SIRC-1 CPU Reference Manual

Featuring the SIRCIS Instruction Set

Technical Specifications

Architecture:	16-bit RISC Load/Store
Address Bus:	24-bit
Data Bus:	16-bit
Instruction Size:	32-bit (fixed)
Registers:	16 x 16-bit registers
Execution:	6-stage pipeline

Version 1.0

January 31, 2026

About This Manual

This reference manual provides comprehensive documentation for the SIRC-1 CPU architecture and the SIRCIS (SIRC Instruction Set) instruction set. It is intended for:

- Assembly language programmers
- Compiler and toolchain developers
- Hardware designers and implementers
- System software developers

Document Conventions

Throughout this manual, the following conventions are used:

- Register names appear in typewriter font: `r1`, `ph`, `sr`
- Hexadecimal values are prefixed with `0x`: `0x1A`
- Immediate values are prefixed with `#`: `#123`
- Instruction mnemonics appear in bold typewriter font: **`ADDI`**
- Addressing modes use parentheses for indirection: `(#4, a)`

Contents

List of Tables

List of Figures

Part I

Architecture Overview

Chapter 1

Introduction

1.1 Overview

The SIRC-1 is a 16-bit RISC processor ideal for general purpose computing.
It implements the SIRCIS (SIRC Instruction Set) instruction set architecture, which provides a comprehensive set of operations while maintaining a clean, consistent design philosophy.

1.2 Design Philosophy

1.2.1 RISC Principles

The SIRC-1 adheres to Reduced Instruction Set Computer (RISC) design principles:

- **Load/Store Architecture:** All ALU operations work exclusively on registers. Memory access is performed only through dedicated load and store instructions.
- **Fixed Instruction Length:** All instructions are exactly 32 bits (2 words) in length, simplifying instruction fetch and decode logic.
- **Simple Instruction Formats:** Only four instruction formats are supported: Implied, Immediate, Short Immediate with Shift, and Register.
- **Register-Rich:** Sixteen addressable registers provide ample workspace for computations without frequent memory access.
- **Consistent Execution Time:** All instructions execute in exactly 6 clock cycles, eliminating the need for microcode and simplifying hardware implementation.

1.2.2 Design Goals

The SIRC-1 was designed with several key goals in order of importance:

Simplicity The CPU should be simple enough to be implemented in hardware without requiring complex microcode engines or multi-level instruction decoders.

Developer Ergonomics It should have powerful instruction encodings that include conditional execution and bit shifting that can help reduce the number of instructions and branches to keep track of.

Performance While simple, the design should allow for reasonable performance through efficient instruction encoding and the potential for future enhancements like pipelining.

Memory Protection Basic privilege levels and memory segmentation provide rudimentary protection for system software.

1.2.3 Applications

The SIRC-1 is designed for ideal for general purpose computing, such as the core of a microcomputer.
It should not be used in memory constrained environments, as the fixed length instructions and lack of byte addressing can potentially mean larger program sizes.

1.3 Key Features

1.3.1 Architecture Characteristics

Feature	Specification
Data Width	16-bit
Addressing	Word Addressing
Address Bus	24-bit (32MB address space)
Instruction Size	32-bit fixed
General Purpose Registers	7 (r1–r7)
Address Register Pairs	4 (l, a, s, p)
Total Addressable Registers	16
Instruction Formats	4
Addressing Modes	8
Condition Codes	16

Table 1.1: SIRC-1 CPU Specifications

1.3.2 Execution Model

The SIRC-1 employs a six-stage execution sequence (one stage per clock):

1. **Instruction Fetch (First Word)** – Fetch bits 31–16 from memory
2. **Instruction Fetch (Second Word)** – Fetch bits 15–0 from memory
3. **Decode and Register Fetch** – Decode the instruction and read source registers
4. **Execute and Address Calculation** – Perform ALU operation or calculate effective address
5. **Memory Access** – Read from or write to memory (or NOP for register operations)
6. **Write Back** – Write results to destination register

This fixed six-stage model means that all instructions take exactly 6 clock cycles to complete, regardless of their complexity. While this may seem wasteful for simple register operations that don't require memory access, it dramatically simplifies the hardware design by eliminating the need for variable-length execution cycles or microcode.

1.3.3 Coprocessors

The SIRC-1 is made up of multiple coprocessors (also known as modules) that all share the same address/data bus and register file. Only one coprocessor can be active at one time and are usually activated using the COP instructions (except for the "exception unit" which can be activated when physical pins are asserted on the chip). All implementations of the SIRC-1 will have the "processing unit" as coprocessor 0x0 and "exception unit" as coprocessor 0x1. Other coprocessors are optional and will vary depending on which SIRC-1 model you have. The instruction set supports up to 16 coprocessors.

ID	Name	Optional	Purpose
0x0	Processing Unit	No	Executes instructions
0x1	Exception Unit	No	Handles interrupts and exceptions
0x2	DMA Unit	Yes	Transfers large amounts of data via the bus
0x3	Maths Unit	Yes	Performs mathematical operations in hardware

Table 1.2: List of coprocessors

Each coprocessor can execute 16 opcodes, the first 8 (0x0-0x7) can be called in any mode. The last 8 (0x8-0xF) can only be called in supervisor mode.

1.3.4 Privilege Levels

The SIRC-1 supports two privilege levels:

Supervisor Mode Full access to all registers and instructions. Can set the high bytes of address register pairs, access privileged status register bits, and execute privileged coprocessor operations.

Protected Mode Restricted access for user programs. Cannot write to the high bytes of address register pairs (providing memory segmentation/protection), cannot access privileged status register bits, and cannot execute privileged coprocessor instructions.

The privilege level is controlled by the Protected Mode bit in the status register.

1.4 Memory Organization

1.4.1 Address Space

The SIRC-1 uses word addressing, where each address refers to a 16-bit word. With a 24-bit address space, this allows access to $2\hat{2}4 = 16{,}777{,}216$ words, or 32 megabytes of memory. Addresses are formed by combining the 8-bit high byte and 16-bit low word of address register pairs:

$$\text{Address} = (\text{RegHigh}[7:0] \ll 16) \mid \text{RegLow}[15:0] \tag{1.1}$$

Note that the upper 8 bits of the high register (bits 15–8) are not used for addressing but are available for storage, enabling techniques like tagged pointers.

1.4.2 Memory Map

The CPU expects certain memory regions to be defined:

- **Reset Vector** (`0x000000`): Initial program counter after reset
- **Fault Vectors** (`0x000001`–`0x000007`): Fault vectors
- **Hardware Vectors** (`0x000010`–`0x000050`): Hardware exception vectors (offset by 0x10 due to hardware implementation)
- **User Vectors** (`0x000060`–`0x0000FF`): User exception vectors

Beyond these reserved regions, the memory layout is flexible and can be defined by system software.

1.4.3 Alignment

- All memory addresses refer to 16-bit words
- There are no alignment restrictions for general memory I/O
- All instructions span 2 words and must begin at an even word address
- Unaligned instruction fetches (odd word address) trigger an alignment fault exception

1.5 Document Organization

This manual is organized into three main parts:

Part I: Architecture Overview Covers the register model, status register, and overall CPU architecture.

Part II: Instruction Set Architecture Details instruction formats, addressing modes, shift operations, and condition codes.

Part III: SIRCIS Instruction Reference Provides detailed documentation for each instruction, including encoding, operation, and examples.

Appendices provide quick reference materials including complete opcode maps, timing diagrams, and notes on undocumented instructions.

Chapter 2

Register Model

2.1 Overview

The SIRC-1 CPU provides sixteen 16-bit registers that serve various purposes. These registers can be classified into three categories:

- General purpose registers (`r1`–`r7`)
- Address register pairs (`l`, `a`, `s`, `p`)
- Status register (`sr`)

All registers are 16 bits wide. However, the address registers can be used in pairs to form 24-bit addresses for memory operations.

2.2 General Purpose Registers

2.2.1 Register Set

The SIRC-1 provides seven general-purpose registers:

Register	Description
`r1`	General purpose register 1
`r2`	General purpose register 2
`r3`	General purpose register 3
`r4`	General purpose register 4
`r5`	General purpose register 5
`r6`	General purpose register 6
`r7`	General purpose register 7

These registers can be used freely for any purpose by the programmer. They hold 16-bit values and can be used as source or destination operands for all ALU instructions.

2.2.2 Usage Conventions

While there are no hardware-enforced conventions, typical usage patterns include:

- `r1`–`r4`: Temporary values and expression evaluation
- `r5`–`r7`: Preserved across function calls (by convention)

These are software conventions only and are not enforced by the CPU.

2.3 Address Register Pairs

2.3.1 Overview

The SIRC-1 provides four address register pairs, each consisting of a high and low 16-bit register. When used together, these pairs form 24-bit addresses by combining the low 8 bits of the high register with the full 16 bits of the low register.

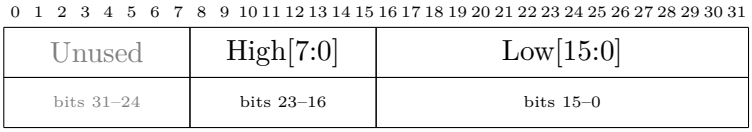

Figure 2.1: Address Register Pair Formation

The upper 8 bits (bits 31–24) of the high register are not used for addressing but remain available for storage, enabling techniques such as tagged pointers.

Although each address register pair has a specific name, and a suggested usage by convention, they are all treated the same by STOR/LOAD instructions and they can all be used for indirect addressing the same way.

In fact, you can perform a jump by simply loading a value in to the Program Counter registers.

However, it is worth noting that the Link Register pair will be updated automatically though specific instructions (LJSR, BRSR) and the Program Counter pair is incremented after every instruction.

2.3.2 Link Register (l)

Component	Register
High Byte	`lh` (privileged)
Low Word	`ll`

The link register pair stores return addresses for subroutine calls. When a subroutine is called using **`LJSR`** or **`BRSR`**, the return address is automatically saved to this register pair.

Typical Use:

- Storing return addresses from subroutine calls
- Alternate storage for general memory addressing

2.3.3 Address Register (a)

Component	Register
High Byte	`ah` (privileged)
Low Word	`al`

A general-purpose address register pair used for pointer operations and memory access.
Typical Use:

- Base pointer for data structures
- Array indexing
- General memory addressing

2.3.4 Stack Pointer (s)

Component	Register
High Byte	`sh` (privileged)
Low Word	`sl`

The stack pointer register pair points to the current top of the stack. Pre-decrement and post-increment addressing modes are particularly useful with this register.
Typical Use:

- Stack operations (push/pop)
- Local variable allocation
- Function parameter passing

2.3.5 Program Counter (p)

Component	Register
High Byte	`ph` (privileged)
Low Word	`pl`

The program counter register pair points to the next instruction to be executed. It is automatically incremented by 1 after each 16-bit word fetch. Since instructions are 32 bits (two words), the PC is incremented twice during a complete instruction fetch, advancing by 2 words total per instruction.
Typical Use:

- Automatically updated during normal execution
- Modified by branch and jump instructions
- Used for position-independent code

2.4 Status Register

The status register (`sr`) is a special 16-bit register that contains CPU status flags and control bits. It is divided into two bytes:

- **Lower Byte (bits 7–0):** Condition flags, accessible in both supervisor and protected modes
- **Upper Byte (bits 15–8):** Control flags, accessible only in supervisor mode (privileged)

The status register is discussed in detail in Chapter 3.

2.5 Privilege Restrictions

2.5.1 High Address Register Restrictions

The high component of each address register pair (`lh`, `ah`, `sh`, `ph`) is considered privileged:

- In **supervisor mode**: Can be read from and written to freely
- In **protected mode**:
  - Reading is allowed
  - Writing triggers a privilege exception

This restriction provides a form of memory segmentation/protection. User programs running in protected mode cannot modify the segment portion of addresses, preventing them from accessing memory outside their designated segment.

2.6 Register Addressing

Registers are addressed using 4-bit register identifiers in instruction encoding:

ID	Register	Privileged
0x0	`sr`	Upper byte only
0x1	`r1`	No
0x2	`r2`	No
0x3	`r3`	No
0x4	`r4`	No
0x5	`r5`	No
0x6	`r6`	No
0x7	`r7`	No
0x8	`lh`	Yes
0x9	`ll`	No
0xA	`ah`	Yes
0xB	`al`	No
0xC	`sh`	Yes
0xD	`sl`	No
0xE	`ph`	Yes
0xF	`pl`	No

Table 2.1: Register Encoding

2.7 Usage Examples

2.7.1 Basic Register Operations

```
; Load immediate value into r1
LOAD r1, #100

; Add 2 to the value stored in r1 and store the result into r2
LOAD r2, r1
ADDI r2, #2

; Add r1 and r2, store in r3
ADDR r3, r1, r2
```

2.7.2 Address Register Usage

```
; Load value from memory at address in 'a' register pair
LOAD r1, (a)

; Store r1 to memory at address (a + 8)
STOR (#8, a), r1

; Push r1 onto stack (pre-decrement)
```

```
8 STOR -(#2, s), r1
9
10 ; Pop stack into r2 (post-increment)
11 LOAD r2, (s)+
```

2.7.3 Subroutine Call

```
1 ; Call subroutine at address in 'a' register
2 ; Return address saved to 'l' register automatically
3 LJSR (a)
4
5 ; ...subroutine code...
6
7 ; Return from subroutine
8 ; Copies 'l' register to 'p' register
9 RETS
```

Chapter 3

Status Register

3.1 Overview

The Status Register (`sr`) is a 16-bit special-purpose register that contains condition flags and control bits for the CPU. It is divided into two bytes with different privilege levels:

- **Lower Byte (bits 7–0):** Condition flags – readable and writable in both supervisor and protected modes
- **Upper Byte (bits 15–8):** Control and mode flags – accessible only in supervisor mode

3.2 Status Register Layout

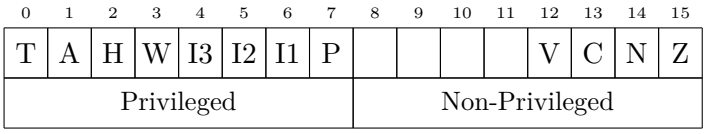

Figure 3.1: Status Register Bit Layout

3.3 Condition Flags (Non-Privileged)

These flags reflect the results of ALU operations and comparisons. They can be read and written by code running in any privilege level.

3.3.1 Zero Flag (Z) – Bit 0

Set when: The result of an operation is zero

Cleared when: The result of an operation is non-zero

Tested by: Equal (`==`), Not Equal (`!=`) condition codes

3.3.2 Negative Flag (N) – Bit 1

Set when: The result of an operation is negative (bit 15 = 1) when interpreted as a signed two's complement value

Cleared when: The result is positive or zero (bit 15 = 0)

Tested by: Negative Set (`NS`), Negative Clear (`NC`), and signed comparison condition codes

3.3.3 Carry Flag (C) – Bit 2

Set when: An addition produces a carry out of bit 15, or a subtraction produces a borrow

Cleared when: No carry/borrow occurs

Tested by: Carry Set (`CS`), Carry Clear (`CC`), Unsigned Higher (`HI`), Unsigned Lower or Same (`LO`) condition codes

Note: For unsigned comparisons, the carry flag acts as an unsigned overflow indicator

3.3.4 Overflow Flag (V) – Bit 3

Set when: A signed arithmetic operation produces a result that cannot be represented in 16 bits (signed overflow)

Cleared when: No signed overflow occurs

Tested by: Overflow Set (`OS`), Overflow Clear (`OC`), and signed comparison condition codes

Note: Overflow occurs when adding two positive numbers yields a negative result, or adding two negative numbers yields a positive result

3.3.5 Reserved Flags – Bits 4–7

Bits 4 through 7 of the lower byte are reserved for future use. They should not be relied upon by software.

3.4 Control Flags (Privileged)

These flags control CPU behavior and can only be modified in supervisor mode. Attempts to write these bits in protected mode will be ignored (the write affects only the lower byte).

3.4.1 Protected Mode (P) – Bit 8

When Clear (0): CPU operates in supervisor mode with full privileges

When Set (1): CPU operates in protected mode with restricted access to:

- High bytes of address register pairs (`lh`, `ah`, `sh`, `ph`)
- Upper byte of status register
- Privileged coprocessor operations (opcodes > 0x07xx)

Note: This bit provides basic memory protection by preventing user code from changing memory segments. With the exception of writing to the status register, preforming a restricted operation will cause a "privilege violation" fault.

3.4.2 Interrupt Mask Bits (I1, I2, I3) – Bits 9–11

These three bits control which interrupt priority levels are masked.

If an exception comes in with a lower priority than the mask, it will not be serviced until the higher priority interrupts are finished.

These bits should not be modified by the user, and are automatically set by the exception coprocessor.

If a level 5 interrupt is triggered while the level 5 interrupt is masked it will cause a fault.

If a fault occurs while handling another fault the CPU will be halted. This is due to the architecture of the exception unit.

Interrupt handling is discussed in detail in Chapter 4.

Value	Interrupt Type
0x0	No Interrupt
0x1	Software Exception
0x2	Level 1 Hardware
0x3	Level 2 Hardware
0x4	Level 3 Hardware
0x5	Level 4 Hardware
0x6	Level 5 Hardware
0x7	Fault

Table 3.1: Interrupt Mask Settings

3.4.3 Waiting for Interrupt (W) – Bit 12

Set when: The `WAIT` instruction is executed

Cleared when: An interrupt or exception occurs

Purpose: Indicates the CPU is in low-power wait state

Note: This bit is primarily for hardware implementation, and is mapped to an external pin. No program will ever read this bit as 1 as the CPU is not executing when this bit is active.

3.4.4 CPU Halted (H) – Bit 13

Set when: A `HALT` instruction is executed (if implemented)

Cleared when: CPU is reset

Purpose: Indicates the CPU has halted execution

Note: This feature may not be implemented in all versions

Note: This bit is primarily for hardware implementation, and is mapped to an external pin. No program will ever read this bit as 1 as the CPU is not executing when this bit is active.

