ChatGPT

Hardware Realization: Interfacing BDI with Silicon

Introduction

Bridging the **Binary Decomposition Interface (BDI)** model from abstract semantics into real hardware requires carefully designing a hardware interface layer. This involves defining a **Hardware Abstraction Layer (HAL)**, a **BDI Instruction Set Architecture (ISA)** with a custom assembly language (CAL), and implementing custom mnemonics/opcodes that map BDI operations to physical processors and devices. The goal is to achieve a thin but crucial abstraction that allows BDI's virtual machine to run on multiple architectures (x86-64, ARM AArch64, RISC-V, FPGAs) without changing its semantics. In traditional computing, a hardware abstraction layer is *"a layer of programming that allows a computer OS to interact with a hardware device at a general or abstract level rather than at a detailed hardware level"* [1] . The BDI HAL follows this principle but is specialized to the BDI virtual machine's needs. It standardizes how BDI's high-level **SYS_* operations** (special BDI opcodes for system/hardware actions) are carried out on each platform, ensuring a stable interface between the BDI virtual runtime and the silicon.

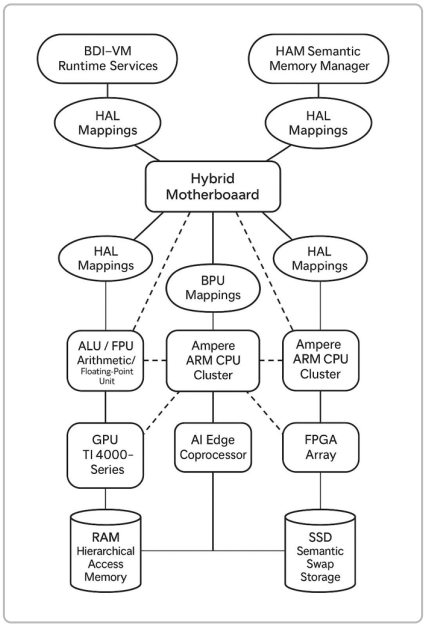

Figure: Simplified depiction of BDI HAL integration in a heterogeneous system. HAL mappings serve as translators between BDI-VM runtime services and various hardware units (CPUs, GPUs, FPGAs, etc.), standardizing commands across x86, ARM, RISC-V, and custom accelerators.

The rest of this chapter details the design and implementation of the BDI HAL and ISA. We discuss how the HAL is defined in a platform-independent manner (via `HardwareAbstractionLayer.hpp`) and how it differs from conventional OS HALs. We then break down examples of implementing BDI's HAL on **x86-64**, **ARM AArch64**, **RISC-V**, and **FPGA** platforms – highlighting use of control registers, special instructions (e.g. MSRs on x86, MRS/MSR on ARM, CSRs on RISC-V), interrupt controllers (APIC/GIC/PLIC), memory

management (MMU page tables and TLB flushes), and FPGA reconfiguration. A comparative summary table is provided to illustrate the challenges and trade-offs across these platforms. Finally, we discuss forward-looking hardware design principles inspired by BDI: using simpler cores with local SRAM, high-bandwidth interconnect fabrics, and direct hardware support for BDI operations and metadata. Throughout, we include references to official documentation from Intel, AMD, ARM, RISC-V, and FPGA vendors to ground the discussion in real-world architectural details.

BDI Instruction Set Architecture (ISA) and Custom Assembly Language (CAL)

A cornerstone of realizing BDI on hardware is defining a **custom ISA and assembly language** for the BDI model. This BDI-ISA includes the unique opcodes and mnemonics that express BDI's graph-based operations and its SYS_* system calls. The custom assembly language (CAL) provides human-readable mnemonics for these opcodes, similar in spirit to how x86 or RISC-V assembly represents machine instructions. However, unlike conventional ISA design (which must remain fixed for decades of processors), the BDI ISA is designed at a higher semantic level – its instructions manipulate BDI graph nodes, metadata, and perform semantic operations (like spawning a subgraph or tagging memory regions) that have no direct analogue in traditional ISAs.

Each BDI opcode is implemented such that it can be **fidelity-preserving** across architectures – i.e., the effect of the opcode is the same no matter the underlying CPU architecture. For example, a BDI instruction to allocate a new memory segment (`SYS_ALLOC`) or to trigger a synchronization barrier in the BDI graph (`SYS_SYNC`) must behave identically whether running on x86-64, ARM, or RISC-V. To achieve this, the CAL defines these instructions abstractly, and the HAL provides the concrete implementation on each platform. This separation is analogous to how high-level language commands are implemented via different system calls on different OS kernels, but here it is at the ISA level. The BDI CAL uses custom mnemonics (e.g. `MOVG r1, r2` for "move graph node", `SYS_RECFG x` for FPGA reconfiguration, etc.) that are not found in standard processor manuals – they are defined by the BDI specification. Implementers must map these to real instructions or sequences thereof on the target CPU. Fortunately, modern CPUs increasingly allow customization (for instance, RISC-V explicitly reserves opcode space for user-defined extensions [2] [3]), meaning BDI's opcodes could even be introduced as custom extensions in future processors for direct hardware execution. In current implementations, they are handled by the HAL at runtime (often by trapping or intercepting and then executing equivalent low-level code).

Designing the BDI ISA took inspiration from existing ISAs but with an eye toward BDI's unique needs. Just as the Intel and AMD manuals enumerate instructions and their encodings in a programmer's reference, the BDI specification enumerates its graph operation codes and SYS calls. For instance, an **Intel x86-64 ISA** instruction like `WRMSR` (Write Model-Specific Register) directly corresponds to a specific opcode and bit pattern [4]. Similarly, the **RISC-V ISA** defines instructions like `CSRRW` for control status registers [5]. In BDI's ISA, we define each instruction's mnemonic and a numeric opcode (or bytecode). A simple example mnemonic could be `GNOP` ("Graph Node Operation") with variants for different operations on graph nodes. Each mnemonic is assigned an opcode value, much like assembly instructions in other architectures. A custom assembler for BDI can translate human-readable assembly into these opcodes for the BDI virtual machine (BDIVM) to execute. Because BDI's machine model is architecture-independent, its CAL is the same regardless of host hardware – which greatly eases portability. The heavy lifting is then done by the HAL and the underlying host, which we describe next.

BDI Hardware Abstraction Layer (HAL) Architecture

The **Hardware Abstraction Layer (HAL)** in BDI is a thin, specialized layer that implements the semantics of BDI's system-level opcodes (the SYS_ *instructions) on actual hardware. Unlike a conventional OS HAL that might abstract general device drivers or provide broad hardware APIs, the BDI HAL is tightly scoped: it knows about BDI's virtual machine semantics and provides just the functionality needed to realize those semantics on a given architecture. The HAL is defined in an architecture-independent header (e.g.* `HardwareAbstractionLayer.hpp`*), which declares a standard interface for all HAL implementations. In essence, it's like an abstract base class defining functions for each SYS_* operation or other hardware access needed by BDI. For example, it may declare methods like `hal_writeRegister(id, value)`, `hal_triggerInterrupt(vector)`, `hal_configureFPGA(bitstream)`, etc., corresponding to BDI operations. Each platform (x86, ARM, RISC-V, etc.) has its own implementation of this interface, translating those abstract operations into the correct low-level sequence of instructions and I/O for that platform.

