0 1 1 0 1 1 | 0 1 |

if **r2** as a signed integer is less than or equal to sign-extended **imm14**, set **r1** to 1, otherwise set **r1** to 0.

```
silei r1, r2, 8191
silei r1, r2, -8192
```

```
gpr[r1] = (gpr[r2] as i64 <= sign_extend(imm14)) as u64;
```
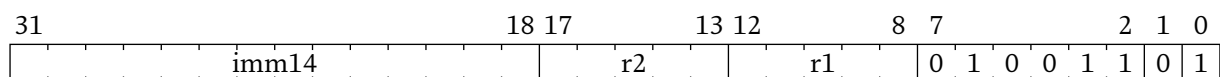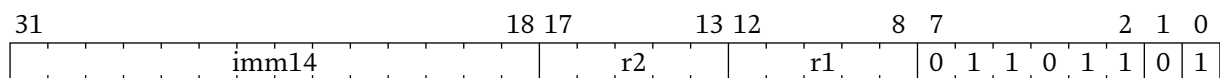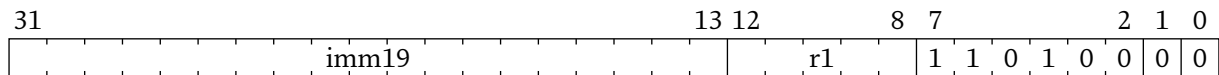
## 6.8. Control Flow

### 6.8.1. BZ - Branch If Zero

| 31 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| imm19 | | r1 | | 1 1 0 1 0 0 | | 0 | 0 |

if **r1** is equal to 0, set **ip** to the sum of sign-extended **imm19** shifted left by 2 and **ip**.

```
bz r1, symbol ; imm19 = (symbol - addr - 4) >> 2
```

```
if gpr[r1] == 0 {
  gpr[IP] = gpr[IP] + sign_extend(imm19) << 2;
}
```

### 6.8.2. BN - Branch If Not Zero

| 31 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| imm19 | | r1 | | 1 1 1 1 0 0 | | 0 | 0 |
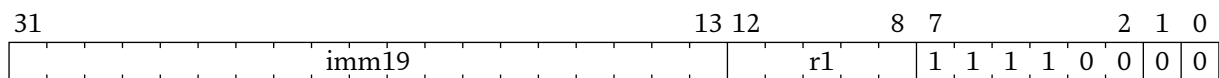
if **r1** is not equal to 0, set **ip** to the sum of sign-extended **imm19** shifted left by 2 and **ip**.

```
bn r1, symbol ; imm19 = (symbol - addr - 4) >> 2
```

```
if gpr[r1] != 0 {
  gpr[IP] = gpr[IP] + sign_extend(imm19) << 2;
}
```

### 6.8.3. JL - Jump and Link

| 31 | 18 | 17 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| imm14 | | r2 | | r1 | | 1 0 1 1 0 0 | | 0 | 1 |

Set **r1** to **ip** and set **ip** to the sum of zero-extended **imm14** shifted left by 2 and **r2**.

```
jl r1, r2, 8191
jl r1, r2, -8192
jl r1, r2        ; imm14 = 0
```

```
let ip = gpr[IP];
gpr[IP] = gpr[r2] + zero_extend(imm14) << 2;
gpr[r1] = ip;
```

### 6.8.4. JLR - Jump and Link Relative

| 31 | 18 17 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|---|
| imm14 | r2 | r1 | 1 0 0 1 0 0 | 0 1 |

Set **r1** to **ip** and set **ip** to the sum of **ip**, zero-extended **imm14** shifted left by 2, and **r2**.

```
jlr r1, r2, 8191
jlr r1, r2, -8192
jlr r1, r2       ; imm14 = 0
```

```
let ip = gpr[IP];
gpr[IP] += gpr[r2] + zero_extend(imm14) << 2;
gpr[r1] = ip;
```

## 6.9. System Control

### 6.9.1. SYSCALL - System Call Interrupt

| 31 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|
| unused | unused | 0 0 0 1 1 1 | 0 0 |

