

FPGA CPU “RG Sonic32”

v1 Instruction Set Architecture Specification

Author: Ryan Gaffere

Version: v1

Document Version: v1

Last Updated: January 26th, 2026

This document defines the architectural contract between software and hardware for v1.

Table of Contents

1. Introduction

1.1 Purpose of This Document

1.2 Scope of the v1 ISA

1.3 Normative Language

2. Architectural Overview

2.1 Programmer's Model

2.2 Address Space

2.3 Endianness

2.4 Instruction Length and Alignment

2.5 Reset State

3. Execution Model

3.1 Instruction Fetch

3.2 Instruction Lifecycle

3.3 Program Counter (PC) Behavior

3.4 Memory Model Overview

4. Memory System

4.1 Unified Instruction/Data Memory

4.2 Memory Interface Protocol

4.3 Memory Access Granularity

4.4 Load Semantics

4.5 Store Semantics

4.6 Alignment Rules and Misaligned Access Policy

4.7 Instruction Fetch Faults and Access Errors

5. Register File

5.1 Register Count and Width

5.2 Register Access Ports

5.3 Register Read and Write Timing

5.4 Read-After-Write Semantics

5.5 Fixed Register Roles

6. Condition Flags

6.1 Flag Definitions

6.2 Flag Update Rules

6.3 Flag-Setting Instructions

6.4 Flag Usage for Branching

7. Instruction Encoding Overview

7.1 Fixed-Length Instruction Encoding

7.2 Opcode Space

7.3 Common Field Placement

7.4 Reserved and Illegal Encodings

8. Instruction Formats

8.1 R-Type Instruction Format

8.2 I-Type Instruction Format

8.3 S-Type Instruction Format

8.4 B-Type Instruction Format

8.5 J-Type Instruction Format

8.6 U-Type Instruction Format

8.7 Immediate Encoding and Sign-Extension Rules

9. Instruction Set Definition

9.1 Data Movement Instructions

9.2 Integer ALU Instructions

9.3 Compare and Test Instructions

9.4 Branch Instructions

9.5 Jump and Call Instructions

9.6 Load Instructions

9.7 Store Instructions

9.8 System Instructions

10. Control Flow Semantics

10.1 Program Counter Update Rules

10.2 Branch Target Computation

10.3 Jump Target Computation

10.4 Delay Slot Policy

11. Trap and Exception Handling

11.1 Trap Vector

11.2 Trap Causes

11.3 Trap Entry Semantics

11.4 Trap Return Instruction

11.5 Architectural Trap Registers

12. Calling Convention (Minimal)

12.1 Register Usage Conventions

12.2 CALL and RET Mechanism

12.3 Stack Behavior

13. Datapath Control Requirements

13.1 ALU Operand Selection

13.2 Writeback Selection

13.3 Program Counter Selection

13.4 Memory Address and Command Signals

13.5 Flag Control Signals

13.6 Instruction Register and Stall Behavior

14. Performance and Microarchitectural Assumptions

14.1 ALU Timing

14.2 Load and Store Timing

14.3 Pipeline Policy

14.4 Stalling and Handshake Behavior

15. Unsupported, Reserved, and Undefined Behavior

15.1 Explicitly Unsupported Instructions

15.2 Reserved Opcode Space

15.3 Undefined Behavior Summary

16. Future Extensions (Non-Normative)

Appendices

Appendix A - Opcode and Function Code Tables

Appendix B - Instruction Encoding Examples

Appendix C - Instruction Pseudocode Summary

Appendix D - Glossary

1. Introduction

1.1 Purpose of This Document

This document serves as the authoritative specification of the Instruction Set Architecture (ISA) for Version 1 (v1) of the RG Sonic32 CPU, implemented on an Artix-7 FPGA.

It defines the architectural contract between software and hardware, including instruction encodings, execution semantics, register behavior, memory access rules, and exception handling. All architectural behaviors described herein are normative and are reflected in the operation of the v1 RG Sonic32 CPU.

This document is intended to be the primary reference for hardware implementation, software development, assembly, emulation, and verification of the v1 architecture.

1.2 Scope of the v1 ISA

This document specifies all architectural design decisions for the v1 RG Sonic32 CPU, including explicit trade-offs and excluded features. It defines what the architecture guarantees, what behaviors are undefined, and which features are unsupported in v1.

This specification applies only to Version 1 of the RG Sonic32 ISA. Future versions may extend, modify, or deprecate architectural features described in this document. Each ISA version shall have its own dedicated specification document located in the [docs/](#) directory. Version 0 (v0) is excluded, as it represents a pre-architectural, module-validation stage rather than a complete ISA.

1.3 Normative Language

The key words “**MUST**”, “**MUST NOT**”, “**REQUIRED**”, “**SHALL**”, “**SHALL NOT**”, “**SHOULD**”, “**MAY**”, and “**OPTIONAL**” in this document are to be interpreted as described in RFC-2119.

The following terms are used throughout this specification:

- **UNDEFINED**: Behavior for which the architecture makes no guarantees. Software must not rely on any specific outcome.
- **RESERVED**: Encodings or behaviors that are not defined in v1 and may be assigned meaning in future ISA versions.
- **OPTIONAL**: Features that are not required for v1 compliance but may be implemented without violating the specification.

2. Architectural Overview

2.1 Programmer's Model

The programmer's model defines the complete set of architectural states that is visible to software executing on the RG Sonic32 v1 CPU. This includes registers, condition flags, the program counter, and the architectural view of memory. All behavior described in this section is independent of microarchitectural implementation.

The RG Sonic32 v1 programmer's model consists of:

- Thirty-two (32) general-purpose registers, each 32 bits wide, where register r0 is hardwired to zero and ignores write attempts.
- A 32-bit program counter (PC) that holds the byte-addressed address of the current instruction.
- A set of condition flags (Z, N, C, V) used for conditional execution and branching.
- A minimal set of architectural trap registers used for exception and interrupt handling.
- A unified, byte-addressable memory space visible to both instruction fetch and data access.

Microarchitectural details such as pipeline stages, internal control states, memory handshaking, and execution timing are not part of the programmer's model and are not visible to software.

2.2 Address Space

The RG Sonic32 v1 architecture defines a 32-bit, flat address space. All addresses generated by software and used by the CPU are 32 bits wide, yielding an architectural address range of 0x00000000 to 0xFFFFFFFF.

The address space is byte-addressed, meaning each address identifies an individual 8-bit byte. Multi-byte data types such as halfwords (16 bits) and words (32 bits) occupy consecutive byte addresses. Software is responsible for ensuring proper alignment of multi-byte accesses, as defined elsewhere in this specification.

Although the architectural address space is 32 bits wide, only a subset of the address space is physically implemented in v1. A compliant v1 implementation shall provide at least 64 MiB of addressable memory; implementations may provide additional memory. Accesses to addresses outside the implemented range result in behavior defined in the memory access and exception handling sections of this document.