3.4.5 Trap on Address Overflow (A) – Bit 14

When Set (1): Memory address calculations that overflow or underflow will trigger an address overflow exception

When Clear (0): Address overflows wrap around without any side effects

Purpose: Helps detect pointer errors during development

Typical Use: Enabled during debugging, disabled in production for performance

3.4.6 Trace Mode (T) – Bit 15

When Set (1): An exception is generated after every instruction

When Clear (0): Normal execution

Purpose: Allows debuggers to single-step through code

Note: Used by SIRCIT (SIRC Instruction Tracer) for debugging

3.5 Status Register Updates

3.5.1 Automatic Updates

The condition flags (Z, N, C, V) are automatically updated by most ALU instructions, unless explicitly disabled:

- **Arithmetic Instructions** (`ADDI`, `SUBI`, `ADCI`, `SBCI`): Update all four condition flags
- **Logical Instructions** (`ANDI`, `ORRI`, `XORI`): Update Z, N and V flags; C flag is cleared
- **Compare Instructions** (`CMPI`, `CMPR`): Same as Arithmetic Instructions but does not update destination
- **Test Instructions** (`TSAI`, `TSAR`, `TSXI`, `TSXR`): Same as Logical Instructions but does not update destination
- **Load Instructions** (`LOAD`, `LDEA`): Does not update condition flags
- **Shift Operations**: Can optionally update flags based on the instruction's status register update source field

3.5.2 Manual Updates

The status register can be directly manipulated like any other register:

```
; Set protected mode bit (only in supervisor mode)
ORRI sr, #0x0100

; Clear interrupt mask
ANDI sr, #0xF1FF

; Restore status register saved to stack
LOAD sr, (s)+
```

Important: When running in protected mode, writes to the status register only affect the lower byte (bits 7–0). The upper byte remains unchanged.

3.6 Reading the Status Register

In supervisor mode, reading `sr` returns the full 16-bit value. In protected mode, reading `sr` returns the lower byte in bits 7–0, with bits 15–8 masked to zero (for security).

```
; In supervisor mode - gets full 16 bits
LOAD r1, sr   ; r1 = 0xXXXX (all bits)

; In protected mode - upper byte masked
LOAD r1, sr   ; r1 = 0x00XX (upper byte forced to 0)
```

3.7 Exception Handling Effects

When an exception occurs:

1. The current status register is saved
2. Protected mode bit (P) is cleared (entering supervisor mode)
3. Trace mode bit (T) is cleared (disabling single-stepping in exception handler)
4. Interrupts may be masked depending on exception type

When returning from an exception with **RETE**, the saved status register is restored.

3.8 Usage Examples

3.8.1 Conditional Execution

```
; Compare r1 with r2
CMPI r1, r2

; Branch if equal
BRAN|== @label_equal

```

```
; Branch if r1 > r2 (unsigned)
BRAN|HI @label_greater

; Add 0x16 to r1 if r1 >= r2 (signed)
ADDI|>= r1, #0x16
```

3.8.2 Manual Status Flag Control

The SIRC-1 provides syntax to manually control how status flags are updated during ALU and shift operations. This is particularly useful for advanced control flow and flag preservation. See Section 7.12 in Chapter 7 for complete details.

```
; Update flags from shift instead of ALU
ADDI[S] r2, #1, LSL #2          ; Flags from shift, not addition

; Preserve flags across operations
CMPI r1, r2                     ; Set comparison flags
ADDI[N] r3, #1                  ; Increment without changing flags
BRAN|== still_uses_cmpi_flags ; Branch uses CMPI flags
```

3.8.3 Testing Bits

```
; Test if bit 5 is set in r1
TSAI r1, #0x0020  ; AND with 0x0020, update flags only

; Branch if bit was set (result non-zero)
BRAN|!= bit_was_set
```

3.8.4 Enabling Protected Mode

```
; Entering user mode (from supervisor mode)
ORRI sr, #0x0100  ; Set protected mode bit
; Now running in protected mode

; Attempting to write privileged register will fault
; ADDI ah, #0x05  ; This would cause exception!
```

3.8.5 Saving and Restoring Status

```
; Save status register
LOAD r7, sr

; Do some operations that change flags
ADDI r1, r2, r3
CMPI r1, #100

; Restore original status
LOAD sr, r7
```

Chapter 4

Exception and Interrupt Handling

4.1 Overview

An exception is an event that causes the program flow to be diverted to an address stored in a predefined vector.

Some are directly caused by an instruction, some are a side effect of an instruction and some are asynchronously caused by external hardware.

They are managed by the exception coprocessor, which will take control of the processor when there is a pending exception.

4.2 Exception Types

4.2.1 Abort Exceptions (Faults)

When an Abort Exception occurs, the instruction causing the exception does not have any effect (it is cancelled after decode stage). The program address stored in the link register is the address of the faulting instruction so it can be retried (RETI will return to the same instruction again) This is important for things like privilege violation because you don't want the illegal instruction to modify any state. These are also referred to as faults. The vectors for abort exceptions are in the 0x1-0xF range.

- **Bus Fault:** Raised when an external device raised an error via the bus error CPU pin.

  This could happen, for example, if a unmapped address is presented by the CPU and another chip detects this and raises an error. It could also be used to implement virtual memory as the instruction is aborted and program counter is not incremented, so the address can be re-calculated and the operation retried.

- **Alignment Fault:** Raised when fetching instructions if the program counter contains an odd word address

  This is to simplify fetching as the second word of an instruction is always at pl | 0x1 and ensures that we never have to worry about instructions overflowing segments (e.g. if the first word is at 0xFFFF and the second word at 0x0000) which is a weird edge case that might complicate fetching.

- **Segment Overflow Fault:** Raised with some instructions if the computed address would go outside the current segment.

E.g. if you are accessing data in the stack segment, and then compute an address that overflows, it is probably a stack overflow. There might be situations where you want address calculations to wrap around so it is only raised if the `TrapOnAddressOverflow` SR bit is set. It is also raised if the program counter overflows, you can tell the difference by reading which phase the fault was raised from the fault metadata register

- **Invalid Opcode Fault:** Raised when a co-processor call is done for a non-existant co-processor or if the co-processor opcode is invalid.

  Can be used for forward compatibility. For example, if the next iteration of the CPU included a floating point co-processor programs written for that co-processor would trap on the older iteration of the CPU and the floating point operation could be emulated in software. Note: There is no invalid opcode detection for CPU instructions outside of co-processor calls This is just to keep things simple. There is a risk of people using undocumented instructions and those programs breaking in future iterations of the CPU but it is expected that the core of the CPU will remain stable and future ISA improvements will be done via co-processors.

- **Privilege Violation Fault:** Raised when not in system mode and a privileged operation is performed:

  1. Writing to the high word of any address registers
  2. Writing to the high byte of the SR register
  3. Triggering exception below 0x80

- **Instruction Trace Fault:** Raised after every instruction when the `TraceMode` SR bit is set

  Used for debugging

- **Level Five Hardware Exception Conflict:** Raised when a level five hardware exception is raised when one is already being handled

  We don't use a stack for handling exceptions, so there is nowhere to store the return address past level five. We could just ignore any level five HW exceptions while it is masked, but that could indicate a hardware misconfiguration, so it is handy so that hardware bugs for things that should not be interrupted are picked up. Level 5 interrupts should be treated like NMIs.

- **Double Fault:** Raised when a fault occurs while another fault is already being handled

  When the CPU is already handling a fault (priority 7) and another fault occurs, there is no additional banked link register available to store the return address. In this situation, a double fault is raised instead. The double fault mechanism overwrites the fault link register (register 6) with the new return address, meaning there is no way to return to the original program unless the original fault handler stored the return address elsewhere.

  The fault metadata register (register 7) is also overwritten with information about the double fault. The metadata includes a double fault flag and stores both the current fault type (which will be `DoubleFault`) and the original fault type that triggered the double fault.

  The CPU jumps to vector 0x06 (the double fault vector). This is typically used as a last-chance handler to dump registers and reset the CPU. A robust operating system might save the original return addresses in every fault handler to enable recovery from double faults.

  If a fault occurs while handling a double fault, it will trigger another double fault, continuously overwriting the link registers in an infinite loop. The double fault vector should therefore be implemented with extreme care to avoid any operations that could trigger additional faults.

4.2.2 Hardware Exceptions (Interrupts)

A hardware exception occurs when interrupt pins are signalled by external hardware. These are also referred to as "interrupts". When a hardware exception occurs the current instruction is completed, and control is transferred to different vectors based on the interrupt "priority". The vectors for abort exceptions are in the 0x10-0x50 range.

4.2.3 User Exceptions (Traps)

A user exception occurs when the program executes a an EXCP instruction that signals the exception coprocessor that it should trigger an exception. They are usually used by programs running in protected mode to switch to supervisor mode in a controlled way. They are also referred to as "traps". The vectors for user exceptions are in the 0x60-0xFF range.

4.3 Exception Registers

The SIRC-1 exception coprocessor maintains several special-purpose registers for exception handling.

4.3.1 Link Registers

The exception unit uses banked link registers to store the processor state when an exception occurs. There are eight link registers in total, indexed 0-7:

- Register 0: Software exceptions (traps)
- Registers 1-5: Hardware exceptions (interrupts), one for each priority level
- Register 6: Faults (abort exceptions)
- Register 7: Fault metadata (stores bus address and execution phase information)

Each link register stores two pieces of information:

- **Return Address:** The 32-bit address to return to when the exception handler completes
- **Return Status Register:** The 16-bit status register value at the time of the exception

When an exception occurs, the appropriate link register (determined by the exception priority level) is populated with the current program counter and status register values. For abort exceptions (faults), the return address is the address of the faulting instruction so it can be retried. For regular exceptions (hardware and software exceptions), the return address is the address after the current instruction.

Link Register Preservation

Important: Link registers can be overwritten when fault occurs during fault handling. A overwrites both the fault link register (register 6) and the fault metadata register (register 7). This means the original return address is lost.

A valid way of handling this is just to declare the CPU state as invalid, do some last chance error handling (e.g. dumping the registers) and reset.

For robust fault handling that can recover from double faults, fault handlers should save the link register contents early in the handler:

```
:fault_handler
    ; Immediately save link register to memory before doing anything
    ; that might trigger another fault
    ETFR #6                    ; Save level 6 exception registers to a
        and r7
    STOR (s)-, r7
    STOR (s)-, ah
    STOR (s)-, al
    ETFR #7                    ; Save level 7 (metadata) exception
        registers to a and r7
    STOR (s)-, r7
    STOR (s)-, ah
    STOR (s)-, al

    ; Now safe to perform operations that might fault
    ; ...

    ; Restore and return
    LOAD al, +(s),
    LOAD ah, +(s)
    LOAD r7, +(s)
    ETTR #7                   ; Restore level 6 exception registers from
        a and r7
    LOAD al, +(s)
    LOAD ah, +(s)
    LOAD r7, +(s)
    ETTR #6                   ; Restore level 7 (metadata) exception
        registers from a and r7

     ; Now save to use RETE
     RETE
```

However, if the stack pointer is invalid in this case, the double fault will still cause the CPU state to be corrupted. You could provide an even more robust implementation by storing/loading from hardcoded memory addresses that are known to always be valid.

Fault Metadata Register Format

The fault metadata register (register 7) stores additional information about faults beyond just the return address. The return_status_register field is used to encode fault-specific metadata in the following format:

- **Bits 0-2:** Execution phase when the fault occurred (3 bits)
  - 0x0: Instruction fetch
  - 0x3: Effective address calculation
  - Other values indicate other pipeline phases
- **Bit 3:** Double fault flag (1 bit) - set to 1 if this is a double fault
- **Bits 4-7:** Current fault type (4 bits) - the type of fault that occurred

- **Bits 8-11:** Original fault type (4 bits) - only meaningful for double faults, indicates the fault type that was being handled when the double fault occurred
- **Bits 12-15:** Unused (reserved for future use)

The return_address field of register 7 stores the bus address that was being accessed when the fault occurred (for faults that involve memory access). For faults like invalid opcode that occur before bus access, this field will be zero.

Exception handlers can use the `ETFR` instruction to read the fault metadata register and decode this information to determine how to handle the fault appropriately.

4.3.2 Cause Register

The cause register is a 16-bit internal register used to communicate between different CPU units and coordinate exception handling. It has the following structure:

- **Bits 15-12 (First Nibble):** Coprocessor ID (0x1 for exception unit, 0x0 for processing unit)
- **Bits 11-8 (Second Nibble):** Coprocessor opcode (the operation to execute)
- **Bits 7-0 (Third and Fourth Nibbles):** Vector or parameter value

The cause register is automatically populated by the exception unit when a pending exception needs to be serviced. The priority of exceptions is determined by examining the first nibble of the vector ID, calculated as 7 minus the value of the first nibble (e.g., 0x00 is priority 7, 0x40 is priority 3, 0x60 and above are all priority 1).

4.3.3 Interrupt Mask

The status register contains a 3-bit interrupt mask field (bits 9-11 in the privileged byte) that determines which exception priority levels are currently enabled. The mask value ranges from 0-7:

- A mask value of 0 means all exceptions are enabled
- A mask value of 7 means only faults (priority 7) can interrupt execution
- When servicing an exception at level N, the mask is automatically set to N to prevent lower-priority exceptions from interrupting

Level 5 hardware exceptions are special - they are treated like non-maskable interrupts (NMIs). If a level 5 exception is raised while another level 5 exception is being handled, a Level Five Hardware Exception Conflict fault is raised instead.

4.4 Exception Processing

4.4.1 Exception Priority

Exceptions are prioritized as follows (highest to lowest):

1. Priority 7: Faults (abort exceptions)

2. Priority 6: Level 5 hardware exceptions (NMI-like)
3. Priority 5: Level 4 hardware exceptions
4. Priority 4: Level 3 hardware exceptions
5. Priority 3: Level 2 hardware exceptions
6. Priority 2: Level 1 hardware exceptions
7. Priority 1: Software exceptions (traps)

4.4.2 Exception Handling Flow

When an exception occurs, the following steps are executed:

1. The exception unit checks if the exception priority is higher than the current interrupt mask. If not, the exception is ignored.
2. For level 5 hardware exceptions being raised when one is already being handled, a Level Five Hardware Exception Conflict fault is raised instead.
3. The current program counter and status register are stored in the appropriate link register based on the exception level.
4. The Protected Mode bit in the status register is cleared, switching the CPU to supervisor mode.
5. The interrupt mask is set to the exception level being serviced to prevent lower-priority interrupts.
6. The vector address is computed by multiplying the vector ID by 2 (since vectors are 32-bit addresses stored as two 16-bit words).
7. The program counter is set to the vector address, transferring control to the exception handler.

4.4.3 Vector Table

The exception vectors are stored starting at address 0x0. Each vector is a 32-bit address stored as two consecutive 16-bit words. The vector table layout is:

- **0x00:** Reset vector
- **0x01:** Bus fault
- **0x02:** Alignment fault
- **0x03:** Segment overflow fault
- **0x04:** Invalid opcode fault
- **0x05:** Privilege violation fault
- **0x06:** Double fault

- **0x07:** Reserved
- **0x08-0x0F:** Reserved for future privileged exceptions
- **0x10:** Level 5 hardware exception vector
- **0x20:** Level 4 hardware exception vector
- **0x30:** Level 3 hardware exception vector
- **0x40:** Level 2 hardware exception vector
- **0x50:** Level 1 hardware exception vector
- **0x60-0xFF:** User exception vectors (128 user-accessible trap vectors)

4.5 Return from Exception

To return from an exception handler, the `RETE` (Return from Exception) instruction is used. This instruction performs the following operations:

1. Reads the link register corresponding to the current interrupt mask level (minus 1, since the mask is set to the exception level)
2. Restores the status register from the link register (including the interrupt mask and protected mode bit)
3. Restores the program counter from the link register
4. Execution continues from the restored address

For abort exceptions (faults), `RETE` returns to the faulting instruction, allowing it to be retried. For regular exceptions, `RETE` returns to the instruction following the one that was executing when the exception occurred.

4.5.1 Accessing Link Registers

The exception coprocessor provides instructions to transfer data to and from the link registers:

- **ETFR (Exception Transfer From Register):** Copies the return address and/or status register from a link register to the address register and/or R7
- **ETTR (Exception Transfer To Register):** Copies the address register and/or R7 to a link register's return address and/or status register

These instructions allow exception handlers to examine or modify the saved processor state before returning, enabling features like system calls, context switching, and debuggers.

4.6 Exception Coprocessor Instructions

The exception coprocessor (coprocessor ID 0x1) provides several specialized instructions for exception handling and system control. These instructions are accessed via the coprocessor instruction format, with the exception coprocessor opcode specified in the instruction encoding.

4.6.1 EXCP - Software Exception (User Exception)

Opcode: 0x1 **Privilege:** User

Syntax: `EXCP #vector`

Triggers a software exception (trap) at the specified vector. The vector must be in the range 0x60-0xFF (user exception vectors). This instruction is used by user-mode programs to invoke system calls or request supervisor mode services.

Operation:

1. Validates that the vector is in the user range (0x60-0xFF)
2. Stores current PC and SR in link register 0 (software exception register)
3. Clears the Protected Mode bit (enters supervisor mode)
4. Sets interrupt mask to 1
5. Jumps to the handler address specified in the vector table

Example:

```
; Trigger user exception at vector 0x80
EXCP #0x80                    ; System call trap
```

4.6.2 WAIT - Wait for Exception

Opcode: 0x9 **Privilege:** Supervisor

Syntax: `WAIT`

Puts the CPU into a low-power waiting state until an exception occurs. This is useful for idle loops in operating systems or when waiting for hardware interrupts.

Operation:

1. Sets the `waiting_for_exception` flag in the exception unit
2. CPU halts instruction fetch until an exception is raised
3. When an exception occurs, normal exception processing resumes

Example:

```
main_loop:
    WAIT                      ; Sleep until interrupt
    ; ... process interrupt ...
    BRAN main_loop
```

4.6.3 RETE - Return from Exception

Opcode: 0xA **Privilege:** Supervisor

Syntax: `RETE`

Returns from an exception handler by restoring the saved processor state from the current link register.

Operation:

1. Determines which link register to use based on current interrupt mask level
2. Restores PC from link register's return address
3. Restores SR from link register's return status register
4. Resumes execution at the restored address

Note: For fault exceptions, this returns to the faulting instruction. For other exceptions, this returns to the instruction after the one that was executing when the exception occurred.

Example:

```
bus_fault_handler:
    ; ... handle bus fault ...
    RETE                        ; Return to faulting instruction
```

4.6.4 RSET - System Reset

Opcode: 0xB **Privilege:** Supervisor

Syntax: `RSET`

Performs a software reset of the processor by clearing the status register and jumping to the reset vector.

Operation:

1. Clears the status register (SR = 0x0)
2. Reads the reset vector (vector 0x00) from the vector table
3. Sets PC to the address specified in the reset vector
4. Execution begins at the reset handler

Example:

```
; Perform software reset
RSET                            ; Jump to reset vector
```

4.6.5 ETFR - Exception Transfer From Register

Opcode: 0xC **Privilege:** Supervisor

Syntax:

```
ETFR #n                         ; Transfer both to a and r7
ETFR a, #n                      ; Transfer return address to a only
ETFR r7, #n                     ; Transfer return SR to r7 only
```

Transfers data from a link register to CPU registers. The instruction encoding specifies which link register (0-7) and which components (address and/or status register) to transfer.

