DREL Architecture Reference Manual

Chapter 1: Instruction Set Architecture (ISA)

1.1 Design Philosophy

The DREL ISA is architected as a streamlined **Load/Store RISC** interface designed to expose the internal micro-architectural capabilities of modern high-performance cores directly to the compiler.

Key design pillars:

- **Fixed-Width Encoding:** All instructions are strictly **32-bit** aligned on 4-byte boundaries. This eliminates the complex length-decoding stage found in legacy x86 (CISC), significantly reducing front-end latency and power consumption.
- **Decoupled Vector Length:** The architecture defines vector operations without hardcoding the vector register width, allowing the same binary to scale from mobile cores (128-bit) to server-grade silicon (512-bit/1024-bit).
- **Simplified Address Generation:** Complex addressing modes are removed to ensure single-cycle address generation and reduce pipeline stalls.

1.2 Instruction Encoding

DREL utilizes a regularized 32-bit format to simplify hardware instruction aligners and pre-decoders.

R-Type (Register to Register): Used for ALU and Vector arithmetic.

```
| 31 ... 26 | 25 ... 21 | 20 ... 16 | 15 ... 11 | 10 .......... 0 |
+-----------+-----------+-----------+-----------+-----------------+
|   Opcode  |     Rd    |    Rs1    |    Rs2    |    Func / Ext   |
+-----------+-----------+-----------+-----------+-----------------+
  (6 bits)    (5 bits)    (5 bits)    (5 bits)       (11 bits)
```

I-Type (Immediate / Memory): Used for Load/Store, Branching, and Immediate arithmetic.

```
| 31 ... 26 | 25 ... 21 | 20 ... 16 | 15 ................... 0 |
+-----------+-----------+-----------+---------------------------+
|   Opcode  |     Rd    |    Rs1    |      Immediate (16b)      |
+-----------+-----------+-----------+---------------------------+
  (6 bits)    (5 bits)    (5 bits)       (Signed 16-bit Imm)
```

1.3 Register File Organization

1.3.1 General Purpose Registers (GPR)

DREL provides 32 integer registers (64-bit width).

- **R0:** Hardwired to **ZERO**. Writes are ignored; reads always return 0. This simplifies dependency breaking and initialization.
- **R1 - R29:** General Purpose (Caller/Callee saved defined in ABI).
- **R30 (LR):** Link Register (Stores return address for `CALL`).
- **R31 (SP):** Stack Pointer.

1.3.2 Scalable Vector Registers (VEC)

Designed for "Write-Once, Run-Anywhere" AI workloads.

- **V0 - V31:** Scalable Vector Registers.
- **VLEN (Vector Length):** An implementation-specific constant (e.g., 128, 256, 512 bits). Software writes code agnostic of VLEN, using Predicate masks and Strip-mining loops provided by the hardware implementation.
- **Zero-Copy Aliasing:** In hybrid x86/DREL implementations, `V0-V15` may alias directly to physical AVX-512/AMX registers to eliminate context switching penalties during mode transitions.

1.3.3 Predicate Registers (P-Mask)

- **P0 - P7:** 1-bit predicate registers used for conditional execution of Vector instructions (masking).
- **P0:** Hardwired to **TRUE**.

1.4 Instruction Classes

1.4.1 Memory Access (Load/Store)

Memory access is strictly strictly decoupled from ALU operations.

- **Supported Addressing Modes:**
    1. **Base + Immediate:** `MEM[Rs1 + simm16]` (Efficient for struct access/stack).
    2. **Base + Index:** `MEM[Rs1 + Rs2]` (Efficient for array traversal).
- *Note:* Complex "scaled" addressing (e.g., `Base + Index*8`) is explicitly removed. The compiler must issue a separate `SLL` (Shift) instruction. This simplifies the AGU (Address Generation Unit).