The RG Sonic32 v1 architecture employs a unified address space for instruction fetch and data access. Instructions and data share the same address space and are accessed using the same

addressing rules. No form of virtual memory, address translation, or segmentation is implemented in v1.

All addresses are interpreted using little-endian byte ordering, as defined in Section 2.3.

2.3 Endianness

The RG Sonic32 v1 architecture uses little-endian byte ordering.

For multi-byte data types, the least significant byte (LSB) is stored at the lowest memory address, and successive bytes are stored at increasing addresses. This convention applies uniformly to all multi-byte memory accesses, including halfword (16-bit) and word (32-bit) loads and stores.

For a 32-bit word stored at address A , memory is organized as follows:

- Memory $A + 0$ contains bits $7 : 0$ (least significant byte)
- Memory $A + 1$ contains bits $15 : 8$
- Memory $A + 2$ contains bits $23 : 16$
- Memory $A + 3$ contains bits $31 : 24$ (most significant byte)

Instruction fetch, data loads, and data stores all interpret memory using the same little-endian byte ordering. No alternative byte-ordering modes are supported in v1.

2.4 Instruction Length and Alignment

All instructions in the RG Sonic32 v1 architecture are fixed-length and 32 bits wide. Instructions are stored in memory as four consecutive bytes and are fetched as a single 32-bit word.

The program counter (PC) is byte-addressed and always points to the address of the first byte of the current instruction. Under normal sequential execution, the PC advances by 4 bytes after each instruction fetch.

Instruction fetches shall be aligned to 32-bit boundaries. The PC must be a multiple of 4, such that:

```
PC[1:0] = 2'b00
```

Instruction fetches from misaligned addresses result in a trap, as defined in the trap and exception handling sections of this specification.

Data alignment requirements for loads and stores are defined separately and are not governed by this section.

2.5 Reset State

Upon reset, the RG Sonic32 v1 CPU enters a well-defined architectural state.

The program counter (PC) is initialized to the reset vector address:

```
RESET_PC = 0xFFFF_0000
```

Execution begins by fetching the instruction located at the reset vector.

All general-purpose registers are cleared to zero on reset. Register r0 remains hardwired to zero and ignores write attempts at all times.

All condition flags (Z, N, C, V) are cleared on reset.

The processor is not in a trap or exception-handling state following reset. Interrupts are disabled, and the architectural trap registers are initialized to zero.

Reset does not define the contents of memory. The initial contents of RAM, ROM, and other memory-mapped regions are platform-defined.

3. Execution Model

3.1 Instruction Fetch

At the start of each instruction's execution, the processor fetches one instruction from memory at the address contained in the program counter (PC).

Instruction fetch produces no architectural side effects. The fetched instruction is presented to the decode stage before any architectural state (registers, flags, or memory) is modified.

If instruction fetch cannot be completed due to an architectural violation (e.g., misaligned PC or access fault), a synchronous trap is raised and normal instruction execution does not proceed.

3.2 Instruction Lifecycle

Each instruction executes atomically with respect to architectural state. Architecturally visible effects occur in program order and are completed before the next instruction begins execution. This includes arithmetic operations whose execution latency exceeds a single cycle.

Conceptually, instruction execution proceeds through the following phases:

1. **Fetch** - The instruction is retrieved from memory.
2. **Decode** - The instruction's operation, operands, and immediates are determined.
3. **Execute** - The instruction's computation, address calculation, or control decision is performed.
4. **Memory Access** - Loads or stores access memory.
5. **Writeback** - Results are committed to registers or flags.

The internal implementation may span multiple cycles or control states; however, software observes each instruction as an indivisible operation.

3.3 Program Counter (PC) Behavior

The program counter (PC) identifies the address of the current instruction.

Under sequential execution, the PC advances to the next instruction following successful completion of the current instruction.

Control-flow instructions modify the PC as follows:

- **Conditional branches** update the PC based on the current condition flags.
- **Jumps and calls** update the PC unconditionally.
- **Trap entry** redirects the PC to the trap vector.
- **Trap return** restores the PC from the saved exception state.

All PC updates take effect immediately upon completion of the instruction. The architecture defines no delay slots.

3.4 Memory Module Overview

Instruction execution may involve memory access for instruction fetch, loads, or stores.

Architecturally, memory operations occur in program order and are completed before subsequent instructions observe their effects. Load instructions produce a value that may be written back to a register, and store instructions update memory as part of the instruction's execution.

Instruction fetch and data access share a single architectural memory space. The architecture permits concurrent instruction fetch and data access, provided architectural ordering and visibility are preserved.

Memory stalls, arbitration, and timing behavior are not architecturally visible. Software observes memory as providing correct values consistent with the architectural rules defined elsewhere in this specification.

4. Memory System

4.1 Unified Instruction/Data Memory

The RG Sonic32 v1 architecture defines a unified memory system in which instructions and data share a single architectural address space.

There is no architectural distinction between instruction memory and data memory. All memory locations are accessed using the same addressing rules and protection semantics. Instruction fetches and data accesses may occur concurrently, provided architectural ordering is preserved.

An implementation may internally use separate instruction and data ports or memories; such distinctions are not visible to software and have no architectural impact.

4.2 Memory Interface Protocol

Architecturally, memory accesses are synchronous operations initiated by instruction execution.

Each memory access is defined by:

- An effective address
- An access size (byte, halfword, or word)
- An operation type (read or write)

The architecture does not expose memory timing, wait states, or handshaking mechanisms to software. An implementation may stall instruction execution while waiting for memory, but such stalls are not architecturally observable.

Memory operations complete in program order. A subsequent instruction observes the effects of all prior completed memory accesses.

4.3 Memory Access Granularity

Memory is byte-addressable.

The RG Sonic32 v1 architecture supports the following access sizes:

Loads:

- Byte (8-bit)
- Halfword (16-bit)
- Word (32-bit)

Stores:

- Byte (8-bit)
- Halfword (16-bit)
- Word (32-bit)

Each load or store accesses exactly the number of bytes implied by its access size. Stores modify only the addressed bytes and do not affect adjacent memory locations.

4.4 Load Semantics

Load instructions read data from memory and write a value to a destination register.

The effective address is computed by the instruction prior to the memory access. The value loaded from memory is extended to the architectural register width according to the instruction variant:

- **LB**: Sign-extend 8-bit value to 32 bits
- **LBU**: Zero-extend 8-bit value to 32 bits
- **LH**: Sign-extend 16-bit value to 32 bits
- **LHU**: Zero-extend 16-bit value to 32 bits
- **LW**: Load full 32-bit value without extension

The destination register is updated only after the memory access completes successfully. If a load instruction triggers a fault, no register state is modified.

4.5 Store Semantics

Store instructions write data from a source register to memory.

The effective address is computed by the instruction prior to the memory access. The least significant bits of the source register are written to memory according to the store size:

- **SB**: Store least significant 8 bits
- **SH**: Store least significant 16 bits
- **SW**: Store full 32-bit value

Store instructions do not produce a register result. If a store instruction triggers a fault, memory contents are not modified.