Crucially, BDI's HAL is *architecture-agnostic* at the interface level – the BDI core logic calls the same HAL methods regardless of running on an Intel Xeon, an ARM Cortex, or a RISC-V core. This differs from conventional OS HALs which typically still assume a single architecture and abstract only devices. Here, the HAL itself has multiple concrete implementations selected at compile-time or runtime based on the host CPU type. The benefit is that BDI's upper layers (the VM runtime, scheduler, memory manager, etc.) do not need to change when porting to a new architecture; only a new HAL implementation needs to be written for that architecture. The HAL ensures that *"platform-specific nuances (x86, ARM, RISC-V, SPIR-V, NVCC, etc.) are abstracted while preserving opcode and binary-level fidelity"* [6] . In other words, the meaning of a BDI opcode is preserved down to the binary behavior, even though the actual instructions executed will differ. The HAL translates the *semantic* intent of the BDI operation into *architecture-specific* actions.

To illustrate, consider a BDI operation `SYS_YIELD` that hints the hardware to switch context or save state. On x86-64, the HAL might implement this by invoking a specific sequence involving writing to a model-specific register or triggering an interrupt to the scheduler. On ARM, it might write to a system register (via MSR) to yield to the hypervisor or use a supervisor call. On RISC-V, it could write to a control CSR or issue an `ecall`. Each of these is different at the machine code level, but the HAL hides those details under one consistent `sys_yield()` interface that BDI core uses. The HAL also manages privileged operations: BDI's VM might run in user mode, so when a BDI SYS opcode needs something privileged (like changing page tables or configuring an interrupt controller), the HAL (running with appropriate privilege or as part of a kernel module) performs that on behalf of BDI.

HAL vs. Conventional OS HALs

It's worth emphasizing how **BDI's HAL differs from a traditional OS HAL**. In a typical operating system, the HAL abstracts hardware differences so that the same OS code can run on different motherboards, CPUs, or devices. For example, Windows' HAL abstracts interrupt controllers and timers so the OS can run on various PC hardware. BDI's HAL, by contrast, is not concerned with abstracting every device or supporting arbitrary OS services – it is laser-focused on enabling the *BDI virtual machine runtime* to interface with hardware. Thus, it doesn't expose a general device driver model or file I/O or network sockets; instead, it exposes operations like "allocate memory chunk for BDI heap," "load this BDI subgraph into an FPGA unit," "notify on-chip scheduler of an event," etc. These operations have no meaning outside the BDI context. The HAL essentially serves as a **bridge between BDI's semantic world and the CPU/SoC's control mechanisms**.

Another difference is stability and portability: An OS HAL often can be bypassed or altered when performance tuning for specific hardware (drivers can have custom code paths). In BDI, the HAL is more strictly the sole gateway to hardware for the VM – by design, it's the *only* place where architecture-specific code resides. This makes the BDI HAL a **stable semantic interface**: as new hardware emerges, one can implement the HAL for it, and the same BDI software (the same bytecode for the BDI-VM) runs on it, leveraging new capabilities without needing changes in the BDI program. In this way, BDI acts as a kind of "virtual ISA" on top of physical ISAs.

Finally, the HAL is extremely thin – it often corresponds to just a handful of instructions or register accesses per operation. It doesn't duplicate large OS subsystems. Instead, it may call into the host OS for help (for instance, to allocate physical memory or to load a driver), but from BDI's perspective those details are hidden. The HAL implementation may live in a privileged firmware or a kernel driver, depending on deployment, whereas the BDI VM can run in user space. This separation is reminiscent of paravirtualization: the HAL is like a lightweight hypervisor that provides BDI with its needed primitives on the hardware.

With the general architecture of the HAL in mind, let us examine how one would implement this on specific platforms.

Implementing BDI HAL on x86-64

On **x86-64 (Intel/AMD64)** platforms, implementing BDI's HAL involves tapping into the rich control features of the architecture: control registers (CR0–CR4), Model-Specific Registers (MSRs), and memory-mapped I/O (MMIO) for devices. The HAL for x86-64 must operate largely at privilege level 0 (ring 0) because it needs to, for example, configure page tables or interrupt controllers. Typically, this means the HAL code might be part of a kernel module or hypervisor that the BDI runtime calls into for SYS operations.

Control Registers: The x86 control registers are used for critical system configuration. For instance, `CR3` holds the physical base address of the page table hierarchy when paging is enabled [7]. A BDI HAL operation like setting up a new memory context might involve writing a new value to CR3 (to switch address spaces or load a BDI-specific page table). Similarly, enabling or disabling certain CPU features could involve CR0 or CR4 flags. The HAL would use privileged instructions like `MOV CRx` to manipulate these. (For example, to activate a new BDI memory mapping, HAL could prepare a page table in memory and then execute `mov cr3, rax` with `rax` pointing to the new PML4 table base – causing the CPU to use that for address translation.) Because writing CR3 also implicitly flushes the TLB (Translation Lookaside Buffer) on x86, this is a convenient way to ensure memory changes take effect [7].

Model-Specific Registers (MSRs): MSRs are another powerful feature on x86-64 that HAL leverages. MSRs control various CPU functionalities (e.g., system interrupts, performance counters, firmware configuration). They are accessed via the `RDMSR` and `WRMSR` instructions. According to Intel's manual, *"the RDMSR (read model-specific register) and WRMSR (write model-specific register) instructions allow a processor's 64-bit model-specific registers (MSRs) to be read and written, respectively."* [4] Many system features in x86 (like the APIC – Advanced Programmable Interrupt Controller – base address, or kernel GS base for segmentation) are exposed as MSRs. Suppose BDI's HAL needs to configure an interrupt vector for a BDI event; on x86 it might write to the local APIC's registers. The local APIC is often accessed either via memory-mapped registers or certain MSRs (in x2APIC mode, APIC registers are accessed as MSRs). The HAL would use `WRMSR` with the appropriate MSR index and write the value to, say, set an interrupt vector or send an inter-processor

interrupt (IPI) to another core where a BDI thread is running. The ability to directly manipulate MSRs is key to implementing BDI's low-level ops on x86. (It goes without saying that the HAL must run with `CR4.SMXE` enabled if touching certain protected MSRs, otherwise a #GP fault would occur; the HAL ensures it has the correct privileges.)

