Document for Pokedex

1980-01-01

Contents

1 The Pokedex Project

1.1 Introduction

Pokedex is an implementation of the RISC-V instruction set architecture (ISA). It provides a simple, maintainable, and flexible ISA design that allows developers to easily add new instructions or make architectural changes—capabilities that are often difficult to achieve with existing tools like `riscv/sail-riscv` or Spike (`riscv/riscv-isa-sim`).

The Pokedex project comprises three main components:

- A formal specification written in the ARM Architecture Specification Language (**The ISA Model**).
- A **simulator** written in Rust, which provides a system platform for co-simulation with the ISA model.
- A **differential test framework** to cross-validate architectural states against Spike and RTL.

You can think of *The ISA Model* as the code that implements the logic defined by the RISC-V ISA Specification. *The Simulator* acts as the host environment; it handles the necessary system-level operations (such as loading ELF files) that are not covered by the ISA specification but are required to run programs. Finally, the *Differential Test Framework* provides the tools necessary to validate and guarantee the correctness of *The ISA Model.*

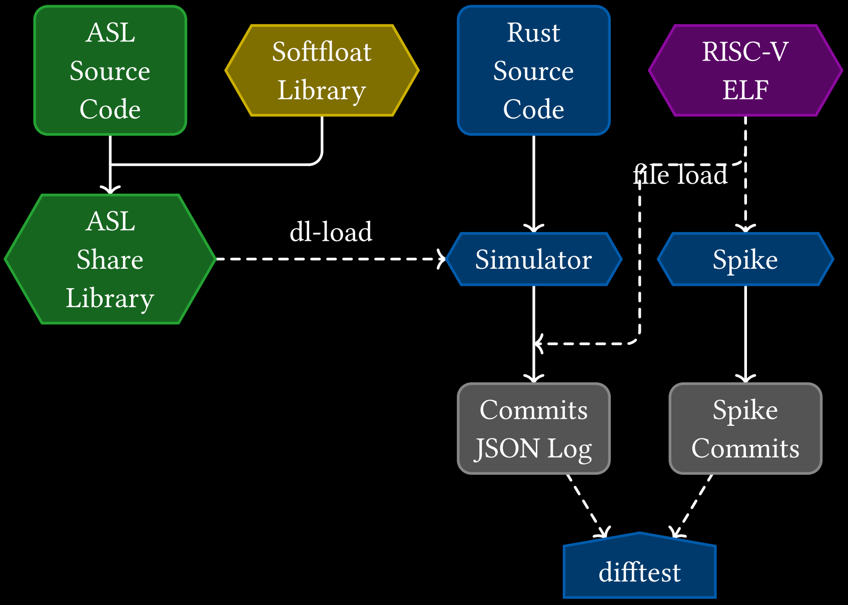

Figure 1: Pokedex Architecture

The primary goal of the Pokedex project is to serve as a “source of truth” for the ISA’s behavior. This enables co-simulation to identify bugs in Register Transfer Level (RTL) designs.

> **Why the name “Pokedex”?**
>
> Our T1 testbench supports a collection of micro-architecture designs, each named after a Pokémon. We chose the name “Pokedex” to align with this theme.

> In the Pokémon world, a Pokédex is an essential tool for understanding the creatures you interact with. In that same spirit, this project provides the tools necessary to understand, maintain, and improve our T1 architectures.

1.2 What's Coming

This document is intended for readers with a basic understanding of software programming. It covers the entire Pokedex architecture and explains key design choices. By the end of this guide, readers will be able to modify and verify the whole project.

1.3 Suggest Reading

- **ASL Reading Guide** (The most important one, this will give you a quick glance on the ASL language): https://intellabs.github.io/asl-interpreter/asl_reading.html
- **RISC-V ISA Specification**: https://github.com/riscv/riscv-isa-manual/
- **ASL Prelude Reference**: https://github.com/IntelLabs/asl-interpreter/blob/master/prelude.asl
- **herdtools7** (The "official" ASL): https://github.com/herd/herdtools7/tree/ASLRefALP3.1/asllib
- **ASL Specification** (The official ASL Specification, optional): https://developer.arm.com/documentation/ddi0626/latest

1.4 Compatibility

Note that we currently use the Intel Labs fork of the ASL Interpreter, which deviates slightly from the official ASL specification.

> A migration to *herdtools7* is planned, pending support for C code generation in that toolchain.

1.5 System Requirement

We use Nix as our primary build system. It orchestrates the build systems for each module and produces final artifacts without requiring manual dependency management.

The only requirement is a system with Nix installed. You can follow the installation guide here: https://nixos.org/download. (Note: You do not need the full NixOS operating system; only the Nix package manager CLI and daemon are required).

After installation, add the following configuration to `~/.config/nix/nix.conf`:

```
extra-experimental-features = flakes nix-command pipe-operators
```

Writing Nix code is outside the scope of this guide. However, the Pokedex project configuration is stable, so you generally will not need to modify Nix files. If you encounter build issues, please file an issue report.

2 ASL Model

We use ARM's Architecture Specification Language (ASL) to formally describe the RISC-V ISA. The core logic is organized within the `model/` directory:

- `aslbuild/`: Contains configuration files for `asl2c` to generate C code.
- `csr/`: Contains code snippets for individual Control and Status Register (CSR) implementations.
- `csrc/`: Contains C source code for ABI compatibility and module integration.
- `data_files/`: Pre-generated files containing deserializable data for the decoder.
- `extensions/`: Holds code snippets defining the semantics for each instruction, organized by ISA extension.
- `handwritten/`: Includes foundational, manually written code, such as architectural state declarations and helper libraries.
- `scripts/`: Includes scripts to generate Ninja build files, data files, etc.