Parameters:

- Bits 3-0: Link register index (0-7)
- Bits 5-4: Register select

  - 0: No transfer (NOP)
  - 1: Transfer return address to address register (A)
  - 2: Transfer return status register to R7
  - 3: Transfer both address to A and status to R7

Operation:

1. Selects the specified link register (0-7)
2. Based on register select bits:
   - If bit 0 set: Copy link register's return address to address register (AH:AL)
   - If bit 1 set: Copy link register's return status register to R7

This instruction is useful for exception handlers that need to examine the saved state, such as debuggers or system call handlers that need to access parameters passed in the calling context.

Examples:

```
; Get both return address and status from fault link register
ETFR #6                          ; a = return addr, r7 = return SR

; Get only the return address to modify it
ETFR a, #6                       ; a = return addr from link reg 6
LOAD al, @after_odd_address      ; Fix alignment issue
ETTR #6, a                       ; Write corrected address back

; Get only the status register to inspect flags
ETFR r7, #7                      ; r7 = fault metadata
ANDI r7, #0x7                    ; Mask off CPU phase bits
```

4.6.6 ETTR - Exception Transfer To Register

Opcode: 0xD **Privilege:** Supervisor

Syntax:

```
ETTR #n                          ; Transfer both a and r7
ETTR #n, a                       ; Transfer a to return address only
ETTR #n, r7                      ; Transfer r7 to return SR only
```

Transfers data from CPU registers to a link register. The instruction encoding specifies which link register (0-7) and which components (address and/or status register) to transfer.

Parameters:

- Bits 3-0: Link register index (0-7)
- Bits 5-4: Register select
  - 0: No transfer (NOP)
  - 1: Transfer address register (A) to return address
  - 2: Transfer R7 to return status register
  - 3: Transfer both A to return address and R7 to status

Operation:

1. Selects the specified link register (0-7)
2. Based on register select bits:
   - If bit 0 set: Copy address register (AH:AL) to link register's return address
   - If bit 1 set: Copy R7 to link register's return status register

This instruction allows exception handlers to modify the saved state before returning.
Examples:

```
; Fix up return address after alignment fault
ETFR a, #6                      ; Get current return address
LOAD al, @after_odd_address     ; Correct the lower word
ETTR #6, a                      ; Write corrected address back

; Clear protected mode bit in saved status register
ETFR r7, #6                     ; Get current return SR
ANDI r7, #0xFEFF                ; Clear protected mode bit
ETTR #6, r7                     ; Write modified SR back

; Transfer both modified address and status
LOAD ah, $PROGRAM_SEGMENT       ; Set up new return address
LOAD al, @return_point
LOAD r7, #0x0000                ; Clear status register
ETTR #6                         ; Write both back to link reg 6
```

4.6.7 Internal Instructions

The following opcodes are used internally by the exception unit and are not directly accessible via user instructions:

- **Fault (0xE):** Automatically invoked when a fault condition is detected
- **HardwareException (0xF):** Automatically invoked when a hardware interrupt is signaled
- **None (0x0):** No operation, used when no exception is pending

These internal operations follow the standard exception handling flow described in the Exception Processing section.

Part II

Instruction Set Architecture

Chapter 5

Instruction Formats

5.1 Overview

All SIRCIS instructions are exactly 32 bits (4 bytes, or 2 words) in length. This fixed instruction size simplifies instruction fetch and decode logic, though it may result in larger code size compared to variable-length instruction sets.

The SIRC-1 supports three distinct instruction formats:

1. Immediate
2. Short Immediate (with Shift)
3. Register

Note: there are meta instructions that do not take any arguments (e.g. NOOP) but they are still assembled to by one of the three instruction formats. There is no "implied" instruction format supported by the CPU.

Each format is optimized for different types of operations and addressing modes.

5.2 Format Overview

Format	**Typical Use**
Immediate	Operations with a 16-bit constant value
Short Immediate	Operations with an 8-bit constant with optional shift
Register	Operations on multiple registers with optional shift

Table 5.1: Instruction Format Summary

5.3 Common Fields

All instruction formats share some common fields:

Opcode (6 bits) Identifies the operation to perform. With 6 bits, up to 64 distinct operations can be encoded.

Condition Flags (4 bits) Specifies under what conditions the instruction should execute. See **Chapter 8** for details.

Additional Flags (2 bits) Used in memory operations to specify address register pairs (a, p, s, l) and in ALU operations to specify how the status register is updated.

5.4 Immediate Format

5.4.1 Structure

The immediate format encodes a full 16-bit immediate value within the instruction.

Note: Due to the large 16-bit operand, shift is not supported in this format.

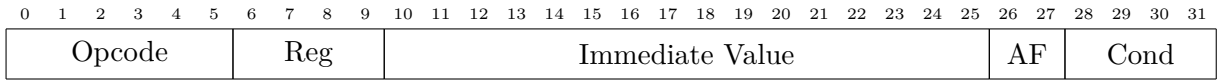

Figure 5.1: Immediate Instruction Format

Opcode (bits 31–26) 6-bit operation code

Register (bits 25–22) 4-bit destination/source register identifier

Immediate (bits 21–6) 16-bit signed or unsigned immediate value

Additional Flags (bits 5–4) 2-bit address register pair selector (for memory operations)

- 00 = l (link register)
- 01 = a (address register)
- 10 = s (stack pointer)
- 11 = p (program counter)

or 2-bit status register update source selector (for ALU operations)

- 00 = Status register not updated
- 01 = Status register updated from ALU operation
- 10 = Status register updated from shift operation
- 11 = Reserved for future use

Condition (bits 3–0) 4-bit condition code

5.4.2 Encoding Examples

Instruction	Opcode	Reg	Immediate	AF	Cond	Hex
`ADDI r1, #100`	0x00 (000000)	0x1 (0001)	0x0064 (0000 0000 0110 0100)	0b01 (01)	0x0 (0000)	`0x00420010`
`CMPI r2, #0x8000`	0x0A (001010)	0x2 (0010)	0x8000 (1000 0000 0000 0000)	0b01 (01)	0x0 (0000)	`0x2A800010`
`LOAD r3, (#16, a)`	0x14 (010100)	0x3 (0011)	0x0010 (0000 0000 0001 0000)	0b01 (01)	0x0 (0000)	`0x53040010`
`STOR (#-2, s), r4`	0x10 (010000)	0x4 (0100)	0xFFFE (1111 1111 1111 1110)	0b10 (10)	0x0 (0000)	`0x41FFF820`
`ADDI|== r5, #42`	0x00 (000000)	0x5 (0101)	0x002A (0000 0000 0010 1010)	0b01 (01)	0x1 (0001)	`0x014A0011`
`SUBI|>= r6, #10`	0x02 (000010)	0x6 (0110)	0x000A (0000 0000 0000 1010)	0b01 (01)	0xB (1011)	`0x0982801B`

Table 5.2: Immediate Format Encoding Examples

Notes:

- Condition code examples: 0x0 (0000) = Always, 0x1 (0001) = Equal (==), 0xB (1011) = Signed >=
- AF field for memory ops: 00=l, 01=a, 10=s, 11=p; for ALU ops: 00=no update, 01=update from ALU, 10=update from shift
- Negative immediates use two's complement (#-2 = 0xFFFE)

5.5 Short Immediate Format (with Shift)

5.5.1 Structure

The short immediate format sacrifices immediate value width to include shift information, allowing for more powerful single-instruction operations.

Only supported for ALU instructions - cannot be used with any of the branching, or load/store instructions.

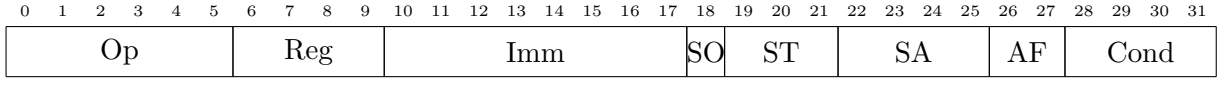

Figure 5.2: Short Immediate with Shift Format

Opcode (bits 31–26) 6-bit operation code

Register (bits 25–22) 4-bit destination/source register identifier

Immediate (bits 21–14) 8-bit signed or unsigned immediate value

Shift Operand (bit 13) Determines how the shift amount field is interpreted

- 0 = Shift amount field contains the literal shift amount
- 1 = Shift amount field refers to register that contains shift amount

Shift Type (bits 12–10) Type of shift operation (see Table 5.6 for encoding details)

Shift Amount (bits 9–6) Shift amount (or register containing shift amount)

Additional Flags (bits 5–4) 2-bit status register update source selector

- 00 = Status register not updated
- 01 = Status register updated from ALU operation
- 10 = Status register updated from shift operation
- 11 = Reserved for future use

Condition (bits 3–0) 4-bit condition code

5.5.2 Encoding Examples

Instruction	Op	Reg	Imm	SO	ST	SA	AF	Cond	Hex
`ADDI r1, #2, LSL #3`	0x20 (100000)	0x1 (0001)	0x02 (00000010)	0 (0)	001 (001)	0x3 (0011)	0b01 (01)	0x0 (0000)	`0x08108C10`
`ORRI r2, #1, LSL #10`	0x25 (100101)	0x2 (0010)	0x01 (00000001)	0 (0)	001 (001)	0xA (1010)	0b01 (01)	0x0 (0000)	`0x09504A10`
`ANDI r3, #7, ASR #2`	0x24 (100100)	0x3 (0011)	0x07 (00000111)	0 (0)	100 (100)	0x2 (0010)	0b01 (01)	0x0 (0000)	`0x09331C210`
`SUBI r4, #5, LSL r2`	0x22 (100010)	0x4 (0100)	0x05 (00000101)	1 (1)	001 (001)	0x2 (0010)	0b01 (01)	0x0 (0000)	`0x08A8A10`
`XORI|!= r7, #15, LSR #4`	0x26 (100110)	0x7 (0111)	0x0F (00001111)	0 (0)	010 (010)	0x4 (0100)	0b01 (01)	0x2 (0010)	`0x9F7C512`
`CMPI|NS r1, #8, RTL r3`	0x2A (101010)	0x1 (0001)	0x08 (00001000)	1 (1)	101 (101)	0x3 (0011)	0b01 (01)	0x5 (0101)	`0xA887D15`
`SHFI r3, #0, RTR #5`	0x20 (100000)	0x3 (0011)	0x00 (00000000)	0 (0)	110 (110)	0x5 (0101)	0b10 (10)	0x0 (0000)	`0x08307520`

Table 5.3: Short Immediate Format Encoding Examples

Notes:

- SO (Shift Operand): 0 = literal shift amount, 1 = register contains shift amount (see rows 4 and 6)
- ST (Shift Type): 000=none, 001=LSL, 010=LSR, 011=ASL, 100=ASR, 101=ROL, 110=ROR
- AF field for ALU: 00=no update, 01=update from ALU, 10=update from shift (see SHFI row 7)
- Condition code examples: 0x0 = Always, 0x2 = Not Equal (!=), 0x5 = Negative (<0)
- SHFI is a meta instruction using ADDI opcode (0x20) with AF=0b10

5.6 Register Format

5.6.1 Structure

The register format allows operations on up to three registers with optional shift.

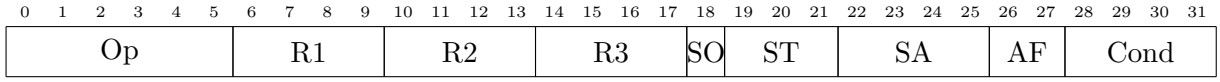

Figure 5.3: Register Instruction Format

Opcode (bits 31–26) 6-bit operation code

Register 1 (bits 25–22) 4-bit destination register

Register 2 (bits 21–18) 4-bit source A register

Register 3 (bits 17–14) 4-bit source B register

Shift Operand (bit 13) Determines how the shift amount field is interpreted

- 0 = Shift amount field contains the literal shift amount
- 1 = Shift amount field refers to register that contains shift amount

Shift Type (bits 12–10) Type of shift operation (see Table 5.6 for encoding details)

Shift Amount (bits 9–6) Shift amount (or register containing shift amount)

Additional Flags (bits 5–4) 2-bit address register pair selector (for memory operations)

- 00 = l (link register)
- 01 = a (address register)
- 10 = s (stack pointer)
- 11 = p (program counter)

or 2-bit status register update source selector (for ALU operations)

- 00 = Status register not updated
- 01 = Status register updated from ALU operation
- 10 = Status register updated from shift operation
- 11 = Reserved for future use

Condition (bits 3–0) Condition code

5.6.2 Register Operand Semantics

For most three-operand instructions:

- R1 = Destination
- R2 = First source operand

- R3 = Second source operand (optional)

Note: If R3 is not specified, the assembler with use R1 as both destination and first source operand when encoding the instruction:

```
ADDR r1, r2, r3   ; r1 = r2 + r3
ADDR r1, r2       ; r1 = r1 + r2 (R3 implicitly = R1)
```

5.6.3 Encoding Examples

Instruction	Op	R1	R2	R3	SO	ST	SA	AF	Cond	Hex
ADDR r1, r2, r3	0x30 (110000)	0x1 (0001)	0x2 (0010)	0x3 (0011)	0 (0)	000 (000)	0x0 (0000)	0b01 (01)	0x0 (0000)	0x0C123010
ADDR r1, r2, r3, LSL #2	0x30 (110000)	0x1 (0001)	0x2 (0010)	0x3 (0011)	0 (0)	001 (001)	0x2 (0010)	0b01 (01)	0x0 (0000)	0x0C1232B10
LOAD r1, (r2, a)	0x15 (010101)	0x1 (0001)	0x2 (0010)	0x0 (0000)	0 (0)	000 (000)	0x0 (0000)	0b01 (01)	0x0 (0000)	0x55200010
STOR (r2, s)+, r1	0x13 (010011)	0x1 (0001)	0x2 (0010)	0x0 (0000)	0 (0)	000 (000)	0x0 (0000)	0b10 (10)	0x0 (0000)	0x4D200020
CMPR r4, r5	0x3A (111010)	0x4 (0100)	0x4 (0100)	0x5 (0101)	0 (0)	000 (000)	0x0 (0000)	0b01 (01)	0x0 (0000)	0x0E945010
XORR r2, r3, r4, RTR #1	0x36 (110110)	0x2 (0010)	0x3 (0011)	0x4 (0100)	0 (0)	110 (110)	0x1 (0001)	0b01 (01)	0x0 (0000)	0x0DA346D10
SUBR|» r1, r2, r3, ASR r4	0x32 (110010)	0x1 (0001)	0x2 (0010)	0x3 (0011)	1 (1)	100 (100)	0x4 (0100)	0b01 (01)	0xD (1101)	0xCA372D1D
ANDR|CS r5, r6, r7, LSL r1	0x34 (110100)	0x5 (0101)	0x6 (0110)	0x7 (0111)	1 (1)	001 (001)	0x1 (0001)	0b01 (01)	0x3 (0011)	0xD567913
SHFR r1, r0, r2, ASL #3	0x30 (110000)	0x1 (0001)	0x0 (0000)	0x2 (0010)	0 (0)	011 (011)	0x3 (0011)	0b10 (10)	0x0 (0000)	0x0C102720

Table 5.4: Register Format Encoding Examples

Notes:

- When R3 is not specified in assembly, it defaults to R1 (e.g., `ADDR r4, r5` uses R1=0x4, R2=0x4, R3=0x5)
- SO (Shift Operand): 0 = literal shift amount, 1 = register contains shift amount (see rows 7 and 8)
- ST (Shift Type): 000=none, 001=LSL, 010=LSR, 011=ASL, 100=ASR, 101=ROL, 110=ROR
- AF for memory ops: 00=l, 01=a, 10=s, 11=p; for ALU ops: 00=no update, 01=update from ALU, 10=update from shift (see SHFR row 9)
- Condition code examples: 0x0 = Always, 0xD = Signed > (»), 0x3 = Carry Set (CS)
- SHFR is a meta instruction using ADDR opcode (0x30) with AF=0b10

5.7 Operand Encoding Details

5.7.1 Condition Codes

Code	Mnemonic	Condition
0000	(none)	Always (unconditional)
0001	==	Equal (Z = 1)
0010	!=	Not Equal (Z = 0)
0011	CS	Carry Set (C = 1)
0100	CC	Carry Clear (C = 0)
0101	NS	Negative Set (N = 1)
0110	NC	Negative Clear (N = 0)
0111	OS	Overflow Set (V = 1)
1000	OC	Overflow Clear (V = 0)
1001	HI	Unsigned Higher (C = 1 AND Z = 0)
1010	LO	Unsigned Lower or Same (C = 0 OR Z = 1)
1011	>=	Signed Greater or Equal (N = V)
1100	<	Signed Less Than (N ≠ V)
1101	»	Signed Greater Than (Z = 0 AND N = V)
1110	«	Signed Less or Equal (Z = 1 OR N ≠ V)
1111	NV	Never (never executes)

Table 5.5: Condition Code Encoding

For more details on condition codes see **Chapter 8**

5.7.2 Shift Types

Code	Type	Mnemonic
000	None (no shift)	–
001	Logical Shift Left	LSL
010	Logical Shift Right	LSR
011	Arithmetic Shift Left	ASL
100	Arithmetic Shift Right	ASR
101	Rotate Left	RTL
110	Rotate Right	RTR
111	Reserved	–

Table 5.6: Shift Type Encoding

For more details on shift operations see **Chapter 7**

5.8 Opcode Encoding Patterns

The SIRCIS instruction set uses systematic opcode patterns to simplify decoding:

5.8.1 ALU Instructions

- **0x0_**: Immediate operand
- **0x1_**: Memory operations
- **0x2_**: Short immediate with shift
- **0x3_**: Register operand

5.8.2 Test vs. Save

All ALU operations have both a "save result" and "test only" variant:

- **0x_0-0x_7**: Save result to destination
- **0x_8-0x_F**: Test only (update flags, don't save result)

To turn a "save result" instruction to a "test only" instruction, you can simply add 0x8 to the opcode.

For example:

- `0x04`: **`ANDI`** – AND immediate, save result
- `0x0C`: **`TSAI`** – Test AND immediate (don't save result)

5.9 Instruction Fetch and Alignment

5.9.1 Fetch Process

Since instructions are 32 bits and the data bus is 16 bits wide, each instruction requires two fetch cycles:

1. Fetch high word (bits 31–16) from address in PC
2. Increment PC by 1
3. Fetch low word (bits 15–0) from address in PC
4. Increment PC by 1

5.9.2 Alignment Requirements

Instructions span two consecutive words and must begin at an even word address. The program counter (`pl`) must always be even.