Memory-Mapped I/O: x86-64 supports both port I/O (the legacy in/out instructions) and memory-mapped I/O. Modern systems (especially in 64-bit mode) prefer MMIO because it can use the normal memory instruction set and larger registers [8]. *"Memory-mapped I/O is preferred in IA-32 and x86-64 architectures because the instructions that perform port-based I/O are limited to one register... any general-purpose register can send or receive data to or from memory-mapped I/O, so memory-mapped I/O uses fewer instructions and can run faster than port I/O."* [9] For the HAL, this means that to communicate with hardware devices (say, an FPGA card or a specialized accelerator attached via PCIe), it will typically write to or read from memory addresses that correspond to device registers. For example, if BDI has a `SYS_FPGA_CONFIG(addr)` opcode to load a partial bitstream into an FPGA, the x86 HAL might map the FPGA's control registers into memory (using e.g. `mmap` on /dev/mem or via a driver) and then perform writes to those MMIO addresses to transfer the bitstream data. It can use normal MOV instructions to do this because the device's registers appear in the physical address space. The HAL must ensure memory writes to MMIO are ordered properly (often using techniques like writing uncached memory or using `sfence` or `mfence` instructions, because the CPU's memory ordering might reorder or buffer writes to devices [10]). Additionally, the HAL might use the `IN`/`OUT` instructions for certain legacy devices (like a debug port or legacy PIC), but those are less likely in a modern BDI scenario. Nonetheless, the HAL is capable of issuing those if needed (since it runs at ring 0, it has permission to use port I/O).

Example – Interrupt Handling: Let's consider how BDI's HAL might implement a `SYS_INT_SEND(vector, core)` operation on x86. If BDI wants to signal another BDI thread (or core) via a software interrupt, the HAL could leverage the local APIC. In x86, each core has a local APIC, and one can send IPIs by writing to the Interrupt Command Register (an MMIO register in the APIC's memory page, or accessible via MSR in x2APIC mode). The HAL would take the `vector` number from the BDI call, place it into the appropriate bits of the APIC ICR, and write to the APIC's memory-mapped address (commonly at `0xFEE00300` for ICR low on xAPIC) [8]. This would cause the hardware to deliver an interrupt to the target core. On the receiving side, the HAL would have set up an IDT entry (through another MSR or through the OS) so that the interrupt vector calls into a BDI handler. All these steps are complex, but encapsulated in the HAL. Official AMD/Intel docs (e.g., the Intel System Programming Guide) detail how to program the APIC and use MSRs like `IA32_APIC_BASE` [11], which the HAL implementer would reference.

Memory Management: For memory, beyond CR3 usage mentioned, the HAL might also use other control registers. CR4 enables various features (e.g. `CR4.PKE` for protection keys, or `CR4.SMEP/SMAP` for access protections), which HAL could toggle if BDI needs a certain memory semantics (like disallowing execution of certain memory regions). These would be done by updating CR4 flags carefully [12]. If BDI's memory model uses tagging or special caching, the HAL might use MSRs like `PAT` (Page Attribute Table MSR) to set memory types for certain physical pages. The AMD64 manuals and Intel manuals are replete with such mechanisms, and the HAL essentially acts as a user of those documented features to implement BDI semantics. It's thin in that it doesn't introduce new mechanisms—just orchestrates existing ones in service of BDI.

Overall, x86-64 offers a very feature-rich but complex environment for HAL. The challenge (and trade-off) here is dealing with legacy aspects and ensuring the HAL doesn't inadvertently interfere with the host OS.

Often, the BDI HAL on x86 might be integrated with a lightweight hypervisor that owns the machine, or run under an OS with driver privileges. The benefit is that almost anything can be accomplished given x86's extensive capabilities (from virtualization extensions to system management mode if needed), but the implementation must be very careful to maintain stability.

Implementing BDI HAL on ARM AArch64

On **ARM 64-bit (AArch64)** platforms, the HAL implementation centers around ARM's cleanly designed system registers and its standardized interrupt controller interface. ARM's privileged architecture is divided into exception levels (EL0 user, EL1 kernel, EL2 hypervisor, EL3 secure monitor), so the BDI HAL typically runs in EL1 or EL2 to have the required access to hardware. Key elements for HAL are the **MRS/MSR instructions** to access system registers, the **Generic Interrupt Controller (GIC)** for interrupt management, and instructions for managing page tables and TLB (such as TLBI and barrier instructions).

System Registers and MRS/MSR: ARM AArch64 doesn't have "control registers" like CR0, CR3, but it has a large bank of system registers (like `SCTLR_EL1` for control bits, `TTBR0_EL1` for page table base, `ESR_EL1` for exception status, etc.). These are accessed using the *MRS* (Move from System Register) and *MSR* (Move to System Register) instructions. For example, reading a system register is done via `MRS Xt, <sys_reg>` and writing via `MSR <sys_reg>, Xt`. As one ARM developer note succinctly says: *"Reading from a system register is done via* `mrs` *instruction."* [13] . The HAL uses these instructions heavily. If BDI's HAL needs to enable a feature in `SCTLR_EL1` (say, disable cache or enable an MMU feature for BDI's address space), it will execute an MSR to set the corresponding bit in `SCTLR_EL1`. If it needs to retrieve the core's ID or timer value, it might use MRS on registers like `TPIDR_EL0` (thread ID) or the counter registers.

For context switching or context saving (if BDI uses its own scheduler on bare metal), the HAL might read/write registers like `ELR_EL1` (return address from exception) or `SP_EL1` (stack pointer) to manage different BDI threads. All of these operations rely on the precise definitions in ARM's Architecture Reference Manual. The HAL implementer would refer to the ARM ARM (for example, the definition of `TTBR0_EL1` states it holds the base address of the translation table for lower addresses [14] , and the HAL would ensure to write the correct physical address there when switching memory context).

Generic Interrupt Controller (GIC): ARM's GIC is a standardized interrupt controller for multi-core ARM systems. For HAL, this is vital when implementing BDI's interrupt or event semantics. The GIC (v2, v3, etc.) consists of a **Distributor** that receives interrupts from devices and **CPU Interfaces** (or **Redistributors** in GICv3) that deliver interrupts to cores. The GIC is configured via memory-mapped registers. According to an ARM document, *"the GIC architecture is divided into two main parts, called the CPU Interface and the Distributor. The CPU Interface is responsible for sending IRQ requests received by the Distributor to the cores; the Distributor receives IRQ signals from I/O peripherals."* [15] . In practical terms, the HAL on ARM will write to GIC registers to enable or disable certain interrupts, set priorities, target interrupts to specific cores, and acknowledge/complete interrupts.

For example, if BDI has a `SYS_INT_ENABLE(dev_id)` to enable an interrupt from a certain device (say a sensor feeding data into the BDI graph), the HAL would write to the GIC Distributor's **Interrupt Set Enable Register** for that interrupt ID. These registers are usually memory-mapped at a known base (for GICv2 typically 0x1F00_0000 region, for GICv3 they are split between Redistributor frames and Distributor). The HAL might perform something like: `LDR X0, =0xF9000000` (base of GIC dist), then set the appropriate

bit in the set-enable register for the dev's ID. Likewise, sending a software-generated interrupt (SGI) to another core (perhaps to notify a BDI thread on core 1) involves writing to the *Software Generated Interrupt Register* with the target core mask and interrupt ID. All these operations use normal memory writes (since GIC registers are memory-mapped). The HAL must also handle the acknowledgement of interrupts: when an IRQ is raised to a core, the HAL's interrupt handler will read from the **Interrupt Acknowledge Register (ICC_IAR1 for EL1)** to get the interrupt ID, then after handling, write to the **End of Interrupt Register (ICC_EOIR1)** to signal completion [16]. By doing so, it interfaces with GIC just like an OS kernel would, but specifically to route interrupts into BDI's event system.

