RISC-4 ISA Specification

Version 0.1.0
Pre-Silicon

Jeremy King

January 21, 2026

Abstract

RISC-4 is a 4-bit load/store RISC architecture designed to demonstrate that RISC principles are independent of datapath width. This specification defines the instruction set architecture (ISA), programmer's model, memory organization, and calling conventions for the RISC-4 processor.

This document represents an alternate history: "What if the RISC revolution occurred during the 4-bit era of the early 1970s?"

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

RISC-4 is a 4-bit Reduced Instruction Set Computer (RISC) architecture that combines the simplicity of early microprocessors like the Intel 4004 with modern RISC design principles. It serves as both an educational tool and a demonstration that RISC concepts apply at any scale.

1.2 Design Principles

The RISC-4 architecture adheres to the following principles:

1. **Fixed-length instructions** – All instructions are 16 bits (4 nibbles)
2. **Load/store architecture** – Only load and store instructions access memory
3. **Simple instruction formats** – Four orthogonal formats: R, I, J, M
4. **Regular encoding** – Consistent field placement across formats
5. **Pipeline-friendly** – Designed for 5-stage pipeline from inception
6. **Compiler-friendly** – Simple, orthogonal instruction set

1.3 Key Specifications

Parameter	Value
Datapath width	4 bits
Instruction width	16 bits (fixed)
Address space	12 bits (4096 nibbles)
General purpose registers	16×4 bits
Pipeline stages	5 (IF, ID, EX, MEM, WB)
Endianness	Little-endian

Table 1.1: RISC-4 Architecture Parameters

1.4 Comparison to Historical Architectures

Table 1.2 compares RISC-4 to its historical inspirations.

Feature	**Intel 4004**	**RISC-4**	**RISC-I**
Year (design)	1971	2025	1982
Datapath	4-bit	4-bit	32-bit
Instruction width	8-16 bits	16 bits	32 bits
Architecture	Accumulator	Load/store	Load/store
Registers	16 × 4-bit	16 × 4-bit	138 × 32-bit
Pipeline	None	5-stage	2-stage
CPI (average)	8.0	1.2	1.3

Table 1.2: Architectural Comparison

Chapter 2

Programmer's Model

2.1 Register File

RISC-4 provides 16 general-purpose registers, each 4 bits wide. While all registers are architecturally equivalent, software conventions assign specific roles to certain registers.

Register	Name	Usage (Convention)
`r0`	zero	Hardwired to zero
`r1`	ra	Return address
`r2`	a0	Argument 0 / return value
`r3`	a1	Argument 1
`r4`	a2	Argument 2
`r5`	a3	Argument 3
`r6`	v0	Return value
`r7`	t0	Temporary (caller-saved)
`r8`	t1	Temporary (caller-saved)
`r9`	t2	Temporary (caller-saved)
`r10`	s0	Saved (callee-saved)
`r11`	s1	Saved (callee-saved)
`r12`	s2	Saved (callee-saved)
`r13`	s3	Saved (callee-saved)
`r14`	sp	Stack pointer
`r15`	fp	Frame pointer

Table 2.1: Register File Organization

2.1.1 Register `r0` (zero)

Register `r0` is hardwired to the constant value zero:

- Reads always return `0x0`
- Writes are silently discarded
- Useful for: comparisons, clearing registers, no-op destinations

2.1.2 Value Ranges

Each register holds values in the range:

- Unsigned: 0 to 15
- Signed (two's complement): -8 to +7

2.2 Effective Addressing & Register Pairs

To address memory beyond the 4-bit limit of a single register, RISC-4 utilizes **Register Pairing**. When a register is used as a base address for memory operations, the processor implicitly accesses the specified register and its neighbor to form an 8-bit address.

Pointer Name	High Nibble (4-bit)	Low Nibble (4-bit)
Pair 0	r0	r1
Pair 2	r2	r3
...	...	...
Frame Pointer (fp)	r12	r13
Stack Pointer (sp)	r14	r15

Table 2.2: Register Pair Organization

Rules for Pairing:

1. **Alignment:** Base registers in load/store instructions must be even-numbered (e.g., r14, r12).
2. **Formation:** `Address = (High_Reg « 4) | Low_Reg`.
3. **Range:** This allows for a Data Address Space of 256 nibbles (`0x00` to `0xFF`).

Chapter 3

Instruction Formats

All RISC-4 instructions are exactly 16 bits wide, divided into four formats based on instruction type.

3.1 Format Summary

The RISC-4 ISA uses fixed 16-bit instructions aligned on 16-bit boundaries. The primary opcode is always located in the most significant 4 bits [15:12].

Figure 3.1 illustrates how the 16 bits are utilized across the different instruction types.

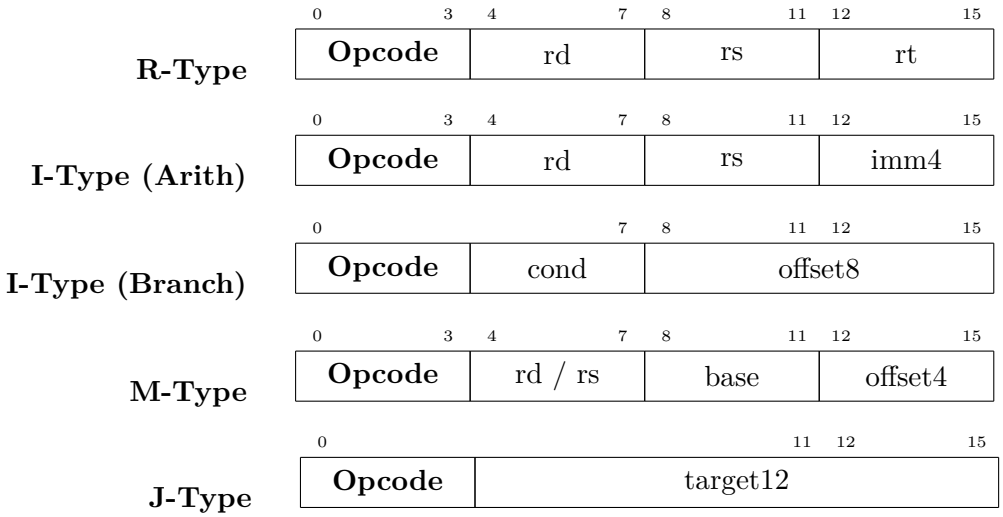


Figure 3.1: RISC-4 Instruction Format Summary

Field Definitions:

- **Opcode:** Primary instruction operation code [15:12].
- **rd:** Destination register operand [11:8].
- **rs:** Source register 1 operand [7:4] (or [11:8] for store).
- **rt:** Source register 2 operand [3:0] (R-Type only).
- **imm4/offset4:** 4-bit immediate value or memory offset [3:0].
- **base:** Register pair base address pointer [7:4].

- **cond:** Branch condition code [11:8] (replaces rd).
- **offset8:** 8-bit signed branch offset [7:0].
- **target12:** 12-bit absolute jump address [11:0].

Chapter 4

Status Flags

4.1 Quick Reference: Flag Behavior

Instruction	C	Z	Notes
Arithmetic			
ADD, ADDI	Set	Set	C = carry out, Z = result is zero
SUB	Set	Set	C = borrow occurred
ADC	Set	Set	Includes previous carry
SBB	Set	Set	Includes previous borrow
NEG	Set	Set	C = 0 only when negating zero
Logical			
AND, ANDI	0	Set	Logical AND, carry cleared
OR, ORI	0	Set	Logical OR, carry cleared
XOR	0	Set	Logical XOR, carry cleared
Comparison			
SLT, SLTI	0	Set	Z = comparison is false
Shift			
SHF	Set	Set	C = last bit shifted out
Memory & Control			
LW, SW	—	—	Flags preserved (not modified)
BEQ, BNE, BCS, BCC	—	—	Flags read but not modified
J	—	—	Flags preserved

Table 4.1: Flag Update Summary

Legend:

- **Set**: Flag updated based on result
- **0**: Flag cleared (forced to 0)
- —: Flag preserved (unchanged)

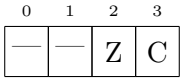
4.2 Overview

RISC-4 maintains a 2-bit status flag register that captures the results of arithmetic and logical operations. While pure academic RISC designs avoid condition codes, RISC-4 follows the commercial RISC tradition (ARM, SPARC, PowerPC) of using flags for efficient comparison and branching.

This design choice reflects the historical context: in the 4-bit microprocessor era of the early 1970s, all architectures used condition codes. RISC-4 demonstrates that RISC principles are compatible with flag-based conditional execution.

4.3 Flag Register

The status register contains two flags:



4.3.1 Carry Flag (C) - Bit 0

The Carry flag indicates unsigned arithmetic overflow or underflow.

Set Conditions:

- **Addition (ADD, ADDI, ADC):** Set if result overflows 4 bits (carry out of bit 3)
  - Example: `0xF + 0x1 = 0x10` → Result `0x0`, C=1
- **Subtraction (SUB, SBB):** Set if borrow occurs (inverted borrow)
  - Example: `0x3 - 0x5 = -2` → Result `0xE`, C=1
- **Shift (SHF):** Contains the last bit shifted out
  - Shift left: C ← original bit 3
  - Shift right: C ← original bit 0

Used By:

- **BCS** (Branch if Carry Set): Branch if C=1
- **BCC** (Branch if Carry Clear): Branch if C=0
- **ADC** (Add with Carry): Adds previous carry into result
- **SBB** (Subtract with Borrow): Subtracts previous borrow from result

Multi-Precision Arithmetic:

The Carry flag enables arithmetic on values larger than 4 bits:

Listing 4.1: 8-bit Addition: r0:r1 = r2:r3 + r4:r5

```
ADD r1, r3, r5      # Add low nibbles: r1 = r3 + r5, sets C
ADC r0, r2, r4      # Add high nibbles with carry: r0 = r2 + r4 + C
```

4.3.2 Zero Flag (Z) - Bit 1

The Zero flag indicates whether the result of an operation is zero.

Set Conditions:

- Set (Z=1) if result equals `0x0`
- Clear (Z=0) if result is non-zero

Updated By:

- All arithmetic operations (ADD, SUB, ADDI, ADC, SBB, NEG)
- All logical operations (AND, OR, XOR, ANDI, ORI)
- Comparison (SLT, SLTI)
- Shift (SHF)

Used By:

- **BEQ** (Branch if Equal): Branch if Z=1
- **BNE** (Branch if Not Equal): Branch if Z=0

Comparison Pattern:

Testing equality requires a subtraction followed by a zero test:

Listing 4.2: If (r1 == r2) goto equal

```
SUB r3, r1, r2        # r3 = r1 - r2, sets Z if equal
BEQ equal             # Branch if Z=1 (difference was zero)
```

4.4 Flag Update Rules

4.4.1 Instructions That Set Flags

Instruction	C	Z	Notes
ADD, ADDI	Yes	Yes	C = carry out, Z = result is zero
SUB	Yes	Yes	C = borrow, Z = result is zero
AND, ANDI	No	Yes	Logical ops don't affect carry
OR, ORI	No	Yes	
XOR	No	Yes	
SLT, SLTI	No	Yes	Z reflects comparison result
SHF	Yes	Yes	C = last bit shifted out
ADC, SBB	Yes	Yes	Extended arithmetic
NEG	Yes	Yes	C set unless negating zero

Table 4.2: Flag Update Behavior

4.4.2 Instructions That Preserve Flags

The following instructions **do not modify** flags:

- **Memory operations:** LW, SW
- **Control flow:** BEQ, BNE, BCS, BCC, J
  - Note: Branches *read* flags but do not modify them

Critical Implication: Load/store operations between an ALU instruction and a branch will preserve flags:

Listing 4.3: Flags preserved across load

```
SUB r3, r1, r2      # Sets Z flag
LW  r4, 0(r14)      # Does NOT change Z flag
BEQ equal           # Still uses Z from SUB
```

4.5 Comparison Operations

4.5.1 Equality (==, !=)

Listing 4.4: Test if r1 == r2

```
SUB r3, r1, r2      # r3 = r1 - r2
BEQ equal           # Branch if Z=1 (r1 == r2)
BNE not_equal       # Branch if Z=0 (r1 != r2)
```

4.5.2 Unsigned Comparison (<, >=)

For unsigned values (0 to 15):

Listing 4.5: Test if r1 < r2 (unsigned)

```
SUB r3, r1, r2      # r3 = r1 - r2
BCS less            # Branch if C=1 (borrow occurred, r1 < r2)
BCC greater_equal   # Branch if C=0 (no borrow, r1 >= r2)
```

4.5.3 Signed Comparison (<, >=)

For signed values (-8 to +7), use SLT:

Listing 4.6: Test if r1 < r2 (signed)

```
SLT r3, r1, r2      # r3 = (r1 < r2) ? 1 : 0
BNE less            # Branch if r3 != 0 (comparison true)
BEQ greater_equal   # Branch if r3 == 0 (comparison false)
```

Rationale: Signed comparison requires checking both the sign bit and magnitude. The SLT instruction implements the full signed comparison logic, outputting a boolean result that can be tested with the zero flag.