Attempting to fetch an instruction from an odd word address will trigger an alignment fault exception.

5.10 Format Selection Guidelines

When writing assembly code, the assembler automatically selects the appropriate format based on operands:

Assembly	**Format Used**	**Notes**
`ADDI r1, #1000`	Immediate	Full 16-bit immediate
`ADDI r1, #10, LSL #2`	Short Immediate	8-bit immediate + shift
`ADDR r1, r2, r3`	Register	Three register operands

Table 5.7: Format Selection Examples

The choice of format affects instruction encoding but is generally transparent to the programmer, as the assembler handles the details.

Chapter 6

Addressing Modes

6.1 Overview

The SIRC-1 CPU supports seven addressing modes that specify how operands are accessed. These modes determine whether an operand is an immediate value, register contents, or data in memory at a computed address.

Mode	Description
Immediate	Operand is a constant value encoded in the instruction
Register Direct	Operand is the contents of a specified register
Indirect Immediate	Memory address = address register + immediate offset
Indirect Register	Memory address = address register + register offset
Post-Increment	Indirect register, then increment address register
Pre-Decrement	Decrement address register, then use as indirect
Short Immediate	8-bit constant with optional shift

Table 6.1: Addressing Modes Summary

Note: There is also an "Implied" addressing mode that is not explicitly supported by the CPU itself, but it is included as many of the assembler meta instructions technically use it.

6.2 Immediate Addressing

6.2.1 Description

The operand is a constant value 16-bit value encoded directly in the instruction.

6.2.2 Syntax

```
ADDI r1, #100          ; 16-bit immediate
ADDI r2, #-50          ; Signed 16-bit immediate
```

6.2.3 Effective Value

16-bit value encoded into the instruction.

6.2.4 Usage

- Loading constant values
- Arithmetic with known values
- Bit manipulation with masks

6.3 Register Direct Addressing

6.3.1 Description

The operand is the contents of a specified register. This is the most common mode for ALU operations in a load/store architecture.

6.3.2 Syntax

```
ADDR r1, r2, r3    ; r1 = r2 + r3
ADDR r4, r5        ; r4 = r4 + r5
```

6.3.3 Effective Value

The 16-bit value currently stored in the specified register.

6.3.4 Usage

- All ALU operations between registers
- Register-to-register data movement
- Testing register values

6.4 Indirect Immediate Addressing

6.4.1 Description

The effective address is computed by adding a 16-bit signed immediate offset to an address register pair. The operand is then loaded from or stored to this computed address.

6.4.2 Syntax

```
LOAD r1, (#8, a)        ; Load from address (a + 8)
STOR (#-4, s), r2       ; Store to address (s - 4)
LDEA p, (#100, p)       ; Jump relative (pc = pc + 100)
```

6.4.3 Effective Address

$$\text{EA} = \text{AddressRegister} + \text{Offset} \tag{6.1}$$

The address register can be: `l` (link), `a` (address), `s` (stack), or `p` (program counter).

6.4.4 Usage

- Accessing structure fields (fixed offset from base)
- Array element access (if index is constant)
- Stack frame access (local variables)
- Position-independent code (PC-relative addressing)

6.5 Indirect Register Addressing

6.5.1 Description

Similar to indirect immediate, but the offset is taken from a register instead of being encoded as an immediate value. This enables runtime-computed offsets.

6.5.2 Syntax

```
LOAD r1, (r2, a)        ; Load from address (a + r2)
STOR (r4, s), r3        ; Store to address (s + r4)
```

6.5.3 Effective Address

$$\text{EA} = \text{AddressRegister} + \text{IndexRegister} \tag{6.2}$$

6.5.4 Usage

- Array indexing with variable index
- Table lookups
- Dynamic offset calculations

6.6 Post-Increment Addressing

6.6.1 Description

The effective address is computed using indirect register addressing, then the address register pair is incremented after the memory operation.

This mode is particularly useful for iterating through arrays or implementing stack operations.

6.6.2 Syntax

```
1 LOAD r1, (r2, s)+      ; Load from (s + r2), then s = s + 1
2 LOAD r1, (#2, s)+      ; Load from (s + 2), then s = s + 1
```

6.6.3 Operation Sequence

1. Compute EA = AddressRegister + Offset
2. Perform memory operation (load)
3. AddressRegister = AddressRegister + 1

6.6.4 Usage

- Stack pop operations
- Forward array traversal
- Buffer reading

6.7 Pre-Decrement Addressing

6.7.1 Description

The address register pair is decremented before computing the effective address for the memory operation.

This mode is the complement of post-increment and is useful for stack push operations.

6.7.2 Syntax

```
1 STOR -(r2, s), r1      ; s = s - 1, then store to (s + r2)
2 STOR -(#2, s), r3      ; s = s - 1, then store to (s + 2)
```

6.7.3 Operation Sequence

1. AddressRegister = AddressRegister - 1
2. Compute EA = AddressRegister + Offset
3. Perform memory operation (store)

6.7.4 Usage

- Stack push operations
- Backward array traversal
- Buffer writing

6.8 Short Immediate Addressing

6.8.1 Description

Similar to immediate addressing, but limited to 8-bit values so shift information can fit into the instruction encoding.

6.8.2 Syntax

```
1 ADDI r1, #10, LSL #2   ; r1 = (r1 << 2) + 10
2 SUBI r2, #1, LSL #8    ; r2 = (r2 << 8) - 1
```

6.8.3 Effective Value

8-bit value encoded into the instruction.

The shift is applied to the source operand, not the constant.

6.8.4 Usage

- Loading constant values
- Arithmetic with known values
- Bit manipulation with masks

6.9 Implied Addressing

6.9.1 Description

No operand is specified; the operation is fully defined by the mnemonic.

This is not an addressing mode supported by the CPU hardware, there is no encoding for an implied instruction. However, it is supported by the assembler for meta instructions that assemble to other instructions that do have operands.

6.9.2 Usage

- Meta-instructions that assemble to instructions with operands

6.9.3 Example

```
NOOP                ; No operation
; Assembles to 'ADDI r0, #0' with status register update set to 0x0 (or
    0x0000_0000)

WAIT                ; Wait for interrupt
; Assembles to 'COPI r1, #0x1900'

RETS                ; Return from subroutine
; Assembles to 'LDEA p, (l)
```

6.10 Address Register Selection

Many addressing modes require specifying which address register pair to use. The 2-bit address register field encodes:

Bits	Symbol	Register Pair
00	l	Link Register (lh, ll)
01	a	Address Register (ah, al)
10	s	Stack Pointer (sh, sl)
11	p	Program Counter (ph, pl)

Table 6.2: Address Register Encoding

6.11 Addressing Mode Examples

6.11.1 Structure Access

```
; Assume 'a' points to a struct:
; struct Point {
;   int16_t x;  // offset 0
;   int16_t y;  // offset 2
; }

LOAD r1, (#0, a)       ; r1 = point.x
LOAD r2, (#2, a)       ; r2 = point.y
```

6.11.2 Array Iteration

```
; Iterate array of 16-bit values
; Assume 'a' points to array start
; r7 = counter
loop:
  LOAD r1, (#0, a)+    ; Load element, advance by 1
  ; ... process r1 ...
```

```
7   SUBI r7, #1
8   BRAN|!= loop
```

6.11.3 Stack Frame

```
1  :begin_subroutine
2
3  ; Function prologue - save registers
4  STOR -(#2, s), r1
5  STOR -(#2, s), r2
6  STOR -(#2, s), r3
7
8  ; Access parameters (above saved regs)
9  LOAD r4, (#6, s)       ; First parameter
10 LOAD r5, (#8, s)       ; Second parameter
11
12 ; Function epilogue - restore registers
13 LOAD r3, (#0, s)+
14 LOAD r2, (#0, s)+
15 LOAD r1, (#0, s)+
16 RETS
```

6.12 Addressing Mode Restrictions

6.12.1 Privilege Mode Restrictions

In protected mode, indirect addressing modes that would write to the high byte of an address register are restricted:

- **LDEA with both registers**: Privileged in protected mode
- **LDEA with low register only**: Allowed (can't change segment)
- **Post-increment/Pre-decrement**: Can only modify low register in protected mode

6.12.2 Alignment

- All memory addresses refer to 16-bit words
- There are no alignment restrictions for general memory I/O
- All instructions span 2 words and must begin at an even word address
- Unaligned instruction fetches (odd word address) trigger an alignment fault exception

Chapter 7

Shift Operations

7.1 Overview

The SIRC-1 CPU supports powerful shift operations that can be applied to operands in many instructions. Shifts can multiply or divide values by powers of 2, extract bit fields, or perform bit rotations.

Shift operations are encoded within Short Immediate and Register format instructions, allowing a value to be shifted before being used in an ALU operation—all within a single instruction.

Note: By default, when shifts are used with ALU instructions (apart from the SHFT meta instruction), the status register flags are updated based on the ALU operation result, not the shift. However, you can manually control this behavior using the status override syntax (see Section 7.12).

7.2 Shift Types

7.2.1 Shift Type Encoding

Code	Mnemonic	Type	Description
000	–	None	No shift performed
001	LSL	Logical Shift Left	Shift left, fill with zeros
010	LSR	Logical Shift Right	Shift right, fill with zeros
011	ASL	Arithmetic Shift Left	Same as LSL (shifts in zeros)
100	ASR	Arithmetic Shift Right	Shift right, preserving sign bit
101	RTL	Rotate Left	Rotate left through carry
110	RTR	Rotate Right	Rotate right through carry
111	–	Reserved	Reserved for future use

Table 7.1: Shift Type Encoding

7.3 Shift Syntax

A shift postamble can come after any short immediate or register instruction. It is applied in the "Decode and Register Fetch" phase of instruction execution, before any ALU operations or memory address calculation occurs.

It is applied to the first source operand of an instruction. If using the three operand form of instruction, it will be the second operand. If using the two operand form of instruction, the second operand is inferred to be the same as the destination.

See this table for some examples of which operand is shifted:

Instruction	How the CPU Interprets It	First Source Operand
ADDR r1, r2, r3, LSL #1	ADDR r1, r2, r3, LSL #1	r2
ADDR r1, r2, LSL #1	ADDR r1, r1, r2, LSL #1	r1
ADDI r1, #2, LSL #1	ADDI r1, r1, #2, LSL #1	r1

Table 7.2: Examples of first source operands

7.4 Shift Type Details

7.4.1 Logical Shift Left (LSL)

Operation: Shift bits to the left, filling vacated bits with zeros.

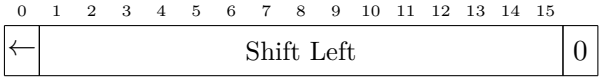

Figure 7.1: Logical Shift Left

- Bit 15 is shifted into the carry flag
- Bit 0 receives 0
- Equivalent to multiplying by 2^n where n is the shift count

Example:

```
; Multiply r1 by 4
ADDI r1, #0, LSL #2   ; r1 = (r1 << 2) = r1 * 4

% TODO: This is broken
; Set bit 10
ORRI r3, #1, LSL #10      ; r3 |= (r3 << 10) with immediate 1 = 0x0400
```

7.4.2 Logical Shift Right (LSR)

Operation: Shift bits to the right, filling vacated bits with zeros.

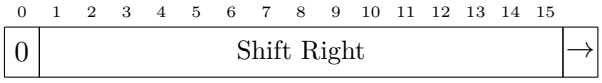

Figure 7.2: Logical Shift Right

- Bit 0 is shifted into the carry flag
- Bit 15 receives 0
- Equivalent to unsigned division by 2^n
- Use for unsigned values only

Example:

```
; Divide unsigned value by 8
ADDI r1, #0, LSR #3   ; r1 = (r1 >> 3) = r1 / 8 (unsigned)

; Extract upper byte
ADDI r3, #0, LSR #8   ; r3 = (r3 >> 8)
```

7.4.3 Arithmetic Shift Left (ASL)

Operation: Identical to LSL. Shift bits to the left, filling with zeros.

- Equivalent to signed multiplication by 2^n
- Can cause signed overflow if sign bit changes
- Overflow flag is set if sign bit changes during shift

Example:

```
; Multiply signed value by 2
ADDI r1, #0, ASL #1   ; r1 = (r1 << 1) = r1 * 2 (signed)
```

7.4.4 Arithmetic Shift Right (ASR)

Operation: Shift bits to the right, preserving the sign bit (bit 15).

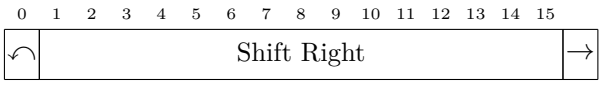

Figure 7.3: Arithmetic Shift Right

- Bit 0 is shifted into the carry flag
- Bit 15 (sign bit) is copied into bit 15 and bit 14
- Equivalent to signed division by 2^n (rounds toward negative infinity)
- Use for signed values only

Example:

```
; Divide signed value by 4
ADDI r1, #0, ASR #2   ; r1 = (r1 >> 2) = r1 / 4 (signed)

; Average two signed values
ADDR r3, r4, r5
ADDI r3, #0, ASR #1   ; r3 = (r3 >> 1) = (r4 + r5) / 2
```

7.4.5 Rotate Left (RTL)

Operation: Rotate bits to the left through the carry flag.

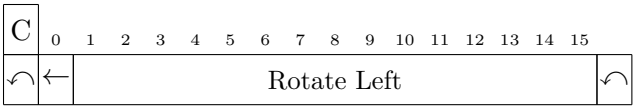

Figure 7.4: Rotate Left

- Bit 15 moves into the carry flag
- Previous carry flag value moves into bit 0
- No data is lost
- Useful for multi-word shifts and bit manipulation

Example:

```
; Rotate r1 left by 1
ADDI r1, #0, ROL #1

; Multi-word left shift (32-bit value in r1:r2)
ADDI r2, #0, LSL #1   ; Shift low word, bit 15 -> carry
ADDI r1, #0, RTL #1   ; Shift high word, carry -> bit 0
```

7.4.6 Rotate Right (RTR)

Operation: Rotate bits to the right through the carry flag.

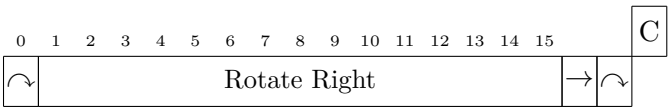

Figure 7.5: Rotate Right

- Bit 0 moves into the carry flag
- Previous carry flag value moves into bit 15
- No data is lost

Example:

```
; Rotate r1 right by 1
ADDI r1, #0, RTR #1

; Check if bit 0 is set
ADDI r2, #0, RTR #1   ; Bit 0 -> carry
; Now branch on carry to test bit 0
```

7.5 Shift Operands

The Shift Operand (SO) bit determines what value is shifted:

SO Bit	Mode	Description
0	Shift by Immediate	R2 (first source operand) is shifted by a literal amount
1	Shift Count in Register	R2 is shifted by the amount in the shift count register

Table 7.3: Shift Operand Modes

7.5.1 Mode 0: Shift by Immediate

The first source operand (R2) is shifted by a literal amount before the ALU operation.

```
; Add (r1 * 4) + 1 to r1
ADDI r1, #1, LSL #2   ; r1 = r1 + (r2 << 2) + 1
                      ; SO=0: r2 is shifted by literal 2
```

7.5.2 Mode 1: Shift by Register Count

The shift count is taken from the shift amount field (interpreted as a register), allowing variable-length shifts. R2 (first source) is still what gets shifted.

```
; Variable shift
; Shift amount register contains shift count
ADDR r1, r2, r3, LSL r4   ; r1 = (r2 << r4) + r3
                          ; SO=1: shift r2 by value in r4

; Barrel shifter emulation
ADDR r4, r5, r6, ASR r7   ; r4 = (r5 >> r7) + r6 (arithmetic)
```

7.6 Shift Count

The shift count is a 4-bit field, allowing shifts of 0–15 positions.

- Shift count of 0 means no shift is performed
- Maximum shift count is 15
- Shift counts $\geq$ 16 would be meaningless for 16-bit values

7.7 Status Flag Effects

7.7.1 Default Behavior

When shift operations are used with ALU instructions, the status flags are **by default** updated based on the **ALU operation result**, not the shift operation. For example, in `CMPI r2, #1, LSL #2`, the flags reflect the comparison result, not the shift.

To update flags based on the shift operation instead, use the `[S]` status override modifier (see Section 7.12), or use the `SHFT` meta-instruction (Chapter refch:meta-instructions).

7.7.2 Carry Flag

For all shifts and rotates:

- The last bit shifted out is placed in the carry flag
- For shifts of more than 1 position, only the final bit shifted out matters
- If shift count is 0, carry flag is not modified

7.7.3 Zero Flag

Set if the result after shifting is zero.

7.7.4 Negative Flag

Set if bit 15 of the result is 1 (result is negative when interpreted as signed).