Page Tables and TLB Management: For memory management, the HAL on AArch64 works with page tables via registers like `TTBR0_EL1` and uses TLBI (TLB Invalidate) instructions. Suppose BDI's HAL needs to map a new memory region for a BDI subgraph's data. The HAL could allocate physical pages (via the OS or hypervisor), construct the necessary page table entries in memory, then write the base address of the new table into `TTBR0_EL1` (if running BDI in an isolated address space). After updating TTBR0, or even after just updating entries in an existing page table, the HAL must ensure old translations are cleared. ARM provides the TLBI instructions (e.g., `TLBI VMALLE1` to invalidate all stage1 entries, or address-specific TLBI instructions). The HAL would issue these at EL1. It also needs a context synchronization barrier: typically one does `DSB SY` (data sync barrier) after modifying page tables, then TLBI, then another `DSB SY` and an `ISB` (instruction sync barrier) to ensure completion. These steps are outlined in ARM's memory management documentation (and often in Linux kernel examples). Essentially, *after updating page tables, software must execute an SFENCE or TLBI equivalent to flush old translations*. In ARM, *"execute an SFENCE.VMA instruction after, or in some cases before, writing satp"* (to borrow RISC-V wording) [17] doesn't directly apply since ARM's "satp" is TTBR and the flush is TLBI – but the concept is analogous across architectures.

Concretely, if BDI's `SYS_ALLOC` opcode creates a new mapping, the HAL might do: write new PTEs in memory, `DSB ISHST`, then `TLBI VAAE1, Xn` (if invalidating a specific address Xn) or a full `TLBI VMALLE1`, then `DSB ISH` and `ISB`. This ensures the MMU sees the updated mapping. The HAL uses MSR/MRS to manipulate TTBR and related control registers (like `TCR_EL1` if it needs to adjust memory region sizes), all according to rules in ARM's manual.

Example – Sending a BDI Event via Interrupt: Imagine a BDI opcode `SYS_EVENT_SEND(core_id, code)` that should raise a lightweight event on another core. On ARM, the HAL might implement this using an IPI through GIC. If core 2 needs to be notified, the HAL will write to the GIC Distributor's SGI register to send an SGI (let's say interrupt ID 1, which by convention could be an IPI) to target core 2. The code might look like writing the value `(1 << 24) | (code & 0xF)` to the **Software Generated Interrupt Register** (where bits encode target list and interrupt ID). This causes GIC to issue interrupt `code` on core 2. The HAL on core 2 would have an ISR that catches this and then queues the event into BDI's scheduler.

In summary, the AArch64 HAL deals with a well-architected set of tools: system registers via MRS/MSR (for core configuration and status), memory-mapped GIC registers (for interrupt control), and explicit cache/TLB maintenance instructions for memory. The trade-offs here: ARM's design makes it relatively straightforward to do these tasks (compared to x86's sometimes arcane MSR indices or segmented memory concerns), but it requires the HAL writer to be familiar with ARM's privileged architecture. Also, ARM systems often run with a hypervisor or Linux under the hood; if BDI HAL is not running bare-metal, it may need to issue hypercalls for some things (e.g., configuring EL2 registers might need EL2 privileges). Still, the HAL's logic on ARM is clean and aligns closely with the official documentation of the ARM architecture, which ensures that implementing according to spec yields reliable behavior.

Implementing BDI HAL on RISC-V

RISC-V, being a modern open ISA, offers a minimalistic and modular approach that can work in BDI's favor. Implementing the HAL on a **RISC-V** platform leverages the ISA's **Control and Status Registers (CSRs)** and its simple interrupt and memory management scheme. RISC-V defines a range of CSRs for machine mode (m-mode), supervisor mode (s-mode), etc., and privileged instructions to access them. Additionally, many RISC-V systems use a **Platform-Level Interrupt Controller (PLIC)** for managing device interrupts, and the `sfence.vma` instruction for TLB management in its virtual memory system.

CSR Access (csrr/csrw): In RISC-V, privileged registers (like `mstatus`, `mtvec`, `satp`, etc.) are accessed via CSR instructions. The instructions `CSRRW`, `CSRRS`, `CSRRC` (and their immediate variants) allow reading/modifying CSRs. As the RISC-V Privileged Spec explains, *"The CSRRW (Atomic Read/Write CSR) instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR … then writes it to the integer register rd. The initial value in rs1 is written to the CSR."* [5] . The BDI HAL running in machine mode or supervisor mode will use these instructions to implement its operations. For example, to set up a new page table base for BDI's address space, the HAL can execute `CSRW satp, a0` (or the pseudo-instruction `csrw satp, a0`) where `a0` holds the new page table root (with proper format bits). To enable or disable interrupts, HAL might write to `mie` or `sie` CSRs (machine or supervisor interrupt enable registers) via CSRRS/CSRRc instructions, setting or clearing specific bits for timer or external interrupts.

If BDI's HAL needs to retrieve some hardware info (say the cycle counter or performance counter), it can read CSRs like `mcycle` or custom CSRs via instructions like `csrr t0, mcycle`. The HAL may also use the `ecall` instruction to invoke higher-level services if running at lower privilege; for instance, if BDI is running in user mode and HAL is partly in the OS, a `SYS_blah` opcode might trap to s-mode via ecall and then be handled there.

Because RISC-V encourages custom extensions, a futuristic approach could even define BDI's SYS ops as custom CSR operations or custom opcodes, but without loss of generality, the HAL can do everything using standard means.

PLIC (Platform-Level Interrupt Controller): The PLIC is an external interrupt controller used in RISC-V systems to handle multiple interrupt sources and route them to hart contexts. The PLIC is memory-mapped and is described in the RISC-V Platform-Level Interrupt Controller specification. At a high level, *"the PLIC prioritizes and distributes global interrupts in a RISC-V system"* [18] . The HAL interacts with the PLIC to implement interrupt-related BDI instructions. For example, to enable an interrupt ID for a device, the HAL writes to the PLIC's enable register for that context. If BDI uses an external device (like a sensor or network input to feed a graph), its interrupt would come through the PLIC. The HAL must set the priority for that interrupt (PLIC priority register), enable it for the target context (PLIC enable bitfield for the HART running BDI), and potentially set a threshold if needed (PLIC threshold register to filter interrupts below a certain priority [19]). When an interrupt arrives, the HAL will get it via the PLIC **claim** register – reading from the claim register gives the ID of the highest-priority pending interrupt and atomically marks it as claimed [20] . After servicing, the HAL writes the ID back to the completion register to signal the PLIC that it's handled [20] . All of this is done with simple memory reads/writes to the PLIC's address space, which the HAL knows (commonly at a fixed address like 0x0C00_0000 in many SoC implementations, but platform-dependent).