4.6 Pipeline Considerations

4.6.1 Flag Hazards

Since flags are updated by instructions in the Execute (EX) stage but may be needed by branches in the Instruction Decode (ID) stage, a **flag hazard** can occur:

```
Cycle:  1    2    3    4    5
SUB     IF   ID   EX   MEM  WB   <- Sets Z flag in cycle 3
BEQ          IF   ID   EX   MEM  <- Needs Z flag in cycle 3
```

Problem: BEQ in ID stage (cycle 3) needs the Z flag, but SUB only produces it in EX stage (also cycle 3). The flag value must be *forwarded* from the EX stage output to the ID stage input in the same cycle.

4.6.2 Solutions

Flag Forwarding (Performance)

Forward flag values directly from EX stage to branch evaluation logic in ID stage:

- **Hardware:** 2-bit bypass path (C and Z flags)
- **Delay:** 0 cycles (resolved within cycle)
- **Complexity:** Low (simpler than register forwarding)

Pipeline Stall (Simplicity)

Detect flag hazard and insert 1-cycle bubble:

- **Detection:** If (previous instruction sets flags) AND (current instruction is branch)
- **Action:** Stall IF and ID stages for 1 cycle
- **Delay:** 1 cycle per flagged branch

Compiler Optimization: Insert independent instruction between comparison and branch:

Listing 4.7: Avoiding stall with instruction scheduling

```
SUB r3, r1, r2      # Sets flags
LW  r4, 0(r14)      # Independent load (fills bubble)
BEQ equal           # Flags ready by this point
```

4.7 Design Rationale

4.7.1 Why RISC-4 Uses Flags

While some RISC designs (MIPS, RISC-V) avoid condition codes, RISC-4 adopts flags for several reasons:

1. Historical Authenticity

- Every 4-bit and 8-bit microprocessor of the 1970s used flags
- Intel 4004 (1971) had Carry and Accumulator Zero flags
- RISC-4 represents "RISC in 1971" flags were the standard then

2. Commercial RISC Precedent

- ARM (1985): NZCV flags, most successful RISC architecture
- SPARC (1987): Integer condition codes
- PowerPC (1990): Condition register
- All three use flags despite being RISC

3. Code Density

- Saves instructions: SUB + BEQ vs SUB + SLT + BNE
- Critical for 4-bit architecture with limited memory
- Typical comparison requires 2 instructions instead of 3

4. Pipeline Equivalence

- Flag hazards and register hazards have identical solutions
- Both require forwarding or stalling
- No performance disadvantage compared to flagless design

5. Simplicity

- Only 2 flags (C and Z), not 8+ like x86
- Clear update rules: ALU sets, Load/Store preserves
- Intuitive for programmers familiar with assembly

4.7.2 What RISC-4 Avoids

RISC-4's flag implementation avoids the complexity criticized in CISC designs:

- **No implicit flag updates:** Load/store don't set flags (unlike x86's MOV)
- **No complex flags:** No overflow (V), parity (P), or auxiliary carry (AC)
- **No every-instruction updates:** Only ALU operations affect flags
- **No condition code register:** Flags are hardware state, not GPRs

This keeps the design simple, regular, and pipeline-friendly — core RISC principles.

Chapter 5

Instruction Set Reference

5.1 Instruction Summary

Table 5.1 lists all RISC-4 instructions.

Opcode	Mnemonic	Format	Description
`0x0`	ADD	R	Add
`0x1`	SUB	R	Subtract
`0x2`	AND	R	Bitwise AND
`0x3`	OR	R	Bitwise OR
`0x4`	XOR	R	Bitwise XOR
`0x5`	SLT	R	Set less than
`0x6`	SHF	I	Shift (Left/Right) Immediate
`0x7`	EXT	R	Extended Math (ADC, SBB, NEG)
`0x8`	ADDI	I	Add immediate
`0x9`	ANDI	I	AND immediate
`0xA`	ORI	I	OR immediate
`0xB`	SLTI	I	Set less than immediate
`0xC`	LW	M	Load word
`0xD`	SW	M	Store word
`0xE`	Bxx	I	Conditional Branch
`0xF`	J	J	Unconditional Jump

Table 5.1: RISC-4 Opcode Map

5.2 Arithmetic Instructions

5.2.1 ADD – Add

```
ADD rd, rs, rt
```

Field	Value
Opcode	`0x0`
Format	R-type
Encoding	`0000 dddd ssss tttt`
Operation	$rd \leftarrow rs + rt$
Flags	C, Z
Cycles	4 (single-cycle impl: 1)

Description: Adds the contents of registers `rs` and `rt`, storing the 4-bit result in `rd`. The carry-out is stored in the **Carry** flag. If result is zero then the **Zero** flag is set.

Flags:

- **C** ← Carry out from bit 3 (unsigned overflow)
- **Z** ← 1 if result == 0, else 0

Examples:

```
ADD r1, r2, r3        # r1 = r2 + r3
# If r2 = 0xF, r3 = 0x1: result = 0x0, C = 1, Z = 1
# If r2 = 0x7, r3 = 0x2: result = 0x9, C = 0, Z = 0
```

5.2.2 SUB – Subtract

```
SUB rd, rs, rt
```

Field	Value
Opcode	`0x1`
Format	R-type
Encoding	`0001 dddd ssss tttt`
Operation	$rd \leftarrow rs - rt$
Flags	C, Z
Cycles	4 (single-cycle impl: 1)

Description: Subtracts the contents of registers `rs` and `rt`, storing the 4-bit result in `rd`. The carry-out is stored in the **Carry** flag. If the result is Zero the **Zero** Flag is set

Flags:

- **C** ← 1 if borrow occurred (unsigned underflow), 0 otherwise
- **Z** ← 1 if result == 0, else 0

Examples:

```
SUB r4, r5, r6        # r4 = r5 - r6
# If r5 = 0x3, r6 = 0x5: result = 0xE (-2), C = 1, Z = 0
# If r5 = 0x8, r6 = 0x8: result = 0x0, C = 0, Z = 1
```

Note: The Z flag is set when operands are equal, making SUB useful for comparisons.

5.2.3 AND – Bitwise AND

```
AND rd, rs, rt
```

Field	Value
Opcode	`0x2`
Format	R-type
Encoding	`0010 dddd ssss tttt`
Operation	$rd \leftarrow rs \& rt$
Flags	Carry, Zero
Cycles	4 (single-cycle impl: 1)

Description: Bitwise AND compare operation on registers **rs** and **rt**, results stored in **rd**. **Carry** flag cleared, **Zero** flag set if result == **0**

Flags:

- **C** ← 0 (cleared)
- **Z** ← 1 if result == 0, else 0

Examples:

```
AND r7, r8, r9       # r7 = r8 & r9
# If r8 = 0b1010, r9 = 0b1100: result = 0b1000, C = 0, Z = 0
# If r8 = 0b1010, r9 = 0b0101: result = 0b0000, C = 0, Z = 1
```

Use Case: Test if specific bits are set:

```
AND r1, r2, r0       # Test if r2 is zero
BEQ  is_zero         # Branch if Z = 1
```

5.2.4 OR – Bitwise OR

```
OR rd, rs, rt
```

Field	Value
Opcode	`0x3`
Format	R-type
Encoding	`0011 dddd ssss tttt`
Operation	$rd \leftarrow rs \mid rt$
Flags	–
Cycles	4 (single-cycle impl: 1)

Description: Bitwise OR compare operation on registers **rs** and **rt**, results store in **rd**. **Carry** flag cleared, **Zero** flag set if result == **0**

Flags:

- **C** ← 0 (cleared)
- **Z** ← 1 if result == 0, else 0

Examples:

```
OR r2, r3, r0        # r2 = r3 | 0 = r3 (MOV pseudo-op)
# Z = 1 if r3 was 0, else Z = 0
# C = 0 always

OR r1, r1, r1        # Explicit way to clear C, update Z
```

5.2.5 XOR – Bitwise Exclusive OR

```
XOR rd, rs, rt
```

Field	Value
Opcode	`0x4`
Format	R-type
Encoding	`0100 dddd ssss tttt`
Operation	$rd \leftarrow rs \veebar rt$
Flags	–
Cycles	4 (single-cycle impl: 1)

Description: Bitwise eXclusive OR compare operation on registers **rs** and **rt**, results store in **rd**. **Carry** flag cleared, **Zero** flag set if result == **0**

Flags:

- **C** ← 0 (cleared)
- **Z** ← 1 if result == 0, else 0

Examples:

```
XOR r1, r1, r1       # Clear r1 (result = 0)
# r1 = 0, C = 0, Z = 1

XOR r2, r3, r4       # r2 = r3 ^ r4
# Z = 1 if r3 == r4, else Z = 0
# C = 0 always
```

Use Case: XOR with self is an idiomatic way to clear a register and set Z flag.

5.2.6 SLT – Set Less Than (Signed)

```
SLT rd, rs, rt
```

Field	Value
Opcode	`0x5`
Format	R-type
Encoding	`0101 dddd ssss tttt`
Operation	$rd \leftarrow (rs < rt)?1 : 0(signedcomparison)$
Flags	–
Cycles	4 (single-cycle impl: 1)

Description: Set Less Than performs a signed comparison of **rs** and **rt** if **rs** is less than **rt** then **rd** is set to 1, if not **rd** is set to 0. **Carry** flag cleared, **Zero** flag set if result == **0**

Flags:

- **C** ← 0 (cleared)
- **Z** ← 1 if result == 0 (comparison false), else 0

Examples:

```
SLT r1, r2, r3       # r1 = (r2 < r3) ? 1 : 0
# If r2 = 0xF (-1), r3 = 0x1 (1): r1 = 1, C = 0, Z = 0
# If r2 = 0x5 (5),  r3 = 0x3 (3): r1 = 0, C = 0, Z = 1

# Use with branch:
SLT r1, r2, r3       # Test r2 < r3
BNE less_than        # Branch if Z = 0 (comparison true)
BEQ greater_equal    # Branch if Z = 1 (comparison false)
```

5.3 Shift Instruction

5.3.1 SHF – Shift

```
SHF rd, rs, imm4
```

Field	Value
Opcode	`0x6`
Format	I-type
Encoding	`0110 dddd ssss iiii`
Operation	$rd \leftarrow shift(rs, imm4)$
Flags	C, Z
Cycles	4

Description: Shifts the value in rs by the amount and direction specified in imm4.