4.6 Alignment Rules and Misaligned Access Policy

Memory accesses are subject to alignment requirements based on access size:

- Byte accesses may be performed at any address.
- Halfword accesses must be aligned to 2-byte boundaries.
- Word accesses must be aligned to 4-byte boundaries.

If a load or store instruction attempts a misaligned access, the behavior is implementation-defined in v1. A compliant implementation may raise a synchronous trap or may treat the behavior as undefined.

Software must not rely on misaligned memory accesses and should ensure proper alignment for all loads and stores.

4.7 Instruction Fetch Faults and Access Errors

Instruction fetch is a memory read operation and is subject to architectural fault conditions.

An instruction fetch fault occurs if:

- The program counter refers to an invalid or inaccessible address
- The program counter violates alignment requirements
- The instruction encoding is illegal or undefined

When an instruction fetch fault occurs, no architectural state associated with the instruction is modified. Control is transferred to the trap handler as defined in the trap and exception handling section of this specification.

Data access faults (for loads or stores) similarly prevent completion of the faulting instruction and do not partially update architectural state.

5. Register File

5.1 Register Count and Width

The RG Sonic32 v1 architecture defines thirty-two (32) general-purpose registers.

Each register is 32 bits wide and is identified by a 5-bit register index. All general-purpose registers are uniformly accessible by integer, control-flow, and memory instructions unless otherwise specified.

Register indices are encoded directly in instruction fields. No banked or privileged register sets are defined in v1.

5.2 Register Access Ports

Architecturally, instructions may reference up to:

- Two source registers
- One destination register

An instruction may read zero, one, or two source registers and may write at most one destination register.

The architecture does not expose the number of physical read or write ports used by an implementation. Any compliant implementation **MUST** preserve the architectural semantics defined in this section regardless of internal porting or scheduling.

5.3 Register Read and Write Timing

Register reads occur during instruction execution and produce the architectural value of the specified source registers.

Register writes occur as part of instruction completion. A destination register is updated only if the instruction completes successfully. If an instruction triggers a trap or exception, no register writes occur.

The effects of a register write are visible to subsequent instructions in program order. No instruction may observe the partial effects of another instruction.

5.4 Read-After-Write Semantics

If an instruction writes a register and a subsequent instruction reads that same register, the subsequent instruction observes the value written by the earlier instruction.

If an instruction both reads and writes the same register, the read value is the architectural value of the register prior to execution of the instruction, unless explicitly specified otherwise by the instruction's semantics.

These rules apply regardless of internal pipeline structure, forwarding mechanisms, or execution timing.

5.5 Fixed Register Roles

Register **r0** is architecturally defined as a constant zero register.

- Reads from r0 always return the value zero.
- Writes to r0 have no effect.

The following register roles are defined by convention and are not enforced by hardware:

Register	Conventional Role
r1	Return address (RA)
r2	Stack Pointer (SP)
r3	Frame Pointer (FP)

Software may use registers other than r0 for any purpose unless restricted by calling convention or system software policy. No additional registers have special architectural behavior in v1.

6. Condition Flags

6.1 Flag Definitions

RG Sonic32 v1 defines four condition flags:

- **Z (Zero)**: Set if the computed result is zero.
- **N (Negative)**: Set if the computed result is negative in two's-complement representation (i.e., the most significant bit of the result is 1).
- **C (Carry)**: Set to indicate an unsigned carry out for addition, or the inverse of borrow behavior for subtraction as defined in Section 6.1.3.
- **V (Overflow)**: Set if a signed two's-complement overflow occurs.

6.1.1 Z (Zero)

$Z = 1$ iff `result == 0`, else $Z = 0$

6.1.2 N (Negative)

$N = \text{result}[31]$

6.1.3 C (Carry / Borrow Convention)

For flag-setting operations, C is defined as follows:

- **Addition:** $C = 1$ iff the unsigned addition produces a carry out of bit 31.
- **Subtraction:** $C = 1$ iff no unsigned borrow is required (i.e., $a \geq b$ in unsigned arithmetic), and $C = 0$ iff a borrow occurs.

This convention enables unsigned comparisons using C as described in Section 6.4.

6.1.4 V (Overflow)

For flag-setting operations:

- **Addition:** $V = 1$ iff adding two operands with the same sign produces a result with a different sign.
- **Subtraction:** $V = 1$ iff subtracting two operands with different signs produces a result whose sign differs from the minuend.

6.2 Flag Update Rules

Condition flags are updated only by flag-setting instructions (Section 6.3). All other instructions **MUST NOT** modify Z, N, C, or V.

If a flag-setting instruction triggers a trap or exception, it **MUST NOT** update the flags.

Flags are updated as part of instruction completion and are visible to subsequent instructions in program order.

6.3 Flag-Setting Instructions

The following instructions update condition flags:

- **CMP** - compare (architecturally equivalent to subtraction for flag purposes)
- **TEST** - test (architecturally equivalent to bitwise AND for flag purposes)

No other instructions update flags in v1 unless explicitly stated in the instruction definitions.

6.3.1 CMP Semantics

CMP *rs1*, *rs2* performs an internal subtraction:

```
tmp = rs1 - rs2
```

It does not write a general-purpose destination register. It updates flags as if the subtraction were performed:

- Z and N from **tmp**
- C using the subtraction carry/borrow convention (Section 6.1.3)
- V from signed overflow on subtraction

6.3.2 TEST Semantics

TEST *rs1*, *rs2* performs an internal bitwise AND:

```
tmp = rs1 & rs2
```

It does not write a general-purpose destination register. It updates flags as follows:

- Z and N from **tmp**
- **C = 0**
- **V = 0**

6.4 Flag Usage for Branching

Conditional branch instructions evaluate conditions based solely on the current condition flags. Branch instructions **MUST NOT** implicitly compute flags.

Unless otherwise specified by the branch instruction definition, a conditional branch that is taken updates the PC to the branch target, and a branch that is not taken continues with sequential execution.

The minimum conditional branch set in v1 uses the following conditions:

- **BEQ**: branch if equal
Taken iff **Z == 1**

- **BNE**: branch if not equal
Taken iff $Z == 0$
- **BLT** (signed less than):
Taken iff $N \neq V$
- **BGE** (signed greater than or equal):
Taken iff $N == V$
- **BLTU** (unsigned less than):
Taken iff $C == 0$
- **BGEU** (unsigned greater than or equal):
Taken iff $C == 1$

Software is responsible for executing an appropriate flag-setting instruction (e.g., **CMP** or **TEST**) prior to any conditional branch that depends on flags.

7. Instruction Encoding Overview

7.1 Fixed-Length Instruction Encoding

All instructions in RG Sonic32 v1 are fixed-length and occupy 32 bits.

Each instruction is encoded as a single 32-bit word and is fetched, decoded, and executed as an indivisible unit. No variable-length, compressed, or multiword instructions are defined in v1.

Instruction boundaries are implicit and determined solely by instruction length; no prefix or extension mechanisms exist in v1.