1.4.2 ALU Operations

- Arithmetic: `ADD`, `SUB`, `MUL`, `DIV`.
- Logical: `AND`, `OR`, `XOR`, `NOT` (Pseudo-op via XOR -1).
- Shifts: `SLL`, `SRL`, `SRA` (Logical/Arithmetic).

1.4.3 Control Flow

- **Branches:** `BEQ`, `BNE`, `BLT`, `BGE` (Compare and branch relative).
- **Jumps:** `JMP` (Unconditional relative), `JAL` (Jump and Link), `JALR` (Register indirect).
- **Hardware Loop Hint:** Special encoding to assist the hardware Loop Stream Detector (LSD) in identifying hot loops.

1.4.4 Vector Operations (AI-Optimized)

- `VADD`, `VSUB`, `VMUL`, `VFMA` (Fused Multiply-Add).
- **Predication:** All vector ops take an optional predicate.
  - Example: `VADD.P1 V1, V2, V3` (Adds V2+V3 into V1 only where P1 is 1).
- **Widening/Narrowing:** Support for mixed-precision (e.g., BF16 inputs -> FP32 accumulation) critical for AI Inference.

Chapter 2: Execution Environment & State Management

2.1 Dual-Mode Operation Principle

The processor operates in one of two mutually exclusive instruction decoding modes. The transition between these modes is hardware-managed and atomic.

1. **Legacy Mode (CISC Host):** The default state upon Reset/Boot. The CPU fetches variable-length x86 instuctions. All Privileged operations (Ring 0), Interrupt handling, and System Management Mode (SMM) routines execute strictly in this mode.
2. **DREL Mode (RISC Guest):** A restricted User-Mode (Ring 3) state. The CPU fetches fixed-width 32-bit DREL instructions. Privileged instructions are illegal.

2.2 Mode Switching Mechanism

2.2.1 Enablement (OS Opt-in)

To prevent legacy software from accidentally entering DREL mode, the OS must explicitly enable the capability via a Model Specific Register (MSR).

- **MSR `IA32_DREL_CTL` (Address `0xC0000080`):**
  - Bit 0 (`ENABLE`): Global enable bit for DREL extensions.
  - Bit 1 (`XSAVE_EXT`): Confirms OS support for saving DREL context via `XSAVE`.

2.2.2 Entering DREL (`DREL_ENTER`)

Entry is voluntary and synchronous, initiated by the application.

- **Instruction:** `DREL_ENTER <Target_RIP>`
- **Semantics:**
    1. Validates that `IA32_DREL_CTL.ENABLE` is set.
    2. Flushes the Legacy Pre-decode queue.
    3. Switches the Instruction Fetch Unit to 4-byte aligned boundaries.
    4. Updates the Instruction Pointer (RIP) to `<Target_RIP>`.
    5. Aliasing: Maps x86 GPRs (RAX, RCX...) to DREL GPRs (R1-R15) to pass arguments without memory overhead.

2.2.3 Exiting DREL (The "Red-Line" Protocol)

Exit behavior is designed to guarantee OS stability.

- **Voluntary Exit:** The instruction `DREL_EXIT` restores the Legacy Mode decoder and returns control to the instruction following the original `DREL_ENTER` call.
- **Involuntary Exit (Interrupts/Exceptions):**
    - Any hardware interrupt (Timer, I/O), Exception (Page Fault, GPF), or Syscall instruction triggers an immediate **Atomic Exit**.
    - **Hardware Behavior:**
        1. The CPU forces a transition back to Legacy Mode.
        2. The architectural state (DREL RIP, Flags) is saved to the interrupt stack frame or a shadow specific area.
        3. The Legacy Interrupt Descriptor Table (IDT) handler is invoked.
    - *Result:* The OS kernel **always** sees a standard x86 exception frame. It does not need to know a DREL instruction caused the fault, preserving kernel ABI compatibility.

2.3 Architectural State Mapping (Aliasing)

To ensure zero-latency transitions, DREL registers are hardware aliases of the physical x86 register file. No data copying occurs during `DREL_ENTER`.