If BDI needs to send a software interrupt between cores, RISC-V also has the concept of *Software Interrupt* (one of the bits in `mie` CSR and delivered via the local Core-Local Interruptor (CLINT) in many systems, or via SBI calls in an OS context). The HAL could trigger a machine-mode software interrupt by writing to the CLINT's MSIP register for the target hart. This is one way to implement a BDI inter-thread signal (similar to an IPI). Alternatively, it could use the PLIC if software interrupts are exposed as global interrupts.

Memory Management and sfence.vma: RISC-V's virtual memory management is controlled by the `satp` CSR (holds page table base and mode) and TLB flushes are done with the `SFENCE.VMA` instruction. After the HAL modifies any page table entries for BDI's memory space, it must perform an SFENCE.VMA to ensure no stale translations remain. The RISC-V spec states: *"The supervisor memory-management fence instruction SFENCE.VMA is used to synchronize updates to in-memory memory-management data structures with current execution... Executing an SFENCE.VMA instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before certain implicit references by subsequent instructions in that hart to the memory-management data structures."* [21]. In simpler terms, SFENCE.VMA flushes the local TLB (for particular addresses or all, depending on arguments) and ensures that page table writes before the fence are seen by future memory accesses. The HAL will execute, for example, `sfence.vma x0, x0` (with x0 registers, meaning flush all translations for all address spaces) after a major change like switching to a new page table. If it only changed one page, it might do `sfence.vma t0, x0` where t0 has the virtual address, to flush just that page's entry. The BDI HAL might wrap this in a function `hal_flush_tlb()` that calls SFENCE.VMA appropriately. We cite the spec to emphasize the defined behavior: SFENCE.VMA flushes "local hardware caches related to address translation" and is analogous to TLB shootdown on other architectures [22] [23].

Example – Context Switch/Thread Scheduling: If BDI is running its own lightweight threading, the HAL might handle a `SYS_SWITCH(thread_id)` opcode by saving registers and switching context. On RISC-V, this could involve writing the current user context's PC and registers to memory, updating `satp` if threads have separate address spaces (or not, if they share address space), and then writing a new `mepc` (machine exception program counter) or `sepc` (supervisor EPC) CSR to point to the new thread's resume point, and finally eret (MRET or SRET) to switch. If BDI's thread scheduler is in m-mode, it might also set a timer via the CLINT (writing to mtimecmp CSR or memory-mapped register) to generate timer interrupts for preemption. All these involve CSR writes: e.g., `csrw mepc, newPC` to set where to resume. The HAL uses RISC-V's defined procedure for context switching, which is far simpler due to the lack of segmented memory or ring transitions (there are only the privilege levels and the trap/return instructions).

In implementing the HAL on RISC-V, one of the **trade-offs** is that RISC-V's simplicity means some features have to be implemented in software if needed. For example, if BDI wanted to use a feature analogous to x86's ring 0/3 separation, RISC-V can do it if running in s-mode vs u-mode, but if more fine-grained permission control is needed (like x86 IOPL or ARM's PAN), the HAL might not have hardware support and would need to emulate or do without. Conversely, RISC-V's clean slate means adding new BDI-specific instructions or registers is feasible: one could imagine a future RISC-V with a BDI extension that provides new CSR registers that directly count graph node activations or a custom accelerator for BDI operations. The current HAL, however, works within the standard means: CSRs, PLIC, and standard RISC-V privileged instructions. The official privileged specification is the guide for all these, and since it's open, BDI's developers could even propose changes to RISC-V if necessary.

Implementing BDI on FPGAs and Reconfigurable Hardware

Perhaps the most unique aspect of BDI's hardware realization is its ability to leverage **FPGAs** and other reconfigurable logic as first-class execution targets. In the BDI model, parts of the computation graph (subgraphs) can be offloaded to custom hardware for acceleration. Implementing the HAL for an FPGA means enabling the runtime to dynamically configure and communicate with the FPGA fabric.

BDI Subgraphs to Verilog: BDI's toolchain can compile subgraphs (segments of the computation that are suitable for hardware acceleration) into HDL (such as Verilog or VHDL) or directly into FPGA bitstreams. In fact, *"BDI emits Verilog or low-level bitstreams as part of its unified graph compilation"* [24] . This means the software side of BDI can generate the logic design for an FPGA that implements a particular operation or set of operations. The role of the HAL is to take that generated bitstream and actually load it into the FPGA at the right time, as well as to provide mechanisms for sending data to/from that FPGA logic.

Partial Reconfiguration via SYS ops: Modern FPGAs (Xilinx, Intel (Altera) etc.) support **partial reconfiguration (PR)**, which allows one portion of the FPGA to be reconfigured on the fly while the rest of the logic continues running [25] . BDI exploits this by treating the FPGA as a dynamically programmable co-processor. For example, if a BDI graph node is detected to be a hotspot, the system might compile it to hardware and then use a `SYS_RECONFIG(fpga_region, bitstream_ptr)` opcode to swap out a region of the FPGA with a new circuit implementing that node. The HAL on an FPGA-equipped system would implement this by interacting with either the FPGA's configuration port or the vendor's driver. On many systems, partial reconfig can be done by writing the bitstream data to a special memory interface (for instance, Xilinx FPGAs often expose a device like `/dev/xdevcfg` or now use PCAP via the ARM processor to load PR bitstreams; Intel FPGAs might use an Avalon interface or JTAG). The HAL could, for instance, memory-map the FPGA configuration registers and perform the sequence to start configuration, feed the bitstream, and finalize.

According to Intel's documentation, *"Partial reconfiguration (PR) allows you to reconfigure a portion of the FPGA dynamically, while the rest of the FPGA design continues to function. The portion that can be reconfigured is referred to as a PR region and the design to be loaded is a PR persona."* [25] . In BDI terms, a PR region might be reserved for BDI's dynamic kernels, and each persona corresponds to a particular subgraph's logic. The HAL manages the personas: it may have precompiled bitstreams for common operations, or compile on the fly (with some overhead). The **trade-off** here is the time it takes to reconfigure, which might be on the order of milliseconds – worth it for long-running changes, but not for very short-lived tasks. BDI mitigates this by overlapping computation and configuration where possible (the concept of dynamic reconfiguration is built into the model).

Communication and MMIO: Once a BDI subgraph is running on FPGA logic, the HAL needs to facilitate data movement. Typically, the FPGA design will include memory-mapped interface registers or FIFO buffers that the CPU can write to/read from to send inputs and get outputs. The HAL might use MMIO writes to deliver function arguments or data to the FPGA and poll for completion or use an interrupt. For example, if a BDI graph node "accelerator" on FPGA produces a result, it might raise an interrupt line that goes to the CPU (perhaps via the same GIC/PLIC mechanisms discussed, since the FPGA could be seen as just another device). The HAL would map that interrupt to a BDI event. Alternatively, the HAL could poll a status register via MMIO until it indicates done. The design choice depends on latency requirements and whether busy-waiting is acceptable. Given BDI's goal of high throughput, using interrupts or DMA signals is likely.