7.2 Opcode Space

The primary opcode field occupies the most significant bits of the instruction word.

- **Opcode width**: 6 bits
- **Primary opcode space**: 2^6 distinct opcode values

Primary opcodes select the general instruction class. Additional instruction differentiation may be provided by secondary fields (e.g., function codes or subfields) within the instruction format.

Not all opcode values are assigned in v1. Unassigned opcode values are reserved for future ISA extensions.

7.3 Common Field Placement

To simplify decoding and enable straightforward hardware implementation, RG Sonic32 v1 uses consistent field placement across instruction formats wherever possible.

Unless otherwise specified by a given format:

- Register index fields (**rd**, **rs1**, **rs2**) occupy fixed bit positions across formats.
- Immediate fields occupy the lower portion of the instruction word and are interpreted according to the instruction format.
- Opcode bits always occupy the most significant portion of the instruction word.

This consistency allows instruction decode to identify register operands and opcode class without format-specific re-alignment or bit rearrangement.

7.4 Reserved and Illegal Encodings

An instruction encoding is classified as one of the following:

- **Defined:** The encoding corresponds to a valid instruction defined in this specification.
- **Reserved:** The encoding is not defined in v1 but may be assigned meaning in a future ISA version.
- **Illegal:** The encoding is architecturally invalid and must not be executed.

Execution of an illegal encoding results in a synchronous trap. No architectural state associated with the instruction is modified.

Execution of a reserved encoding produces behavior equivalent to an illegal encoding unless explicitly stated otherwise in a future ISA revision.

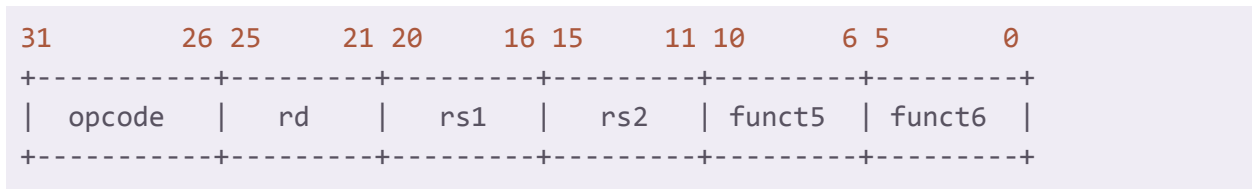
Software must not rely on the behavior of reserved or illegal instruction encodings.

8. Instruction Formats

8.1 R-Type Instruction Format

The R-type format is used for register-to-register operations, including integer arithmetic, logical operations, and register-based shifts.

R-Type Layout



Field Definitions

- **opcode [31:26]**: Primary opcode
- **rd [25:21]**: Destination register
- **rs1 [20:16]**: First source register
- **rs2 [15:11]**: Second source register
- **funct5 [10:6]**: Subfunction or shift-related field
- **funct6 [5:0]**: Function selector within the opcode

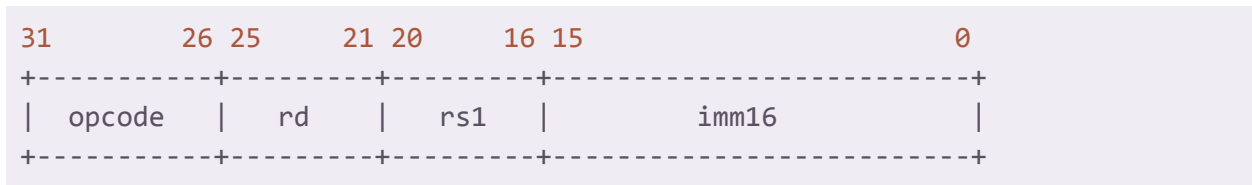
Semantics

R-type instructions read **rs1** and **rs2**, perform the specified operation, and write the result to **rd**. Flag-setting behavior, if any, is defined by the instruction semantics.

8.2 I-Type Instruction Format

The I-type format is used for register–immediate operations, loads, register-relative jumps, and shift-immediate instructions.

I-Type Layout



Field Definitions

- **opcode [31:26]**: Primary opcode
- **rd [25:21]**: Destination register
- **rs1 [20:16]**: Source register
- **imm16 [15:0]**: Immediate value

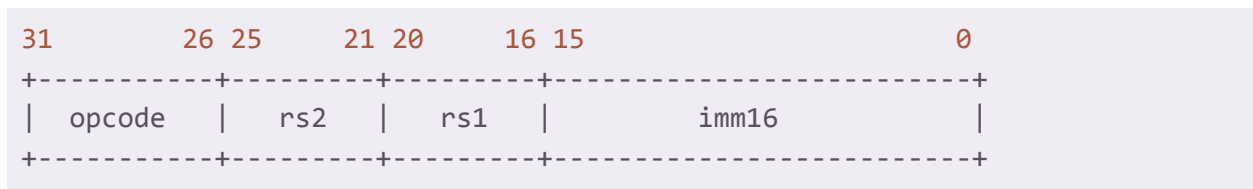
Semantics

The immediate field is sign-extended to 32 bits unless otherwise specified.
For shift-immediate instructions, only `imm16[4:0]` is used as the shift amount.

8.3 S-Type Instruction Format

The S-type format is used for store instructions.

S-Type Layout



Field Definitions

- **opcode [31:26]**: Primary opcode
- **rs2 [25:21]**: Source register containing store data
- **rs1 [20:16]**: Base register
- **imm16 [15:0]**: Signed offset

Semantics

The effective address is computed as:

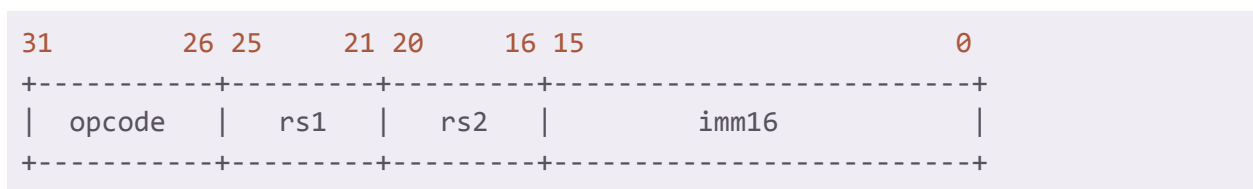
$$EA = rs1 + \text{signext}(imm16)$$

The value in `rs2` is written to memory at the effective address.

8.4 B-Type Instruction Format

The B-type format is used for conditional branch instructions.

B-Type Layout



Field Definitions

- **opcode [31:26]**: Primary opcode
- **rs1 [25:21]**: Reserved (may be unused in v1)
- **rs2 [20:16]**: Reserved (may be unused in v1)
- **imm16 [15:0]**: Signed PC-relative offset

Semantics

Branch conditions are evaluated using condition flags (Section 6).