DREL Register	x86 Physical Alias	Usage Convention
R0	*Hardware Zero*	Constant Zero
R1	RAX	Return Value / Arg 0
R2	RCX	Argument 1
R3	RDX	Argument 2

R4 - R15	RBX, RSI, RDI, R8-R15	General Purpose
R16 - R29	*Extended Physical File*	DREL-only storage (Saved via XSAVE)
R30 (LR)	*Internal Shadow Reg*	Link Register
R31 (SP)	RSP	Stack Pointer (Shared)

- **Vector Registers (V0-V31):** Directly alias to the physical register file backing AVX-512/AMX (ZMM0-ZMM31).

2.4 Privilege Levels

DREL is strictly a **CPL=3 (User Mode)** technology.

- Attempting to execute privileged instructions (e.g., `WRMSR`, `HLT`, `MOV CR3`) inside DREL mode triggers an `EXIT_REASON_PRIV_VIOLATION` exception, handing control back to the OS kernel trap handler.

Chapter 3: System Integration (OS & Loader)

3.1 Binary Container Format

DREL does not require a proprietary executable format. Instead, it leverages existing formats (PE/COFF for Windows, ELF for Linux) by introducing a dedicated section type. This ensures backward compatibility with existing linkers, debuggers, and build tools.

3.1.1 The `.drel` Section

Compilers targeting DREL shall place all RISC machine code into a specific section named `.drel` (or `.xrisc`).

- **Attributes:**
  - `READ` (Readable): Yes.
  - `EXECUTE` (Executable): Yes.
  - `WRITE` (Writable): **NO**. (Strict W^X enforcement).
- **Alignment:** 4KB (Page Aligned). This allows the OS to apply granular page table permissions strictly to DREL code pages, distinct from x86 code pages.

3.2 OS Loader Responsibilities

The Operating System Loader (e.g., `ntdll.dll` or `ld-linux.so`) requires minimal modification to support DREL.

1. **Mapping:** When loading the binary, the loader maps the `.drel` section into the virtual address space.
2. **Permission Hardening:** The loader **must** mark the pages backing the `.drel` section as `PAGE_EXECUTE_READ`.
3. **Relocations:** Since DREL code is position-independent (PIC), usage of absolute addresses is discouraged. If relocations are necessary, the standard relocation table of the host binary is used to patch the DREL section *before* marking it executable.

3.3 Security & Integrity

Because DREL bypasses the legacy instruction decoder (a common point of deep packet inspection for AV software), the integrity of the DREL code block is paramount.

3.3.1 Immutable Code Policy

DREL explicitly forbids **Self-Modifying Code (SMC)**.

- The hardware instruction fetch unit for DREL Mode will verify that the memory page has the `Dirty` bit cleared (or Read-Only attribute set).
- Attempting to execute code from a Writable page in DREL mode triggers an `EXIT_REASON_SECURITY_FAULT`.

3.3.2 Code Signing

To prevent "Polyglot" malware (where safe x86 code hides malicious DREL code), the OS Loader should verify a cryptographic signature covering the `.drel` section.

- **Mechanism:** Standard OS code signing (Authenticode / ELF signatures).
- **Enforcement:** In high-security environments (e.g., Kernel Mode DREL or Driver context), the loader must reject the binary if the DREL section hash does not match the signature.

Chapter 4: Security & Virtualization

4.1 Control Flow Integrity (CFI)

DREL implements hardware-enforced Control Flow Integrity to mitigate modern exploit techniques such as Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP).

4.1.1 Shadow Stack (Return Integrity)

Upon entering DREL mode (`DREL_ENTER`), the hardware automatically activates a Shadow Stack mechanism if enabled by the OS.