FPGA as a Compute Unit: In the context of the overall system, the FPGA might be one of several heterogeneous units. Recall the earlier diagram (Figure above) showing a "FPGA Array" alongside CPU clusters and GPUs in a **Hybrid Motherboard**. The HAL is responsible for treating the FPGA as another execution target. This means scheduling work onto it, ensuring data is moved to its local memory if needed (some FPGAs have attached RAM or use main memory via PCIe), and handling errors (if a bitstream fails or if the operation cannot be placed on FPGA due to resource limits, HAL must fall back to CPU execution perhaps).

Example – BDI Graph Execution on FPGA: Suppose a BDI operation is identified to run on FPGA. The HAL's steps could be: (1) Allocate a region on the FPGA (maybe choose a PR region or an available slot), (2) load the corresponding bitstream (HAL might call an FPGA driver or, if running in a soft SoC environment, directly manipulate configuration registers), (3) initialize any needed registers – for instance, write input data to BRAM mapped at some addresses, (4) start the operation by writing to a control register (like setting a "start" bit), (5) either busy-wait or wait for interrupt that indicates completion, (6) read back results from output registers or memory, and (7) pass results back to the BDI VM. All these steps happen under the hood of a single BDI `SYS_FPGARUN` or similar opcode invocation. To the BDI program, it looks like a single instruction that offloaded work and got an answer; the HAL hides the complexity of programming the FPGA.

FPGAs also bring **opportunities**: they could implement certain BDI operations more directly. For instance, if BDI's model has a concept of **metadata propagation** (extra information flowing along with data), the FPGA design could physically carry that metadata on wires alongside the data. In a CPU, metadata handling might take extra instructions and memory, but an FPGA can have a wider bus or custom logic to naturally propagate it. This hints at the future design principles we will discuss – that hardware can evolve to support BDI semantics natively.

The challenges in FPGA HAL implementation include the need for careful synchronization (partial reconfiguration must be carefully sequenced, and one must ensure the rest of the system is in a stable state or using fallback paths while a region is reconfiguring) and the management of bitstream sizes and speeds (a large bitstream can be tens of megabytes, which if sent over PCIe introduces latency). Techniques like compressing bitstreams or using incremental reconfiguration help. Also, ensuring security (one must not load an untrusted bitstream) and consistency (if two BDI threads both want the FPGA, some arbitration must happen) are practical concerns.

Nonetheless, the fact that BDI integrates FPGA reconfiguration as a routine operation is forward-looking. It treats hardware as malleable – an extreme form of hardware abstraction where even the logic gates can be repurposed at runtime via the HAL. Few systems aside from high-end reconfigurable computing frameworks attempt this today.

Comparative Summary of HAL Implementations across Platforms

Each platform – x86-64, AArch64, RISC-V, FPGA – offers distinct mechanisms and poses unique challenges for implementing the BDI HAL. Below is a comparative summary highlighting key aspects:

Aspect	x86-64 (Intel/ AMD)	ARM AArch64	RISC-V	FPGA (Reconfigurable)
Privileged Register Access	Multiple control registers (CR0–CR4, CR8) for global CPU state; MSRs for fine-grained control. HAL uses `MOV CRx` and `RDMSR/WRMSR` to configure CPU features [4]. Example: update `CR3` for new page table [7].	Dedicated system registers (SCTLR, TTBR, etc.) accessed via MRS/MSR [13]. Orthogonal design (separate registers for separate functions). Example: set `TTBR0_EL1` for new page table base, via `MSR TTBR0_EL1, Xn`.	Control and Status Registers (CSRs) accessed with atomic CSR instructions [5]. Fewer registers, but custom ones can be added. Example: write `satp` CSR for page table base with `csrw satp, a0`.	No traditional registers; configuration via special interfaces. The "registers" to control FPGA are memory-mapped config ports or driver APIs. HAL writes to config control to start PR, etc.
Interrupt Handling	Local APIC and I/O APIC (or x2APIC). HAL writes to APIC MSRs or MMIO to send interrupts. Legacy PIC for older systems. Must manage interrupt vectors in IDT. Complex but very flexible (IPI, etc.).	Generic Interrupt Controller (GIC). Standard MMIO interface [15]. HAL enables/disables interrupts by writing to Distributor registers, uses SGIs for IPIs. Must manage interrupt priorities and CPU interface registers (ICC_*). Straightforward due to standardization.	Platform-Level Interrupt Controller (PLIC) for external interrupts [18]. CLINT for timer and software interrupts. HAL writes to PLIC registers to route interrupts, reads claim/complete [20]. Simpler priority scheme. Software interrupts via CLINT MSIP (memory-mapped per hart).	Typically no built-in interrupt controller (unless SoC FPGA). If FPGA logic needs to interrupt CPU, it often connects to CPU's interrupt controller as an external line. HAL must integrate FPGA interrupts into APIC/GIC/PLIC of host. Otherwise uses polling.

Aspect	x86-64 (Intel/AMD)	ARM AArch64	RISC-V	FPGA (Reconfigurable)
Memory Management	4-level page tables (in 64-bit) with hardware walker. HAL creates/modifies PML4, etc. Use `INVLPGA` or setting CR3 to flush TLB. Supports advanced features (huge pages, PCID tags for TLB, etc.). More complex, legacy baggage (e.g., segmentation still partly exists).	2-3 levels of page tables (depending on granule). HAL writes new entries in memory, updates TTBR, uses TLBI instructions to invalidate TLB. Supports ASIDs (TLB tags) to avoid full flush on context switch. Generally simpler, with required barriers (DSB,ISB).	2-3 levels of page tables (Sv32, Sv39, Sv48 as needed). HAL writes entries, updates `satp`. Must execute `SFENCE.VMA` after changes to flush TLB [26]. Supports ASIDs (called VMIDs in satp in newer versions). Very minimal design.	If using soft CPUs on FPGA, similar to above. But if using pure logic, "memory management" is custom – e.g., explicit BRAM buffering. Partial reconfig can swap out logic connected to certain memory regions. HAL might manage DMA to move data in/out of FPGA memory. No TLB in pure logic (unless implemented), so no cache coherence unless via host.
I/O and Device Access	Both Port I/O (in/out) and MMIO. Modern usage is MMIO for devices (e.g., writing to PCIe config or BAR space) [8]. HAL can use any general register for MMIO, making it efficient. Needs memory barriers (e.g., `MFENCE`) to ensure ordering [10].	Exclusively memory-mapped I/O for on-chip devices. HAL writes to device registers (e.g., GIC, UART) through normal store instructions. Coherency with devices handled via explicit barriers (`DMB/DSB` for I/O). Simpler model (no separate I/O space).	Memory-mapped I/O as well. Simplicity of ordering – usually rely on `FENCE` instructions if needed for device I/O ordering. No separate I/O instructions, which aligns well with high-level abstraction (just use regular loads/stores).	FPGA logic may expose device-like registers. HAL likely goes through a driver (e.g., PCIe driver to talk to FPGA). The HAL might map an FPGA's control register and send commands by writing there. Bandwidth to FPGA might be lower than CPU local bus – uses DMA for bulk data.