If the branch is taken:

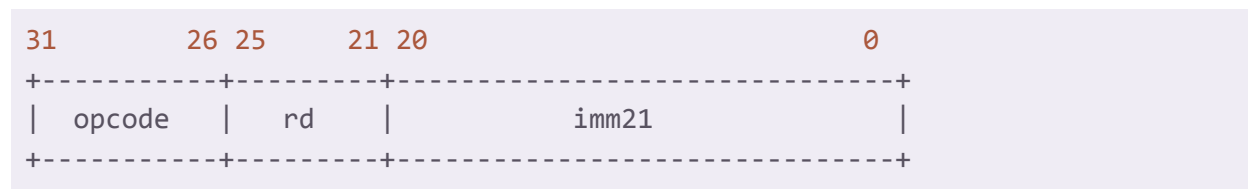
```
PC_next = PC + 4 + signext(imm16)
```

If the branch is not taken, execution continues sequentially.

8.5 J-Type Instruction Format

The J-type format is used for unconditional jumps and jump-and-link instructions.

J-Type Layout



Field Definitions

- **opcode [31:26]**: Primary opcode
- **rd [25:21]**: Destination register for link address
- **imm21 [20:0]**: Signed PC-relative offset

Semantics

The jump target is computed as:

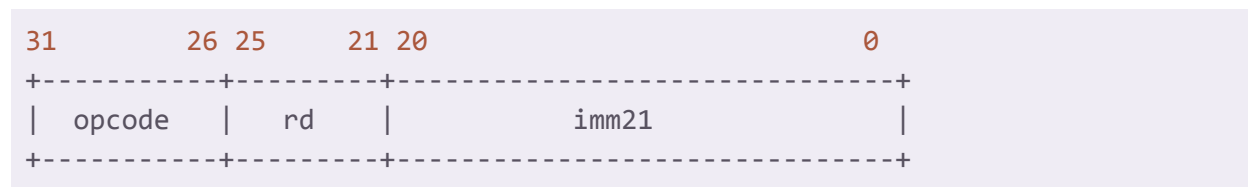
```
PC_next = PC + 4 + signext(imm21)
```

If $rd \neq r0$, the return address ($PC + 4$) is written to rd .

8.6 U-Type Instruction Format

The U-type format is used for upper-immediate instructions.

U-Type Layout



Field Definitions

- **opcode** [31:26]: Primary opcode
- **rd** [25:21]: Destination register
- **imm21** [20:0]: Immediate value

Semantics

For U-type instructions, the immediate is shifted left by a fixed amount before use. The shift amount is instruction-defined and chosen to facilitate construction of full-width constants.

8.7 Immediate Encoding and Sign-Extension Rules

Immediate fields are interpreted as signed or unsigned values depending on instruction type.

Immediate Widths

- **imm16**: 16-bit immediate (I-, S-, B-type)
- **imm21**: 21-bit immediate (J-, U-type)

Sign Extension

- Branch and jump offsets are sign-extended.
- Load, store, and arithmetic immediates are sign-extended.
- Zero-extension applies only to loaded data, not to immediates.

Shift Amounts

- Shift amounts are 5 bits wide.
- For shift-immediate instructions, **shamt** = **imm**[4:0].

- For register-based shifts, `shamt = rs2[4:0]`.

Immediate values are applied directly in byte units; no implicit scaling or shifting is performed unless explicitly stated by the instruction definition.

9. Instruction Set Definition

9.1 Data Movement Instructions

MOV - Register Move

Format: R-type

Operation:

```
rd ← rs1
```

Description:

Copies the value from `rs1` into `rd`.

Does not update condition flags.

ADDI - Add Immediate

Format: I-type

Operation:

```
rd ← rs1 + signext(imm16)
```

Description:

Adds a signed immediate to a register value.

Does not update condition flags.

LUI - Load Upper Immediate

Format: U-type

Operation:

```
rd ← imm21 << K
```

Description:

Loads an immediate value into the upper portion of a register.

The shift amount K is architecturally defined (e.g., 16) to facilitate construction of full-width constants.

Does not update condition flags.

9.2 Integer ALU Instructions

The following instructions perform register-to-register integer operations.

ADD - Add

```
rd ← rs1 + rs2
```

SUB - Subtract

```
rd ← rs1 - rs2
```

AND - Bitwise AND

```
rd ← rs1 & rs2
```

OR - Bitwise OR

```
rd ← rs1 | rs2
```

XOR - Bitwise XOR

```
rd ← rs1 ^ rs2
```

SLL - Shift Left Logical

```
rd ← rs1 << rs2[4:0]
```

SRL - Shift Right Logical

```
rd ← rs1 >> rs2[4:0]
```

SRA - Shift Right Arithmetic

```
rd ← rs1 >>> rs2[4:0]
```

MUL - Multiply

```
rd ← (rs1 * rs2)[31:0]
```

Notes:

- All ALU instructions write a result to **rd**.
- Some integer ALU instructions may require multiple cycles to complete.
- None of these instructions update condition flags unless explicitly specified elsewhere.

9.3 Compare and Test Instructions

CMP - Compare

Format: R-type

Operation:

```
tmp ← rs1 - rs2
```

Description:

Performs a subtraction for the purpose of setting condition flags.
No register result is written.

Flags Updated: Z, N, C, V

TEST - Test Bits

Format: R-type

Operation:

```
tmp ← rs1 & rs2
```

Description:

Performs a bitwise AND for the purpose of testing bits.
No register result is written.

Flags Updated:

- Z, N from result
- C = 0
- V = 0

9.4 Branch Instructions

Branch instructions alter control flow based on condition flags. Branch instructions do not modify flags.

All branch targets are PC-relative.

BEQ - Branch if Equal

Taken iff $Z == 1$

BNE - Branch if Not Equal

Taken iff $Z == 0$

BLT - Branch if Less Than (signed)

Taken iff $N \neq V$

BGE - Branch if Greater or Equal (signed)

Taken iff $N == V$

BLTU - Branch if Less Than (unsigned)

Taken iff $C == 0$

BGEU - Branch if Greater or Equal (unsigned)

Taken iff $C == 1$

PC Update:

```
PC_next = PC + 4 + signext(imm16)
```

9.5 Jump and Call Instructions

J - Unconditional Jump

Format: J-type (encoded as `JAL r0, imm`)

Operation:

```
PC ← PC + 4 + signext(imm21)
```

JAL - Jump and Link

Format: J-type

Operation:

```
rd ← PC + 4
PC ← PC + 4 + signext(imm21)
```

JALR - Jump and Link Register

Format: I-type

Operation:

```
rd ← PC + 4
PC ← rs1 + signext(imm16)
```

Notes:

- If $rd = r0$, no link is written.
- Implementations may optionally clear bit 0 of the target address.

9.6 Load Instructions

Load instructions read data from memory into a register.

LB - Load Byte (signed)

```
rd ← signext(mem8[EA])
```

LBU - Load Byte (unsigned)

```
rd ← zeroext(mem8[EA])
```

LH - Load Halfword (signed)

```
rd ← signext(mem16[EA])
```

LHU - Load Halfword (unsigned)

```
rd ← zeroext(mem16[EA])
```

LW - Load Word

$$rd \leftarrow \text{mem32}[EA]$$

Effective Address:

$$EA = rs1 + \text{signext}(\text{imm16})$$

9.7 Store Instructions

Store instructions write register data to memory.