- **Operation:** Every `CALL` instruction pushes the return address to both the architectural stack (R31/SP) and a protected, hardware-managed internal Shadow Stack.
- **Verification:** Every `RET` instruction compares the return address on the architectural stack against the Shadow Stack.
- **Fault:** A mismatch triggers an immediate `EXIT_REASON_CFI_VIOLATION`, preventing ROP chain execution.

4.1.2 Indirect Branch Tracking (Forward Edge)

To prevent JOP attacks (jumping to arbitrary code gadgets), DREL enforces strict landing rules for indirect branches (`JALR`, `JMP Register`).

- **Landing Pad (`LPAD`):** The target of any indirect jump *must* be a specific `LPAD` instruction.
- **Enforcement:** If the Instruction Fetch Unit detects a jump target that is not `LPAD`, it triggers a security fault. This restricts attackers to valid function entry points only.

4.2 Side-Channel Defense (Spectre/Meltdown)

Legacy x86 CPUs suffer from speculative execution vulnerabilities due to the complexity of speculative access permission checks. DREL simplifies this surface.

1. **Strict Ring 3 Isolation:** DREL is architecturally incapable of generating Privileged (Ring 0) memory requests. The hardware memory management unit (MMU) for DREL mode can essentially hard-wire the "User Mode" bit, making Meltdown-style kernel data leaks architecturally impossible.
2. **Speculative Barriers:** DREL introduces a lightweight `FENCE.SPEC` instruction. This instruction halts speculative execution until all prior branch resolutions are retired, allowing compiler-inserted mitigations for Spectre V1/V2 without the heavy cost of a full pipeline flush.

4.3 Virtualization Support

DREL is designed to be "Virtualization Native," requiring minimal changes to Hypervisors (KVM, Hyper-V, Xen).

4.3.1 Zero-Cost State Switching

Because DREL registers (R0-R15, V0-V31) alias directly to the physical x86/AVX register file (as defined in Chapter 2), the Hypervisor does not need to manage a separate "DREL State".

- **VMEXIT/VMENTER:** The standard x86 `XSAVE/XRSTOR` logic automatically saves and restores the guest's DREL context. The Hypervisor treats a DREL guest simply as a standard x86 guest utilizing AVX registers.

4.3.2 Nested Virtualization

DREL fully supports nested virtualization.

- **CPUID Virtualization:** A Hypervisor can toggle the `IA32_DREL_CTL` availability bits in the virtualized CPUID. This allows cloud providers to emulate DREL presence on legacy hardware (via binary translation) or hide it on supported hardware for live migration compatibility.
- **Exit Forwarding:** If a guest executes `DREL_ENTER` and the Hypervisor traps it (e.g., for instrumentation), it generates a standard VMEXIT, allowing the Hypervisor to inspect or emulate the request.

Chapter 5: Heterogeneous Compute & GPU Bridge

5.1 Philosophy: The CPU as a Compute Controller

In traditional x86 architectures, the CPU is merely a driver host, spending significant cycles marshalling data and API calls (CUDA/OpenCL) through heavy userspace drivers. DREL inverts this model. By exposing a low-latency RISC interface, the CPU acts as a direct **Command Processor** for the GPU, capable of feeding the GPU at the speed of the hardware interconnect (PCIe/CXL).

5.2 Shared Virtual Memory (SVM)

To eliminate the "Memory Wall" between Host RAM and Device VRAM, DREL mandates a Unified Addressing Model.

5.2.1 IOMMU Integration

When a process enters DREL mode via `DREL_ENTER`, the hardware enforces strict synchronization with the IOMMU (Input-Output Memory Management Unit).

- **Single Address Space:** A pointer `0x1000` in DREL CPU code points to the exact same physical data as pointer `0x1000` in GPU code.
- **Zero-Copy Passing:** DREL applications pass data to the GPU by passing pointers, not by copying buffers. The GPU accesses system memory directly via PCIe (or CXL) caching protocols.

5.3 Direct Doorbell Mechanism

DREL eliminates the high latency of kernel launch found in traditional drivers (the "Driver Overhead").