Immediate Encoding:

- **Bit 3:** Direction (0 = left, 1 = right)
- **Bits 2-0:** Shift amount (0-7)

Operation:

- **Shift Left:** rd ← rs « amount, fill with 0
- **Shift Right:** rd ← rs » amount, fill with 0 (logical)

Flags:

- **C** ← Last bit shifted out
  - Shift left: C ← bit 3 of original value (before shift)
  - Shift right: C ← bit 0 of original value (before shift)
  - For multi-bit shifts, C contains the last bit shifted out
- **Z** ← 1 if result == 0, else 0

Examples:

```
SHF r1, r2, 0x1      # Shift left by 1: r1 = r2 << 1
# If r2 = 0b1010: r1 = 0b0100, C = 1, Z = 0

SHF r1, r2, 0x9      # Shift right by 1: r1 = r2 >> 1
# If r2 = 0b1010: r1 = 0b0101, C = 0, Z = 0

SHF r1, r2, 0xA      # Shift right by 2: r1 = r2 >> 2
# If r2 = 0b1100: r1 = 0b0011, C = 0, Z = 0

SHF r1, r2, 0x4      # Shift left by 4 (clear register)
# r1 = 0b0000, C = ?, Z = 1
```

Use Cases:

```
# Multiply by 2 (shift left 1)
SHF r1, r2, 0x1      # r1 = r2 * 2

# Divide by 2 (shift right 1, unsigned)
SHF r1, r2, 0x9      # r1 = r2 / 2

# Multi-precision shift: shift 8-bit value r0:r1 left
SHF r1, r1, 0x1      # Shift low nibble left, C = old bit 3
# (Need to implement rotate through carry for high nibble)
```

Note: RISC-4 does not have arithmetic right shift (sign-extension). For signed division by 2, additional instructions are needed to handle the sign bit.

5.4 Extended Arithmetic Instructions

These instructions share Opcode 7 and use a 2-operand destructive format (`rd` is both source and destination) to allow for a 4-bit function code.

5.4.1 ADC – Add with Carry

```
ADC rd, rs
```

Field	Value
Opcode	`0x7`
Format	R-type
Encoding	`0111 dddd ssss 0000`
Operation	$rd \leftarrow rd + rs + Carry$
Flags	Carry
Cycles	4

Description: Adds `rs` and the current **Carry** flag to `rd` **Zero** flag set if result == **0**. Used for multi-nibble addition.

Flags:

- **C** ← Carry out from bit 3
- **Z** ← 1 if result == 0, else 0

Multi-Precision Example:

```
# 8-bit addition: r0:r1 = r2:r3 + r4:r5
ADD r1, r3, r5      # Low nibble: r1 = r3 + r5, sets C
ADC r0, r2, r4      # High nibble: r0 = r2 + r4 + C
# Final Z flag reflects whether entire 8-bit result is zero
```

5.4.2 SBB – Subtract with Borrow

```
SBB rd, rs
```

Field	Value
Opcode	`0x7`
Format	R-type
Encoding	`0111 dddd ssss 0001`
Operation	$rd \leftarrow rd - rs - Carry$
Flags	Carry
Cycles	4

Description: Subtracts `rs` and the Carry flag (borrow) from `rd` **Zero** flag set if result == **0**. Used for multi-nibble subtraction.

Flags:

- **C** ← 1 if borrow occurred, else 0
- **Z** ← 1 if result == 0, else 0

Multi-Precision Example:

```
# 8-bit subtraction: r0:r1 = r2:r3 - r4:r5
SUB r1, r3, r5      # Low nibble: r1 = r3 - r5, sets C if borrow
SBB r0, r2, r4      # High nibble: r0 = r2 - r4 - C
# Z flag reflects whether entire 8-bit result is zero
```

5.4.3 NEG – Negate (Two's Complement)

```
NEG rd, rs
```

Field	Value
Opcode	`0x7`
Format	R-type
Encoding	`0111 dddd ssss 0010`
Operation	$rd \leftarrow 0 - rs$
Flags	C, Z
Cycles	4

Description: Computes the two's complement of `rs` and stores it in `rd`. (Pseudo-equivalent to `SUB rd, r0, rs`). **Carry** flag set to 0 if operand was zero else **Carry** set to 1. **Zero** flag set if result == **0**

Flags:

- **C** ← 0 if operand was zero, else 1
- **Z** ← 1 if result == 0 (only when negating zero), else 0

Examples:

```
NEG r1, r2          # r1 = -r2 (two's complement)
# If r2 = 0x5 (5):  r1 = 0xB (-5), C = 1, Z = 0
# If r2 = 0x0 (0):  r1 = 0x0 (0),  C = 0, Z = 1
# If r2 = 0x8 (-8): r1 = 0x8 (-8), C = 1, Z = 0 (overflow!)
```

Note: Negating -8 produces -8 due to two's complement overflow in 4-bit signed arithmetic.

5.5 Immediate Arithmetic Instructions

5.5.1 ADDI – ADD Immediate

```
ADDI rd,rs, imm4
```

Field	Value
Opcode	`0x8`
Format	I-type
Encoding	`1000 dddd ssss iiii`
Operation	$rd \leftarrow rs + sign_extend(imm4)$
Flags	C, Z
Cycles	4 (single-cycle impl: 1)

Description: ADD Immediate adds the sign-extended 4-bit immediate `imm4` to the contents of register `rs`, storing the result in `rd`. This allows for incrementing and decrementing (using negative immediates).

Flags:

- **C** ← Carry out from bit 3
- **Z** ← 1 if result == 0, else 0

Examples:

```
ADDI r1, r2, 5       # r1 = r2 + 5
# If r2 = 0xB (11): r1 = 0x0 (0), C = 1, Z = 1

ADDI r3, r3, -1      # r3 = r3 - 1 (decrement)
# If r3 = 0x1: r3 = 0x0, C = 0, Z = 1
# If r3 = 0x0: r3 = 0xF (-1), C = 1, Z = 0
```

Note on Negative Immediates: Since the immediate field is 4 bits wide, it can represent values from -8 to +7. The processor treats the binary value as a signed two's complement number.

- **Subtraction:** To subtract 1, use the immediate `0xF` (`0b1111`), which represents -1 in two's complement.
- **Calculation:** `0b0011` (3) + `0b1111` (-1) = `0b10010` → Result `0b0010` (2), Carry Set.

5.5.2 ANDI – AND Immediate

```
ANDI rd, rs, imm4
```

Field	Value
Opcode	`0x9`
Format	I-type
Encoding	`1001 dddd ssss iiii`
Operation	$rd \leftarrow rs \;\&\; zero\_extend(imm4)$
Flags	C, Z
Cycles	4

Description: Performs a bitwise AND between register `rs` and the 4-bit immediate, storing the result in `rd`. **Note:** Since the register is 4 bits, the mask `imm4` covers all bits. **Carry** flag is cleared, **Zero** flag is set if result == **0**

Flags:

- **C** ← 0 (cleared)
- **Z** ← 1 if result == 0, else 0

Examples:

```
ANDI r1, r2, 0x1     # Extract bit 0 of r2
# If r2 = 0b1010: r1 = 0b0000, C = 0, Z = 1
# If r2 = 0b1011: r1 = 0b0001, C = 0, Z = 0

ANDI r1, r2, 0x8     # Extract bit 3 (sign bit)
BNE  negative        # Branch if Z = 0 (bit was set)
```

5.5.3 ORI – OR Immediate

```
ORI rd, rs, imm4
```

Field	Value
Opcode	`0xA`
Format	I-type
Encoding	`1010 dddd ssss iiii`
Operation	$rd \leftarrow rs \mid zero\_extend(imm4)$
Flags	C, Z
Cycles	4

Description: Performs a bitwise OR between register `rs` and the 4-bit immediate, storing the result in `rd`. **Carry** flag cleared, **Zero** flag set if result == **0**

Flags:

- **C** ← 0 (cleared)
- **Z** ← 1 if result == 0, else 0

Examples:

```
ORI r1, r0, 0x5      # Load immediate: r1 = 0x5
# r1 = 5, C = 0, Z = 0

ORI r2, r3, 0x8      # Set bit 3 (sign bit)
# Z = 0 (result always non-zero since bit 3 set)
```

5.5.4 SLTI – Set Less Than Immediate

```
SLTI rd, rs, imm4
```

Field	Value
Opcode	`0xB`
Format	I-type
Encoding	`1011 dddd ssss iiii`
Operation	$rd \leftarrow (rs < sign\_extend(imm4))?1 : 0$
Flags	C, Z
Cycles	4

Description: Compares `rs` to the signed 4-bit immediate. If `rs` is strictly smaller, `rd` is set to 1, otherwise 0. **Carry** flag cleared, **Zero** flag set if result == **0**

Flags:

- **C** ← 0 (cleared)
- **Z** ← 1 if result == 0 (comparison false), else 0

Examples:

```
SLTI r1, r2, 5       # r1 = (r2 < 5) ? 1 : 0
# If r2 = 0x3 (3):   r1 = 1, C = 0, Z = 0 (true)
# If r2 = 0x7 (7):   r1 = 0, C = 0, Z = 1 (false)
# If r2 = 0xF (-1): r1 = 1, C = 0, Z = 0 (true, signed)
```

5.6 Memory Instructions

5.6.1 LW – Load Word

```
LW rd, offset(base)
```

Field	Value
Opcode	`0xC`
Format	M-type
Encoding	`1100 dddd bbbb oooo`
Operation	$rd \leftarrow mem[reg\_pair(base) + sign\_extend(offset)]$
Flags	–
Cycles	5

Description: Loads a 4-bit nibble from data memory into `rd`. **Addressing:** The effective address is calculated by taking the 8-bit value in the register pair specified by `base` (must be even) and adding the sign-extended 4-bit `offset`.

Example:

```
# Assume r14=0x8, r15=0x0 (Stack Pointer = 0x80)
LW r1, 4(r14)        # Address = 0x80 + 4 = 0x84
                     # r1 gets value at mem[0x84]
```

5.6.2 SW – Store Word

```
SW rs, offset(base)
```

Field	Value
Opcode	`0xD`
Format	M-type
Encoding	`1101 ssss bbbb oooo`
Operation	$mem[reg\_pair(base) + sign\_extend(offset)] \leftarrow rs$
Flags	–
Cycles	5

Description: Stores the 4-bit value from `rs` into data memory. **Note:** `base` must be an even-numbered register representing a pair.

Example:

```
# Push r2 to stack
# 1. Decrement SP (r14:r15) - Requires multi-step math or software
    convention
# 2. Store
SW r2, 0(r14)       # mem[sp] = r2
```

5.7 Control Flow Instructions

5.7.1 Bxx – Conditional Branch

```
BEQ offset8    # Branch if Equal (Z=1)
BNE offset8    # Branch if Not Equal (Z=0)
BCS offset8    # Branch if Carry Set (C=1)
BCC offset8    # Branch if Carry Clear (C=0)
```

Field	Value
Opcode	`0xE`
Format	I-type
Encoding	`1110 cccc oooooooo`
Operation	$if(cond) pc \leftarrow pc + sign\_extend(offset8)$
Flags	–
Cycles	3

Description: Branches relative to the current PC if the condition specified by the 4-bit `cond` field is met.

- **Encoding (cccc):** 0000 (EQ), 0001 (NE), 0010 (CS/LT), 0011 (CC/GE).
- **Offset:** 8-bit signed offset allows jumping forward/backward by 128 nibbles.

5.7.2 J – Jump (Unconditional)

```
J target12
```

Field	Value
Opcode	`0xF`
Format	J-type
Encoding	`1111 aaaa aaaa aaaa`
Operation	$pc \leftarrow target12$
Flags	–
Cycles	3

Description: Unconditional absolute jump to the 12-bit address `target12`. This allows jumping to any location in the 4096-nibble instruction memory.

Chapter 6

Pipeline Architecture

6.1 Overview

RISC-4 implements a classic 5-stage RISC pipeline designed for simplicity and efficiency. The pipeline enables an ideal throughput of one instruction per cycle (CPI = 1.0) while maintaining a straightforward control path suitable for educational implementation and FPGA synthesis.