Aspect	x86-64 (Intel/ AMD)	ARM AArch64	RISC-V	FPGA (Reconfigurable)
HAL Implementation Complexity	**High:** x86 HAL is challenging due to decades of legacy and intricacies (must handle protected mode, 64-bit mode differences, etc.). However, x86 provides many hooks (VMX for virtualization if needed, etc.). Development aided by extensive Intel/ AMD documentation [4] [7].	**Moderate:** ARM HAL is cleaner; well-documented system registers and GIC make implementation straightforward if one knows ARM ARM. Needs careful barrier use. Multi-EL (if not running at EL1) can complicate (might need EL2 help). Overall less quirky than x86.	**Low to Moderate:** RISC-V HAL is relatively small – the ISA is simple and one can run BDI in machine mode for full control. Biggest effort might be writing a simple PLIC driver, which is simpler than APIC/GIC. Also, one may extend ISA if needed. The openness of spec [27] [18] helps.	**High (different sense):** FPGA HAL complexity is not in code volume but in managing hardware configurations and timing. Must interface with vendor tools or IPs. Debugging is tricky (issues can be electrical/timing). However, logically the HAL code just orchestrates bitstreams and data movement.

Table: Comparison of HAL implementation aspects across x86-64, AArch64, RISC-V, and FPGA. Each platform offers different primitives to achieve BDI's requirements, leading to distinct challenges and optimizations.

As seen above, x86-64 provides powerful mechanisms but at a cost of complexity (e.g., dealing with MSRs and legacy modes), whereas ARM's and RISC-V's streamlined designs make the HAL more predictable and maintainable. FPGAs introduce a completely different paradigm of "programming hardware on the fly," which is powerful for BDI but requires careful management. Despite their differences, all platforms can support the BDI semantics through their HAL implementations – a testament to BDI's architecture-independent philosophy.

Future Hardware Design Principles Inspired by BDI

The experience of implementing BDI on current hardware reveals both the possibilities and limitations of today's CPUs and accelerators. BDI as a model hints at **future hardware design principles** that could make such semantic interfaces even more efficient. Some key principles include:

- **Simple, Decoupled Cores:** BDI doesn't require extremely complex out-of-order superscalar cores. In fact, a greater number of *simpler cores* might better serve a BDI system by allowing massive parallelism for graph node execution. These cores could be in-order or modestly superscalar but kept simple to reduce power and improve predictability. They serve as the "worker bees" executing fine-grained BDI tasks. This is in line with emerging designs like *Tesla's Dojo* and many-core accelerators, where *"the core is very simple... It has its own local SRAM, a dedicated interface to the wide*

network" [28] [29]. Simpler cores also switch context faster and can be specialized more easily (possibly even synthesized on FPGAs as needed).

- **Local SRAM and Scratchpads:** Rather than a huge shared cache hierarchy, BDI suggests using local memory close to each core to store the working set of graph nodes. Each core (or each small group of cores) might have a slice of **SRAM** that it uses as a scratchpad or an extended L1. This aligns with designs like Cerebras or Dojo, where each core includes a substantial local memory (e.g., **Dojo's core has 1.25 MB SRAM with multiple ports for high bandwidth** [30]). Local SRAM allows deterministic, high-speed data access for the core's portion of the BDI graph, reducing reliance on slower off-chip DRAM. In BDI, local memories could hold the state of subgraphs or frequently accessed metadata, enabling faster updates and reducing traffic on the global memory fabric.

- **High-Bandwidth Interconnect Fabrics:** To connect these many simple cores (and possibly specialized units like FPGAs or tensor units), a high-bandwidth network-on-chip or fabric is crucial. BDI workloads can involve frequent communication between graph nodes; if those nodes reside on different cores or accelerators, the interconnect must support large volumes of fine-grained messages. Future hardware could use mesh or torus networks, optical interconnects, or dedicated fabrics optimized for graph communication. The **wide network links** in systems like Dojo (which connects tiles over a proprietary mesh with high bisection bandwidth) are an example [31]. For BDI, one might design the fabric with hardware support for common communication patterns (like scatter-gather of data to multiple consumers, or reduction of results from multiple producers) since graph computing often requires those.

- **Direct Hardware Support for BDI Ops:** Perhaps the most impactful principle is building hardware instructions or units that directly implement BDI operations and manage BDI metadata. For instance, if BDI heavily uses a "graph node scheduler" that takes into account priority and aging (as hinted by the HAM memory manager's use of *entropy, attention, and RL to reorganize memory* [32]), hardware could accelerate that. A simple example: a hardware queue that prioritizes tasks based on a weight could replace a lot of HAL software logic for scheduling. Metadata like "age" or "priority" could be stored in special registers or attached to cache lines, allowing hardware to, say, evict least-used BDI objects first (implementing an intelligent cache according to BDI's needs).

Another example is **tagged memory**: BDI might tag memory with semantic identifiers. A future CPU might allow tags to be associated with each cache line or pointer, and have instructions that operate taking those tags into account (for security or GC or other purposes). Some research architectures (like CHERI for security or HWASan for debugging) already play with tagged memory – BDI could repurpose that idea for semantic tags. If such support existed, the HAL's job in managing metadata would simplify, as the hardware would carry and preserve tags automatically through loads/stores.

- **Flexible and Unified Memory Hierarchy:** BDI's HAL tries to implement a Hierarchical Access Memory (HAM) above the hardware. Future hardware might integrate the concept of multi-tier memory (fast critical memory vs slower archive memory) with hardware management. For example, a CPU could have both SRAM and stacked DRAM accessible, with the ability to pin certain data in SRAM for "critical" tasks, leaving less critical data to DRAM. This looks like cache QoS or tiered memory allocation in today's terms, but driven by semantic importance from BDI. If hardware offered ways to mark memory regions with priority or expected access frequency (perhaps via page

table bits or memory types), the BDI HAL could simply use those, rather than manually shuffling data between RAM and SSD as done now [33].

- **Reconfigurability as a First-Class Feature:** Instead of treating FPGAs as external devices, future CPUs might incorporate regions of reconfigurable fabric on-die (as some SoCs already do with FPGA fabric next to cores). This fabric could be context-switched or partially reconfigured quickly under program control. If BDI could tell the hardware "compile and load this function into reconfigurable logic," and the hardware had built-in capabilities to do so (perhaps by having a library of "FPGA" configurations or using HLS in real-time for small circuits), it would drastically cut the HAL complexity. We might envision a CPU with an internal FPGA whose configuration is managed by issuing special instructions or writing to certain control registers (with the heavy lifting done by microcode or dedicated configuration engines). BDI's frequent use of reconfiguration [24] would then be accelerated by hardware support for fast reconfig.