2.1 Quick Glance at the Build Process

ASL provides limited support for polymorphic types. Consequently, implementing the full RISC-V ISA manually is difficult due to the significant amount of redundant logic required for instruction execution. To address this, we use Jinja to define and reuse code snippets.

The script `scripts/buildgen.py` acts as the primary build entry point. It scans all source files and generates a Ninja build file to perform the following tasks:

1. Process all `.j2` templates using JSON data from `data_files` to generate the final ASL code.
2. Combine the generated ASL code with the handwritten ASL code, then use the `asl2c` tool to compile them into C source code.
3. Compile the resulting C code alongside the interface code in `csrc/`, linking them with the `softfloat` library to produce the final dynamic library (`libpokedex_model.so`).

Now, we will step through the build process to explain the ASL model structure.

2.2 How we "decode"

Fundamentally, an instruction can be viewed as a sequence of bytes representing a "command" (opcode) paired with "arguments" (operands). The **decoder's** responsibility is to interpret this sequence: it identifies which "command" to execute and extracts the necessary "arguments", ensuring strict adherence to the ISA specification.

2.2.1 Instructions Decoding

Instruction encodings are defined in the `data_files/inst_encoding.json` file, which is generated by `scripts/datagen.py`. Each instruction entry requires the following fields to define an encoding:

```
{
  "name": "addi",
  "encoding": "-----------------000-----0010011",
  "extension": "rv_i"
}
```

The encoding data is derived from the riscv/riscv-opcodes project. We vendor the upstream parsing tool at `scripts/riscv_opcodes_util.py`, allowing us to accept any *riscv-opcodes* source. This enables us to easily define encodings for unratified extensions.

The encoding field is utilized by the `templates/inst_dispatch.asl.j2` template, which defines the instruction decoding entry point. Each encoding string becomes a bit-pattern match arm. When an instruction matches a specific bit pattern, the `DecodeAndExecute` function delegates execution to the corresponding function identified by the name field. If no bit pattern matches, an `IllegalInstruction` result is returned.

```
// example of code generated inst_dispatch.asl
func DecodeAndExecute(instruction : bits(32)) => Result
begin
  case instruction of
    when '-----------------000-----0010011' =>
      return Execute_ADDI(instruction);

    // ... other bit patterns and dispatcher ...

    otherwise =>
      return IllegalInstruction();
  end
end
```

In the example above, the decoder is defined as a function. A typical function block follows this structure:

```
function FunctionName(argument : type)
begin
  // function body
end
```

The input type `bits(32)` denote that `instruction` is a `bitvector` of length `32`. Pattern matching is achieved using the `case...when` expression. Each `when` keyword defines a match arm; if the pattern matches the value, the associated statements are executed. The `otherwise` keyword acts as a “catch-all” default arm; if no pattern matches the value, the statements defined in `otherwise` will be executed.

```
// ...
  case <expression> of
    when <pattern> =>
      <statement>
    otherwise =>
      <statement>
  end
// ...
```

As the number of supported ISA extensions grows, hand-writing these pattern matches becomes unmanageable. That’s why we use Jinja to parse the list of encodings and automatically generate this instruction dispatch logic.

Document of all instruction handler can be found in Section 5.

2.2.2 Control and Status Register Decoding

CSR handling follows the same logic as instruction decoding: a CSR definition file feeds an ASL Jinja template to generate the pattern matching and dispatch code.

The standard RISC-V ISA sets aside a 12-bit encoding space (`csr[11:0]`) for up to 4,096 CSRs. By convention, the upper 4 bits of the CSR address (`csr[11:8]`) are used to encode the read and write accessibility of the CSRs according to privilege level. The top two bits (`csr[11:10]`) indicate whether the register is read/write (`00`, `01`, or `10`) or read-only (`11`). The next two bits (`csr[9:8]`) encode the lowest privilege level that can access the CSR.

The CSR encoding data is derived from the `riscv-opcodes` project and stored in `data_files/csr.json`. Each CSR is defined using the following JSON format:

```json
{
  "name": "fflags",
  "mode": "urw",
  "addr": 1,
  "bin_addr": "000000000001",
  "read_write": true
}
```

This JSON data is processed by the `templates/csr_dispatch.asl.j2` template to generate CSR operations as follows:

```
func ReadCSR(csr : bits(12)) => CsrReadResult
begin
  case csr of
    when '000000001000' =>
      return Read_VSTART();

    // other csr

    otherwise =>
      return CsrReadIllegalInstruction();
  end
end

func WriteCSR(csr : bits(12)) => Result
begin
  case csr of
    when '000000001000' =>
      return Write_VSTART();

    // other csr

    otherwise =>
      return IllegalInstruction();
  end
end
```

Note that the `Write_XXX` function call is only generated if the CSR is defined as writable (based on the `read_write` fields).

Details of CSR can be found in Section 2.6.1.

2.3 Stepping Instructions

We have seen how instructions are decoded and dispatched to their handler functions. Next, we examine how instructions are fetched and passed to the decoder.

The `Step` function, defined in `handwritten/step.asl`, serves as the primary entry point for driving the model. It processes instructions sequentially, one at a time. Below is a simplified pseudo-code representation of this function:

```
// pseudo example code
func Step() => StepResult
begin
  if HasInterrupt() then
    return INTERRUPT;
  end

  let instruction = FFI_instruction_fetch(PC);
  let result = DecodeAndExecute(instruction);
  if result.is_ok then
    return COMMITTED;
  else
    return EXCEPTION;
  end
end
```

We will discuss how the Foreign Function Interface (FFI) interoperates with the model in Section 2.7. For now, treat any function with the `FFI_` prefix as a black box that performs the necessary external operations.