Trigger a **SYSCALL** interrupt.
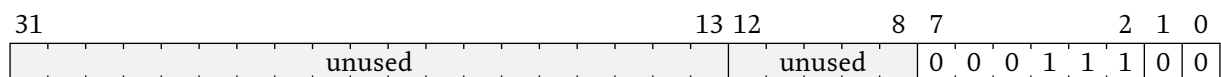
```
syscall
```

```
trigger_interrupt(SYSCALL);
```

### 6.9.2. BREAKPT - Breakpoint Interrupt

| 31 | 13 12 | 8 7 | 2 1 0 |
|---|---|---|---|
| unused | unused | 0 0 1 1 1 1 | 0 0 |

Trigger a **BREAKPT** interrupt.

```
breakpt
```

```
trigger_interrupt(BREAKPT);
```

### 6.9.3. WAIT - Wait for Interrupt

| 31 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|----|----|----|---|---|---|---|---|
| unused | | unused | | 1 1 1 1 1 1 | | 0 | 0 |

Pause execution and idle until an interrupt occurs. Must be in kernel mode to execute.

```
wait
```

```
if ctrl[STAT] & STAT_U != 0 {
   trigger_interrupt(INVALID);
}
pause_execution();
```

### 6.9.4. SPIN - Hint Spin-wait Loop

| 31 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|----|----|----|---|---|---|---|---|
| unused | | unused | | 0 1 0 1 1 1 | | 0 | 0 |

Hint a spin-wait loop to the processor. On complex implementations, this instruction may lower power consumption in spin-wait loops, mitigate performance issues due to execution pipelining, signal that this LP is currently not busy for hyper-threading mechanisms, etc.

```
spin
```

```
hint_spin();
```

### 6.9.5. IRET - Interrupt Return

| 31 | 13 | 12 | 8 | 7 | 2 | 1 | 0 |
|----|----|----|---|---|---|---|---|
| unused | | unused | | 1 0 0 1 1 1 | | 0 | 0 |

Return from an interrupt handler or jump into user mode. Must be in kernel mode to execute.

```
iret
```

```
if ctrl[STAT] & STAT_U != 0 {
   trigger_interrupt(INVALID);
}
ctrl[STAT] = ctrl[INTSTAT];
gpr[IP] = ctrl[INTIP];
```

### 6.9.6. LCTRL - Load Control Register

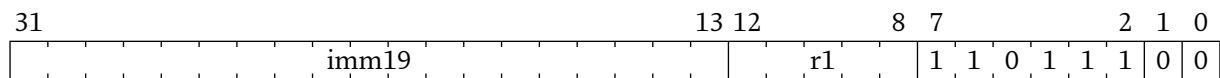| 31 | | 13 | 12 | | 8 | 7 | | | | | | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| | imm19 | | | r1 | | 1 | 0 | 1 | 1 | 1 | 1 | | 0 | 0 |
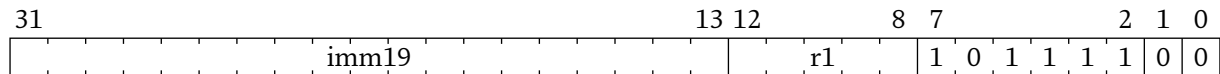
Load from control register **imm19** to **r1**. Must be in kernel mode to execute.

```
lctrl r1, int0
```

```
if ctrl[STAT] & STAT_U != 0 || !is_valid_ctrl(imm19) {
   trigger_interrupt(INVALID);
}
gpr[r1] = ctrl[imm19];
```

### 6.9.7. SCTRL - Store Control Register

| 31 | | 13 | 12 | | 8 | 7 | | | | | | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| | imm19 | | | r1 | | 1 | 1 | 0 | 1 | 1 | 1 | | 0 | 0 |

Store **r1** to control register **imm19**. Must be in kernel mode to execute.

```
sctrl int0, r1
```

```
if ctrl[STAT] & STAT_U != 0 || !is_valid_ctrl(imm19) {
   trigger_interrupt(INVALID);
}
ctrl[imm19] = gpr[r1];
```

## 6.10. Pseudo-instructions

Assemblers for the Aphelion ISA must implement the following pseudo-instructions to be compliant.