In essence, BDI's needs shine a light on what an ideal hardware for dynamic, graph-oriented, intelligent workloads might look like: lots of modest cores each with fast local memory, all tied together with an ultra-fast fabric, and augmented with configurable logic and rich metadata support. Intriguingly, these trends are echoed by some cutting-edge architectures in AI and HPC: for instance, Graphcore's IPU (Intelligence Processing Unit) uses many cores with local scratchpad memories for graph computation, and Cerebras's wafer-scale engine dedicates huge silicon area to memory close to compute. The industry is moving toward more specialized, heterogeneous designs – BDI simply provides a unifying semantic layer that could make use of them in a portable way.

Conclusion

The **Binary Decomposition Interface (BDI)** presents a novel approach to system architecture: it defines a stable, architecture-independent semantic layer (through its ISA and HAL) that can leverage an increasingly heterogeneous hardware landscape. By implementing a thin HAL for each platform, BDI ensures that its core abstractions – binary graph nodes, semantic memory management, SYScalls for dynamic behavior – remain consistent and reliable, irrespective of whether the underlying silicon is a 30-year-old x86 processor, a cutting-edge ARM SoC, a RISC-V core on an FPGA, or a custom accelerator. The HAL carries the weight of translation, mapping BDI's graph-centric operations to **control registers and MSRs on x86** [4], **MRS/MSR-accessible system registers and GIC on ARM** [15], **CSR instructions and PLIC on RISC-V** [18], and even **partial reconfiguration on FPGAs** [25]. In doing so, it allows the BDI-VM and its custom assembly language to function as a *"virtual hardware"* that is consistent across systems.

This hardware realization strategy yields a powerful advantage: **BDI becomes a stable semantic interface for evolving silicon.** As new hardware paradigms emerge – be it more advanced multi-core CPUs, novel memory technologies, or integrated photonic processors – one needs only to implement the HAL for those, and the wealth of BDI software and algorithms can immediately run on them, gaining benefits without a rewrite. It is analogous to how the C language allowed programs to be recompiled for any new CPU, or how the Java VM allowed "write once run anywhere" – but taken to the systems level, where even kernel-level behaviors and scheduling can be carried over. BDI's ISA and HAL essentially form a *virtual machine architecture* that decouples the progress of software semantics from the churn of hardware innovation.

Furthermore, by studying the HAL on current platforms, we identify bottlenecks and inefficiencies which guide future hardware designs (as discussed). The long-term vision is hardware-software co-design: future

CPUs might incorporate explicit support for the kinds of operations BDI needs (just as GPUs added tensor cores for AI because frameworks like TensorFlow demanded them). If BDI's model of computation proves powerful for AI and complex software (as the early research suggests, with concepts like **semantic swap storage and introspective memory management** [33]), hardware vendors could adopt its principles – e.g., include graph execution units, or memory hardware that directly supports hierarchical aging and eviction policies.

In the interim, the BDI HAL is the unsung hero enabling this experiment in universal computation across architectures. It is kept as minimal as possible – just enough to implement BDI's SYScalls and nothing more – yet it must be meticulously engineered using the best knowledge from Intel manuals, AMD manuals, ARM architecture docs, RISC-V specs, and FPGA datasheets. By citing those sources throughout this chapter, we underlined that every HAL action has a counterpart in a hardware manual: the HAL engineer must be both a "systems plumber" (connecting BDI to hardware pipes) and a "language translator" (ensuring BDI's intent is expressed in the chip's own idioms).

In closing, **BDI's hardware realization through HAL and custom ISA shows that we can achieve portability without sacrificing performance**, by intelligently exploiting low-level features of each architecture. It provides a level of indirection – a semantic indirection – that lets hardware and software evolve somewhat independently. As hardware becomes more diverse and powerful, BDI is poised to ride that wave, acting as a stable platform for the 21st-century computation paradigms. The lessons learned here foreshadow a future where operating systems, or even language runtimes, could define their own "virtual ISA" and have chips optimized for them. BDI may well be a glimpse of that future, where the boundary between software logic and hardware capability is blurred, and a new contract – based on dynamic binary dynamics – governs the cooperation between the two.

---

[1] 101.1. Determine and configure hardware settings | LPIC1 Exam Guide
https://borosan.gitbook.io/lpic1-exam-guide/1011-determine-and-configure-hardware-settings

[2] tech-composable-custom-extensions@lists.riscv.org | Messages
https://lists.riscv.org/g/tech-composable-custom-extensions/messages?expanded=1&msgnum=201

[3] Trap-and-emulate for hardware forward-compatibility
https://lists.riscv.org/g/tech-profiles/topic/101153812

[4] [7] [11] [12] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1
https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

[5] people.eecs.berkeley.edu
https://people.eecs.berkeley.edu/~krste/papers/riscv-priv-spec-1.7.pdf

[6] [24] [32] [33] "BDI: A New VM Runtime for Heterogeneous Units" | Tariq Mohammed posted on the topic | LinkedIn
https://www.linkedin.com/posts/tariq-mohammed-b70a1b179_the-binary-decomposition-interface-vm-runtime-activity-7326132425460199425-Pzi0

[8] [9] [10] Memory-mapped I/O and port-mapped I/O - Wikipedia
https://en.wikipedia.org/wiki/Memory-mapped_I/O_and_port-mapped_I/O

13 AArch64 memory and paging | Welcome to the Mike’s homepage!
https://krinkinmu.github.io/2024/01/14/aarch64-virtual-memory.html

14 Updating/Changing MMU Page Tables - Raspberry Pi Forums
https://forums.raspberrypi.com/viewtopic.php?t=268543

15 16 Using the ARM* Generic Interrupt Controller
https://fog.misty.com/perry/cpe2/ARM_GIC.pdf

17 21 22 23 26 27 The RISC-V Instruction Set Manual, Volume II: Privileged Architecture | Five EmbedDev
https://five-embeddev.com/riscv-priv-isa-manual/Priv-v1.12/supervisor.html

18 19 20 The RISC-V Instruction Set Manual, Volume II: Privileged Architecture | Five EmbedDev
https://five-embeddev.com/riscv-priv-isa-manual/Priv-v1.12/plic.html

25 4.10. Partial Reconfiguration
https://www.intel.com/content/www/us/en/docs/programmable/683389/22-4/partial-reconfiguration.html

28 Tesla's Dojo Supercomputer Deep Dive - by Dr. Ian Cutress
https://morethanmoore.substack.com/p/teslas-dojo-supercomputer-deep-dive?utm_source=substack&utm_medium=email&utm_content=share&action=share

29 30 31 Hot Chips 34 – Tesla’s Dojo Microarchitecture
https://chipsandcheese.com/p/hot-chips-34-teslas-dojo-microarchitecture