6.2 Pipeline Stages

6.2.1 Stage 1: Instruction Fetch (IF)

Duration: 1 cycle

Operations:

- Fetch 16-bit instruction from instruction memory at address PC
- Increment PC by 1 (RISC-4 uses nibble addressing, instructions are 4 nibbles = 16 bits)
- Forward instruction to IF/ID pipeline register

Critical Path:

```
1 PC $\rightarrow$ Instruction Memory $\rightarrow$ Instruction Register $\
    rightarrow$ IF/ID Register
```

Stall Conditions:

- Instruction memory not ready (cache miss in advanced implementations)
- Pipeline stall propagated from later stages

6.2.2 Stage 2: Instruction Decode (ID)

Duration: 1 cycle

Operations:

- Decode opcode [15:12] to determine instruction type
- Read up to two source registers from register file
- Sign-extend or zero-extend immediate values
- Generate control signals for subsequent stages
- Evaluate branch conditions (for conditional branches)

Register File Access:

```
Read Port 1: rs [7:4]
Read Port 2: rt [3:0] (R-type) or imm4 [3:0] (I-type)
```

Branch Evaluation:

- For conditional branches (BEQ, BNE, BCS, BCC), condition is checked in ID stage
- Branch target address calculated: PC + sign_extend(offset8)
- If branch taken, flush IF stage and redirect PC

Stall Conditions:

- Load-use hazard: Previous instruction is LW and current instruction uses the loaded register
- Flag hazard: Previous instruction sets flags and current instruction is a conditional branch

6.2.3 Stage 3: Execute (EX)

Duration: 1 cycle

Operations:

- Perform ALU operation (ADD, SUB, AND, OR, XOR, SLT, shifts)
- Calculate memory address for load/store (base + offset)
- Update Carry (C) and Zero (Z) flags based on ALU result
- Forward result to ID stage (for forwarding) and to EX/MEM register

ALU Operations:

Instruction Type	ALU Operation
Arithmetic (ADD, SUB, ADC, SBB)	4-bit adder/subtractor
Logical (AND, OR, XOR)	Bitwise operations
Comparison (SLT, SLTI)	Signed comparison
Shift (SHF)	Barrel shifter (0-7 positions)
Memory (LW, SW)	Address calculation

Flag Generation:

- Carry flag: Set by arithmetic/shift operations
- Zero flag: Set if ALU result == 0x0
- Flags forwarded to ID stage for immediate use by branches

6.2.4 Stage 4: Memory Access (MEM)

Duration: 1 cycle

Operations:

- Load: Read 4-bit nibble from data memory at calculated address
- Store: Write 4-bit nibble to data memory at calculated address
- Non-memory instructions: Pass ALU result through unchanged

Memory Interface:

```
Address: 8-bit (from register pair + offset)
Data: 4-bit nibble
Control: Read enable, Write enable
```

Memory Map:

- Data memory: 256 nibbles (0x00 - 0xFF)
- Accessed via 8-bit addresses formed by register pairs

6.2.5 Stage 5: Write Back (WB)

Duration: 1 cycle

Operations:

- Select result source (ALU result or memory data)
- Write result to destination register rd
- Register r0 writes are ignored (hardwired to zero)

Write Sources:

Instruction Type	Write Source
Arithmetic/Logical	ALU result from EX stage
Load (LW)	Memory data from MEM stage
Store (SW), Branches, Jump	No write (rd unused)

6.3 Pipeline Registers

Data is passed between stages via pipeline registers:

Register	Contents
IF/ID	Instruction[15:0], PC+1
ID/EX	Control signals, Register values (rs, rt), Immediate, PC+1, rd address
EX/MEM	Control signals, ALU result, rt value (for stores), rd address, Flags
MEM/WB	Control signals, Memory data or ALU result, rd address

6.4 Hazards and Solutions

6.4.1 Data Hazards

RAW (Read After Write) Hazards

Problem: Instruction needs a register that hasn't been written yet.

Example:

```
ADD r3, r1, r2    # r3 written in WB stage (cycle 5)
SUB r4, r3, r5    # r3 needed in EX stage (cycle 3)
                  # Hazard: r3 not ready when needed
```

Solution 1: Forwarding (Bypassing)

Forward data from later pipeline stages to earlier stages:

- **EX-to-EX forwarding:** Forward ALU result from EX/MEM register back to ALU inputs
- **MEM-to-EX forwarding:** Forward memory result from MEM/WB register to ALU inputs

Forwarding Paths:

```
1. EX/MEM.ALUResult $\rightarrow$ ID/EX (ALU operand)
2. MEM/WB.Result $\rightarrow$ ID/EX (ALU operand)
```

Hardware Cost:

- 2 multiplexers per ALU input (4 total)
- Comparators to detect hazard conditions
- Minimal: 50 gates

Load-Use Hazard

Problem: Load instruction followed immediately by use of loaded data.

Example:

```
LW  r1, 0(r14)    # r1 available in MEM stage (cycle 4)
ADD r3, r1, r2    # r1 needed in EX stage (cycle 3)
                  # Hazard: r1 not ready - 1 cycle gap
```

Solution: 1-Cycle Stall

- Detect hazard in ID stage: (previous instruction is LW) AND (LW.rd == current.rs or current.rt)
- Insert 1-cycle bubble (NOP) into pipeline
- Allow load to complete before use
- Performance: CPI increases by 0.1 (assuming 10% loads with immediate use)

Compiler Optimization:

Insert independent instruction between load and use:

```
LW  r1, 0(r14)    # Load r1
ORI r4, r0, 5     # Independent operation (fills bubble)
ADD r3, r1, r2    # r1 now ready (no stall)
```

Flag Hazards

Problem: Conditional branch needs flags that haven't been computed yet.

Example:

```
ADD r1, r2, r3    # Sets C and Z flags in EX stage (cycle 3)
BEQ target        # Needs Z flag in ID stage (cycle 3)
                  # Hazard: flags not ready
```

Solution 1: Flag Forwarding

Forward flags from EX stage to branch evaluation logic in ID stage:

- Hardware: 2-bit bypass path (C and Z flags)
- Timing: Flags available same cycle they're computed
- Delay: 0 cycles (resolved within cycle)

Solution 2: 1-Cycle Stall

- Detect hazard: (previous instruction sets flags) AND (current instruction is conditional branch)
- Insert 1-cycle stall
- Simpler hardware than forwarding
- Performance cost: 0.05 CPI increase

6.4.2 Control Hazards

Problem: Branch outcome not known until ID stage, but next instruction already fetched.

Branch Penalty

Timeline:

```
Cycle:  1    2    3    4    5
BEQ     IF   ID   EX   MEM  WB   <- Branch taken/not decided in ID
next    IF   XX   --   --   --   <- Fetched but may be wrong
        --   IF   --   --   --   <- Correct instruction if branch taken
```

Penalty: 1 cycle for taken branches (IF stage fetched wrong instruction)

Solution 1: Branch Prediction (Static)

Predict Not Taken:

- Assume all branches fall through
- Continue fetching sequentially
- If branch actually taken: Flush IF stage, redirect PC
- Cost: 1 cycle penalty only when branch taken

Performance:

- Best case (branch not taken): 0 penalty
- Worst case (branch taken): 1 cycle penalty
- Average: 0.5 cycles per branch (assuming 50% taken rate)

Solution 2: Branch Delay Slot

Architecture Change:

- Instruction immediately after branch **always** executes
- Compiler fills delay slot with useful instruction or NOP
- No penalty if slot filled productively

Example:

```
    SUB r3, r1, r2    # Sets flags
    BEQ equal         # Branch
    ADD r4, r5, r6    # DELAY SLOT: Always executes
equal:
    ...
```

RISC-4 Decision: Not implemented in v0.1 (simpler pipeline), but reserved for future versions.

6.5 Performance Analysis

6.5.1 Ideal Performance

- **Ideal CPI:** 1.0 (one instruction per cycle)
- **Clock frequency:** 100 MHz (estimated for SKY130 process)
- **Ideal throughput:** 100 MIPS

6.5.2 Realistic Performance

Hazard Impact:

Hazard Type	Frequency	CPI Impact
Load-use stall	10% of instructions	+0.10
Branch misprediction	15% of instructions, 50% taken	+0.075
Flag stall (if no forwarding)	5% of instructions	+0.05
Total		**+0.225**

Realistic CPI: 1.2 to 1.3

Comparison:

- Intel 4004: CPI = 8.0 (8-16 cycles per instruction)
- RISC-I: CPI = 1.3
- RISC-4: CPI = 1.2 - 1.3

Speedup vs. 4004: 6× to 7× at same clock frequency

6.5.3 Optimization Opportunities

Compiler Techniques:

- Instruction scheduling to avoid load-use hazards
- Loop unrolling to reduce branch frequency
- Register allocation to minimize memory access

Hardware Enhancements (Future):

- Branch delay slots (reduce branch penalty to 0)
- Dynamic branch prediction (reduce misprediction rate)
- Dual-ported register file (eliminate some forwarding delays)

6.6 Implementation Considerations

6.6.1 Critical Paths

Longest paths in each stage:

Stage	Critical Path
IF	PC → Instruction Memory → IF/ID Register
ID	Register File Read → Sign Extend → ID/EX Register
EX	ALU Input Mux → 4-bit Adder → Flag Generation → EX/MEM Register
MEM	Address → Data Memory → MEM/WB Register
WB	Result Mux → Register File Write

Design Goal: Balance stage delays to maximize clock frequency.

6.6.2 Forwarding Logic

EX Hazard Detection:

```
if (EX/MEM.RegWrite && EX/MEM.rd != 0 && EX/MEM.rd == ID/EX.rs)
    ForwardA = 01  // Forward from EX/MEM
if (EX/MEM.RegWrite && EX/MEM.rd != 0 && EX/MEM.rd == ID/EX.rt)
    ForwardB = 01  // Forward from EX/MEM
```

MEM Hazard Detection:

```
if (MEM/WB.RegWrite && MEM/WB.rd != 0 && MEM/WB.rd == ID/EX.rs)
    ForwardA = 10  // Forward from MEM/WB
if (MEM/WB.RegWrite && MEM/WB.rd != 0 && MEM/WB.rd == ID/EX.rt)
    ForwardB = 10  // Forward from MEM/WB
```

6.6.3 Stall Logic

Load-Use Hazard Detection:

```
if (ID/EX.MemRead &&
    ((ID/EX.rd == IF/ID.rs) || (ID/EX.rd == IF/ID.rt)))
    Stall = 1  // Insert bubble, freeze IF and ID stages
```

6.7 Design Rationale

6.7.1 Why 5 Stages?

- **Classic RISC:** Proven design from MIPS, ARM
- **Balanced stages:** Each stage equal delay
- **Simple control:** Straightforward forwarding and hazard detection
- **Educational:** Easy to understand and implement

6.7.2 Alternatives Considered

3-Stage Pipeline (IF, EX, WB):

- Simpler, fewer hazards
- Longer critical path (lower clock frequency)
- Less instructive (too simple for learning)

7-Stage Pipeline (split EX into multiple stages):

- Higher clock frequency potential
- More forwarding complexity
- Overkill for 4-bit datapath

Decision: 5 stages balances performance, complexity, and educational value.

Chapter 7

Addressing

7.1 Overview

RISC-4 faces a fundamental challenge: a 4-bit datapath cannot directly address more than 16 locations (2^4), yet the architecture requires access to:

- **Instruction memory:** 4096 nibbles (12-bit address space)
- **Data memory:** 256 nibbles (8-bit address space)

To bridge this gap, RISC-4 employs **Register Pairing**, where consecutive registers combine to form wider addresses. This approach maintains the simplicity of a 4-bit datapath while enabling practical memory access.