In each execution of Step, the model fetches a single instruction using the external FFI function. The `PC` (Program Counter) is a global architectural state variable (32-bit) representing the physical memory address of the current instruction. This `PC` value is passed to `FFI_instruction_fetch` to retrieve the instruction data at that specific address.

The execution flow is as follows:

1. Instruction is fetched through the `FFI_instruction_fetch()` function.
2. The instruction data is passed to `DecodeAndExecute` (implemented in Section 2.2.1) to obtain the execution result.
3. If the result indicates success (no exceptions), a signal is use as response to indicate instruction is committed.
4. If an exception is raised, an exception signal is returned.

Note that the actual `Step` implementation handles significantly more complexity than shown above, including Compressed Instructions (the RISC-V “C” extension) and detailed exception logic. Please refer to `handwritten/step.asl` for the complete source code.

2.4 Error Handling and Result Type

Due to ASL’s limited support for polymorphic types and our preference to avoid standard `try...catch` mechanisms, we define a custom `Result` type to encapsulate operation status and error details:

```
record Result {
  is_ok : boolean;
  cause : integer{0..31};
  payload : bits(XLEN);
};

type CsrReadResult of record {
  data: bits(XLEN),
  result: Result
};
```

The `Result` structure indicates whether an operation triggered an exception.

- When `is_ok` is `TRUE`, the operation was successful.
- When `is_ok` is `FALSE`, the operation failed, and the structure contains necessary exception details (such as the `cause`).

If an exception does not require a `payload`, the field is typically set to zero. However, developers should treat the `payload` as undefined in these cases and ignore it.

We provide helper APIs to construct `Result` instances (undefined fields should be ignored), see `model/handwritten/exception.asl` for details.

2.5 Architecture States

We have established the main logic for driving the model to process instructions sequentially. However, instructions rarely execute in isolation; they often exhibit **data dependencies**, where the current instruction requires the result of a previous one.

To manage this, the model requires persistent internal storage to hold these results between steps. This collection of information is known as the **Architectural State**.

These states are defined in file `handwritten/states.asl` and `handwritten/states_v.asl`.

2.6 CSR (Control and Status Register)

2.6.1 CSR vs Architecture States

The relationship between an **Architectural State** and a **Control and Status Register (CSR)** is that of implementation versus interface.

Developer can think of **Architectural States** as the actual, physical variables in our model. These are custom-sized registers that hold the state of the model.

In contrast, **CSRs** are the standardized, abstract interface used to access those underlying architectural states. Because of this separation, we only implement the architectural state bits that are necessary for our model's features, rather than defining storage for every bit of every CSR in the specification.

The RISC-V specification defines several types of CSRs, such as **WPRI** (Write-Preserve, Read-Ignore), **WLRL** (Write-Legal, Read-Legal), and **WARL** (Write-Any, Read-Legal). These types dictate how writes are handled—for instance, some fields in a write may be consider invalid and be ignored.

To ensure our model correctly adheres to these rules, we do not expose the raw architectural states directly. Instead, we provide constrained getter and setter APIs for each state. This design imposes the specification's rules at the access layer, guaranteeing that any register write performed by your instruction logic is automatically handled correctly according to the CSR's type, and consider any invalid write as "model implementation bugs".

As a **Write-Any, Read-Legal (WARL)** register, the `mtvec` CSR provides a clear example of separating the public-facing CSR from its underlying architectural states.

The `mtvec` CSR is composed of two architectural states:

- `MTVEC_BASE`: A 30-bit field for the trap address.
- `MTVEC_MODE`: A 2-bit field for the trap mode.

The key distinction lies in how writes are handled:

1. **CSR Write Handler (Permissive):** As a WARL register, any 32-bit value can be written to the `mtvec` CSR. The CSR's write logic is responsible for sanitizing this input. For example, the `MODE` field only supports values `0b00` and `0b01`. If a write contains `0b10` or `0b11` in the mode bits, the handler correctly ignores the update for that field, treating it as a no-op.
2. **Architectural State (Strict):** The internal `MODE` state itself should **never** contain an illegal value like `0b10` or `0b11`. The sanitization logic in the CSR handler guarantees this. Therefore, our model's internal logic will use an **assertion** to validate the `MODE` state. If this assertion ever fails, it signals a critical bug in the CSR's write-handling or some explicit write logic that must be fixed.

Document for CSR R/W handler can be founded in Section 5.

2.7 FFI and the Platform

The model, as presented so far, is a collection of standalone functions and variables, lacking any command-line interface (CLI) handling, ELF file parsing, or even a main function. This is by design. The model's sole responsibility is to implement the ISA specification. It acts as a core component that is driven externally, exposing its inputs and requesting data from other modules, analogous to hardware wiring. These connections are implemented using a C Foreign Function Interface (FFI).

2.7.1 Export API from Model

All ASL code is compiled into C using the `asl2c` tool. Since `asl2c` does not namespace the compiled symbols, an additional C layer, `csrc/pokedex_interface.c`, is used to manage the Application Binary Interface (ABI). By default, symbols are hidden using the GNU GCC compile flag `-fvisibility=hidden` (see `dylib_cflags` in `scripts/buildgen.py`). The sole exposed symbol, `EXPORT_pokedex_get_model_export()`, returns a reference to the `model_export` variable shown as follow.

```c
static const struct pokedex_model_export model_export = {
    .abi_version = POKEDEX_ABI_VERSION,
    .create = model_create,
    // Other Constructor

    .step = model_step_trace,
    // Other Executor

    .get_pc = model_read_pc,
    // Other Accessor
}

__attribute__((visibility("default")))
const struct pokedex_model_export* EXPORT_pokedex_get_model_export() {
  return &model_export;
}
```

The `model_export` variable is a `pokedex_model_export` struct that serves as the central access point for all exported functions. Developers can obtain its address to invoke the corresponding APIs for driving the model or accessing architectural states stored within the ASL model.