7.7.5 Overflow Flag

- For **ASL**: Set if the sign bit changes during the shift (signed overflow)
- For other shifts: Behavior is undefined; typically unchanged or cleared

7.8 Common Use Cases

7.8.1 Multiplication by Constants

```
; Multiply by 2
ADDI r1, #0, LSL #1    ; r1 = (r1 << 1) = r1 * 2

; Multiply by 3
ADDR r1, r2, r2, LSL #1    ; r1 = (r2 << 1) + r2 = r2 * 3

; Multiply by 5
ADDR r1, r2, r2, LSL #2    ; r1 = (r2 << 2) + r2 = r2 * 5
```

7.8.2 Division by Powers of 2

```
; Unsigned divide by 4
LOAD r1, r2          ; r1 = r2
ADDI r1, #0, LSR #2  ; r1 = (r1 >> 2) = r1 / 4 (unsigned)

; Signed divide by 8
LOAD r1, r2          ; r1 = r2
ADDI r1, #0, ASR #3  ; r1 = (r1 >> 3) = r1 / 8 (signed)
```

7.8.3 Bit Field Extraction

```
; Extract bits 8-11 from r1
LOAD r2, r1          ; r2 = r1
ADDI r2, #0, LSR #8  ; Shift down
ANDI r2, #0x0F           ; Mask to 4 bits

; Extract and sign-extend bit 7
LOAD r2, r1          ; r2 = r1
ADDI r2, #0, LSL #8  ; Shift bit 7 to bit 15
ADDI r2, #0, ASR #8  ; Arithmetic shift to sign-extend
```

7.8.4 Bit Manipulation

```
; Set bit N (where N is in r2)
LOAD r3, #1              ; r3 = 1
ADDI r3, #0, LSL r2      ; r3 = (r3 << r2) = 1 << N
ORRR r1, r3              ; r1 |= (1 << N)

; Clear bit N
LOAD r3, #1
ADDI r3, #0, LSL r2      ; r3 = (r3 << r2) = 1 << N
LOAD r4, #0xFFFF
XORR r3, r4, r3          ; r3 = 0xFFFF ^ (1 << N) = ~(1 << N)
ANDR r1, r1, r3          ; r1 = r1 & ~(1 << N)

; Test bit N
LOAD r3, #1
ADDI r3, #0, LSL r2      ; r3 = (r3 << r2) = 1 << N
TSAR r1, r3              ; Test r1 & (1 << N)
BRAN|!= bit_was_set      ; Branch if bit set
```

7.9 Multi-Word Shifts

For operations on values larger than 16 bits, shifts and rotates can be combined:

7.9.1 32-bit Left Shift

```
; Shift 32-bit value in r1:r2 left by 1
; r1 = high word, r2 = low word
ADDI r2, #0, LSL #1    ; Shift low, MSB -> carry
ADDI r1, #0, RTL #1    ; Rotate high with carry
```

7.9.2 32-bit Right Shift (Unsigned)

```
; Shift 32-bit value in r1:r2 right by 1 (unsigned)
ADDI r1, #0, LSR #1    ; Shift high, LSB -> carry
ADDI r2, #0, RTR #1    ; Rotate low with carry
```

7.9.3 32-bit Right Shift (Signed)

```
; Shift 32-bit value in r1:r2 right by 1 (signed)
ADDI r1, #0, ASR #1    ; Arithmetic shift high
ADDI r2, #0, RTR #1    ; Rotate low with carry
```

7.10 Performance Considerations

- All shift operations complete in the same number of cycles regardless of shift count
- Shifts can be performed "for free" as part of another instruction (no extra cycles)
- Using shifts for multiplication/division by powers of 2 is much faster than using a multiply/-divide instruction (if one existed)
- Barrel shifter implementation in hardware allows any shift count in constant time

7.11 Limitations

- Maximum shift count is 15 (4-bit field)
- For shifts > 15, multiple shift instructions must be used
- Rotates include the carry flag, so rotating by 16 does not equal original value
- Some shift types (e.g., reserved type 111) may have undefined behavior

7.12 Status Flag Update Control

By default, ALU instructions with shift operations update the status register flags based on the **ALU operation result**, not the shift operation. However, the SIRC-1 provides manual control over which operation updates the flags.

7.12.1 Status Override Syntax

A status override modifier can be added immediately after an instruction mnemonic using square brackets:

Modifier	Meaning	Description
[A]	ALU	Update flags from ALU result (default)
[S]	Shift	Update flags from shift result, not ALU
[N]	None	Do not update status flags at all

Table 7.4: Status Flag Override Modifiers

7.12.2 Examples

Default Behavior (ALU flags):

```
; Without override, flags reflect the ALU addition result
ADDI r2, #1, LSL #2           ; r2 = (r2 << 2) + 1
                              ; Flags: based on addition result
```

Shift Flag Override:

```
; Update flags based on shift result instead of ALU
ADDI[S] r2, #1, LSL #2        ; r2 = (r2 << 2) + 1
                              ; Flags: based on (r2 << 2), not the +1

; Check if shifted value is zero
CMPI[S] r3, #0, LSR #4        ; Compare (r3 >> 4) with 0
                              ; Flags: based on shift, not comparison
```

No Flag Update:

```
; Perform operation without affecting status flags
ADDI[N] r1, #5, LSL #1        ; r1 = (r1 << 1) + 5
                              ; Flags: unchanged

; Useful when preserving flags from previous operation
CMPI r4, r5                   ; Set flags based on comparison
ADDI[N] r6, #1                ; Increment r6 without changing flags
BRAN|== equal_branch          ; Branch using flags from CMPI
```

7.12.3 Use Cases

Testing Shift Results

When you need to know properties of a shifted value without performing a subsequent comparison:

```
; Check if r1 << 3 would be negative
ADDI[S] r1, #0, LSL #3        ; r1 = r1 << 3, flags from shift
BRAN|NS overflow_detected     ; Branch if sign bit set
```

Preserving Flags

When performing operations that should not affect condition codes:

```
; Test condition
CMPI r1, r2                    ; Compare r1 and r2

; Do some work without affecting flags
ADDI[N] r3, #1                 ; Increment counter (flags unchanged)
LOAD[N] r4, (#0, a)+           ; Load next value (flags unchanged)

; Branch using original comparison flags
BRAN|>= r1_was_greater
```

SHFT Meta-Instruction

The **SHFT** meta-instruction (see Chapter 14) uses the [S] modifier internally:

```
SHFT r1, ASL #3                ; Expands to: ORRI[S] r1, #0, ASL #3
```

This provides a clean syntax for shift-only operations with flag updates.

7.12.4 Assembler Syntax Notes

- The status override modifier must appear immediately after the instruction mnemonic with no spaces: `ADDI[S]`
- The modifier is optional; omitting it defaults to [A] (ALU flag updates)
- The override applies to the entire instruction, including any conditional execution
- Explicit [A] is rarely needed but can improve code readability

Chapter 8

Condition Codes

8.1 Overview

Most SIRCIS instructions can be executed conditionally based on the state of the condition flags in the status register. This allows for efficient implementation of conditional logic without requiring explicit branch instructions.

The 4-bit condition code field in each instruction specifies under what conditions the instruction should execute. If the condition is not met, the instruction behaves as a `NOOP` (no operation).

8.2 Condition Code Encoding

Code	Mnemonic	Condition (when instruction executes)
0000	AL (or none)	Always (unconditional)
0001	==	Equal (Z = 1)
0010	!=	Not Equal (Z = 0)
0011	CS	Carry Set (C = 1)
0100	CC	Carry Clear (C = 0)
0101	NS	Negative Set (N = 1)
0110	NC	Negative Clear (N = 0)
0111	OS	Overflow Set (V = 1)
1000	OC	Overflow Clear (V = 0)
1001	HI	Unsigned Higher (C = 1 AND Z = 0)
1010	LO	Unsigned Lower or Same (C = 0 OR Z = 1)
1011	>=	Signed Greater or Equal (N = V)
1100	<	Signed Less Than (N ≠ V)
1101	»	Signed Greater Than (Z = 0 AND N = V)
1110	«	Signed Less or Equal (Z = 1 OR N ≠ V)
1111	NV	Never (never executes)

Table 8.1: Condition Code Encoding

8.3 Condition Code Categories

8.3.1 Simple Conditions

These conditions test a single status flag:

Always (default) No condition code specified; instruction always executes.

== (Equal) Executes when Z = 1. Typically used after `CMP` to test equality.

!= (Not Equal) Executes when Z = 0. Typical use: test inequality.

CS (Carry Set) Executes when C = 1. Can indicate unsigned overflow or borrow.

CC (Carry Clear) Executes when C = 0. No unsigned overflow/borrow.

NS (Negative Set) Executes when N = 1. Tests if result is negative (signed).

NC (Negative Clear) Executes when N = 0. Tests if result is positive or zero (signed).

OS (Overflow Set) Executes when V = 1. Signed overflow occurred.

OC (Overflow Clear) Executes when V = 0. No signed overflow.

8.3.2 Unsigned Comparisons

These conditions are used after comparing unsigned integers:

HI (Unsigned Higher) Executes when C = 1 AND Z = 0. Tests if first operand > second operand (unsigned).

LO (Unsigned Lower or Same) Executes when C = 0 OR Z = 1. Tests if first ≤ second (unsigned).

Note: For unsigned comparisons:

- Use `HI` and `LO` after `CMP`
- `HI` tests if first > second (unsigned)
- `LO` tests if first ≤ second (unsigned)
- `CS` is equivalent to unsigned ≥
- `CC` is equivalent to unsigned <

8.3.3 Signed Comparisons

These conditions are used after comparing signed integers:

>= (Signed Greater or Equal) Executes when N = V. First operand ≥ second operand (signed).

« (Signed Less Than) Executes when N ≠ V. First operand < second operand (signed).

» (Signed Greater Than) Executes when Z = 0 AND N = V. First operand > second operand (signed).

<= (Signed Less or Equal) Executes when Z = 1 OR N ≠ V. First operand ≤ second operand (signed).

8.3.4 Special Conditions

Never (NV) Never executes; equivalent to `NOOP`. Can be used for instruction padding or to temporarily disable instructions during debugging.

8.4 Assembly Syntax

Condition codes are specified by appending a pipe character (`|`) followed by the condition mnemonic to the instruction:

```
; Unconditional
ADDI r1, #10

; Conditional - execute only if equal
ADDI|== r2, #20

; Conditional branch
BRAN|!= loop_start

; Multiple conditions
ADDR|HI r3, r4, r5
STOR|NS (#0, s), r6
```

8.5 Usage with Compare Instructions

The most common use of condition codes is with compare (`CMP`) instructions:

8.5.1 Unsigned Comparison

```
; if (r1 > r2) { r3 = r4; } (unsigned)
CMPR r1, r2           ; Compare r1 and r2
LOAD|HI r3, r4        ; r3 = r4 if r1 > r2 (unsigned)

; if (r1 <= 100) { branch } (unsigned)
CMPI r1, #100
BRAN|LO target
```

8.5.2 Signed Comparison

```
; if (r1 >= r2) { r3 = r4; } (signed)
CMPR r1, r2           ; Compare r1 and r2
LOAD|>= r3, r4        ; r3 = r4 if r1 >= r2 (signed)

; if (r1 < -10) { branch } (signed)
CMPI r1, #-10
BRAN|<< negative_case
```

8.5.3 Equality Testing

```
; if (r1 == r2) { branch }
CMPR r1, r2
BRAN|== equal_case

; if (r1 != 0) { branch }
CMPI r1, #0
BRAN|!= non_zero_case
```

8.6 Condition Code Truth Tables

8.6.1 After Unsigned Comparison (CMPR rA, rB)

Relationship	Z	C	Condition	Code	Meaning
rA = rB	1	*	==	0001	Equal
rA ≠ rB	0	*	!=	0010	Not Equal
rA > rB (unsigned)	0	1	HI	1001	Higher
rA < rB (unsigned)	0	0	CC	0100	Lower
rA ≥ rB (unsigned)	*	1	CS	0011	Higher or Same
rA ≤ rB (unsigned)	*	*	LO	1010	Lower or Same

Table 8.2: Unsigned Comparison Results

8.6.2 After Signed Comparison (CMPR rA, rB)

Relationship	Z	N	V	Condition	Code	Meaning
rA = rB	1	*	*	==	0001	Equal
rA ≠ rB	0	*	*	!=	0010	Not Equal
rA > rB (signed)	0	0	0	»	1101	Greater
rA > rB (signed)	0	1	1	»	1101	Greater
rA < rB (signed)	*	0	1	«	1100	Less
rA < rB (signed)	*	1	0	«	1100	Less
rA ≥ rB (signed)	*	0	0	>=	1011	Greater or Equal
rA ≥ rB (signed)	*	1	1	>=	1011	Greater or Equal
rA ≤ rB (signed)	1	*	*	<=	1110	Less or Equal
rA ≤ rB (signed)	0	0	1	<=	1110	Less or Equal
rA ≤ rB (signed)	0	1	0	<=	1110	Less or Equal

Table 8.3: Signed Comparison Results

8.7 Advanced Usage

8.7.1 Conditional Assignment

```
; max = (a > b) ? a : b (unsigned)
CMPR r1, r2                ; Compare a and b
LOAD|HI r3, r1             ; r3 = a if a > b
LOAD|LO r3, r2             ; r3 = b if a <= b
```

8.7.2 Conditional Increment

```
; if (condition) count++
CMPI r1, #0
ADDI|!= r2, #1             ; Increment r2 if r1 != 0
```

8.7.3 Conditional Function Call

```
; if (r1 > 0) call_function()
CMPI r1, #0
LJSR|>> (#0, a)            ; Call function if r1 > 0 (signed)
```

8.7.4 Loop Construction

```
; for (i = 10; i > 0; i--)
ADDI r1, #0, #10           ; i = 10
loop:
    ; ... loop body ...
    SUBI r1, #1            ; i--
    BRAN|HI loop           ; Continue if i > 0 (unsigned)
```

8.8 Condition Code Selection Guide

Intent	After CMP	Condition Code
If equal	`CMPR a, b`	`==`
If not equal	`CMPR a, b`	`!=`
If a > b (unsigned)	`CMPR a, b`	`HI`
If a ≥ b (unsigned)	`CMPR a, b`	`CS`
If a < b (unsigned)	`CMPR a, b`	`CC`
If a ≤ b (unsigned)	`CMPR a, b`	`LO`
If a > b (signed)	`CMPR a, b`	`»`
If a ≥ b (signed)	`CMPR a, b`	`>=`
If a < b (signed)	`CMPR a, b`	`«`
If a ≤ b (signed)	`CMPR a, b`	`<=`
If negative	`CMPI a, #0`	`NS`
If positive or zero	`CMPI a, #0`	`NC`
If overflow occurred	After arithmetic	`OS`
If no overflow	After arithmetic	`OC`

Table 8.4: Condition Code Selection Guide

8.9 Performance Considerations

- Conditional instructions do not branch, avoiding pipeline flushes
- When a condition is false, the instruction still takes 6 cycles but performs no operation
- For short conditional sequences, conditional execution can be faster than branching
- For longer conditional blocks, explicit branches may be more efficient

8.10 Common Pitfalls

- **Signed vs. Unsigned**: Using unsigned conditions (`HI`, `LO`) for signed values or vice versa produces incorrect results
- **Flag Clobbering**: Flags are modified by most ALU instructions. Ensure the condition flags haven't been modified between the comparison and conditional instruction:

```
; WRONG - flags clobbered
CMPR r1, r2
ADDI r3, #100          ; Modifies flags!
BRAN|HI target         ; Tests wrong flags

; CORRECT - flags preserved
CMPR r1, r2
BRAN|HI target
ADDI r3, #100          ; After the branch
```

- **Inverted Logic**: Remember that `CMP A, B` computes A - B, so condition codes test A relative to B, not B relative to A

Part III

SIRCIS Instruction Reference

Chapter 9

Instruction Summary

9.1 Overview

The SIRCIS instruction set consists of 64 possible opcodes (6-bit opcode field), of which approximately 52 are documented and assigned. The instructions are organized into logical groups:

- **ALU Instructions (0x00–0x0F, 0x20–0x2F, 0x30–0x3F)**: Arithmetic, logical, and comparison operations
- **Memory Instructions (0x10–0x17)**: Load and store operations
- **Control Flow (0x18–0x1F)**: Branches, jumps, and address loading
- **Coprocessor (0x0F, 0x2F, 0x3F)**: Coprocessor interface

9.2 Complete Instruction List

Opcode	Mnemonic	Format	Flags	Operation
0x00	ADDI	Immediate	NZCV	Add immediate
0x01	ADCI	Immediate	NZCV	Add immediate with carry
0x02	SUBI	Immediate	NZCV	Subtract immediate
0x03	SBCI	Immediate	NZCV	Subtract immediate with carry
0x04	ANDI	Immediate	NZ	AND immediate
0x05	ORRI	Immediate	NZ	OR immediate
0x06	XORI	Immediate	NZ	XOR immediate
0x07	LOAD	Immediate	–	Load immediate (move)
0x08	–	–	–	Undocumented
0x09	–	–	–	Undocumented
0x0A	CMPI	Immediate	NZCV	Compare immediate
0x0B	–	–	–	Undocumented
0x0C	TSAI	Immediate	NZ	Test AND immediate
0x0D	–	–	–	Undocumented
0x0E	TSXI	Immediate	NZ	Test XOR immediate
0x0F	COPI	Immediate	–	Coprocessor call immediate
0x10	STOR	Indirect Imm	–	Store to (addr + imm)
0x11	STOR	Indirect Reg	–	Store to (addr + reg)
0x12	STOR	Pre-Dec	–	Store with pre-decrement
0x13	STOR	Pre-Dec Reg	–	Store with pre-dec (reg)
0x14	LOAD	Indirect Imm	–	Load from (addr + imm)
0x15	LOAD	Indirect Reg	–	Load from (addr + reg)
0x16	LOAD	Post-Inc	–	Load with post-increment
0x17	LOAD	Post-Inc Reg	–	Load with post-inc (reg)
0x18	LDEA	Indirect Imm	–	Load effective address
0x19	LDEA	Indirect Reg	–	Load effective address (reg)
0x1A	BRAN	Indirect Imm	–	Branch (imm displacement)
0x1B	BRAN	Indirect Reg	–	Branch (reg displacement)
0x1C	LJSR	Indirect Imm	–	Long jump to subroutine
0x1D	LJSR	Indirect Reg	–	Long jump to subroutine (reg)
0x1E	BRSR	Indirect Imm	–	Branch to subroutine
0x1F	BRSR	Indirect Reg	–	Branch to subroutine (reg)
0x20	ADDI	Short Imm+Shift	NZCV	Add short immediate
0x21	ADCI	Short Imm+Shift	NZCV	Add short imm with carry
0x22	SUBI	Short Imm+Shift	NZCV	Subtract short immediate
0x23	SBCI	Short Imm+Shift	NZCV	Subtract short imm with carry
0x24	ANDI	Short Imm+Shift	NZ	AND short immediate
0x25	ORRI	Short Imm+Shift	NZ	OR short immediate
0x26	XORI	Short Imm+Shift	NZ	XOR short immediate
0x27	LOAD	Short Imm+Shift	–	Load short immediate
0x28	–	–	–	Undocumented
0x29	–	–	–	Undocumented
0x2A	CMPI	Short Imm+Shift	NZCV	Compare short immediate
0x2B	–	–	–	Undocumented
0x2C	TSAI	Short Imm+Shift	NZ	Test AND short immediate
0x2D	–	–	–	Undocumented
0x2E	TSXI	Short Imm+Shift	NZ	Test XOR short immediate
0x2F	COPI	Short Imm+Shift	–	Coprocessor call short imm
0x30	ADDR	Register	NZCV	Add register
0x31	ADCR	Register	NZCV	Add register with carry
0x32	SUBR	Register	NZCV	Subtract register
0x33	SBCR	Register	NZCV	Subtract register with carry
0x34	ANDR	Register	NZ	AND register
0x35	ORRR	Register	NZ	OR register
0x36	XORR	Register	NZ	XOR register
0x37	LOAD	Register	–	Load from register (move)
0x38	–	–	–	Undocumented
0x39	–	–	–	Undocumented
0x3A	CMPR	Register	NZCV	Compare register
0x3B	–	–	–	Undocumented
0x3C	TSAR	Register	NZ	Test AND register
0x3D	–	–	–	Undocumented
0x3E	TSXR	Register	NZ	Test XOR register
0x3F	COPR	Register	–	Coprocessor call register

Table 9.1: Complete SIRCIS Instruction Set

9.3 Instruction Grouping Patterns

9.3.1 Opcode Organization

The opcode space is systematically organized:

- **0x0_**: Immediate operand format
- **0x1_**: Memory and control flow operations
- **0x2_**: Short immediate with shift format
- **0x3_**: Register operand format

9.3.2 Save vs. Test Variants

Many ALU instructions have two variants:

- **+0x00 to +0x07**: Save result to destination register
- **+0x08 to +0x0F**: Test only (update flags, discard result)

Examples:

- 0x04 = **ANDI** (save result), 0x0C = **TSAI** (test only)
- 0x06 = **XORI** (save result), 0x0E = **TSXI** (test only)

9.4 Meta-Instructions

Several commonly-used operations are implemented as meta-instructions (pseudo-instructions) that assemble to specific opcodes:

Meta-Instruction	Assembles To	Purpose
NOOP	ADDI r1, #0	No operation
RETS	LDEA p, (#0, l)	Return from subroutine
WAIT	COP #0x1F00	Wait for interrupt
RETE	COP #0x1A00	Return from exception

Table 9.2: Meta-Instructions

9.5 Instruction Timing

All instructions execute in exactly 6 clock cycles:

1. Instruction Fetch (High Word)
2. Instruction Fetch (Low Word)

3. Decode and Register Fetch
4. Execute/Address Calculate
5. Memory Access (or NOP)
6. Write Back

This fixed timing simplifies hardware design and makes execution time predictable.