7.2 Memory Organization

7.2.1 Memory Map

RISC-4 uses separate instruction and data memories (Harvard architecture):

Memory Type	Address Range	Size
Instruction Memory (ROM)	0x000 - 0xFFF	4096 nibbles (2048 bytes)
Data Memory (RAM)	0x00 - 0xFF	256 nibbles (128 bytes)

Table 7.1: RISC-4 Memory Map

7.2.2 Address Formation

Instruction Addresses:

- Formed by 12-bit Program Counter (PC)
- Automatically incremented by fetch unit
- Modified by jumps and branches

Data Addresses:

- Formed by combining two consecutive 4-bit registers
- Explicitly specified in load/store instructions
- Calculated as: `Address = (High_Reg « 4) | Low_Reg`

7.3 Register Pairs

7.3.1 Pair Formation Rules

Register pairs combine two consecutive registers to form an 8-bit value:

1. **Base register must be even-numbered** (r0, r2, r4, ..., r14)
2. **High nibble:** Base register (e.g., r14)
3. **Low nibble:** Base register + 1 (e.g., r15)
4. **Formation:** `Value = (r[n] « 4) | r[n+1]`

7.3.2 Standard Register Pairs

Pair	High Nibble	Low Nibble	Conventional Use
r0:r1	r0	r1	General purpose (always reads 0x0n)
r2:r3	r2	r3	General purpose / argument pointer
r4:r5	r4	r5	General purpose
r6:r7	r6	r7	General purpose
r8:r9	r8	r9	General purpose
r10:r11	r10	r11	General purpose
r12:r13	r12	r13	Frame pointer (fp)
r14:r15	r14	r15	Stack pointer (sp)

Table 7.2: Register Pair Organization

Note: Pair r0:r1 always has high nibble = 0 (since r0 is hardwired to zero), limiting it to addresses 0x00-0x0F.

7.3.3 Pair Manipulation

Loading a Constant Address

To set a register pair to a specific address (e.g., 0x84):

```
ORI r14, r0, 0x8     # r14 = 0x8 (high nibble)
ORI r15, r0, 0x4     # r15 = 0x4 (low nibble)
                     # r14:r15 now = 0x84
```

Incrementing a Pointer

To increment r14:r15 by 1:

```
ADDI r15, r15, 1     # r15 = r15 + 1
# If no carry (r15 didn't overflow):
#   Done - pointer incremented
# If carry (r15 overflowed from 0xF to 0x0):
#   Need to increment high nibble:
ADDI r14, r14, 0     # Add 0 to propagate previous carry
# Or use ADC if carry is set:
# ADC r14, r0         # r14 = r14 + 0 + Carry
```

Proper implementation with carry handling:

```
ADDI r15, r15, 1     # Increment low nibble, sets C if overflow
# Check if we need to increment high nibble:
# Option 1: Conditional (if branches supported):
BCC  no_carry        # Skip if no carry
ADDI r14, r14, 1     # Increment high nibble
no_carry:
    # Continue...

# Option 2: Always add carry (simpler):
ADC r14, r0, r0      # r14 = r14 + 0 + C (adds carry unconditionally)
```

Decrementing a Pointer

To decrement r14:r15 by 1:

```
ADDI r15, r15, -1    # r15 = r15 - 1, sets C if borrow
# If borrow (r15 underflowed from 0x0 to 0xF):
BCC no_borrow        # Skip if no borrow
ADDI r14, r14, -1    # Decrement high nibble
no_borrow:
    # Continue...
```

Adding an Offset to a Pointer

To add a multi-nibble offset (e.g., 0x12 = 18 decimal) to r14:r15:

```
# Add 0x12 to r14:r15
ADDI r15, r15, 0x2   # Add low nibble (2), sets C if overflow
ADC  r14, r14, 0x1   # Add high nibble (1) + carry
                     # Result: r14:r15 = original + 0x12
```

7.4 Load/Store Addressing Modes

7.4.1 Base + Offset Addressing

RISC-4 load/store instructions use base-plus-offset addressing:

```
LW rd, offset(base)
SW rs, offset(base)
```

Address Calculation:

```
Effective Address = reg_pair(base) + sign_extend(offset)
```

Where:

- `base`: Even-numbered register (specifies register pair)
- `offset`: 4-bit signed immediate (-8 to +7)
- `reg_pair(base)`: 8-bit value from registers `base` and `base+1`

7.4.2 Addressing Examples

Stack Access

Assuming stack pointer sp (r14:r15) = 0x90:

```
LW   r1, 0(r14)       # Load from [0x90 + 0] = 0x90
LW   r2, 1(r14)       # Load from [0x90 + 1] = 0x91
LW   r3, -1(r14)      # Load from [0x90 + (-1)] = 0x8F
SW   r4, 2(r14)       # Store to [0x90 + 2] = 0x92
```

Array Access

Accessing elements of an array starting at 0x40:

```
# Set up base pointer to array start (0x40)
ORI r10, r0, 0x4     # r10 = 0x4 (high nibble)
ORI r11, r0, 0x0     # r11 = 0x0 (low nibble)
                     # r10:r11 = 0x40

# Access array elements
LW r1, 0(r10)        # array[0] from 0x40
LW r2, 1(r10)        # array[1] from 0x41
LW r3, 2(r10)        # array[2] from 0x42

# Access with negative offset (before base)
LW r4, -1(r10)       # Address 0x3F (before array)
```

Structure Field Access

Accessing fields in a structure at address 0x60:

```
# Structure layout:
#   +0: status  (1 nibble)
#   +1: value   (1 nibble)
#   +2: flags   (1 nibble)

ORI r8, r0, 0x6      # r8 = 0x6
ORI r9, r0, 0x0      # r9 = 0x0
                     # r8:r9 = 0x60 (structure base)

LW r1, 0(r8)         # Load status from 0x60
LW r2, 1(r8)         # Load value from 0x61
LW r3, 2(r8)         # Load flags from 0x62

# Modify a field
ADDI r2, r2, 1       # Increment value
SW r2, 1(r8)         # Store back to 0x61
```

7.5 Multi-Precision Arithmetic

Register pairs enable arithmetic on values larger than 4 bits by chaining operations through the Carry flag.

7.5.1 8-Bit Addition

Add two 8-bit values: (r0:r1) = (r2:r3) + (r4:r5)

```
ADD r1, r3, r5       # Add low nibbles: r1 = r3 + r5
                     # Sets C if sum >= 16
ADC r0, r2, r4       # Add high nibbles with carry:
                     # r0 = r2 + r4 + C
# Result: r0:r1 = (r2:r3) + (r4:r5)
```

Example:

```
r2:r3 = 0x9F (159 decimal)
r4:r5 = 0x23 (35 decimal)

Step 1: r1 = r3 + r5 = 0xF + 0x3 = 0x12 $\rightarrow$ r1 = 0x2, C = 1
Step 2: r0 = r2 + r4 + C = 0x9 + 0x2 + 1 = 0xC
Result: r0:r1 = 0xC2 (194 decimal) $\checkmark$
```

7.5.2 8-Bit Subtraction

Subtract two 8-bit values: (r0:r1) = (r2:r3) - (r4:r5)

```
SUB r1, r3, r5       # Subtract low nibbles: r1 = r3 - r5
                     # Sets C if borrow occurred
SBB r0, r2, r4       # Subtract high nibbles with borrow:
                     # r0 = r2 - r4 - C
# Result: r0:r1 = (r2:r3) - (r4:r5)
```

Example:

```
r2:r3 = 0x84 (132 decimal)
r4:r5 = 0x37 (55 decimal)

Step 1: r1 = r3 - r5 = 0x4 - 0x7 = -3 $\rightarrow$ r1 = 0xD (13), C = 1 (
    borrow)
Step 2: r0 = r2 - r4 - C = 0x8 - 0x3 - 1 = 0x4
Result: r0:r1 = 0x4D (77 decimal) $\checkmark$
```

7.5.3 16-Bit Operations

For 16-bit (4-nibble) values, chain four registers:

```
# 16-bit addition: (r0:r1:r2:r3) = (r4:r5:r6:r7) + (r8:r9:r10:r11)

ADD r3, r7, r11      # Nibble 0 (LSB)
ADC r2, r6, r10      # Nibble 1
ADC r1, r5, r9       # Nibble 2
ADC r0, r4, r8       # Nibble 3 (MSB)
# Result: r0:r1:r2:r3 contains 16-bit sum
```

7.6 Stack Management

7.6.1 Stack Organization

The stack grows downward from high memory toward low memory:

```
High Memory (0xFF)
    $\uparrow$
    | Stack grows downward
    | (SP decrements on push)
    $\downarrow$
Low Memory (0x00)
```

Convention:

- Stack pointer (sp = r14:r15) points to **next free location**
- Push: Decrement SP, then store
- Pop: Load, then increment SP

7.6.2 Push Operation

Push register r1 onto stack:

```
# Decrement stack pointer (r14:r15)
ADDI r15, r15, -1    # Decrement low nibble
BCC  no_borrow       # If no borrow, skip high nibble decrement
ADDI r14, r14, -1    # Decrement high nibble due to borrow
no_borrow:

# Store value
SW r1, 0(r14)        # mem[sp] = r1
```

Optimized (assuming no overflow):

```
ADDI r15, r15, -1    # SP--
SW   r1, 0(r14)      # Push r1
# Note: This assumes SP low nibble doesn't underflow
# Full implementation needs borrow handling
```

7.6.3 Pop Operation

Pop from stack into register r1:

```
# Load value
LW r1, 0(r14)        # r1 = mem[sp]

# Increment stack pointer
ADDI r15, r15, 1     # Increment low nibble
BCC  no_carry        # If no carry, done
ADDI r14, r14, 1     # Increment high nibble due to carry
no_carry:
```

7.6.4 Stack Frame Example

Typical function stack frame layout:

```
High addresses
    [local variable n]  $\leftarrow$ FP + n
    [local variable 2]  $\leftarrow$ FP + 2
    [local variable 1]  $\leftarrow$ FP + 1
    [saved r12]         $\leftarrow$ FP + 0  (old frame pointer)
    [return address]    $\leftarrow$ FP - 1
    [saved registers]   $\leftarrow$ FP - 2, FP - 3, ...
    [free space]        $\leftarrow$ SP (current stack pointer)
Low addresses
```

7.7 Addressing Limitations and Workarounds

7.7.1 Limited Offset Range

Load/store offsets are only 4 bits signed (-8 to +7). To access beyond this range:

Problem: Access memory at base + 20

Solution: Adjust base pointer

```
# Want to access: base_ptr + 20 (0x14)
# But offset limited to -8 to +7

# Option 1: Adjust pointer temporarily
ADDI r11, r11, 0x4   # Add 0x14 to low nibble (will overflow)
ADC  r10, r10, 0x1   # Add 0x1 to high nibble + carry
LW   r1, 0(r10)      # Load from adjusted address

# Restore pointer if needed
ADDI r11, r11, -0x4
SBB  r10, r10, 0x1

# Option 2: Use separate pointer
ORI r8, r0, <high>   # Calculate new base
ORI r9, r0, <low>    # = original + 20
LW  r1, 0(r8)        # Load from offset address
```

7.7.2 Pointer Arithmetic Complexity

Multi-nibble pointer arithmetic requires careful carry handling:

```
# Add 0x2F to pointer r14:r15

# Naive (WRONG - ignores carry):
ADDI r15, r15, 0xF   # Low nibble
ADDI r14, r14, 0x2   # High nibble (WRONG - missed carry)

# Correct (handles carry):
ADDI r15, r15, 0xF   # Low nibble, may set carry
ADC  r14, r14, 0x2   # High nibble + any carry from low
```

7.7.3 r0:r1 Pair Limitation

Since r0 is hardwired to zero, pair r0:r1 can only address 0x00-0x0F:

```
# r0:r1 can only form addresses 0x00 through 0x0F
ORI r1, r0, 0x7      # r1 = 0x7
LW  r2, 0(r0)        # Loads from 0x07 (since r0 = 0)

# Cannot use r0:r1 for addresses >= 0x10
# Must use a different register pair
```

Recommendation: Avoid using r0 as base register for load/store instructions.