Document of exported API can be found at `model/handwritten/external.asl`.

2.7.2 Required API for Model

To function, the ASL model relies on external functions for operations such as requesting data from the system memory bus or sending status updates to a debug channel. These external requirements are grouped into three categories:

- Memory APIs: For fetching data from an external memory implementation.
- Debugging APIs: For printing debugging messages.
- Architectural State Post-Write Hooks: For recording changes to the architectural state.

All required APIs are declared in `handwritten/external.asl`, containing only their function signatures. These APIs are then compiled into C header declarations, with the suffix `_0` appended to each function name. Developers can implement these API functions by including the generated C header.

2.8 Floating Point Arithmetic

Intel Lab's compiler runtime does not include built-in support for floating-point arithmetic. Consequently, all floating-point arithmetic within the ASL model is delegated to the Berkeley Softfloat project (see https://github.com/ucb-bar/berkeley-softfloat-3).

The Softfloat project is compiled into a standalone C library, and our `csrc/softfloat_wrapper.c` file provides an FFI glue layer. This wrapper implements the floating-point functions declared in the ASL model. It works by converting data types, calling the corresponding Softfloat API, and returning the result.

The result of floating-point arithmetic is wrapped in a `F32_Flags` struct. This struct includes a FLEN `value` field, which stores the operation's result, and an `fflags` field, which stores any associated exception flags.

```
type F32_Flags of record {
  value: bits(32),
  fflags: bits(5)
};
```

Each floating-point instruction calls its corresponding arithmetic API. The returned value is then stored in the floating-point register, and the `fflags` are used to update the FCSR state.

3 Simulator

Thanks to the stable C ABI, the platform language can be chosen based on developer preference. We selected Rust for its robust type system and extensive community support. The platform-specific code resides in the `simulator` directory.

3.1 Interact with Model

The aspect of the project that likely holds the most interest for readers is the interaction between the simulator and the model. Before diving into the complexities of the FFI, the following diagram illustrates the basic flow of this interaction.

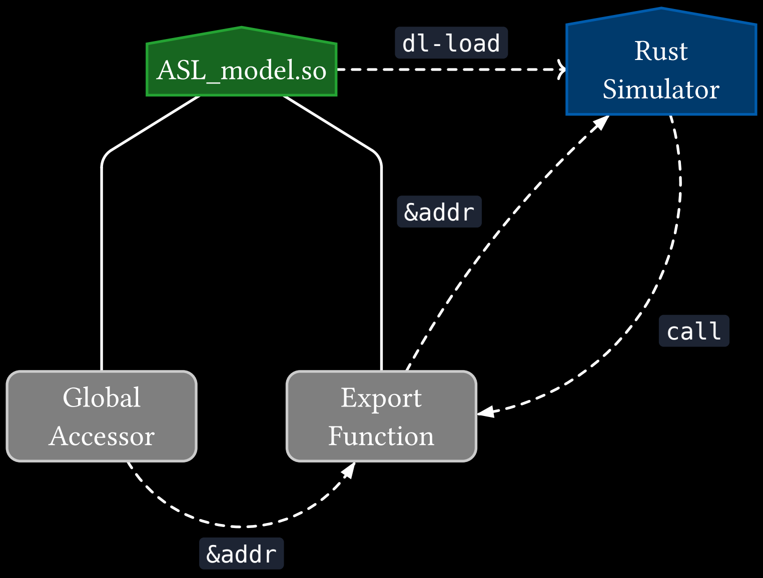

Figure 2: Simulator Interaction w/ Model

As shown in Section 2.7, the simulator obtains the API wrapper by calling `EXPORT_pokedex_get_model_export()`. The simulator uses `libc::dlopen` to load the shared object file, locates the exported C symbol, and transmutes the function pointer into a Rust function. Finally, it invokes this function to initialize the API wrapper, facilitating data transfer across the language boundary.

Memory access from the ASL model follows a reverse pattern to Section 2.7: the simulator exposes its memory APIs via a C vtable wrapper, `struct pokedex_mem_callback_vtable`. This structure contains function pointers that map to the underlying Rust implementations. A global instance of this vtable is declared in `pokedex_interface.c` and initialized by the simulator, making it accessible to the ASL model.

For example, when the ASL model requires a 32-bit memory read, the process proceeds as follows:

1. ASL Call: The model calls the ASL function `FFI_read_physical_memory_32bits`.
2. C Binding: This compiles to the C header declaration: `FFI_read_physical_memory_32bits_0`.
3. Vtable Delegation: Implementation in `pokedex_interface.c` delegate this request to vtable.

```
static const struct pokedex_mem_callback_vtable* mem_cb_vtable = NULL;

FFI_ReadResult_N_32 FFI_read_physical_memory_32bits_0(uint32_t addr) {
    uint32_t data = 0;
    int ret = mem_cb_vtable->read_mem_4(mem_cb_data, addr, &data);
    FFI_ReadResult_N_32 value = {
        .success = !ret,
        .data = data,
    };
    return value;
}
```

The `mem_cb_vtable` has a static address after linking, which is used by the simulator to initializes a new instance of `pokedex_mem_callback_vtable` type at that address:

```
struct pokedex_mem_callback_vtable {
    int (*read_mem_4)(void* cb_data, uint32_t addr, uint32_t* ret);
    // ... Other memory APIs ...
};
```

On the Rust side, the simulator implements the `read_mem_4` function. It casts the opaque `bus` pointer back to a mutable Bus reference and performs the read operation.

```
unsafe extern "C" fn read_mem_4(bus: *mut c_void, addr: u32, ret: *mut u32) ->
c_int {
    let bus = unsafe { &mut *(bus as *mut T) };
    bus.read_mem_u32(addr).as_int_ret(ret)
}
```

Finally, the Bus implementation handles all the byte-level access:

```
impl Bus {
    fn read_mem_u32(&mut self, addr: u32) -> BusResult<u32> {
        assert!(addr % 4 == 0);

        let mut data = [0; 4];
        self.bus
            .read(addr, &mut data)
            .map(|_| u32::from_le_bytes(data))
    }
}
```

For further details on the internal Bus implementation, please refer to the following section.