- **Memory-Mapped Command Queues:** The GPU's command ring buffers are mapped directly into the DREL process address space.
- **The `DBELL` Instruction:** DREL introduces a specialized `DBELL` (Doorbell) instruction.
    - Syntax: `DBELL Rs1, Rs2` (Write value Rs2 to MMIO address Rs1 with "Write Combining" hint).
    - **Effect:** This performs an atomic, non-cached write to the GPU's doorbell register, signaling that new work is ready. This bypasses the OS kernel and the GPU userspace driver entirely for work submission.

5.4 Smart Streaming: Breaking the "VRAM Wall"

This feature targets the limitation where large AI models (e.g., 70B+ parameters) cannot fit into consumer GPU VRAM.

5.4.1 Pinned Memory Pre-fetching

DREL introduces "Smart Streaming" logic to utilize System RAM as an efficient Tier-2 memory for the GPU.

- **Mechanism:** Instead of demanding the entire model reside in VRAM, the DREL CPU executes a helper thread that manages a rolling buffer in VRAM.
- **Operation:**
    1. DREL CPU calculates which model layers are needed next.
    2. Using DMA engines (via `DBELL`), it asynchronously pushes the next layers from System RAM (DDR5) to GPU VRAM over PCIe Gen5/6.
    3. Compute overlaps with transfer.
- **Result:** A GPU with 24GB VRAM can execute a 100GB model with minimal performance penalty, as the DREL CPU "feeds" the GPU just-in-time, far more efficiently than a bloated CISC driver could.

5.5 The "Canonical" Compute Pipeline

To bypass proprietary stacks (like CUDA), DREL defines a standard compute flow:

1. **Source:** Compute Kernels are compiled to **SPIR-V** (industry standard).
2. **Host Code:** Written in DREL Assembly/C.

3. **Execution:**
    - DREL CPU loads the SPIR-V kernel.
    - DREL CPU writes the kernel pointer to the GPU Ring Buffer.
    - DREL CPU executes `DBELL`.
    - GPU executes the kernel.
    - *No NVIDIA/AMD proprietary driver logic is involved in the hot path.*

Chapter 6: Ecosystem & Verification

6.1 Compiler Infrastructure

Support for DREL is designed to integrate seamlessly into existing toolchains (LLVM, GCC, MSVC) as a secondary target architecture alongside the primary x86 host.

6.1.1 The `-mdrel` Target Flag

Compilers shall implement a specific code generation flag (e.g., `-mdrel` or `__attribute__((drel))`) to mark functions eligible for RISC compilation.

- **Function Multi-Versioning (FMV):** When `-mdrel` is active, the compiler should generate two versions of the function:
    1. **Legacy Path:** Standard x86 instructions (fallback).
    2. **DREL Path:** Optimized 32-bit fixed-width instructions.
- **Runtime Dispatch:** A lightweight resolver checks the `IA32_DREL_CTL` MSR bit at runtime to select the optimal path, ensuring binaries remain portable across non-DREL hardware.

6.1.2 Intrinsic Support

To allow fine-grained control over DREL specific features (like the `DBELL` instruction or `FENCE.SPEC`), the compiler must provide C-level intrinsics.

- Example: `__drel_dbell(void* addr, uint64_t val)` maps directly to the `DBELL` opcode.

6.2 Assembler Syntax Standard

To ensure interoperability between different assemblers (GNU as, NASM, LLVM MC), DREL mandates a standardized assembly syntax based on established RISC conventions.