7.8 Design Rationale

7.8.1 Why Register Pairs?

Alternatives Considered:

1. **Dedicated address registers:** Add special 8-bit address registers

- CONS: Increases ISA complexity, requires new instructions
- CONS: Breaks load/store architecture simplicity

2. **Implicit addressing:** Use accumulator-style addressing

- CONS: Not RISC (accumulator architecture)
- CONS: Limits parallelism and compiler optimization

3. **Register pairs (chosen):**

- PROS: Uses existing general-purpose registers
- PROS: Maintains RISC load/store simplicity
- PROS: Enables multi-precision arithmetic with same mechanism
- CONS: Requires even-numbered base registers

7.8.2 Why Even-Numbered Base Registers?

Hardware Simplification:

- Base register [7:4] determines pair: `(base, base+1)`
- If odd base allowed: Would require `(base, base+1)` OR `(base-1, base)`
- Alignment simplifies decode logic and register file access

Software Impact:

- 8 usable pairs (r0:r1 through r14:r15)
- Sufficient for: SP, FP, and 6 general-purpose pointers
- Compiler must track register alignment

Chapter 8

Calling Convention

8.1 Overview

A calling convention defines the rules for how functions call each other, pass parameters, preserve state, and return values. RISC-4's calling convention balances the constraints of a 4-bit architecture with the need for practical, efficient function calls.

Key Challenges:

- Limited register count (16 registers, but r0 is zero)
- 4-bit values limit parameter and return value sizes
- 8-bit stack pointer requires multi-instruction manipulation
- No dedicated link register or call instruction (must use software convention)

8.2 Register Usage Convention

8.2.1 Register Roles

Register	Name	Role	Preserved?
r0	zero	Constant zero	N/A (always zero)
r1	ra	Return address	Caller saves
r2	a0	Argument 0 / Return value	No
r3	a1	Argument 1	No
r4	a2	Argument 2	No
r5	a3	Argument 3	No
r6	v0	Return value (alternate)	No
r7	t0	Temporary	No (caller-saved)
r8	t1	Temporary	No (caller-saved)
r9	t2	Temporary	No (caller-saved)
r10	s0	Saved register	Yes (callee-saved)
r11	s1	Saved register	Yes (callee-saved)
r12	s2/fp_hi	Saved / Frame pointer high	Yes (callee-saved)
r13	s3/fp_lo	Saved / Frame pointer low	Yes (callee-saved)
r14	sp_hi	Stack pointer high	Yes (callee-saved)
r15	sp_lo	Stack pointer low	Yes (callee-saved)

Table 8.1: Register Calling Convention

8.2.2 Caller-Saved vs Callee-Saved

Caller-Saved Registers (r1-r9):

- Function may freely modify these
- Caller must save before calling if values are needed afterward
- Used for: arguments, return values, temporary computations

Callee-Saved Registers (r10-r15):

- Function must preserve these if used
- Callee must save on entry and restore on exit
- Used for: local variables that must survive across function calls

8.3 Parameter Passing

8.3.1 Basic Parameter Passing (1-4 nibbles)

Parameters are passed in registers r2-r5:

Register	Parameter
r2 (a0)	First argument (or bits [3:0] of multi-nibble value)
r3 (a1)	Second argument (or bits [7:4] of multi-nibble value)
r4 (a2)	Third argument
r5 (a3)	Fourth argument

Example: Single 4-bit parameter

```
# Call: result = add_one(value)
# Input: r2 = value (4-bit)
# Output: r2 = result (4-bit)

ORI   r2, r0, 0x5     # Set argument: a0 = 5
# Save r1 if needed (return address)
SW    r1, -1(r14)     # Save return address to stack
ORI   r1, r0, <ret_addr_high>  # Load return address
ORI   r1, r0, <ret_addr_low>   # (simplified; real implementation uses
    computed address)
J     add_one         # Call function
# Return here
```

8.3.2 Multi-Nibble Parameters

For 8-bit (2-nibble) parameters, use register pairs:

```
# Call: result = multiply_by_two(value)
# Input: r2:r3 = value (8-bit)
# Output: r2:r3 = result (8-bit)

ORI   r2, r0, 0x1     # High nibble = 0x1
ORI   r3, r0, 0x7     # Low nibble = 0x7
                      # r2:r3 = 0x17 (23 decimal)
# Call multiply_by_two
# r2:r3 = 0x2E (46 decimal) on return
```

8.3.3 More Than 4 Parameters

When more than 4 nibble parameters are needed, use the stack:

```
# Call: result = sum_six(a, b, c, d, e, f)
# First 4 in registers: r2=a, r3=b, r4=c, r5=d
# Remaining 2 on stack: e, f

# Caller:
ADDI r15, r15, -2    # Allocate stack space for 2 params
BCC  no_carry1
ADDI r14, r14, -1    # Handle carry
no_carry1:

SW   r6, 0(r14)      # Push e (5th parameter)
SW   r7, 1(r14)      # Push f (6th parameter)

ORI  r2, r0, <a>     # Load first 4 params
ORI  r3, r0, <b>
ORI  r4, r0, <c>
ORI  r5, r0, <d>

# Call function
# ...

# Callee (sum_six):
sum_six:
    # Parameters: r2, r3, r4, r5, [sp+0], [sp+1]
    ADD  r2, r2, r3      # a + b
    ADD  r2, r2, r4      # + c
    ADD  r2, r2, r5      # + d
    LW   r7, 0(r14)      # Load e from stack
    ADD  r2, r2, r7      # + e
    LW   r7, 1(r14)      # Load f from stack
    ADD  r2, r2, r7      # + f
    # Result in r2
    # Return...
```

8.4 Return Values

8.4.1 Single Nibble Return

Return values are placed in r2 (a0):

```
# Function: returns 4-bit value
get_status:
    # ... computation ...
    ORI r2, r0, 0x5      # Return value = 5
    # Jump to return address (in r1)
    # (Pseudo-instruction: RET would be J r1, but RISC-4 v0.1 lacks
       indirect jump)
```

Note: RISC-4 v0.1 lacks indirect jump, so return requires storing r1 in a known location or using a software convention.

8.4.2 Multi-Nibble Return

For 8-bit return values, use r2:r3:

```
# Function: returns 8-bit value
compute_large:
    # ... computation ...
    ORI r2, r0, 0xA      # High nibble = 0xA
    ORI r3, r0, 0x7      # Low nibble = 0x7
                         # Return value = 0xA7 (167 decimal)
    # Return to caller
```

8.4.3 Returning Multiple Values

For multiple return values (unusual but possible):

```
# Function: returns quotient and remainder
# Output: r2 = quotient, r3 = remainder
divide:
    # ... division logic ...
    ORI r2, r0, 0x5      # quotient = 5
    ORI r3, r0, 0x2      # remainder = 2
    # Return
```

8.5 Stack Frame Structure

8.5.1 Stack Frame Layout

A typical stack frame contains:

```
High addresses
    [Caller's frame]
    [................]
    [Argument n]        $\leftarrow$ FP + n (arguments beyond 4)
    [Argument 5]        $\leftarrow$ FP + 5
    [Return address]    $\leftarrow$ FP + 4
    [Saved FP high]     $\leftarrow$ FP + 3
    [Saved FP low]      $\leftarrow$ FP + 2
    [Saved r10]         $\leftarrow$ FP + 1
    [Saved r11]         $\leftarrow$ FP + 0 (FP points here)
    [Local var 0]       $\leftarrow$ FP - 1
    [Local var 1]       $\leftarrow$ FP - 2
    [Temp storage]      $\leftarrow$ SP (current stack pointer)
Low addresses
```

8.5.2 Frame Pointer (FP)

The frame pointer (r12:r13) provides a stable reference point for accessing:

- Local variables (negative offsets from FP)
- Saved registers (small positive offsets from FP)
- Stack parameters (larger positive offsets from FP)

Without FP, all references would be relative to SP, which changes during the function.

8.6 Function Prologue and Epilogue

8.6.1 Complete Function Prologue

The prologue sets up the stack frame:

```
function_name:
    # Step 1: Allocate stack space
    # Allocate N nibbles (e.g., 6 nibbles for saved regs + locals)
    ADDI r15, r15, -6    # Decrement SP low nibble
    BCC  no_borrow1      # Check for borrow
    ADDI r14, r14, -1    # Decrement SP high nibble
no_borrow1:

    # Step 2: Save callee-saved registers
    SW   r10, 0(r14)     # Save r10 at [sp+0]
    SW   r11, 1(r14)     # Save r11 at [sp+1]

    # Step 3: Save old frame pointer
    SW   r12, 2(r14)     # Save FP high at [sp+2]
    SW   r13, 3(r14)     # Save FP low at [sp+3]

    # Step 4: Save return address
    SW   r1, 4(r14)      # Save RA at [sp+4]

    # Step 5: Set new frame pointer
    # FP = SP + 2 (points to saved r11)
    ADD  r13, r15, r0    # FP_lo = SP_lo
    ADD  r12, r14, r0    # FP_hi = SP_hi
    ADDI r13, r13, 2     # FP += 2
    BCC  no_carry1
    ADDI r12, r12, 1     # Handle carry
no_carry1:

    # Function body starts here
    # Local variables at [FP-1], [FP-2], etc.
```

8.6.2 Complete Function Epilogue

The epilogue tears down the stack frame and returns:

```
    # Function body ends here

    # Step 1: Restore callee-saved registers
    LW   r10, 0(r14)     # Restore r10
    LW   r11, 1(r14)     # Restore r11

    # Step 2: Restore old frame pointer
    LW   r12, 2(r14)     # Restore FP high
    LW   r13, 3(r14)     # Restore FP low

    # Step 3: Load return address
    LW   r1, 4(r14)      # Load RA into r1

    # Step 4: Deallocate stack space
    ADDI r15, r15, 6     # Increment SP low nibble by 6
    BCC  no_carry2       # Check for carry
    ADDI r14, r14, 1     # Increment SP high nibble
no_carry2:

    # Step 5: Return (pseudo-instruction)
    # RISC-4 v0.1 lacks indirect jump, so this is simplified
    # In real implementation, r1 would contain return address
    # and we'd need a software-managed return mechanism
    # RET  (pseudo: J r1, not supported in v0.1)
```

8.7 Complete Function Example

8.7.1 Leaf Function (No Calls)

A leaf function doesn't call other functions, so it doesn't need to save RA:

```
# Function: add_three(a, b, c) -> a + b + c
# Input: r2 = a, r3 = b, r4 = c
# Output: r2 = result

add_three:
    # No prologue needed (leaf function, no locals)

    # Function body
    ADD r2, r2, r3       # r2 = a + b
    ADD r2, r2, r4       # r2 = a + b + c

    # No epilogue needed
    # Return (assuming return mechanism exists)
    RET                  # Pseudo-instruction
```

8.7.2 Non-Leaf Function (Makes Calls)

A function that calls other functions must save RA:

```
# Function: square_sum(a, b) -> square(a + b)
# Calls: square(x)

square_sum:
    # Prologue
    ADDI r15, r15, -3    # Allocate 3 nibbles
    BCC  no_borrow_p
    ADDI r14, r14, -1
no_borrow_p:

    SW   r1, 0(r14)      # Save return address
    SW   r10, 1(r14)     # Save r10 (we'll use it)

    # Function body
    ADD  r10, r2, r3     # r10 = a + b (save before calling)

    # Call square(r10)
    ADD  r2, r10, r0     # Move argument to r2
    # Save our RA before calling
    # (Our RA is already on stack)

    # Pseudo-call to square
    # In real implementation, this would be:
    # Store current PC+1 in r1
    # J square

    # After return from square, r2 = result

    # Epilogue
    LW   r10, 1(r14)     # Restore r10
    LW   r1, 0(r14)      # Restore return address

    ADDI r15, r15, 3     # Deallocate
    BCC  no_carry_e
    ADDI r14, r14, 1
no_carry_e:

    RET                  # Return
```