3.2 Execution

Upon launch, the simulator initializes the system memory bus. It reads the ELF file from the path specified in the CLI arguments and loads it into memory.

After loading, the simulator identifies the ELF entry address and sets the model's `PC` state to this address using the `model_export.reset()` C API.

Finally, the simulator enters its main execution loop. It repeatedly calls the `model.step()` C API until the model throws an error or writes an exit code to a specific bus device.

3.3 Device Bus

Memory components (such as SRAM) and other MMIO devices are abstracted as address spaces paired with corresponding handlers. When a request falls within a specific address space, the bus routes it to the corresponding device.

```
pub struct Bus {
    address_space: Vec<(Range<u32>, Box<dyn Addressable>)>,
}
```

The `Addressable` trait defines the interface for bus-compatible devices. Implementing types must handle both read and write operations.

```
pub trait Addressable: Send {
    fn do_bus_read(&mut self, offset: u32, dest: &mut [u8]) -> Result<(),
BusError>;
    fn do_bus_write(&mut self, offset: u32, data: &[u8]) -> Result<(),
BusError>;
}
```

- Reads: The device populates the mutable `dest` buffer.
- Writes: The device consumes the input `data` slice.

As shown in the trait definition, devices receive a relative `offset` rather than the absolute physical address. The Bus calculates this offset relative to the device's base address before delegating the request. The original address is transparent to the device.

For example, if a device is mapped to `0x8000_0000` and a request targets `0x8000_0044`, the device receives an offset of `0x44`.

3.3.1 The Main Memory

The simulator implements a basic SRAM model called `NaiveMemory`, which serves as the main memory component. Internally, it wraps a raw byte vector.

```
pub struct NaiveMemory {
    memory: Vec<u8>,
}
```

`NaiveMemory` is initialized as a zero-filled buffer. Its implementation of the `Addressable` trait handles data transfer and safety checks:

```rust
impl Addressable for NaiveMemory {
    fn do_bus_read(&mut self, offset: u32, dest: &mut [u8]) -> Result<(),
BusError> {
        let length = self.memory.len() as u32;
        if offset >= length || offset + dest.len() as u32 > length {
            return Err(BusError::DeviceError {
                id: "NaiveMemoryRead",
            });
        }

        dest.copy_from_slice(&self.memory[offset as usize..offset as usize +
dest.len()]);

        Ok(())
    }

    fn do_bus_write(&mut self, offset: u32, data: &[u8]) -> Result<(),
BusError> {
        let length = self.memory.len() as u32;
        if offset >= length || offset + data.len() as u32 > length {
            return Err(BusError::DeviceError {
                id: "NaiveMemoryWrite",
            });
        }

        self.memory[offset as usize..offset as usize +
data.len()].copy_from_slice(data);

        Ok(())
    }
}
```

- Bounds Checking: Both read and write operations first verify that the requested address range is valid. If the range exceeds the memory size, a `BusError` is returned.
- Read: Copies data from the internal `memory` vector at the specified `offset` into the `dest` buffer.
- Write: Copies data from the input `data` slice into the internal `memory` vector at the specified `offset`.

3.3.2 MMIO Registers

MMIO registers consist of pairs mapping a memory offset to a specific handler. In this RV32I architecture, every MMIO register stores a 32-bit value, each occupies one byte address space.

The `MMIOAddrDecoder` handles request from the bus using the following logic:

```rust
pub struct MMIOAddrDecoder {
    // offset, type
    regs: Vec<(u32, MmioRegs)>,
}

impl Addressable for MMIOAddrDecoder {
    fn do_bus_write(&mut self, offset: u32, data: &[u8]) -> Result<(),
BusError> {
        // a non 32-bit write is consider as implementation bug and should be
immediately bail out
        let new_val = u32::from_le_bytes(data.try_into().unwrap());
        let index = self.regs.binary_search_by(|(reg, _)| reg.cmp(&offset));

        if let Ok(i) = index {
            self.regs[i].1.store(new_val);

            Ok(())
        } else {
            debug!("unhandle MMIO write to offset={offset} with value
{new_val}");
            Err(BusError::DeviceError { id: "MMIOWrite" })
        }
    }

    fn do_bus_read(&mut self, offset: u32, dest: &mut [u8]) -> Result<(),
BusError> {
        let index = self.regs.binary_search_by(|(reg, _)| reg.cmp(&offset));

        if let Ok(i) = index {
            // a non 32-bit read is consider as implementation bug and should
be immediately bail out
            dest.copy_from_slice(&self.regs[i].1.load().to_le_bytes());

            Ok(())
        } else {
            debug!("unhandle MMIO read to offset={offset}");
            Err(BusError::DeviceError { id: "MMIORead" })
        }
    }
}
```

- Lookup: The decoder performs a search to check if an MMIO register exists at the requested offset.
- Read: If found, the handler loads the 32-bit value from the register, converts it to Little Endian bytes, and copies it to the `dest` buffer.
- Write: If found, the handler converts the input `data` (Little Endian bytes) into a 32-bit integer and stores it in the register.