9.6 Next Chapters

The following chapters provide detailed documentation for each instruction group:

- **Chapter 10**: ALU instructions (arithmetic, logical, compare)
- **Chapter 11**: Load and store operations
- **Chapter 12**: Branches, jumps, and address operations
- **Chapter 13**: Coprocessor interface
- **Chapter 14**: Meta-instructions and pseudo-ops

Chapter 10

ALU Instructions

10.1 Overview

The ALU (Arithmetic Logic Unit) instructions perform arithmetic and logical operations on register operands. All ALU instructions follow the load/store architecture principle: they operate only on registers, not directly on memory.

ALU instructions are available in three format variants:

- **Immediate**: Operation with 16-bit constant
- **Short Immediate with Shift**: Operation with 8-bit constant and optional shift
- **Register**: Operation between registers with optional shift

10.2 ALU Instruction Families

10.2.1 Arithmetic Instructions

- `ADD` – Addition
- `ADC` – Add with Carry
- `SUB` – Subtraction
- `SBC` – Subtract with Carry (Borrow)

10.2.2 Logical Instructions

- `AND` – Bitwise AND
- `ORR` – Bitwise OR
- `XOR` – Bitwise XOR (Exclusive OR)

10.2.3 Data Movement

- `LOAD` – Load immediate or copy register

10.2.4 Test and Compare

- `CMP` – Compare (subtract without saving result)
- `TSA` – Test AND (AND without saving result)
- `TSX` – Test XOR (XOR without saving result)

10.3 Status Flag Updates

Most ALU instructions update the status register flags:

Arithmetic (ADD, ADC, SUB, SBC, CMP) Update all four flags: N, Z, C, V

Logical (AND, ORR, XOR, TSA, TSX) Update N and Z; clear C and V

Load (LOAD) Do not update flags

10.3.1 Manual Flag Update Control

ALU instructions support manual control over how status flags are updated using the status override syntax. See Section 7.12 in Chapter 7 for complete details.

- `INSTR[A]` – Update flags from ALU result (default behavior)
- `INSTR[S]` – Update flags from shift result instead of ALU
- `INSTR[N]` – Do not update status flags

Example:

```
; Update flags from shift operation
ADDI[S] r1, #0, LSL #3        ; Flags reflect (r1 << 3), not the ALU

; Preserve flags during operation
CMPI r2, r3                   ; Set comparison flags
ADDI[N] r4, #1                ; Increment without changing flags
BRAN|>= branch_if_r2_gte_r3   ; Use flags from CMPI
```

10.4 ADD – Addition

ADDI / ADDR – Add Immediate / Add Register

Opcodes: 0x00 (Imm), 0x20 (Short Imm), 0x30 (Reg)
Syntax:

```
ADDI rD, #imm16                 ; rD = rD + imm16
ADDI rD, #imm8, shift           ; rD = (rD << shift) + imm8
ADDR rD, rS1, rS2               ; rD = rS1 + rS2
ADDR rD, rS1, rS2, shift        ; rD = (rS1 << shift) + rS2
```

Operation:

```
result = operand1 + operand2
destination = result
SR.N = result[15]
SR.Z = (result == 0)
SR.C = carry_out
SR.V = signed_overflow
```

Description:
Adds two values and stores the result in the destination register. In short immediate format with shift, the destination register is shifted before adding the immediate. The carry flag is set if an unsigned overflow occurs (carry out of bit 15). The overflow flag is set if a signed overflow occurs (the sign of the result is incorrect for the operation).
Flags: N Z C V
Example:

```
; Simple addition
ADDI r1, #100               ; r1 = r1 + 100

; Add two registers
ADDR r3, r1, r2             ; r3 = r1 + r2

; Scaled addition (multiply by 3)
ADDR r4, r5, r5, LSL #1     ; r4 = r5 + (r5 * 2) = r5 * 3

; Build large constant
ADDI r1, #1, LSL #12        ; r1 = (r1 << 12) + 1
```

Notes:

- For multi-precision arithmetic, use **ADC** to propagate carry
- Adding zero (`ADDI r1, #0`) can be used to move values or test flags
- Shift operations allow efficient constant multiplication

10.5 ADC – Add with Carry

ADCI / ADCR – Add with Carry

Opcodes: 0x01 (Imm), 0x21 (Short Imm), 0x31 (Reg)
Syntax:

```
ADCI rD, #imm16                 ; rD = rD + imm16 + C
ADCI rD, #imm8, shift           ; rD = (rD << shift) + imm8 + C
ADCR rD, rS1, rS2               ; rD = rS1 + rS2 + C
ADCR rD, rS1, rS2, shift        ; rD = (rS1 << shift) + rS2 + C
```

Operation:

```
result = operand1 + operand2 + SR.C
destination = result
SR.N = result[15]
SR.Z = (result == 0)
SR.C = carry_out
SR.V = signed_overflow
```

Description:
Adds two values plus the carry flag and stores the result. In short immediate format with shift, the destination register is shifted before adding the immediate. This instruction is used for multi-precision (32-bit, 48-bit, etc.) arithmetic to propagate the carry from lower-order additions to higher-order additions.
Flags: N Z C V
Example:

```
; 32-bit addition (r1:r2 = r3:r4 + r5:r6)
; Low words
ADDR r2, r4, r6             ; r2 = r4 + r6, set carry

; High words (with carry propagation)
ADCR r1, r3, r5             ; r1 = r3 + r5 + carry
```

Notes:

- Always use after **ADD** for multi-word arithmetic
- The carry flag must be in the correct state before **ADC**
- Can be used to add 1 conditionally based on carry flag

10.6 SUB – Subtraction

SUBI / SUBR – Subtract Immediate / Subtract Register

Opcodes: 0x02 (Imm), 0x22 (Short Imm), 0x32 (Reg)
Syntax:

```
SUBI rD, #imm16                 ; rD = rD - imm16
SUBI rD, #imm8, shift           ; rD = (rD << shift) - imm8
SUBR rD, rS1, rS2               ; rD = rS1 - rS2
SUBR rD, rS1, rS2, shift        ; rD = (rS1 << shift) - rS2
```

Operation:

```
result = operand1 - operand2
destination = result
SR.N = result[15]
SR.Z = (result == 0)
SR.C = NOT(borrow)  ; Set if no borrow
SR.V = signed_overflow
```

Description:
Subtracts the second operand from the first and stores the result. In short immediate format with shift, the destination register is shifted before subtracting the immediate. The carry flag is cleared if a borrow occurs (operand2 > operand1 for unsigned). The overflow flag indicates signed overflow.
Flags: N Z C V
Example:

```
; Decrement by 1
SUBI r1, #1                 ; r1 = r1 - 1

; Subtract registers
SUBR r3, r1, r2             ; r3 = r1 - r2

; Loop counter
SUBI r7, #1                 ; Decrement counter
BRAN|!= loop                ; Branch if not zero
```

Notes:

- Carry flag behavior is inverted compared to ADD (set when no borrow)
- For multi-precision subtraction, use `SBC`
- Subtracting a value from itself (`SUBR r1, r1, r1`) clears the register

10.7 SBC – Subtract with Carry (Borrow)

SBCI / SBCR – Subtract with Carry

Opcodes: 0x03 (Imm), 0x23 (Short Imm), 0x33 (Reg)
Syntax:

```
SBCI rD, #imm16                 ; rD = rD - imm16 - !C
SBCR rD, rS1, rS2               ; rD = rS1 - rS2 - !C
SBCR rD, rS1, rS2, shift        ; rD = (rS1 << shift) - rS2 - !C
```

Operation:

```
result = operand1 - operand2 - (1 - SR.C)
destination = result
SR.N = result[15]
SR.Z = (result == 0)
SR.C = NOT(borrow)
SR.V = signed_overflow
```

Description:
Subtracts the second operand and borrow (inverse of carry) from the first operand. In short immediate format with shift, the destination register is shifted before subtracting the immediate. Used for multi-precision subtraction to propagate borrows from lower-order to higher-order subtractions.
Flags: N Z C V
Example:

```
; 32-bit subtraction (r1:r2 = r3:r4 - r5:r6)
; Low words
SUBR r2, r4, r6             ; r2 = r4 - r6, set/clear carry

; High words (with borrow propagation)
SBCR r1, r3, r5             ; r1 = r3 - r5 - borrow
```

Notes:

- Always use after **SUB** for multi-word subtraction
- Remember: carry flag is inverted for subtraction (C=1 means no borrow)

10.8 AND – Bitwise AND

ANDI / ANDR – AND Immediate / AND Register

Opcodes: 0x04 (Imm), 0x24 (Short Imm), 0x34 (Reg)
Syntax:

```
ANDI rD, #imm16                 ; rD = rD & imm16
ANDI rD, #imm8, shift           ; rD = (rD << shift) & imm8
ANDR rD, rS1, rS2               ; rD = rS1 & rS2
ANDR rD, rS1, rS2, shift        ; rD = (rS1 << shift) & rS2
```

Operation:

```
result = operand1 AND operand2
destination = result
SR.N = result[15]
SR.Z = (result == 0)
SR.C = 0
SR.V = 0
```

Description:
Performs bitwise AND operation. Each bit in the result is 1 only if the corresponding bits in both operands are 1. In short immediate format with shift, the destination register is shifted before ANDing with the immediate. Commonly used for bit masking and clearing specific bits.
Flags: N Z (C and V cleared)
Example:

```
; Mask lower byte
ANDI r1, #0x00FF            ; r1 = r1 & 0xFF

; Clear bit 5
ANDI r2, #~(1<<5)           ; r2 = r2 & 0xFFDF

; Check if two registers share any set bits
ANDR r3, r1, r2             ; r3 = r1 & r2
CMPI r3, #0
BRAN|!= common_bits         ; Branch if any bits in common
```

Notes:

- ANDing with 0xFFFF leaves value unchanged
- ANDing with 0x0000 clears the register
- Use **TSA** to test bits without modifying the register

10.9 ORR – Bitwise OR

ORRI / ORRR – OR Immediate / OR Register

Opcodes: 0x05 (Imm), 0x25 (Short Imm), 0x35 (Reg)
Syntax:

```
ORRI rD, #imm16                 ; rD = rD | imm16
ORRI rD, #imm8, shift           ; rD = (rD << shift) | imm8
ORRR rD, rS1, rS2               ; rD = rS1 | rS2
ORRR rD, rS1, rS2, shift        ; rD = (rS1 << shift) | rS2
```

Operation:

```
result = operand1 OR operand2
destination = result
SR.N = result[15]
SR.Z = (result == 0)
SR.C = 0
SR.V = 0
```

Description:
Performs bitwise OR operation. Each bit in the result is 1 if either corresponding bit in the operands is 1. In short immediate format with shift, the destination register is shifted before ORing with the immediate. Commonly used for setting specific bits.
Flags: N Z (C and V cleared)
Example:

```
; Set bit 7
ORRI r1, #0x0080            ; r1 = r1 | 0x80

; Set multiple bits
ORRI r2, #0xF000            ; Set upper nibble

; Combine two registers
ORRR r3, r1, r2             ; r3 = r1 | r2

; Set bit N (variable)
ADDI r4, #1
ADDR r4, r4, r5, LSL        ; r4 = 1 << r5
ORRR r1, r4                 ; r1 |= (1 << r5)
```

Notes:

- ORing with 0x0000 leaves value unchanged
- ORing with 0xFFFF sets all bits
- Use for enabling flag bits or combining bit fields

10.10 XOR – Bitwise Exclusive OR

XORI / XORR – XOR Immediate / XOR Register

Opcodes: 0x06 (Imm), 0x26 (Short Imm), 0x36 (Reg)
Syntax:

```
XORI rD, #imm16                 ; rD = rD ^ imm16
XORI rD, #imm8, shift           ; rD = (rD << shift) ^ imm8
XORR rD, rS1, rS2               ; rD = rS1 ^ rS2
XORR rD, rS1, rS2, shift        ; rD = (rS1 << shift) ^ rS2
```

Operation:

```
result = operand1 XOR operand2
destination = result
SR.N = result[15]
SR.Z = (result == 0)
SR.C = 0
SR.V = 0
```

Description:
Performs bitwise XOR (exclusive OR) operation. Each bit in the result is 1 if the corresponding bits in the operands are different. In short immediate format with shift, the destination register is shifted before XORing with the immediate. Used for toggling bits, bitwise comparison, and simple encryption.
Flags: N Z (C and V cleared)
Example:

```
; Toggle bit 3
XORI r1, #0x0008            ; r1 = r1 ^ 0x08

; Invert all bits
XORI r2, #0xFFFF            ; r2 = ~r2

; Check if registers are equal
XORR r3, r1, r2             ; r3 = r1 ^ r2
; If r3 == 0, then r1 == r2

; Swap using XOR (no temporary)
XORR r1, r1, r2             ; r1 = r1 ^ r2
XORR r2, r1, r2             ; r2 = r1 ^ r2 = (r1 ^ r2) ^ r2 = r1
XORR r1, r1, r2             ; r1 = (r1 ^ r2) ^ r1 = r2
```

Notes:

- XORing with 0x0000 leaves value unchanged
- XORing with 0xFFFF inverts all bits (bitwise NOT)
- XORing a value with itself always gives 0
- Use **TSX** to check for differences without modifying register

10.11 LOAD – Load Immediate / Move Register

LOAD – Load/Move

Opcodes: 0x07 (Imm), 0x27 (Short Imm), 0x37 (Reg)
Syntax:

```
LOAD rD, #imm16                 ; rD = imm16 (via add)
LOAD rD, #imm8, shift           ; rD = imm8 << shift
LOAD rD, rS                     ; rD = rS (via add)
```

Operation:

```
; Implemented as: rD = 0 + operand
destination = operand
; Flags NOT updated
```

Description:
Loads an immediate value into a register or copies a register value. This is actually implemented as an ADD with an implicit zero operand, but flags are not updated. This is the primary way to load constants or move data between registers.
Flags: None
Example:

```
; Load constant
LOAD r1, #1234              ; r1 = 1234

; Copy register
LOAD r2, r1                 ; r2 = r1

; Load with arithmetic
LOAD r3, #1
ADDI r3, #0, LSL #8         ; r3 = (r3 << 8) + 0 = 256

; Zero a register
LOAD r4, #0                 ; r4 = 0

; Load large constant (combine)
LOAD r5, #0x12
ADDI r5, #0, LSL #8         ; r5 = (r5 << 8) + 0 = 0x1200
ORRI r5, #0x34              ; r5 = 0x1234
```

Notes:

- Does not update flags, unlike `ADD`
- Most efficient way to move data between registers
- For 16-bit constants, use immediate form
- For larger or shifted constants, use short immediate with shift

10.12 CMP – Compare

CMPI / CMPR – Compare Immediate / Compare Register

Opcodes: 0x0A (Imm), 0x2A (Short Imm), 0x3A (Reg)
Syntax:

```
CMPI rS, #imm16                 ; Compare rS with imm16
CMPI rS, #imm8, shift           ; Compare (rS << shift) with imm8
CMPR rS1, rS2                   ; Compare rS1 with rS2
CMPR rS1, rS2, shift            ; Compare (rS1 << shift) with rS2
```

Operation:

```
result = operand1 - operand2
; Result is discarded, only flags updated
SR.N = result[15]
SR.Z = (result == 0)
SR.C = NOT(borrow)
SR.V = signed_overflow
```

Description:
Performs subtraction but discards the result, only updating the status flags. In short immediate format with shift, the source register is shifted before comparing with the immediate. This is the primary instruction for implementing conditional execution and branches. The flags can be tested with condition codes to determine the relationship between the operands.
Flags: N Z C V
Example:

```
; Test if r1 is zero
CMPI r1, #0
BRAN|== is_zero             ; Branch if r1 == 0

; Test if r1 > r2 (unsigned)
CMPR r1, r2
BRAN|HI r1_greater

; Test if r1 >= r2 (signed)
CMPR r1, r2
BRAN|>= r1_greater_or_equal

; Range check: if (r1 >= 10 && r1 < 20)
CMPI r1, #10
BRAN|<< out_of_range        ; Branch if r1 < 10
CMPI r1, #20
BRAN|>= out_of_range        ; Branch if r1 >= 20
; r1 is in range [10, 20)
```

Notes:

- Does not modify any register, only flags
- Use with conditional instructions or branches
- Remember: `CMP A, B` computes A - B
- For signed comparisons, use `>=`, `<`, `»`, `«`
- For unsigned comparisons, use `>`, `CS`, `CC`, `<=`

10.13 TSA – Test AND

TSAI / TSAR – Test AND Immediate / Test AND Register

Opcodes: 0x0C (Imm), 0x2C (Short Imm), 0x3C (Reg)
Syntax:

```
TSAI rS, #imm16                 ; Test rS & imm16
TSAI rS, #imm8, shift           ; Test (rS << shift) & imm8
TSAR rS1, rS2                   ; Test rS1 & rS2
TSAR rS1, rS2, shift            ; Test (rS1 << shift) & rS2
```

Operation:

```
result = operand1 AND operand2
; Result is discarded, only flags updated
SR.N = result[15]
SR.Z = (result == 0)
SR.C = 0
SR.V = 0
```