SB - Store Byte

$$\text{mem8}[EA] \leftarrow rs2[7:0]$$

SH - Store Halfword

$$\text{mem16}[EA] \leftarrow rs2[15:0]$$

SW - Store Word

$$\text{mem32}[EA] \leftarrow rs2$$

Effective Address:

$$EA = rs1 + \text{signext}(\text{imm16})$$

9.8 System Instructions

NOP - No Operation

Operation:

No architectural effect.

HALT - Halt Processor

Operation:

Stops instruction execution.

Further behavior is implementation-defined and platform-specific.

10. Control Flow Semantics

10.1 Program Counter Update Rules

The program counter (PC) holds the byte address of the current instruction.

Upon completion of an instruction, the PC is updated according to one of the following rules:

1. **Sequential execution**

If the instruction does not alter control flow:

```
PC_next = PC + 4
```

2. **Taken branch**

If a conditional branch instruction is taken:

```
PC_next = branch_target
```

3. **Jump or call**

If a jump or call instruction is executed:

```
PC_next = jump_target
```

4. **Trap or exception entry**

If a trap or exception is taken:

```
PC_next = TRAP_VECTOR
```

5. **Trap return**

If a trap return instruction is executed:

```
PC_next = EPC
```

The PC is updated atomically as part of instruction completion. No instruction observes an intermediate or partially updated PC value.

10.2 Branch Target Computation

All conditional branch instructions use PC-relative addressing.

The branch target is computed as:

```
branch_target = PC + 4 + signext(imm16)
```

Where:

- **PC** is the address of the branch instruction
- **imm16** is the signed 16-bit immediate encoded in the instruction
- The immediate value is interpreted in byte units

Branch conditions are evaluated solely using the current condition flags as defined in Section 6. Branch instructions do not compute or modify flags.

If the branch condition is not satisfied, execution proceeds with sequential execution (**PC + 4**).

10.3 Jump Target Computation

Jump and call instructions alter control flow unconditionally.

PC-Relative Jumps (J, JAL)

For PC-relative jump instructions, the target is computed as:

```
jump_target = PC + 4 + signext(imm21)
```

If the instruction writes a link register, the value written is:

```
link = PC + 4
```

Register-Relative Jumps (JALR)

For register-relative jump instructions, the target is computed as:

```
jump_target = rs1 + signext(imm16)
```

The value written to the link register, if any, is:

```
link = PC + 4
```

If the computed jump target is misaligned, the resulting behavior is defined by the alignment and trap rules specified elsewhere in this document.

10.4 Delay Slot Policy

The RG Sonic32 v1 architecture defines no delay slots.

When a branch or jump instruction alters control flow, the PC update takes effect immediately upon completion of the instruction. No subsequent sequential instruction is executed as a side effect of control transfer.

Software must not assume the existence of delay slots, and implementations must not architecturally expose delayed control flow behavior.

11. Trap and Exception Handling

11.1 Trap Vectors

The RG Sonic32 v1 architecture defines a single trap vector address:

```
TRAP_VECTOR = 0xFFFF_0000
```

When a trap is taken, control is transferred to the instruction located at the trap vector address. The trap handler is responsible for determining the cause of the trap and taking appropriate action.

The trap vector address is fixed in v1. No vectored or priority-based trap dispatch mechanism is defined.

11.2 Trap Causes

A trap may be generated by either a synchronous exception or an asynchronous interrupt.

11.2.1 Synchronous Exceptions

Synchronous exceptions are generated as a direct result of instruction execution. The following synchronous exceptions are defined in v1:

- **Illegal Instruction**
An instruction encoding is invalid, reserved, or unsupported.
- **Misaligned Instruction Fetch**
The program counter is not aligned to a 4-byte boundary at instruction fetch.
- **Misaligned Load or Store**
A memory access violates alignment requirements.
- **Instruction Fetch Access Fault**
Instruction fetch accesses an invalid or inaccessible memory address.
- **Load or Store Access Fault**
A data access references an invalid or inaccessible memory address.

Synchronous exceptions are precise: the faulting instruction does not modify architectural state.

11.2.2 Asynchronous Interrupts

Asynchronous interrupts are external events that may occur independently of instruction execution.

The RG Sonic32 v1 architecture defines support for at least one external interrupt source. The presence, number, and triggering conditions of interrupt sources are implementation-defined.

Interrupts are taken only when interrupts are enabled, as determined by architectural status state.

11.3 Trap Entry Semantics

When a trap is taken, the following architectural actions occur atomically:

1. Save Program Counter

```
EPC ← PC
```

2. Record Trap Cause

```
CAUSE ← trap_id
```

3. Update Status

- The current interrupt enable state is saved.

- Interrupts are disabled to prevent nested traps unless explicitly re-enabled by software.

4. Redirect Control Flow

```
PC ← TRAP_VECTOR
```

No instruction following the faulting or interrupted instruction is executed before control is transferred to the trap handler.

11.4 Trap Return Instruction

The RG Sonic32 v1 ISA defines a trap return instruction:

ERET - Exception Return

Operation:

```
PC ← EPC
restore interrupt enable state
```

Description:

Returns execution to the instruction address saved in **EPC**. The interrupt enable state is restored to its value prior to trap entry.

Execution resumes as if the trapped instruction had completed normally, unless the trap handler modifies architectural state prior to return.

11.5 Architectural Trap Registers

The following architectural registers are defined for trap handling:

EPC - Exception Program Counter

- Width: 32 bits
- Holds the program counter value saved at trap entry.

CAUSE - Trap Cause Register

- Width: 32 bits
- Encodes the reason for the trap. The encoding of trap cause values is implementation-defined but must be consistent within a given implementation.

STATUS - Status Register

- Width: 32 bits
- Contains control and state information related to trap handling.

At minimum, **STATUS** includes:

- **IE** - Global interrupt enable
- **PIE** or equivalent state to restore interrupt enable status on trap return

Additional bits in **STATUS** are reserved for future use.

12. Calling Convention (Minimal)

This section defines a minimal calling convention for software executing on the RG Sonic32 v1 architecture. These conventions are not enforced by hardware, but are required for interoperability between independently compiled code, including compiled C programs, libraries, and system software.

The conventions defined here are sufficient to support function calls, stack-based local storage, and return semantics required by v1 software workloads.

12.1 Register Usage Conventions

The following register usage conventions are defined:

Register	Conventional Role	Notes
r0	Constant zero	Hardwired to zero
r1	Return address (RA)	Holds link address for CALL/RET
r2	Stack pointer (SP)	Points to top of stack
r3	Frame pointer (FP)	Optional; may be used as general-purpose
r4-r31	General-purpose	No fixed roles in v1

All registers other than **r0** may be freely used by software unless restricted by higher-level ABI or operating system policy.

This specification does not define caller-saved vs callee-saved register classes. Software is responsible for preserving registers as required by its own conventions.