The `MmioRegs` `enum` defines the supported types of MMIO registers. The following implementation demonstrates how these registers interact with the system platform.

```rust
pub enum MmioRegs {
    Exit(Syscon),
}

impl MmioRegs {
    fn load(&self) -> u32 {
        match self {
            Self::Exit(_) => {
                panic!("unexpected read from exit MMIO device");
            }
        }
    }

    fn store(&mut self, v: u32) {
        match self {
            Self::Exit(eic) => {
                eic.0.store(v, Ordering::Release);
            }
        }
    }
}
```

In this example, the `Exit` variant act as a write-only register. Writing to it updates its internal `Syscon` state.

`Syscon` wraps an atomic value that is shared between the Bus and the system controller. This allows the system to detect state changes (such as a shutdown request) triggered by the emulated device.

```rust
pub struct Syscon(Arc<AtomicU64>);

fn try_build_from(config: &[(String, u32)]) -> anyhow::Result<(Self,
ExitStateRef)> {
    let mut regs = Vec::new();
    let mut exit_state = None;
    for (name, offset) in config {
        match name.as_str() {
            "exit" => {
                let s = Arc::new(AtomicU64::new(0));
                let exit_rc = Syscon::new(s.clone());
                exit_state = Some(s);
                regs.push((*offset, MmioRegs::Exit(exit_rc)))
            }
            name => bail!("unsupported MMIO device {name}"),
        }
    }

    Ok((Self { regs }, exit_state.unwrap()))
}
```

When the MMIO mapping is initialized via `try_build_from`:

- A new `Syscon` (wrapping an `AtomicU64`) is created.
- One clone of the atomic reference is stored inside the `MmioRegs::Exit` variant.
- Another clone is returned to the system controller (`ExitStateRef`).

This architecture allows the system controller to inspect the status of the device externally via the shared reference.

3.4 Commit Events

To facilitate debugging and verification, the simulator records architectural state changes in a JSON log. This is achieved by interacting with a global trace buffer defined in `pokedex_interface.c`.

The logging lifecycle operates at the instruction boundary:

- Pre-execution: The trace buffer is cleared to ensure it only captures side effects from the current instruction.
- Execution: As the instruction executes, state modifications are recorded in the C trace buffer.
- Post-execution: The simulator retrieves the data via FFI, parses the changes, and serializes them into the JSON log.

The simulator produces four types of log entries:

- Reset: Indicates the ASL model has been reset (includes the reset PC).
- Exit: Indicates an exit signal was received (includes the software exit code).
- Commit: Indicates an instruction completed execution (includes modified states).
- Exception: Indicates a hardware exception occurred (includes modified states).

A Commit log entry contains the following metadata:

```rust
#[derive(Debug, Serialize, Deserialize)]
pub struct CommitLog {
    pub pc: u32,
    pub is_compressed: bool,
    pub instruction: u32,
    pub states_changed: Vec<StateWrite>,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
#[serde(tag = "dest", rename_all = "lowercase")]
pub enum StateWrite {
    Xrf { rd: u8, value: u32 },
    Frf { rd: u8, value: u32 },
    Vrf { rd: u8, value: Vec<u8> },
    Csr { name: String, value: u32 },
}
```

- `pc`: The program counter of the committed instruction.
- `is_compressed`: Boolean flag indicating if the instruction is compressed (16-bit).
- `instruction`: The raw 32-bit representation of the instruction.
- `states_changed`: A list of architectural state changes caused by this instruction.
  - `Xrf`: Write to a *General Purpose Register*, (includes register index `rd` and new `value`).
  - `Frf`: Write to a *Floating Point Register*, (includes register index `rd` and new `value`).
  - `Vrf`: Write to a *Vector Register*, (includes register index `rd` and the raw byte vector `value`).
  - `Csr`: Write to a *Control and Status Register*, (includes the CSR `name` and the new `value`).

3.5 Configurations

The simulator features a flexible bus configuration system powered by the KDL Document Language. KDL was selected for its readability and XML-like hierarchical structure. A typical configuration file is shown below:

```
sram {
  base 0x80000000
  length 0x20000000
}

mmio {
  base 0x40000000
  length 0x1000

  mmap "exit" offset=0x4
}
```

The parser interprets each top-level block as a distinct address space, where the block name (e.g., `sram`, `mmio`) determines the component type.

Required Nodes:

- `base`: Specifies the starting address of the component.
- `length`: Specifies the size of the address space.

Additional Nodes:

- `mmap` (MMIO only): Configures individual registers within the MMIO block. In the example above, an “exit” register is mapped to offset 0x4.

4 Differential Test

Implementing a complex ISA based solely on textual specifications is prone to human error and ambiguity. To ensure correctness, we validate our implementation against the Spike simulator, which serves as the industry-standard RISC-V reference model.

We have designed a differential testing framework to automate this verification. This framework executes the same instruction stream on both simulators and compares their architectural states at the per-instruction level to detect discrepancies immediately.

4.1 Interfaces

To avoid coupling the verification process directly to Spike, we designed a flexible Differential Testing Framework. Instead of comparing raw internal states, we abstract the comparison logic through the `DiffBackend` trait.

```rust
pub trait DiffBackend {
    fn description(&self) -> String;

    fn diff_reset(&mut self, pc: u32) -> anyhow::Result<()>;
    fn diff_step(&mut self) -> anyhow::Result<Status>;

    fn state(&self) -> &CpuState;
}
```

Any simulator integrating with this framework must implement `DiffBackend`. This allows the framework to verify correctness using a unified interface, regardless of the underlying simulator implementation.