8.7.3 Recursive Function

```
# Function: factorial(n) -> n!
# Input: r2 = n
# Output: r2 = n!

factorial:
    # Prologue
    ADDI r15, r15, -4    # Allocate space
    BCC  no_borrow_fp
    ADDI r14, r14, -1
no_borrow_fp:

    SW   r1, 0(r14)      # Save RA
    SW   r10, 1(r14)     # Save r10 (for n)
    SW   r2, 2(r14)      # Save n

    # Base case: if (n <= 1) return 1
    SLTI r7, r2, 2       # r7 = (n < 2) ? 1 : 0
    BEQ  base_case       # If r7 == 0 (n >= 2), skip to recursive

    # Base case: return 1
    ORI  r2, r0, 1       # result = 1
    J    factorial_return

base_case:
    # Recursive case: return n * factorial(n-1)
    ADD  r10, r2, r0     # Save n in r10

    ADDI r2, r2, -1      # r2 = n - 1
    # Recursive call to factorial(r2)
    # (Pseudo-call, would need real call mechanism)

    # After return, r2 = factorial(n-1)
    # Multiply by n (in r10)
    # For 4-bit, multiplication is complex, simplified here:
    # result = n * factorial(n-1)
    # (Assume multiply routine exists)

factorial_return:
    # Epilogue
    LW   r10, 1(r14)     # Restore r10
    LW   r1, 0(r14)      # Restore RA

    ADDI r15, r15, 4     # Deallocate
    BCC  no_carry_fe
    ADDI r14, r14, 1
no_carry_fe:

    RET
```

8.8 Advanced Topics

8.8.1 Accessing Stack Parameters

When more than 4 parameters are passed:

```
# Function: sum_six(a, b, c, d, e, f)
# Params: r2=a, r3=b, r4=c, r5=d, stack[sp+4]=e, stack[sp+5]=f

sum_six:
    # After prologue, stack layout:
    # [sp+0] = saved registers
    # [sp+1] = saved FP
    # [sp+2] = saved FP
    # [sp+3] = saved RA
    # [sp+4] = caller's stack param: e
    # [sp+5] = caller's stack param: f

    # To access e and f, need to account for our frame
    # FP points to saved r11, so:
    # [FP+3] = saved RA
    # [FP+4] = parameter e
    # [FP+5] = parameter f

    ADD  r2, r2, r3      # a + b
    ADD  r2, r2, r4      # + c
    ADD  r2, r2, r5      # + d
    LW   r7, 4(r12)      # Load e from [FP+4]
    ADD  r2, r2, r7      # + e
    LW   r7, 5(r12)      # Load f from [FP+5]
    ADD  r2, r2, r7      # + f
    # Result in r2
```

8.8.2 Variable-Length Argument Lists

For functions like printf with variable arguments:

```
# Convention: First parameter is count, rest are values
# varargs(count, arg0, arg1, ...)

varargs:
    # r2 = count
    # r3 = arg0
    # r4 = arg1
    # r5 = arg2
    # [FP+4] = arg3
    # [FP+5] = arg4
    # ...

    # Loop through stack parameters if count > 4
    SLTI r7, r2, 5      # if (count < 5), no stack params
    BNE  no_stack

    # Process stack parameters
    # Calculate number of stack params: count - 4
    ADDI r6, r2, -4     # r6 = count - 4
    ORI  r8, r0, 4      # r8 = offset (starting at FP+4)

loop:
    LW   r9, 0(r12)     # Load param at [FP + r8]
                        # (Simplified: would need to add r8 to r12:r13)
    # Process r9...

    ADDI r8, r8, 1      # offset++
    ADDI r6, r6, -1     # remaining--
    BNE  loop           # Continue if more params

no_stack:
    # Done processing
```

8.8.3 Nested Function Calls

Each function call adds a new frame:

```
# Call chain: main -> func_a -> func_b

main:
    # main's frame
    # SP = 0xF0, FP = 0xF0
    # Call func_a
    # ...

func_a:
    # Prologue creates frame
    # SP = 0xE8, FP = 0xE8
    # Saved registers, RA point to main

    # Call func_b
    # ...

func_b:
    # Prologue creates frame
    # SP = 0xE0, FP = 0xE0
    # Saved registers, RA point to func_a

    # Work...

    # Epilogue restores func_a's frame
    # SP back to 0xE8
    # Return to func_a

    # func_a epilogue restores main's frame
    # SP back to 0xF0
    # Return to main
```

8.9 Limitations and Workarounds

8.9.1 No Indirect Jump (v0.1)

RISC-4 v0.1 lacks `JALR` (jump and link register), making returns difficult.

Workaround Options:

Option 1: Fixed Return Addresses

```
# Caller knows where it will return to
call_site_1:
    ORI  r1, r0, <ret1_high>
    ORI  r1, r0, <ret1_low>  # r1 = address of ret1
    J    function
ret1:
    # Returned here

function:
    # Do work...
    # Jump to address in r1 (but can't: no JALR)
    # Instead, jump to fixed return trampoline
```

Option 2: Return Trampoline

```
# Shared return mechanism at fixed address
return_trampoline:
    # Load RA from stack
    LW   r1, -1(r14)      # Assuming RA is at [SP-1]
    # Dispatch based on r1 value
    # (Requires jump table or series of comparisons)
```

Recommendation: For v1.0, add `JALR rd, rs` instruction:

- `rd ← PC + 1` (save return address)
- `PC ← rs` (jump to register value)

8.9.2 Limited Parameter Count

Only 4 register parameters forces stack usage for larger signatures.

Workaround: Pass pointer to structure:

```
# Instead of: func(a, b, c, d, e, f, g, h)
# Use: func(params_ptr)

# Caller:
# Build parameter structure at 0x80:
ORI   r10, r0, 0x8
ORI   r11, r0, 0x0        # r10:r11 = 0x80

SW    r2, 0(r10)          # params[0] = a
SW    r3, 1(r10)          # params[1] = b
# ... etc

# Pass pointer
ADD   r2, r10, r0         # Only need low byte if < 16 params
J     func

# Callee:
func:
    # r2 = params pointer (low byte)
    # Reconstruct full pointer (assuming high byte is 0x8)
    ORI   r10, r0, 0x8
    ADD   r11, r2, r0      # r10:r11 = params pointer

    LW    r3, 0(r10)       # Load params[0]
    LW    r4, 1(r10)       # Load params[1]
    # ...
```

8.10 Performance Considerations

8.10.1 Call Overhead

Typical function call overhead:

Operation	**Cycles**
Allocate stack (with carry handling)	2-4
Save registers (4 registers)	4
Save FP (2 nibbles)	2
Save RA	1
Update FP (with carry handling)	2-4
Total Prologue	**11-15 cycles**
Restore registers (4 registers)	4
Restore FP	2
Restore RA	1
Deallocate stack	2-4
Return (future JALR)	1
Total Epilogue	**10-12 cycles**
Total Call Overhead	**21-27 cycles**

8.10.2 Optimization Strategies

1. Leaf Function Optimization

- Don't save RA if function doesn't call others
- Don't save callee-saved registers if not used
- Reduces overhead to 0 cycles for trivial functions

2. Inline Small Functions

- If function body < 10 instructions, consider inlining
- Eliminates call overhead entirely
- Trade-off: code size increases

3. Tail Call Optimization

```
# Instead of:
func_a:
    # Work...
    # Call func_b
    # Epilogue
    # Return

# Use tail call:
func_a:
    # Work...
    # Setup args for func_b
    # Epilogue (restore our frame)
    # Jump directly to func_b (not call)
    J func_b    # func_b returns directly to our caller
```

4. Register Allocation

- Prefer caller-saved registers (r7-r9) for temporaries
- Only use callee-saved registers (r10-r11) when values must survive calls
- Reduces save/restore overhead

8.11 Future Enhancements

8.11.1 Proposed v1.0 Additions

1. JALR Instruction

```
JALR rd, rs     # rd <- PC+1, PC <- rs
```

Enables true function calls with return addresses.

2. Hardware Stack Support

- Dedicated SP register (auto-increment/decrement)
- PUSH/POP instructions
- Simplifies stack management

3. Link Register Convention

- Use r1 as dedicated link register
- JAL instruction: `r1 ← PC+1, PC ← target`
- RET instruction: `PC ← r1`

8.12 Design Rationale

8.12.1 Why This Convention?

Register Allocation:

- 4 argument registers matches typical function signatures
- 3 temporaries (t0-t2) sufficient for most expressions
- 2 saved registers (s0-s1) for critical locals
- Dedicated FP and SP essential for stack management

Stack Frame Design:

- FP provides stable reference despite SP changes
- Saved RA at predictable offset enables debuggers
- Compatible with C calling conventions (adapted)

Caller vs Callee Saved:

- Caller-saved for values not needed after call (avoids unnecessary saves)
- Callee-saved for long-lived locals (saves once, use many times)
- Balance minimizes total save/restore operations

Appendix A

Assembly Language Syntax

A.1 Instruction Format

RISC-4 assembly follows a simple, consistent syntax:

```
[label:] mnemonic [operands]  [# comment]
```

Components:

- **label:** Optional identifier followed by colon
- **mnemonic:** Instruction name (ADD, SUB, LW, etc.)
- **operands:** Comma-separated registers, immediates, or addresses
- **comment:** Optional text following # character

A.2 Operand Types

A.2.1 Registers

Registers are specified by name or number:

```
# By number:
ADD r1, r2, r3

# By conventional name:
ADD ra, a0, a1       # Equivalent to above

# Zero register:
ADD r4, r0, r5       # r4 = 0 + r5 (copy r5 to r4)
ORI r1, zero, 0x5    # r1 = 5 (load immediate)
```

A.2.2 Immediate Values

Immediates can be decimal, hexadecimal, or binary:

```
ADDI r1, r2, 5       # Decimal: 5
ADDI r1, r2, 0x5     # Hexadecimal: 5
ADDI r1, r2, 0b0101  # Binary: 5

ADDI r3, r3, -1      # Negative immediate (two's complement)
ORI  r4, r0, 0xF     # Hexadecimal: 15
```

A.2.3 Memory Operands

Load/store instructions use offset(base) syntax:

```
LW  r1, 0(r14)       # Load from [r14:r15 + 0]
SW  r2, 4(r14)       # Store to [r14:r15 + 4]
LW  r3, -1(r12)      # Load from [r12:r13 - 1]

# Base must be even-numbered register (pair)
LW  r1, 0(r14)       # Valid: r14 is even
LW  r1, 0(r15)       # INVALID: r15 is odd
```

A.2.4 Branch Targets

Branches use labels or offsets:

```
loop:
    ADDI r1, r1, 1
    SLTI r2, r1, 10
    BNE  loop            # Branch to label

    BEQ  +5              # Branch forward 5 instructions (rare)
    BCS  -3              # Branch backward 3 instructions (rare)
```

A.2.5 Jump Targets

Jumps use labels or absolute addresses:

```
    J start              # Jump to label
    J 0x100              # Jump to absolute address (rare)

start:
    # Program begins here
```

A.3 Pseudo-Instructions

Assemblers often provide pseudo-instructions that expand to one or more real instructions:

A.3.1 NOP - No Operation

```
NOP                  # Pseudo-instruction
# Expands to:
ADD r0, r0, r0       # Does nothing (writes to r0 are ignored)
```

A.3.2 MOV - Move Register

```
MOV rd, rs          # Pseudo-instruction: rd = rs
# Expands to:
ADD rd, rs, r0      # rd = rs + 0
# Or:
OR  rd, rs, r0      # rd = rs | 0
```