12.2 CALL and RET Mechanism

Function calls and returns are implemented using existing control-flow instructions.

CALL

A function call is performed using a jump-and-link instruction:

```
CALL target
≡ JAL r1, target
```

Semantics:

- The return address ($PC + 4$) is written to $r1$
- Control transfers to the call target

RET

A function return is performed using a register-relative jump:

```
RET
≡ JALR r0, r1, 0
```

Semantics:

- Control transfers to the address stored in $r1$
- No link register is written

12.3 Stack Behavior

The stack is a contiguous region of memory addressed using the stack pointer ($r2$).

The following stack conventions are defined:

- The stack grows downward, toward lower memory addresses.
- The stack pointer always points to the top of the current stack frame.
- Stack allocation is performed by decrementing $r2$.
- Stack deallocation is performed by incrementing $r2$.

A typical function prologue may:

- Save the return address and/or frame pointer
- Allocate space for local variables by adjusting $r2$

A typical function epilogue may:

- Restore saved registers
- Deallocate the stack frame
- Return using **RET**

The specific layout of stack frames, argument passing conventions, and register save/restore policies are intentionally left unspecified and may be defined by higher-level ABI or operating system conventions.

13. Datapath Control Requirements

13.1 ALU Operand Selection

The datapath **MUST** support selection of ALU input operands sufficient to implement all instruction semantics.

At minimum, the ALU input selection logic must support:

ALU Operand A Selection

- Register operand (**rs1**)
- Program counter (PC)
- Constant zero

ALU Operand B Selection

- Register operand (**rs2**)
- Sign-extended immediate
- Constant value 4 (for PC increment)
- Shifted immediate (for upper-immediate instructions)

These selections enable:

- Integer arithmetic and logic operations
- Effective address computation for loads and stores
- PC-relative branch and jump target computation

13.2 Writeback Selection

The datapath **MUST** support selection of writeback data to the destination register.

At minimum, the following writeback sources are required:

- ALU result
- Memory read data
- $PC + 4$ (for jump-and-link instructions)
- Shifted immediate value (for upper-immediate instructions)

Writeback occurs only if the instruction semantics specify a destination register update. If an instruction does not produce a register result, no writeback occurs.

13.3 Program Counter Selection

The datapath **MUST** support selection of the next program counter value from the following sources:

- Sequential PC increment ($PC + 4$)
- PC-relative target (branches and PC-relative jumps)
- Register-relative target (JALR)
- Trap vector address
- Exception return address (EPC)

The control logic **MUST** ensure that only one PC source is selected per instruction completion and that the PC update occurs atomically.

13.4 Memory Address and Command Signals

The datapath **MUST** support memory access for instruction fetch and data operations.

At minimum, the following capabilities are required:

Memory Address Selection

- PC (for instruction fetch)
- ALU result (effective address for load/store)

Memory Command Signals

- Read enable
- Write enable
- Access size (byte, halfword, word)
- Load sign control (sign-extend vs zero-extend)

Store instructions must write only the addressed bytes. Load instructions must apply the appropriate extension semantics before writeback.

13.5 Flag Control Signals

The datapath **MUST** support explicit control over condition flag updates.

At minimum:

- A flag write enable signal must exist
- Flags are updated only by flag-setting instructions
- Flag values are derived from the ALU result and operation type

Instructions that do not explicitly set flags **MUST NOT** modify flag state.

13.6 Instruction Register and Stall Behavior

The datapath **MUST** support holding the current instruction stable across multiple cycles when required by the implementation.

At minimum:

- An instruction register write enable must exist to control when a new instruction is latched
- Instruction execution may stall if required to complete an operation (e.g., memory access or multicycle execution)

Stalls are not architecturally visible. During a stall:

- The program counter does not advance
- No new instruction is fetched
- No partial architectural state updates occur

Once execution resumes, instruction completion proceeds normally.

14. Performance and Microarchitectural Assumptions

This section documents performance-related assumptions and microarchitectural behaviors that are relevant to software and system design. These assumptions describe the intended

characteristics of a compliant RG Sonic32 v1 implementation but do not impose strict timing requirements unless explicitly stated.

Unless otherwise specified, all behaviors described in this section are not architecturally visible and may vary across implementations.

14.1 ALU Timing

Simple integer ALU operations (e.g., ADD, SUB, logical ops, shifts) are assumed to complete within a finite number of cycles.

In the reference v1 implementation, integer ALU operations (e.g., ADD, SUB, logical operations, shifts) are expected to complete in a single execution step. However, the ISA does not require single-cycle completion.

Software must not assume any specific ALU latency beyond the guarantee that:

- Each instruction completes atomically
- Results are visible to subsequent instructions in program order

Longer-latency operations may be implemented as multi-cycle operations, provided architectural semantics are preserved.

14.2 Load and Store Timing

Load and store instructions are assumed to complete in a finite amount of time.

In the reference v1 implementation, memory accesses are expected to complete without additional wait states. However, the ISA does not mandate zero-latency memory.

An implementation may stall instruction execution while waiting for a memory operation to complete. Such stalls are not architecturally visible, and no instruction may observe partial memory effects.

Memory operations complete in program order. No form of memory reordering or speculation is defined or permitted in v1.

14.3 Pipeline Policy

The RG Sonic32 v1 ISA does not define or expose a pipeline model.

A compliant implementation may be:

- Non-pipelined
- Multi-cycle
- Pipelined
- Or use any internal execution structure

Regardless of internal implementation, the following architectural guarantees must hold:

- Instructions appear to execute sequentially
- Architectural state updates occur in program order
- No instruction observes the partial execution of another instruction

Software must not assume the presence or absence of pipelining, forwarding, or speculation.

14.4 Stalling and Handshake Behavior

The ISA permits instruction execution to stall internally when required to satisfy instruction semantics.

Stalls may occur due to:

- Memory access latency
- Multicycle execution
- Internal resource contention

During a stall:

- The program counter does not advance
- No new instruction is architecturally fetched
- No architectural state is partially updated

Once the stall condition is resolved, execution resumes normally.

The ISA does not expose handshake, ready/valid, or wait-state signals to software. Any such mechanisms are strictly microarchitectural and implementation-defined.

15. Unsupported, Reserved, and Undefined Behavior

15.1 Explicitly Unsupported Instructions

The following instruction classes are explicitly unsupported in RG Sonic32 v1:

- Integer division instructions (DIV, MOD)

- Floating-point instructions of any kind
- Atomic or read-modify-write memory operations
- Vector or SIMD instructions
- Privileged system instructions beyond those explicitly defined in this specification

Any attempt to execute an instruction belonging to an unsupported class results in an illegal instruction trap.

The absence of these instructions is a deliberate design choice for v1. Future ISA revisions may define extensions that introduce additional instruction classes.

15.2 Reserved Opcode Space

Not all opcode values and instruction encodings are assigned in RG Sonic32 v1.

Opcode values and instruction encodings that are not explicitly defined in this specification are classified as reserved. Reserved encodings have no defined behavior in v1.