4.1.1 Shadow State Construction

The framework reconstructs the architectural state by replaying execution logs provided by the simulators. Rather than comparing the logs directly, the framework updates a canonical CpuState (a “shadow state”) and performs comparisons on this structure.

```rust
pub struct CpuState {
    pub gpr: [u32; 32],
    pub fpr: [u32; 32],
    pub vregs: Vec<u8>,

    pub pc: u32,

    pub csr: CsrState,

    pub(crate) is_reset: bool,
    pub(crate) reset_vector: u32,
    pub(crate) is_poweroff: bool,
}
```

During execution, if a simulator commit log indicates a state change, a `DiffRecord` is produced to update the *shadow state*. The comparison tool then verifies that the `CpuState` of the simulator matches the Golden Model.

4.1.2 CSR Comparison

CSRs require special handling because many values are implementation-defined or updated implicitly (side effects). By default, the framework compares the full CSR state to ensure accuracy. However, to optimize performance, the comparison scope is narrowed in specific scenarios. For example, during Floating-Point operations, we may strictly compare relevant registers (such as `fcsr` and `fflags`) rather than the entire CSR set.

4.2 Spike Hacks

To ensure compatibility with the differential testing framework, Spike must be invoked with specific command-line arguments. These settings align Spike's behavior and memory layout with the Pokedex simulator.

```
spike --isa=rv32imafc_zvl256b_zve32x_zifencei \
  --priv=m --log-commits -p1 --hartids=0 --triggers=0 \
  -m0x80000000:0x20000000,0x40000000:0x1000 \
  --log=commits.log \
  "$casePath"
```

- `--isa`: Restricts the Instruction Set Architecture to the specific extensions supported by Pokedex.
- `--priv=m`: Restricts execution to Machine (M) mode.
- `--log-commits`, `--log`: Enables execution logging and specifies the output file
- `-p1`, `--hartid=0`: Configures Spike for single-core execution assigned to Hart ID 0.
- `--triggers=0`: Disables hardware debug triggers.
- `-m`: Replicates the Pokedex memory layout (SRAM and MMIO regions) within Spike.

5 Model Reference

5.1 Instruction Reference

5.1.1 ADDI

- **Bit Pattern**: `-----------------000-----0010011`
- **Extension Set**: `rv_i`

ADDI extracts RD(inst[11:7]), RS1(inst[19:15]), and IMM(inst[31:20]) from the 32-bit instruction. It performs a bitwise ADD operation on the value stored in register X[RS1] and the sign-extended immediate value, then stores the result in register X[RD].

Formulated as:

```
X[RD] = X[RS1] + SExt(IMM)
```

5.1.2 ANDI

- **Bit Pattern**: `-----------------111-----0010011`
- **Extension Set**: `rv_i`

ANDI extracts RD(inst[11:7]), RS1(inst[19:15]), and IMM(inst[31:20]) from the 32-bit instruction. It performs a bitwise AND operation on the value stored in register X[RS1] and the sign-extended immediate value, then stores the result in register X[RD].

Formulated as:

```
X[RD] = X[RS1] `AND` SExt(IMM)
```

5.1.3 ORI

- **Bit Pattern**: `-----------------110-----0010011`
- **Extension Set**: `rv_i`

ORI extracts RD(inst[11:7]), RS1(inst[19:15]), and IMM(inst[31:20]) from the 32-bit instruction. It performs a bitwise OR operation on the value stored in register X[RS1] and the sign-extended immediate value, then stores the result in register X[RD].

Formulated as:

```
X[RD] = X[RS1] `OR` SExt(IMM)
```

5.1.4 XORI

- **Bit Pattern**: `-----------------100-----0010011`
- **Extension Set**: `rv_i`

XORI extracts RD(inst[11:7]), RS1(inst[19:15]), and IMM(inst[31:20]) from the 32-bit instruction. It performs a bitwise XOR operation on the value stored in register X[RS1] and the sign-extended immediate value, then stores the result in register X[RD].

Formulated as:

```
X[RD] = X[RS1] `OR` SExt(IMM)
```

5.1.5 SLTI

- **Bit Pattern**: `-----------------010-----0010011`
- **Extension Set**: `rv_i`

SLTI extracts RD(inst[11:7]), RS1(inst[19:15]), and IMM(inst[31:20]) from the 32-bit instruction. It place the value `1` in register X[RD] if value in register RS1(X[RS1]) is less than the sign-extended immediate value when they both treated as **signed number**.

Formulated as:

```
let v1 = SInt(X[RS1]);
let v2 = SInt(SExt(IMM));
if v1 < v2 then
  X[RD] = 1;
else
  X[RD] = 0;
end
```

5.1.6 SLTIU

- **Bit Pattern**: `-----------------011-----0010011`
- **Extension Set**: `rv_i`

SLTIU extracts RD(inst[11:7]), RS1(inst[19:15]), and IMM(inst[31:20]) from the 32-bit instruction. It place the value `1` in register X[RD] if value in register RS1(X[RS1]) is less than the sign-extended immediate value when they both treated as **unsigned number**.

Formulated as:

```
let v1 = UInt(X[RS1]);
let v2 = UInt(SExt(IMM));
if v1 < v2 then
  X[RD] = 1;
else
  X[RD] = 0;
end
```

5.2 CSR Reference

5.2.1 MTVEC

- **Bit Pattern**: `001100000101`
- **Mode**: `mrw`
- **Number**: `773`

The mtvec (Machine Trap-Vector Base-Address Register) is a read/write register accessible exclusively in Machine Mode. Attempts to access this register from lower privilege levels result in an Illegal Instruction Exception.

mtvec adheres to WARL (Write Any Values, Read Legal Values) semantics. The simulator implements the following behavior:

- Supported Modes: Only Direct (00) and Vectored (01) modes are valid.
- Invalid Writes: If a write operation specifies an unsupported mode, the entire

write is ignored, and the register retains its previous value.