A.3.3 LI - Load Immediate

```
LI rd, imm4         # Pseudo-instruction: rd = immediate
# Expands to:
ORI rd, r0, imm4    # rd = 0 | imm4
```

A.3.4 NOT - Bitwise NOT

```
NOT rd, rs          # Pseudo-instruction: rd = ~rs
# Expands to:
XOR rd, rs, 0xF     # rd = rs ^ 0b1111 (flip all bits)
```

A.3.5 CLR - Clear Register

```
CLR rd              # Pseudo-instruction: rd = 0
# Expands to:
XOR rd, rd, rd      # rd = rd ^ rd = 0
# Or:
AND rd, rd, r0      # rd = rd & 0 = 0
```

A.3.6 INC/DEC - Increment/Decrement

```
INC rd              # Pseudo-instruction: rd++
# Expands to:
ADDI rd, rd, 1

DEC rd              # Pseudo-instruction: rd--
# Expands to:
ADDI rd, rd, -1
```

A.3.7 RET - Return from Function

```
RET                 # Pseudo-instruction: return to caller
# In v0.1 (no JALR):
# Requires software convention (see Chapter 10)

# In future v1.0 with JALR:
# Expands to:
JALR r0, r1         # PC = r1, r0 = PC+1 (r0 write ignored)
```

A.3.8 PUSH/POP - Stack Operations

```
PUSH rs              # Pseudo-instruction: push rs to stack
# Expands to:
ADDI r15, r15, -1    # SP--
SW   rs, 0(r14)      # mem[SP] = rs
# Note: Simplified; real implementation needs carry handling

POP rd               # Pseudo-instruction: pop from stack to rd
# Expands to:
LW   rd, 0(r14)      # rd = mem[SP]
ADDI r15, r15, 1     # SP++
# Note: Simplified; real implementation needs carry handling
```

A.3.9 BLT/BGE/BGT/BLE - Signed Comparison Branches

```
BLT rs, rt, label    # Pseudo: branch if rs < rt (signed)
# Expands to:
SLT  r7, rs, rt      # r7 = (rs < rt) ? 1 : 0
BNE  r7, label       # Branch if r7 != 0

BGE rs, rt, label    # Pseudo: branch if rs >= rt (signed)
# Expands to:
SLT  r7, rs, rt
BEQ  r7, label       # Branch if r7 == 0

BGT rs, rt, label    # Pseudo: branch if rs > rt (signed)
# Expands to:
SLT  r7, rt, rs      # r7 = (rt < rs) ? 1 : 0
BNE  r7, label

BLE rs, rt, label    # Pseudo: branch if rs <= rt (signed)
# Expands to:
SLT  r7, rt, rs
BEQ  r7, label
```

A.4 Assembler Directives

Common assembler directives for controlling assembly:

A.4.1 .org - Set Origin Address

```
    .org 0x000       # Start assembling at address 0x000
start:
    ORI r14, r0, 0xF
    ORI r15, r0, 0xF     # Initialize SP = 0xFF
```

A.4.2 .data - Data Section

```
    .data           # Switch to data section
array:
    .nibble 0x1, 0x2, 0x3, 0x4  # Define nibble array
value:
    .nibble 0xA     # Single nibble

    .text           # Switch back to code section
    LW r1, array    # Load first element
```

A.4.3 .equ - Define Constant

```
    .equ STACK_TOP, 0xFF
    .equ MAX_COUNT, 10

    ORI r14, r0, (STACK_TOP >> 4)
    ORI r15, r0, (STACK_TOP & 0xF)
```

A.4.4 .align - Alignment

```
    .align 4        # Align to next 4-nibble boundary
function:
    ADD r1, r2, r3
```

A.5 Commenting Conventions

A.5.1 Inline Comments

```
ADD r1, r2, r3      # r1 = r2 + r3
ADDI r4, r4, 1      # Increment counter
```

A.5.2 Block Comments

```
#=============================================
# Function: multiply
# Description: Multiply two 4-bit numbers
# Input: r2 = multiplicand, r3 = multiplier
# Output: r2:r3 = 8-bit product
# Clobbers: r4, r5, r6
#=============================================
multiply:
    # Implementation...
```

A.5.3 Section Headers

```
#-----------------------------------------------
# Initialization
#-----------------------------------------------
init:
    ORI r14, r0, 0xF     # Set up stack pointer
    ORI r15, r0, 0xF

#-----------------------------------------------
# Main Loop
#-----------------------------------------------
main_loop:
    # Loop body...
```

A.6 Register Naming Conventions

Number	Name	Usage
r0	zero	Constant zero
r1	ra	Return address
r2	a0	Argument 0 / Return value
r3	a1	Argument 1
r4	a2	Argument 2
r5	a3	Argument 3
r6	v0	Return value (alternate)
r7	t0	Temporary 0
r8	t1	Temporary 1
r9	t2	Temporary 2
r10	s0	Saved register 0
r11	s1	Saved register 1
r12	s2/fp_hi	Saved 2 / Frame pointer high
r13	s3/fp_lo	Saved 3 / Frame pointer low
r14	sp_hi	Stack pointer high
r15	sp_lo	Stack pointer low

Table A.1: Register Name Aliases

A.7 Example Assembly Program

```
#================================================
# Program: Sum Array
# Description: Sum elements of a 4-element array
#================================================

    .org 0x000

    #-----------------------------------------
    # Initialization
    #-----------------------------------------
start:
```

```
    # Initialize stack pointer to 0xFF
    LI   sp_hi, 0xF
    LI   sp_lo, 0xF

    # Set up array pointer
    LI   r10, 0x8           # Array at 0x80
    LI   r11, 0x0

    # Initialize sum to zero
    CLR  r2                 # r2 = sum

    # Initialize counter
    LI   r3, 4              # r3 = count

    #-----------------------------------------
    # Sum Loop
    #-----------------------------------------
sum_loop:
    LW   r4, 0(r10)         # Load array[i]
    ADD  r2, r2, r4         # sum += array[i]

    # Increment pointer
    INC  r11                # ptr++

    # Decrement counter
    DEC  r3                 # count--
    BNE  sum_loop           # Continue if count != 0

    #-----------------------------------------
    # Store Result
    #-----------------------------------------
    SW   r2, 0(r10)         # Store sum at array[4]

done:
    J    done               # Infinite loop

    #-----------------------------------------
    # Data Section
    #-----------------------------------------
    .org 0x80
array:
    .nibble 0x3, 0x5, 0x2, 0x7  # Array elements
result:
    .nibble 0x0                  # Result location
```

Appendix B

Example Programs

B.1 Fibonacci Sequence

```
# Compute fib(n)
# Input:  r2 (n)
# Output: r6 (result)
fib:
    # Base Case: if n < 2, return n
    SLTI r7, r2, 2        # r7 = (n < 2) ? 1 : 0
    ADDI r7, r7, -1       # r7 = (n < 2) ? 0 : -1
    BEQ  r7, return_n     # If r7 == 0 (meaning n < 2), jump

    # Recursive step would go here...
    # (Omitted for brevity due to complexity of
    # saving context in 4-bit mode)

return_n:
    ADD r6, r2, r0        # return n
    RET                   # Pseudo for J r1 (if implemented)
```

Appendix C

Revision History

Version	Date	Changes
0.0.1	2026-01-20	Initial draft - Basic ISA definition - 16 instructions documented - Register model established - Preliminary calling convention
0.1.0	2026-01-21	Major documentation expansion - Added Chapter 4: Status Flags (comprehensive) - Added Chapter 8: Pipeline Architecture (5-stage pipeline) - Expanded Chapter 9: Addressing (register pairs, multi-precision) - Expanded Chapter 10: Calling Convention (complete prologue/epilogue) - Expanded Appendix A: Assembly Syntax (pseudo-instructions, directives) - Fixed all instruction flag documentation (C, Z flags) - Reorganized instruction order (opcode sequence) - Improved document formatting and spacing - Added comprehensive examples throughout

Table C.1: Document Revision History

C.1 Version 0.0.1 (Initial Draft)

Date: January 20, 2026

Status: Solidifying

Major Components:

- Defined 4-bit RISC architecture with 16 instructions
- Established 5-stage pipeline design (concept)
- Documented register file (16 × 4-bit registers)

- Defined four instruction formats (R, I, M, J)
- Basic calling convention outlined
- Fibonacci example program

Known Limitations:

- Pipeline documentation incomplete
- Flag behavior not fully specified
- Calling convention needs expansion
- Missing examples for common operations
- No JALR instruction (returns difficult)

C.2 Version 0.1.0 (Documentation Expansion)

Date: January 21, 2026

Status: Pre-Silicon

Major Additions:

C.2.1 Chapter 4: Status Flags

- Complete flag register specification (Carry and Zero)
- Flag update rules for all instructions
- Comparison operation patterns (equality, signed, unsigned)
- Pipeline hazard analysis (flag forwarding vs stalls)
- Design rationale (why RISC-4 uses flags)
- Comparison to MIPS, ARM, SPARC flag implementations

C.2.2 Chapter 8: Pipeline Architecture

- Complete 5-stage pipeline description (IF, ID, EX, MEM, WB)
- Pipeline register definitions
- Data hazard analysis (RAW, load-use, flag hazards)
- Control hazard solutions (branch prediction, delay slots)
- Forwarding logic specifications
- Performance analysis (realistic CPI estimation)
- Implementation considerations

C.2.3 Chapter 9: Addressing (Expanded)

- Memory organization (Harvard architecture)
- Register pair formation rules
- Pair manipulation techniques (increment, decrement, offset addition)
- Complete addressing mode examples (stack, arrays, structures)
- Multi-precision arithmetic (8-bit, 16-bit operations)
- Stack management (push, pop with carry handling)
- Limitations and workarounds
- Design rationale for register pairing

C.2.4 Chapter 10: Calling Convention (Expanded)

- Complete register usage convention
- Parameter passing (1-4 parameters, stack parameters)
- Return value conventions
- Stack frame structure and layout
- Complete function prologue and epilogue
- Example functions (leaf, non-leaf, recursive)
- Advanced topics (variable arguments, nested calls)
- Performance analysis (call overhead)
- Optimization strategies
- Future enhancements (JALR proposal)

C.2.5 Appendix A: Assembly Syntax (Expanded)

- Complete operand type documentation
- Pseudo-instruction catalog (NOP, MOV, LI, NOT, etc.)
- Assembler directives (.org, .data, .equ, .align)
- Commenting conventions
- Register naming conventions
- Complete example program

C.2.6 All Instructions Updated

- Comprehensive flag documentation (C and Z)
- Consistent formatting across all instruction descriptions
- Detailed examples for each instruction
- Flag behavior explicitly stated

Bug Fixes:

- Fixed AND instruction: corrected operation to `rd ← rs & rt`
- Fixed ADDI instruction: corrected parameter order in encoding
- Corrected flag naming inconsistencies throughout
- Fixed instruction ordering (SHF moved to correct position)

Formatting Improvements:

- Added spacing optimizations (reduced excessive whitespace)
- Implemented raggedbottom (no vertical stretching)
- Added subsection grouping (improved page breaks)
- Standardized table formatting
- Improved code listing presentation

Remaining for v1.0:

- Add JALR instruction for proper function returns
- Additional example programs (bubble sort, string operations)
- Hardware implementation guide
- Verilog/VHDL reference implementation
- Assembler specification
- Complete instruction timing diagrams
- Test suite definition

C.3 Future Versions

C.3.1 Planned for v1.0 (Silicon)

- JALR instruction (indirect jump with link)
- Hardware verification complete
- Tapeout-ready RTL
- Complete test suite
- Assembler and toolchain
- Reference implementation in Verilog

C.3.2 Potential v2.0 Enhancements

- Hardware multiply/divide
- Interrupt support
- Memory-mapped I/O specification
- DMA controller
- Cache hierarchy
- Floating-point coprocessor (4-bit mantissa experiments)