Execution of a reserved encoding results in behavior equivalent to execution of an illegal instruction, unless explicitly stated otherwise by a future ISA revision.

Software must not emit or rely on reserved encodings.

15.3 Undefined Behavior Summary

Certain behaviors are intentionally left undefined in RG Sonic32 v1. In such cases, the architecture makes no guarantees regarding program behavior, and implementations may handle these situations arbitrarily.

Undefined behavior includes, but is not limited to:

- Execution of instructions with undefined or reserved encodings
- Misaligned memory accesses, unless otherwise specified as trapping
- Control flow to misaligned instruction addresses
- Use of uninitialized memory
- Reliance on memory access timing, stall behavior, or internal execution order
- Assumptions about instruction latency or cycle counts

Programs that exhibit undefined behavior are not portable and may produce unpredictable results.

16. Future Extensions (Non-Normative)

Instruction Set Extensions

Future ISA revisions may introduce additional instruction classes, including but not limited to:

- **Integer division**
 - May be introduced as multicycle operations
 - Likely to produce only the lower 32 bits of results in early revisions
- **Extended integer arithmetic**
 - Saturating arithmetic
 - Bit manipulation instructions
- **Floating-point support**
 - Single-precision floating point as a possible initial target
 - No commitment to IEEE-754 compliance in early revisions
- **Atomic and synchronization instructions**
 - Required for advanced multitasking or SMP systems
 - Not required for early embedded or single-core use cases

System and Privileged Extensions

Future versions of the architecture may expand system-level functionality, including:

- **Expanded trap and interrupt model**
 - Multiple interrupt sources
 - Priority or vectored interrupt handling
- **Additional system registers**
 - Timer registers
 - Performance counters
 - Debug and tracing facilities
- **Privilege levels**
 - Separation between user and supervisor execution
 - Support for operating system isolation

Accelerator and Coprocessor Integration

The RG Sonic32 architecture is intended to support heterogeneous compute models in future revisions.

Possible extensions include:

- **Neural network accelerators**
 - Custom instructions to configure, launch, and synchronize with accelerators
 - Memory-mapped control registers and DMA engines

- **Custom opcode space**
 - Reserved opcode regions may be used for application-specific extensions
 - Such extensions may be implementation-defined and non-portable

These extensions are expected to coexist with the base ISA without breaking binary compatibility for v1 software.

Platform-Specific Extensions

Certain features may be defined outside the core ISA in platform-specific documentation, including:

- Memory-mapped I/O address maps
- Peripheral interfaces
- Boot ROM layout
- Debug interfaces

Separating platform details from the core ISA allows the RG Sonic32 architecture to be reused across multiple systems and applications.

Compatibility and Versioning

Future ISA revisions will preserve backward compatibility with v1 software wherever practical. New features will be introduced using reserved opcode space or new instruction encodings.

Each ISA revision will be accompanied by a dedicated specification document identifying the supported feature set and compatibility guarantees.

Appendices

Appendix A - Opcode and Function Code Tables

This appendix summarizes the opcode space and function code usage for RG Sonic32 v1. All opcode values not explicitly listed are reserved.

A.1 Primary Opcode Map (Summary)

Opcode [31:26]	Instruction Class
000000	Integer ALU (R-type)
000001	Integer Immediate / Loads (I-type)
000010	Stores (S-type)

000011	Branches (B-type)
000100	Jumps / Calls (J-type)
000101	Upper Immediate (U-type)
111111	System Instructions

A.2 R-Type Function Codes (Example)

funct6	Operation
000000	ADD
000001	SUB
000010	AND
000011	OR
000100	XOR
000101	SLL
000110	SRL
000111	SRA
001000	CMP
001001	TEST
001010	MUL

A.3 System Instructions

Opcode	Mnemonic	Description
111111	NOP	No operation
111111	HALT	Halt execution
111111	ERET	Return from trap

Appendix B - Instruction Encoding Examples

This appendix provides illustrative examples of instruction encodings. Bit patterns are shown for clarity and are not normative.

B.1 ADD r5, r6, r7 (R-type)

```
opcode = 000000
rd      = 00101 (r5)
rs1     = 00110 (r6)
rs2     = 00111 (r7)
funct   = ADD
```

B.2 ADDI r4, r3, -8 (I-type)

```
opcode = 000001
rd      = 00100 (r4)
rs1     = 00011 (r3)
imm16   = 0xFFF8
```

B.3 LW r8, 12(r2) (I-type load)

```
opcode = 000001
rd      = 01000 (r8)
rs1     = 00010 (r2)
imm16   = 0x000C
```

B.4 BEQ label (B-type)

```
opcode = 000011
imm16   = PC-relative offset
```

Branch taken if $Z == 1$.

B.5 CALL function (J-type)

```
CALL function
≡ JAL r1, imm21
```

Appendix C - Instruction Pseudocode Summary

This appendix summarizes instruction behavior using pseudocode. These definitions are descriptive and must be consistent with the normative instruction semantics.

C.1 Arithmetic and Logic

```
ADD:   rd = rs1 + rs2
SUB:   rd = rs1 - rs2
AND:   rd = rs1 & rs2
OR:    rd = rs1 | rs2
XOR:   rd = rs1 ^ rs2
MUL:   rd = (rs1 * rs2)[31:0]
```

C.2 Shifts

```
SLL:   rd = rs1 << rs2[4:0]
SRL:   rd = rs1 >> rs2[4:0]
SRA:   rd = rs1 >>> rs2[4:0]
```

C.3 Compare / Test

```
CMP:   tmp = rs1 - rs2 ; update flags
TEST:  tmp = rs1 & rs2 ; update flags (C=0, V=0)
```

C.4 Control Flow

```
BEQ:   if Z == 1 then PC = PC + 4 + imm
BNE:   if Z == 0 then PC = PC + 4 + imm
JAL:   rd = PC + 4 ; PC = PC + 4 + imm
JALR:  rd = PC + 4 ; PC = rs1 + imm
```

C.5 Loads / Stores

```
LB:    rd = signext(mem8[EA])
LBU:   rd = zeroext(mem8[EA])
LH:    rd = signext(mem16[EA])
LW:    rd = mem32[EA]
SB:    mem8[EA] = rs2[7:0]
```

```
SH:    mem16[EA] = rs2[15:0]  
SW:    mem32[EA] = rs2
```

Appendix D - Glossary

ABI

Application Binary Interface. A set of conventions governing function calls, register usage, and binary compatibility.

ALU

Arithmetic Logic Unit. Performs integer arithmetic and logical operations.

Architectural State

Processor state visible to software, including registers, flags, memory, and control registers.

Condition Flags

Z, N, C, V flags used for conditional branching and comparisons.

ISA

Instruction Set Architecture. The contract between hardware and software.

PC (Program Counter)

Holds the address of the current instruction.

Trap

A transfer of control caused by an exception or interrupt.

Undefined Behavior

Program behavior for which the architecture makes no guarantees.