Description:
Performs bitwise AND but discards the result, only updating flags. In short immediate format with shift, the source register is shifted before ANDing with the immediate. Used to test if specific bits are set without modifying the register. The zero flag indicates whether any tested bits were set.
Flags: N Z (C and V cleared)
Example:

```
; Test if bit 5 is set
TSAI r1, #0x0020            ; Test r1 & 0x20
BRAN|!= bit5_set            ; Branch if bit 5 was set

; Test if any upper byte bits are set
TSAI r2, #0xFF00
BRAN|!= has_upper_bits

; Test if r1 and r2 have any bits in common
TSAR r1, r2
BRAN|== no_common_bits      ; Branch if no bits in common

; Test multiple bits
TSAI r3, #0xF000            ; Test upper nibble
BRAN|== upper_clear         ; Branch if all tested bits clear
```

Notes:

- Does not modify any register, only flags
- Z=1 means all tested bits were 0
- Z=0 means at least one tested bit was 1
- Use for bit testing without side effects
- More efficient than AND+CMP for simple tests

10.14 TSX – Test XOR

TSXI / TSXR – Test XOR Immediate / Test XOR Register

Opcodes: 0x0E (Imm), 0x2E (Short Imm), 0x3E (Reg)
Syntax:

```
TSXI rS, #imm16                 ; Test rS ^ imm16
TSXI rS, #imm8, shift           ; Test (rS << shift) ^ imm8
TSXR rS1, rS2                   ; Test rS1 ^ rS2
TSXR rS1, rS2, shift            ; Test (rS1 << shift) ^ rS2
```

Operation:

```
result = operand1 XOR operand2
; Result is discarded, only flags updated
SR.N = result[15]
SR.Z = (result == 0)
SR.C = 0
SR.V = 0
```

Description:
Performs bitwise XOR but discards the result, only updating flags. In short immediate format with shift, the source register is shifted before XORing with the immediate. Used to test if bits match a specific pattern or to compare values for equality. Z=1 indicates the values are identical.
Flags: N Z (C and V cleared)
Example:

```
; Test if r1 equals specific value
TSXI r1, #0x1234            ; Test r1 ^ 0x1234
BRAN|== is_0x1234           ; Branch if r1 == 0x1234

; Test if two registers are equal
TSXR r1, r2                 ; Test r1 ^ r2
BRAN|== registers_equal     ; Branch if r1 == r2

; Test if bit pattern matches
TSAI r3, #0xFF              ; Mask to lower byte
TSXI r3, #0xAA              ; Test if lower byte == 0xAA
BRAN|== pattern_match
```

Notes:

- Does not modify any register, only flags
- Z=1 means values are identical (all bits match)
- Can be used as equality test (alternative to CMP for == only)
- More efficient than XOR + CMP for simple equality tests

10.15 Summary

The ALU instructions form the computational core of the SIRCIS instruction set. Key points:

- All ALU operations work only on registers (load/store architecture)
- Three format variants provide flexibility (immediate, short+shift, register)
- Test variants (CMP, TSA, TSX) allow non-destructive testing
- Shift operations can be combined with ALU ops for powerful single-cycle operations
- Consistent flag behavior makes conditional execution predictable

Chapter 11

Memory Access Instructions

11.1 Overview

Memory access instructions transfer data between registers and memory. Following the load/store architecture, these are the only instructions that can access memory—all arithmetic and logical operations must work on registers.

The SIRC-1 provides two primary memory operations:

- `LOAD` – Read from memory into a register
- `STOR` – Write from a register to memory

Both instructions support multiple addressing modes:

- Indirect with immediate offset: `(#offset, addr)`
- Indirect with register offset: `(reg, addr)`
- Post-increment: `(#offset, addr)+`
- Pre-decrement: `-(#offset, addr)`

Important: When using indirect addressing modes (with memory access), `LOAD` and `STOR` instructions support optional shift operations on the data being loaded or stored. See Section 11.6 for details. Note that shifts are **not** supported when loading immediate values or copying register values directly.

11.2 Effective Address Calculation

For all memory operations, the effective address (EA) is calculated as:

$$\text{EA} = \text{AddressRegister} + \text{Offset} \tag{11.1}$$

Where:

- Address Register is one of: `l`, `a`, `s`, or `p`
- Offset is either an immediate value or register contents

The resulting 24-bit address is used to access memory.

11.3 LOAD Instructions

LOAD – Load from Memory

Opcodes: 0x14 (Indirect Imm), 0x15 (Indirect Reg), 0x16 (Post-Inc Imm), 0x17 (Post-Inc Reg)
Syntax:

```
LOAD rD, (#offset, addr)        ; rD = memory[addr + offset]
LOAD rD, (rS, addr)             ; rD = memory[addr + rS]
LOAD rD, (#offset, addr)+       ; rD = memory[addr + offset]; addr += 1
LOAD rD, (rS, addr)+            ; rD = memory[addr + rS]; addr += 1

; With optional shift (memory addressing modes only)
LOAD rD, (#offset, addr), shift       ; Load and shift
LOAD rD, (rS, addr), shift            ; Load and shift
LOAD rD, (#offset, addr)+, shift      ; Load, shift, then increment
LOAD rD, (rS, addr)+, shift           ; Load, shift, then increment
```

Description:
Loads a 16-bit value from memory into the destination register. Post-increment forms update the address register after the load, useful for array traversal.
Examples:

```
; Load from fixed offset
LOAD r1, (#4, a)                ; r1 = memory[a + 4]

; Load using variable offset
LOAD r2, (r3, a)                ; r2 = memory[a + r3]

; Array iteration
LOAD r4, (#0, a)+               ; r4 = *a; a += 1

; Stack pop
LOAD r5, (#0, s)+               ; r5 = *s; s += 1

; Load and shift (for bit manipulation)
LOAD r3, (r2, a), LSL #2        ; r3 = memory[a + r2] << 2
LOAD r6, (#0, a), ASR #2        ; r6 = memory[a] >> 2 (signed)
```

11.4 STOR Instructions

STOR – Store to Memory

Opcodes: 0x10 (Indirect Imm), 0x11 (Indirect Reg), 0x12 (Pre-Dec Imm), 0x13 (Pre-Dec Reg)
Syntax:

```
STOR (#offset, addr), rS        ; memory[addr + offset] = rS
STOR (rD, addr), rS             ; memory[addr + rD] = rS
STOR -(#offset, addr), rS       ; addr -= 1; memory[addr + offset] =
    rS
STOR -(rD, addr), rS            ; addr -= 1; memory[addr + rD] = rS

; With optional shift (memory addressing modes only)
STOR (#offset, addr), rS, shift       ; Shift and store
STOR (rD, addr), rS, shift            ; Shift and store
STOR -(#offset, addr), rS, shift      ; Decrement, shift, and store
STOR -(rD, addr), rS, shift           ; Decrement, shift, and store
```

Description:
Stores a 16-bit value from a register into memory. Pre-decrement forms update the address register before the store, useful for stack push operations.
Examples:

```
; Store to fixed offset
STOR (#8, a), r1                ; memory[a + 8] = r1

; Store using variable offset
STOR (r3, a), r2                ; memory[a + r3] = r2

; Store with shift (data shifted before storing)
STOR (r2, a), r1, LSL #1        ; memory[a + r2] = r1 << 1
STOR (#0, a), r1, LSR #3        ; memory[a] = r1 >> 3

; Stack push
STOR -(#0, s), r4               ; s -= 1; memory[s] = r4
```

11.5 Common Memory Access Patterns

11.5.1 Structure Field Access

```
; struct { int16_t x, y, z; } point;
; 'a' points to point
LOAD r1, (#0, a)                ; r1 = point.x
LOAD r2, (#1, a)                ; r2 = point.y
LOAD r3, (#2, a)                ; r3 = point.z
```

11.5.2 Array Iteration

```
; Process array of 16-bit values
; 'a' = array base, r7 = count
loop:
    LOAD r1, (#0, a)+          ; Load element, advance
    ; ... process r1 ...
    SUBI r7, #1
    BRAN|!= loop
```

11.5.3 Stack Operations

```
; Push registers
STOR -(#0, s), r1
STOR -(#0, s), r2
STOR -(#0, s), r3

; Pop registers (reverse order)
LOAD r3, (#0, s)+
LOAD r2, (#0, s)+
LOAD r1, (#0, s)+
```

11.6 Shift Operations with Memory Instructions

When using indirect addressing modes (memory access), both **LOAD** and **STOR** instructions support optional shift operations. This feature allows data to be shifted during the load or store operation, which is particularly useful for bit manipulation and working with the CPU's word-addressed memory architecture.

11.6.1 LOAD with Shift

When loading from memory, the data read from memory is shifted **after** being loaded but **before** being written to the destination register:

```
LOAD r3, (r2, a), LSL #2       ; r3 = (memory[a + r2]) << 2
LOAD r4, (#0, a), LSR #4       ; r4 = (memory[a]) >> 4
LOAD r5, (r2, a), ASL #3       ; r5 = (memory[a + r2]) << 3
LOAD r6, (#0, a), ASR #2       ; r6 = (memory[a]) >> 2 (sign-extend)
```

11.6.2 STOR with Shift

When storing to memory, the source register value is shifted **before** being written to memory. The original register value is **not** modified:

```
STOR (r2, a), r1, LSL #1       ; memory[a + r2] = r1 << 1 (r1 unchanged)
STOR (#0, a), r1, LSR #3       ; memory[a] = r1 >> 3 (r1 unchanged)
STOR (r2, a), r1, ASL #2       ; memory[a + r2] = r1 << 2 (r1 unchanged)
STOR (#0, a), r1, ASR #4       ; memory[a] = r1 >> 4 (r1 unchanged)
```

11.6.3 Supported Shift Types

All standard shift types are supported:

- **LSL** – Logical Shift Left (fill with zeros)
- **LSR** – Logical Shift Right (fill with zeros)
- **ASL** – Arithmetic Shift Left (same as LSL)
- **ASR** – Arithmetic Shift Right (sign-extend)
- **RTL** – Rotate Left (through carry)
- **RTR** – Rotate Right (through carry)

11.6.4 Shift Limitations

Important: Shift operations are **only** supported when using indirect addressing modes (accessing memory). The following `LOAD` modes do **not** support shifts:

- Loading immediate values: `LOAD r1, #0x1234` – No shift allowed
- Copying registers: `LOAD r1, r2` – No shift allowed

For these non-memory operations, use regular ALU instructions with shifts instead (e.g., `ADDI r1, #0, LSL #2`).

11.6.5 Use Cases

Word-Addressed Memory

The SIRC-1 uses word addressing, so shifting during load/store can save cycles when converting between byte and word addresses:

```
; Convert byte offset to word address and load
LOAD r3, (r2, a), LSR #1        ; Divide byte offset by 2
```

Bit Manipulation

Shifting during memory access enables efficient bit manipulation:

```
; Extract upper byte from memory
LOAD r1, (#0, a), LSR #8        ; r1 = (memory[a] >> 8) & 0xFF

; Store scaled value
STOR (#0, a), r2, LSL #2        ; memory[a] = r2 * 4
```

Combined with Auto-Increment/Decrement

Shifts work seamlessly with post-increment and pre-decrement modes:

```
; Load, shift, and advance pointer
LOAD r3, (r2, a)+, LSL #1      ; r3 = (memory[a + r2]) << 1; a += 1

; Decrement, shift, and store
STOR -(r2, a), r1, LSR #2      ; a -= 1; memory[a + r2] = r1 >> 2
```

11.6.6 Status Flags

Note: Unlike shifts in ALU instructions, shifts performed during **LOAD** and **STOR** operations can **not** update the status register flags. If you need flag updates based on a shift result, use an ALU instruction or the **SHFT** meta-instruction (see Chapter 14).

Chapter 12

Control Flow Instructions

12.1 Overview

Control flow instructions modify the program counter to change the execution flow. The SIRC-1 provides several types of control flow operations:

- **`BRAN`** – Branch (unconditional or conditional jump)
- **`LJSR`** – Long Jump to Subroutine (with link)
- **`BRSR`** – Branch to Subroutine (relative, with link)
- **`LDEA`** – Load Effective Address (can be used for jumps)

All control flow instructions can be conditional, allowing efficient implementation of if-then-else and loop constructs.

12.2 Branch Instructions

BRAN – Branch

Opcodes: 0x1A (Imm Displacement), 0x1B (Reg Displacement)
Syntax:

```
BRAN (#disp, p)                 ; PC = PC + disp
BRAN (rS, p)                    ; PC = PC + rS
BRAN|cond (#disp, p)            ; Conditional branch
```

Description:
Branches to a new location by adding a displacement to the program counter. The displacement can be an immediate value (for fixed jumps) or a register value (for computed jumps). Branches are typically PC-relative. **Examples:**

```
; Unconditional branch forward
BRAN (#20, p)                   ; Jump ahead 20 words

; Backward branch (loop)
loop:
    ; ... loop body ...
    BRAN|!= (#-16, p)           ; Jump back if not zero

; Conditional branch
CMPI r1, #10
BRAN|>= (#8, p)                 ; Branch if r1 >= 10
```

12.3 Subroutine Call Instructions

LJSR – Long Jump to Subroutine

Opcodes: 0x1C (Imm), 0x1D (Reg)
Syntax:

```
LJSR (#addr, a)                 ; l = PC; PC = a + addr
LJSR (#addr, p)                 ; l = PC; PC = PC + addr (rel call)
```

Description:
Calls a subroutine by saving the return address to the link register (`l`) and jumping to the target address. The return address can later be restored using **`RETS`**.
Examples:

```
; Call function at address in 'a'
LJSR (#0, a)

; PC-relative call
LJSR (#function_offset, p)

; Return from subroutine
RETS                            ; Assembles to: LDEA p, (#0, l)
```

12.4 Load Effective Address

LDEA – Load Effective Address

Opcodes: 0x18 (Imm), 0x19 (Reg)
Syntax:

```
LDEA dest, (#offset, src)       ; dest = src + offset
LDEA dest, (rS, src)            ; dest = src + rS
```

Description:
Computes an effective address and loads it into the destination address register pair. Can be used for pointer arithmetic or, when the destination is the program counter, as a jump instruction.
Examples:

```
; Pointer arithmetic
LDEA a, (#16, a)                ; a = a + 16

; Computed jump
LDEA p, (r1, p)                 ; PC = PC + r1 (jump table)

; Return from subroutine
LDEA p, (#0, l)                 ; PC = l (RETS pseudo-instruction)
```

12.5 Control Flow Patterns

12.5.1 If-Then-Else

```
; if (r1 > r2) { ... } else { ... }
CMPR r1, r2
BRAN|<= else_part
; Then part
; ...
BRAN (#end, p)
else_part:
; Else part
; ...
end:
```

12.5.2 While Loop

```
; while (r1 > 0) { ... }
while_start:
    CMPI r1, #0
    BRAN|<= while_end
    ; Loop body
    ; ...
    BRAN (#while_start, p)
while_end:
```

12.5.3 Switch/Jump Table

```
; Switch on r1 (0-3)
LOAD r2, r1                    ; r2 = r1
ADDI r2, #0, LSL #1            ; r2 = (r2 << 1) = r2 * 2 (word size)
LDEA p, (r2, p)                ; Jump via table

jump_table:
    DW case0 - jump_table
    DW case1 - jump_table
    DW case2 - jump_table
    DW case3 - jump_table
```

(Additional control flow details would follow)

Chapter 13

Coprocessor Instructions

13.1 Overview

The SIRC-1 CPU includes a coprocessor interface that allows delegation of certain operations to specialized coprocessors. The primary coprocessor is the Exception Unit (Coprocessor 1), which handles exceptions, interrupts, and privileged operations.

COP – Coprocessor Call

Opcodes: 0x0F (Imm), 0x2F (Short Imm), 0x3F (Reg)
Syntax:

```
COPI #opcode                    ; Call coprocessor with 16-bit opcode
COPR rS                         ; Call coprocessor with register
    value
```

Description:
Transfers control to a coprocessor. The coprocessor ID and operation code are encoded in the immediate value or register. Coprocessor operations may be privileged.
Format:

- Bits 15–12: Coprocessor ID
- Bits 11–0: Operation code

Examples:

```
; Wait for interrupt (Exception Unit)
COPI #0x1900                    ; Coprocessor 1, opcode 0x900
; Assembled as WAIT meta-instruction

; Return from exception
COPI #0x1A00                    ; Coprocessor 1, opcode 0xA00
; Assembled as RETE meta-instruction

; User exception (system call)
COPI #0x1180                    ; Coprocessor 1, opcode 0x180 (vector
    0x80)
; Assembled as EXCP #0x80 meta-instruction
```

13.2 Exception Unit (Coprocessor 1)

The Exception Unit handles:

- Hardware exceptions (faults, traps)
- Software exceptions
- Interrupts (multiple priority levels)
- Privileged operations

13.2.1 Common Operations

Opcode	Mnemonic	Operation
0x1100-0x11FF	EXCP	User exception (trap) at vector 0x60-0xFF
0x1900	WAIT	Wait for interrupt
0x1A00	RETE	Return from exception
0x1B00	RSET	System reset
0x1C00-0x1C7F	ETFR	Transfer from exception link register
0x1D00-0x1D7F	ETTR	Transfer to exception link register

Table 13.1: Exception Unit Operations

See Chapter 4 for detailed documentation of exception handling and all exception coprocessor instructions.

Chapter 14

Meta-Instructions

14.1 Overview

Meta-instructions (also called pseudo-instructions) are assembly mnemonics that don't directly correspond to unique opcodes. Instead, they assemble to specific forms of existing instructions, providing more intuitive syntax for common operations.

14.2 NOOP – No Operation

Syntax: `NOOP`

Assembles to: `ADDI r1, #0`

Description: Does nothing for 6 cycles. Useful for timing delays, instruction padding, or placeholders during development.

Example:

```
1 NOOP                          ; Wait 6 cycles
2 NOOP
3 NOOP                          ; Total 18 cycles delay
```

14.3 RETS – Return from Subroutine

Syntax: `RETS`

Assembles to: `LDEA p, (#0, l)`

Description: Returns from a subroutine by copying the link register to the program counter. Used after **`LJSR`** or **`BRSR`**.

Example:

```
1 my_function:
2     ; ... function body ...
3     RETS                      ; Return to caller
```

14.4 EXCP – User Exception (Trap)

Syntax: `EXCP #vector`

Assembles to: `COPI #0x1100 | vector`

Description: Triggers a software exception (trap) at the specified user vector (0x60-0xFF). Used by user-mode programs to invoke system calls or request supervisor mode services.