Some instructions may produce **linker relocations** so that symbol values that change during linking are accurately updated where they are referenced. Relocation types are detailed in the Aphelion ABI Specification.

Note that the value of **addr** in the following expansions is a shorthand for the address of the *first* instruction in the expansion.

### 6.10.1. CALL - Procedure Call

Perform a relative procedure call to **symbol** and save the return address to **r1**, using **r2** as a scratch register. Produces a **CALL** relocation at the address of the **ssi.c** instruction.

```
call r1, r2, symbol
call r1, symbol      ; r2 = r1
call symbol          ; r1 = lp, r2 = lp
  ; Expands to:
ssi.c r2, ((symbol - addr) >> 16) & 0xFFFF, 16
jlr   r1, r2, ((symbol - addr) & 0xFFFF) >> 2
```

Optionally, if **symbol** is a symbol or a symbol offset that *does not* require a relocation (within the same section, between pinned sections, etc.), assemblers may produce a smaller, logically equivalent expansion if possible.

### 6.10.2. FCALL - Far Procedure Call

Perform an absolute procedure call to **symbol** and save the return address to **r1**, using **r2** as a scratch register. Produces an **FCALL** relocation at the address of the **ssi.c** instruction.

```
fcall r1, r2, symbol
fcall r1, symbol      ; r2 = r1
fcall symbol          ; r1 = lp, r2 = lp
  ; Expands to:
ssi.c r2, (symbol >> 48) & 0xFFFF, 48
ssi   r2, (symbol >> 32) & 0xFFFF, 32
ssi   r2, (symbol >> 16) & 0xFFFF, 16
jl    r1, r2, (symbol & 0xFFFF) >> 2
```

### 6.10.3. LI - Load Immediate

Load a constant, symbol, or symbol offset (notated as **val**) into **r1**.

If **val** is a constant, a sequence of **ssi** instructions shall be used. The assembler should aim to minimize the size of the sequence, though simple implementations may choose to always use the full four-instruction expansion, shown below:

```
li r1, val
   ; Expands to:
ssi.c r1, val >> 48, 48
ssi   r1, (val >> 32) & 0xFFFF, 32
ssi   r1, (val >> 16) & 0xFFFF, 16
ssi   r1, val & 0xFFFF, 0
```

If **val** is a symbol or a symbol offset, the above expansion shall be used and an **LI** relocation shall be produced at the address of the **ssi.c** instruction.

Optionally, if **val** is a symbol or a symbol offset that *does not* require a relocation (within the same section, between pinned sections, etc.), arithmetic involving the instruction pointer may be used to produce a smaller expansion.

### 6.10.4. RET - Return from Procedure Call

Return from a standard procedure call by performing a jump to the address in **r1**.

```
ret r1
ret     ; r1 = lp
   ; Expands to:
jl zr, r1, 0
```

### 6.10.5. NOP - No Operation

The canonical way to do nothing.

```
nop
   ; Expands to:
or zr, zr, zr
```

### 6.10.6. MOV - Move Register

Copy the value of **r2** into **r1**.

```
mov r1, r2
   ; Expands to:
or r1, r2, zr
```

# 7. Glossary

- **word**: The size of a register. 8 bytes, 64 bits.
- **half-word**: 4 bytes, 32 bits.
- **quarter-word**: 2 bytes, 16 bits.
- **logical processor, LP**: a distinct Aphelion context that executes instructions independently from other contexts in a system.
- **page**: a contiguous block of memory 4KiB ($2^{12}$B) in size.
- **weak consistency**: a memory model which allows global reordering of memory accesses, restricted by memory fence operations.
- **data cache, d-cache**: internal memory used to speed up memory loads/stores.
- **instruction cache, i-cache**: internal memory used to speed up instruction fetches and aid the execution pipeline.
- **(linker) relocation**: An object file entry that inserts/replaces a value in a section such that symbol values that change during linking are accurately updated where they are referenced.