6 ASLc Compiler Passes

6.1 Filter unreachable code from exports

This command discard any code not reachable from the list of exported functions.

```
:filter_reachable_from --keep-builtins exports
```

Remove `--keep-builtins` flag to remove unreachable prelude functions. Be aware the removal of built-ins functions better to be used at the end of the pass pipeline to avoid missing functions after optimization.

6.2 Eliminate `typedef`

```
:xform_named_type
```

6.3 Eliminate bit and int arithmetic operation

Eliminate bit,int arithmetic operations like `'000' + 3`.

```
:xform_desugar
```

6.4 Eliminate bit vector concatenate

Eliminate bit-tuples like `[x,y] = z;` and `x[7:0, 15:8]`.

```
:xform_bittuples
```

6.5 Convert bit-slice operation

Convert all bit-slice operations to use the +: syntax, e.g., `x[7:0]` –> `x[0 +: 8]`.

```
:xform_lower
```

6.6 Eliminate slices of integers

Eliminate slices of integers by first converting the integer to a bitvector. E.g., if `x` is of `integer` type, then convert `x[1 +: 8]` to `cvt_int_bits(x, 9)[1 +: 8]`

```
:xform_int_bitslices
```

6.7 Convert getter/setter

Convert use of getter/setter syntax to function calls. E.g., `Mem[a, sz] = x` –> `Mem_set(a, sz, x)`

```
:xform_getset
```

6.8 TODO

```
:xform_valid track-valid
```

6.9 Constant propagation

Perform constant propagation without unrolling loops. This helps identify potential targets for the monomorphicalize pass.

```
:xform_constprop --nounroll
```

6.10 Monomorphicalize functions

Create specialized versions of every bitwidth-polymorphic function and change all function calls to use the appropriate specialized version. (Note that this performs an additional round of constant propagation.)

```
:xform_monomorphize --auto-case-split
```

6.11 Lift let-expressions

Lift let-expressions as high as possible out of an expression e.g., `F(G(let t = 1 in H(t))) -> let t = 1 in F(G(H(t)))`.

(This makes later transformations work better if, for example, they should match against `G(H(..))`)

Note that source code is not expected to contain let-expressions. They only exist to make some transformations easier to write.

```
:xform_hoist_lets
```

6.12 Convert bit vector slice operation to bit operation

Convert bitslice operations like “x[i] = ‘1’;” to a combination of AND/OR and shift operations like “x = x OR (1 << i);” This works better after constant propagation/monomorphization.

```
:xform_bitslices
```

Add flag `--notransform` when using `ac`, `sc` backend.

6.13 Convert match to if

Any case statement that does not correspond to what the C language supports is converted to an if statement. This works better after constant propagation/monomorphization because that can eliminate/simplify guards on the clauses of the case statement.

```
:xform_case
```

6.14 Wrap variable

```
:xform_wrap
```

6.15 Create bounded int

```
:xform_bounded
```

6.16 Filter function not listed in imports

The `imports` field should be defined in configuration in following form:

```
{
  "imports": [
    "TraceMemRead",
    "TraceMemWrite"
  ]
}
```

```
:filter_unlisted_functions imports
```

6.17 Check monomorphic

Check that all definitions are bitwidth-monomorphic and report a useful error message if they are not. The code generator will produce an error message if it finds a call to a polymorphic functions but we can produce a much more useful error message if we scan for all polymorphic functions and organize the list of functions into a call tree so that you can see which functions are at the roots of the tree (and therefore are the ones that you need to fix).

```
:check_monomorphization --fatal --verbose
```

7 How to build this document

7.1 Doc Comment

Most implementation details are extracted directly from ASL **doc comments**. We define a “doc comment” as any comment block where each line begins with two forward slashes and a bang (`//!`).

For example:

```
// This is a normal comment and is ignored by the documentation generator.

//! This is a doc comment and must follow the documenting rules.
```

To structure the generated documentation, we adopt the frontmatter convention commonly used by Markdown static site generators. Each doc comment block requires two components:

- Frontmatter: A metadata block written in YAML, located at the very beginning of the comment. It is enclosed by triple hyphens (`---`).
- Content: The main body of the documentation, written in Typst.

A valid ASL doc comment follows this structure:

````
//! ---
//! section: instruction
//! label: rv32-andi
//! title: RV32 ANDI
//! ---
//! ANDI extract RD, RS1 and IMM from the 32bits instruction, do logical
//! and operation to the 32-bits value store in X register RS1 and the
//! sign extended 32-bits immediate value, store the result in X register RD.
//!
//! Formulated as
//! ```text
//! X[RD] = X[RS1] `AND` SExt(IMM)
//! ```
//!
//! See @arg-luts for args extraction.
func Execute_ANDI(instruction: bits(32)) => Result
begin
{{- arith_imm_instruction_body("andi", "riscv_and") -}}
end
````

In the example above, the YAML block serves as the frontmatter. Developers can define custom context fields, but the following three fields are mandatory:

- `section`: Defines the document type, allowing the Typst compiler to anchor this comment correctly in the final document.
- `label`: A valid Typst label used for cross-referencing this section from other components.
- `title`: Declares the heading for the section (each doc comment is treated as an individual section).

The generated documentation for the example above will appear as follows:

~~~
### RV32 ANDI <rv32-andi>

(Defined in `extensions/rv_im_zifencei/addi.asl.j2:39`)

ANDI extracts RD, RS1, and IMM from the 32-bit instruction. It performs a
bitwise AND operation on the value stored in register X[RS1] and the
sign-extended immediate value, then stores the result in register X[RD].

Formulated as:
```text
X[RD] = X[RS1] `AND` SExt(IMM)
```

See @arg-luts for argument extraction details.
~~~

7.2 Extracting Doc Comment

Run script `model/scripts/docomment.py`.