- **Format:** `MNEMONIC Destination, Source1, Source2`
- **Register Naming:**
    - Integer: `%r0` - `%r31` (or aliases `%sp`, `%lr`)

    - Vector: `%v0` - `%v31`
    - Predicate: `%p0` - `%p7`
- **Example Snippet:**

```
.section .drel
my_fast_loop:
    LD.W   %r1, [%r2 + 4]      ; Load word from R2+4 into R1
    ADD    %r3, %r1, %r5       ; R3 = R1 + R5
    VADD.P1 %v1, %v2, %v3      ; Vector Add (masked by P1)
    BNE    %r3, %r0, my_fast_loop ; Branch if R3 != 0
    DREL_EXIT                  ; Return to Legacy Mode
```

6.3 Conformance & Verification

For a hardware implementation to be certified as "DREL Compliant", it must pass the **DREL Architecture Compliance Suite (DACS)**.

6.3.1 Architectural Litmus Tests

The DACS includes tests specifically targeting the DREL memory consistency model (TSO).

- **Store Buffer Visibility:** Verifies that stores in DREL mode are visible to legacy x86 threads in the correct TSO order.
- **Atomicity:** Verifies that `LOCK` prefixed instructions in x86 and Atomic AMOs in DREL respect the same cache coherence protocol (MESI).

6.3.2 Illegal Encoding Fuzzing

Hardware must strictly adhere to the decoding rules defined in Chapter 1. The verification suite includes a "Fuzzer" that feeds random 32-bit patterns to the DREL decoder to ensure that *any* undefined pattern triggers an `EXIT_REASON_ILLEGAL_ENC` and does not hang the pipeline or cause undefined behavior.

6.4 Debugging Support (DWARF Extensions)

Debuggers (GDB, LLDB) require extensions to handle the mode switching.

- **Mixed-Mode Unwinding:** Stack unwinding information (DWARF `.debug_frame`) must support frames where the Instruction Pointer (IP) is in the `.drel` section.
- **Register View Switching:** When the debugger detects the CPU is in DREL mode, the UI should automatically switch from showing EAX/EBX to showing R0-R31, reflecting the aliasing defined in Chapter 2.

Chapter 7: Governance & Roadmap

7.1 Specification Governance

The DREL specification is maintained as an Open Standard. Evolution of the architecture follows Semantic Versioning (SemVer) principles to ensure stability for hardware implementers and software toolchains.

- **Version Format:** `Major.Minor` (e.g., v1.0).
  - **Major:** Introducing breaking changes to the encoding or memory model (Requires recompilation).
  - **Minor:** Backwards-compatible extensions (e.g., new vector instructions, new MSRs).
- **Extension Policy:** All future extensions must be **Additive**. Removing or changing the behavior of existing opcodes is strictly prohibited to guarantee binary compatibility of DREL applications across hardware generations.

7.2 Development Roadmap

The path from concept to silicon is defined in four distinct phases:

- **Phase 1: Specification (Current Status)**
  - Definition of ISA, Encoding, and System Interface.
  - Publication of "Prior Art" to establish open architectural baseline.
- **Phase 2: Emulation & Validation**
  - Development of a QEMU fork supporting `DREL_ENTER` and DREL instruction decoding.
  - Creation of the DREL Compliance Suite (DACS) unit tests.
- **Phase 3: FPGA Implementation**
  - Soft-core implementation (e.g., modified RISC-V core acting as a DREL co-processor) to measure die area and power efficiency.
- **Phase 4: Silicon Integration**
  - Integration into commercial x86 silicon as an experimental "Dev Mode" feature.
  - Full OS enablement in Linux Kernel mainline.

Chapter 8: License & Attribution

8.1 Licensing Model

This architecture specification is explicitly released into the public domain of engineering under a permissive license to encourage adoption, innovation, and implementation by any hardware vendor (Intel, AMD, VIA, Zhaoxin, etc.) or software developer.

8.2 Attribution Requirement

8.3 Patent Disclaimer (Prior Art)

This document constitutes **Prior Art**. The specific mechanisms described herein—including but not limited to the Dual-Mode x86/RISC transition, the `DREL_ENTER` mechanism, and the aliasing of x86 GPRs to a fixed-width RISC ISA—are publicly disclosed to prevent the monopolization of these concepts via future patents by third parties.