Example:

```
; System call to print character
LOAD r1, #'A'                   ; Character to print
EXCP #0x80                      ; Trap to OS print routine
```

14.5 WAIT – Wait for Interrupt

Syntax: `WAIT`

Assembles to: `COPI #0x1900`

Description: Puts the CPU into a low-power wait state until an exception occurs. Sets the Waiting for Interrupt bit in the status register. Only valid in supervisor mode.

Example:

```
main_loop:
    WAIT                        ; Sleep until interrupt
    ; ... process interrupt ...
    BRAN main_loop
```

14.6 RETE – Return from Exception

Syntax: `RETE`

Assembles to: `COPI #0x1A00`

Description: Returns from an exception handler by restoring the saved program counter and status register. Only valid in supervisor mode.

Example:

```
exception_handler:
    ; ... handle exception ...
    RETE                        ; Return to interrupted code
```

14.7 RSET – System Reset

Syntax: `RSET`

Assembles to: `COPI #0x1B00`

Description: Performs a software reset of the processor by clearing the status register and jumping to the reset vector. Only valid in supervisor mode.

Example:

```
fatal_error:
    ; Log error...
    RSET                        ; Reset the system
```

14.8 ETFR – Exception Transfer From Register

Syntax:

```
ETFR #n                          ; Transfer both to a and r7
ETFR a, #n                       ; Transfer address to a only
ETFR r7, #n                      ; Transfer status to r7 only
```

Assembles to: `COPI #0x1Cxy` (where x = register select, y = link register)

Description: Copies the return address and/or status register from a link register (0-7) to CPU registers. Used in exception handlers to examine saved processor state. Only valid in supervisor mode.

Example:

```
alignment_fault_handler:
    ETFR a, #6                   ; Get faulting return address
    LOAD al, @fixed_address      ; Correct the address
    ETTR #6, a                   ; Write corrected address back
    RETE
```

14.9 ETTR – Exception Transfer To Register

Syntax:

```
ETTR #n                          ; Transfer both a and r7
ETTR #n, a                       ; Transfer a to address only
ETTR #n, r7                      ; Transfer r7 to status only
```

Assembles to: `COPI #0x1Dxy` (where x = register select, y = link register)

Description: Copies CPU registers to a link register's return address and/or status register. Used to modify saved processor state before returning from an exception. Only valid in supervisor mode.

Example:

```
privilege_fault_handler:
    ETFR r7, #6                  ; Get saved status register
    ANDI r7, #0xFEFF             ; Clear protected mode bit
    ETTR #6, r7                  ; Save modified status back
    RETE
```

14.10 SHFT – Shift with Status Update

Syntax: `SHFT rD, shift`

Assembles to: `ORRI[S] rD, #0, shift`

Description: Shifts the value in a register and updates the status register flags based on the shift result (not the ALU operation). This is useful when you need to shift a value and have the flags reflect the shifted result.

Normally, when using shift operations with ALU instructions like `CMPI r2, #1, LSL #2`, the status flags are updated according to the ALU operation result, not the shift. The **`SHFT`** instruction allows the status flags to be updated from the shift operation itself.

Examples:

```
; Shift r1 left by 3 and update flags
SHFT r1, LSL #3                ; r1 = r1 << 3, flags = status of shift

; Shift right and test result
SHFT r3, LSR #2                ; r3 = r3 >> 2
BRAN|== zero_result            ; Branch if shifted result is zero

; Arithmetic shift with sign check
SHFT r6, ASR #4                ; r6 = r6 >> 4 (signed)
BRAN|NS negative_value         ; Branch if result is negative
```

Notes:

- Behind the scenes, **SHFT** uses the status override syntax (see Section 7.12) to force flag updates from the shift operation
- All shift types are supported (LSL, LSR, ASL, ASR, RTL, RTR)
- Both immediate and register-based shift counts are supported

(Note: Check assembler documentation for supported pseudo-instructions)

Appendix A

Opcode Map

A.1 Complete Opcode Reference

This appendix provides a complete map of all 64 possible opcodes in the SIRCIS instruction set.

Hex	Binary	Mnemonic	Description
ALU Immediate (0x00–0x0F)			
00	000000	ADDI	Add Immediate
01	000001	ADCI	Add with Carry Immediate
02	000010	SUBI	Subtract Immediate
03	000011	SBCI	Subtract with Carry Immediate
04	000100	ANDI	AND Immediate
05	000101	ORRI	OR Immediate
06	000110	XORI	XOR Immediate
07	000111	LOAD	Load Immediate
08	001000	–	Undocumented
09	001001	–	Undocumented
0A	001010	CMPI	Compare Immediate
0B	001011	–	Undocumented
0C	001100	TSAI	Test AND Immediate
0D	001101	–	Undocumented
0E	001110	TSXI	Test XOR Immediate
0F	001111	COPI	Coprocessor Immediate
Memory and Control (0x10–0x1F)			
10	010000	STOR	Store Indirect Immediate
11	010001	STOR	Store Indirect Register
12	010010	STOR	Store Pre-Decrement Immediate
13	010011	STOR	Store Pre-Decrement Register
14	010100	LOAD	Load Indirect Immediate
15	010101	LOAD	Load Indirect Register
16	010110	LOAD	Load Post-Increment Immediate
17	010111	LOAD	Load Post-Increment Register
18	011000	LDEA	Load Effective Address Immediate
19	011001	LDEA	Load Effective Address Register
1A	011010	BRAN	Branch Immediate
1B	011011	BRAN	Branch Register
1C	011100	LJSR	Long Jump to Subroutine Immediate
1D	011101	LJSR	Long Jump to Subroutine Register
1E	011110	BRSR	Branch to Subroutine Immediate
1F	011111	BRSR	Branch to Subroutine Register
ALU Short Immediate (0x20–0x2F)			
20	100000	ADDI	Add Short Immediate
21	100001	ADCI	Add with Carry Short Immediate
22	100010	SUBI	Subtract Short Immediate
23	100011	SBCI	Subtract with Carry Short Immediate
24	100100	ANDI	AND Short Immediate
25	100101	ORRI	OR Short Immediate
26	100110	XORI	XOR Short Immediate
27	100111	LOAD	Load Short Immediate
28	101000	–	Undocumented
29	101001	–	Undocumented
2A	101010	CMPI	Compare Short Immediate
2B	101011	–	Undocumented
2C	101100	TSAI	Test AND Short Immediate
2D	101101	–	Undocumented

A.2 Opcode Patterns

A.2.1 Format Identification

- Bits 5–4 = 00: Immediate format (0x00–0x0F)
- Bits 5–4 = 01: Memory/Control operations (0x10–0x1F)
- Bits 5–4 = 10: Short Immediate with Shift (0x20–0x2F)
- Bits 5–4 = 11: Register format (0x30–0x3F)

A.2.2 Save vs. Test

- Bit 3 = 0: Save result (0x00–0x07, 0x20–0x27, 0x30–0x37)
- Bit 3 = 1: Test only, discard result (0x08–0x0F, 0x28–0x2F, 0x38–0x3F)

A.2.3 Memory Operations Decoding

For opcodes 0x10–0x1F, bits encode memory operation type:

- Bit 4 = 1: Always set for memory/control section
- Bits 3–2: Operation type
  - 00 = Store
  - 01 = Load
  - 10 = Load Effective Address
  - 11 = Load EA with Link (jump to subroutine)
- Bit 1: Determines if both registers in address pair are updated
- Bit 0: Immediate (0) or Register (1) offset

A.3 Undocumented Instructions

The following opcodes are undocumented and should not be used in production code:

- 0x08, 0x09, 0x0B, 0x0D (Immediate format, test variants)
- 0x28, 0x29, 0x2B, 0x2D (Short Immediate format, test variants)
- 0x38, 0x39, 0x3B, 0x3D (Register format, test variants)

These opcodes may execute in hardware but their behavior is implementation-defined and may change between CPU revisions. See Appendix C for details.

A.4 Quick Lookup Table

Operation	Immediate	Short+Shift	Register
ADD	0x00	0x20	0x30
ADC	0x01	0x21	0x31
SUB	0x02	0x22	0x32
SBC	0x03	0x23	0x33
AND	0x04	0x24	0x34
OR	0x05	0x25	0x35
XOR	0x06	0x26	0x36
LOAD	0x07	0x27	0x37
CMP	0x0A	0x2A	0x3A
TSA	0x0C	0x2C	0x3C
TSX	0x0E	0x2E	0x3E
COP	0x0F	0x2F	0x3F

Table A.2: Instruction Variant Quick Reference

Appendix B

Instruction Timing

B.1 Overview

All SIRCIS instructions execute in exactly 6 clock cycles, following the six-stage pipeline architecture. This fixed timing greatly simplifies software development and hardware design.

B.2 Six-Stage Pipeline

1. **Instruction Fetch (High Word)** – 1 cycle
   - Fetch bits 31–16 from memory address in PC
   - Increment PC by 1
2. **Instruction Fetch (Low Word)** – 1 cycle
   - Fetch bits 15–0 from memory address in PC
   - Increment PC by 1
   - Instruction is now fully available for decode
3. **Decode and Register Fetch** – 1 cycle
   - Decode instruction opcode and operands
   - Read source registers
   - Evaluate condition codes
4. **Execute and Address Calculation** – 1 cycle
   - Perform ALU operation, or
   - Calculate effective memory address
   - Apply shift operations if specified
5. **Memory Access** – 1 cycle
   - Read from memory (LOAD), or
   - Write to memory (STOR), or

- No operation (register-only instructions)

6. **Write Back** – 1 cycle
   - Write result to destination register
   - Update status register flags if applicable
   - Update address registers for post-increment/pre-decrement

B.3 Timing Characteristics

Instruction Type	Cycles
ALU (Register)	6
ALU (Immediate)	6
Load from Memory	6
Store to Memory	6
Branch (taken)	6
Branch (not taken)	6
Conditional (executed)	6
Conditional (not executed)	6
NOOP	6

Table B.1: Instruction Timing (All instructions)

B.4 Conditional Execution

When a conditional instruction's condition is not met:

- The instruction still takes 6 cycles to complete
- All pipeline stages execute normally
- The write-back stage is disabled (no register or memory modification)
- Status flags are not updated

This means conditional instructions do not branch and avoid pipeline stalls, but failed conditions still consume execution time.

B.5 Program Timing Examples

B.5.1 Simple Loop

```
; Process 10 elements
ADDI r7, #0, #10                ; 6 cycles
loop:
    LOAD r1, (#0, a)+           ; 6 cycles
    ADDI r1, #1                 ; 6 cycles
    STOR (#-2, a), r1           ; 6 cycles
    SUBI r7, #1                 ; 6 cycles
    BRAN|!= (#loop, p)          ; 6 cycles
; Total per iteration: 30 cycles
; Total for 10 iterations: 6 + (10 * 30) = 306 cycles
```

B.5.2 Function Call Overhead

```
; Call function
LJSR (#0, a)                    ; 6 cycles

; Function body executes...

; Return
RETS                            ; 6 cycles
; Total overhead: 12 cycles
```

B.5.3 Conditional Block

```
CMPR r1, r2                     ; 6 cycles
ADDR|HI r3, r4, r5              ; 6 cycles (exec if r1 > r2)
ADDR|LO r3, r6, r7              ; 6 cycles (exec if r1 <= r2)
; Total: 18 cycles (one conditional executes, one doesn't)
```

B.6 Performance Considerations

B.6.1 No Pipeline Stalls

The SIRC-1 does not implement pipelining in the traditional sense where multiple instructions are in flight simultaneously. Instead:

- Each instruction completes fully before the next begins
- No data hazards or pipeline bubbles
- No branch prediction penalties
- Execution time is completely predictable

B.6.2 Memory Access

- Memory access always takes 1 cycle within the instruction
- No wait states or cache misses (in the simulation)
- Real hardware implementations may have variable memory timing

B.6.3 Optimization Guidelines

1. **Use conditional execution for short sequences**

```
; Instead of:
CMPR r1, r2
BRAN|<= skip
ADDI r3, #1
skip:

; Use:
CMPR r1, r2
ADDI|HI r3, #1              ; Saves branch overhead
```

2. **Combine operations with shifts**

```
; Instead of:
ADDR r1, r2, r2
ADDI r1, #0, LSL #1

; Use:
ADDR r1, r2, r2, LSL #1     ; Multiply by 3 in one instruction
```

3. **Use post-increment/pre-decrement for iteration**

```
; Instead of:
LOAD r1, (#0, a)
ADDI a, #2

; Use:
LOAD r1, (#0, a)+           ; Same result, fewer cycles
```

4. **Loop unrolling reduces overhead**

```
; Instead of processing one item per iteration,
; process multiple items to reduce branch overhead
loop:
    LOAD r1, (#0, a)+
    LOAD r2, (#0, a)+
    LOAD r3, (#0, a)+
    LOAD r4, (#0, a)+
    ; Process 4 items...
    SUBI r7, #4
    BRAN|HI loop
```

B.7 Theoretical Performance

At a given clock frequency:

- **Instructions per second:** $f_{clock}/6$
- **At 10 MHz:** $\approx$ 1.67 million instructions/second

- **At 25 MHz:** ≈ 4.17 million instructions/second

These are theoretical maximums assuming perfect code with no inefficiencies.

B.8 Comparison to Other CPUs

CPU	Avg CPI	Notes
SIRC-1	6.0	Fixed, all instructions
MOS 6502	2-7	Variable by instruction
Motorola 68000	4-158	Highly variable
ARM6	1.0	Single-cycle with pipeline
MIPS R2000	1.0	Single-cycle with pipeline

Table B.2: Cycles Per Instruction Comparison

The SIRC-1's fixed 6-cycle execution makes it slower than pipelined RISC CPUs but more predictable. It's comparable to early microprocessors while being simpler to implement in hardware.

Appendix C

Undocumented Instructions

C.1 Overview

The SIRCIS instruction set includes 12 undocumented instruction opcodes. These opcodes are valid and will execute in hardware, but their behaviour is not officially specified and may vary between CPU implementations or revisions.

Warning: Use of undocumented instructions is not recommended for production code.

C.2 Undocumented Opcode List

Opcode	Format	Pattern	Notes
0x08	Immediate	ADD-like + 0x8	Test variant (no save)
0x09	Immediate	ADC-like + 0x8	Test variant (no save)
0x0B	Immediate	SUB-like + 0x8	Test variant (no save)
0x0D	Immediate	SBC-like + 0x8	Test variant (no save)
0x28	Short Imm	ADD-like + 0x8	Test variant (no save)
0x29	Short Imm	ADC-like + 0x8	Test variant (no save)
0x2B	Short Imm	SUB-like + 0x8	Test variant (no save)
0x2D	Short Imm	SBC-like + 0x8	Test variant (no save)
0x38	Register	ADD-like + 0x8	Test variant (no save)
0x39	Register	ADC-like + 0x8	Test variant (no save)
0x3B	Register	SUB-like + 0x8	Test variant (no save)
0x3D	Register	SBC-like + 0x8	Test variant (no save)

Table C.1: Undocumented Instruction Opcodes

C.3 Likely Behavior

Based on the systematic opcode organization, these undocumented instructions likely behave as "test" variants of arithmetic instructions:

C.3.1 0x08, 0x28, 0x38 (ADD Test)

Probably performs addition but discards the result, only updating flags. Similar to `CMP` but using ADD semantics instead of SUB.

```
; Hypothetical behavior
result = operand1 + operand2  ; Calculate
; result is discarded (not written to register)
; Flags updated based on result
```

Utility: Limited. `CMP` already provides comparison functionality.

C.3.2 0x09, 0x29, 0x39 (ADC Test)

Probably performs add-with-carry but discards the result, updating flags.

```
; Hypothetical behavior
result = operand1 + operand2 + C
; result is discarded
; Flags updated
```

Utility: Possibly useful for testing carry propagation without modifying registers.

C.3.3 0x0B, 0x2B, 0x3B (SUB Test)

Probably identical to `CMP` (which is already SUB without save).
Utility: None. Use `CMP` instead.

C.3.4 0x0D, 0x2D, 0x3D (SBC Test)

Probably performs subtract-with-borrow but discards the result.

```
; Hypothetical behavior
result = operand1 - operand2 - (1 - C)
; result is discarded
; Flags updated
```

Utility: Possibly useful for testing borrow propagation in multi-precision arithmetic.

C.4 Implementation Notes

C.4.1 Current Simulator

In the current SIRC-VM simulator (as of version 1.0):

- Undocumented instructions are recognized and decoded
- They execute following the "test variant" pattern (no write-back)

- Flag updates may or may not be implemented correctly
- Behavior may change in future versions

C.4.2 Hardware Implementation

In a hardware implementation:

- The decoder treats bit 3 = 1 as "don't write back"
- ALU operations still execute normally
- Flags are still updated
- The write-back stage is simply disabled

This means the instructions *will* work, but their exact behavior depends on the hardware implementation details.

C.5 Why Undocumented?

These instructions were left undocumented for several reasons:

1. **Limited utility**: Most have no practical use beyond documented instructions
2. **Incomplete semantics**: The designer hasn't fully defined their behavior
3. **Opcode space reservation**: May be used for future features
4. **Implementation flexibility**: Allows hardware optimizations

C.6 Recommendations

C.6.1 For Application Developers

Do not use undocumented instructions in production code.

- Behavior may change between CPU revisions
- Assemblers may reject them or emit warnings
- Code may not be portable
- Debugging tools may not recognize them

C.6.2 For Compiler Writers

- Do not generate undocumented instructions
- Use only documented instructions for all operations
- If optimization requires new operations, request formal documentation

C.6.3 For Hardware Implementers

- You may implement these opcodes following the logical pattern
- Document any implementation-specific behavior
- Consider reserving them for future standardization

C.6.4 For Researchers/Hackers

If you must experiment with undocumented instructions:

- Test thoroughly on your specific hardware/simulator version
- Document your findings
- Isolate usage in test code, not production
- Be prepared for behavior changes

C.7 Future Standardization

Some undocumented instructions may be formally specified in future SIRCIS revisions. Candidates for standardization include:

- **ADC Test (0x09/0x29/0x39)**: For carry flag testing in multi-precision
- **SBC Test (0x0D/0x2D/0x3D)**: For borrow flag testing in multi-precision

Others (SUB Test, which duplicates CMP) will likely remain undocumented or be repurposed.

C.8 Opcode Space for Expansion

The SIRCIS instruction set has limited expansion space:

- 12 undocumented opcodes (could be formally specified)
- Reserved shift type (0x111) could enable additional shift modes
- Reserved address register encoding could enable more address modes
- Additional coprocessors could provide extended functionality

Any future expansion must maintain backward compatibility with existing code.