HCL Descriptions of Y86 Processors (Rust impl.)

Weiyao Huang

Oct 25, 2024

Notice

This document describes the Rust-implemented version of the Hardware Control Language (**HCL-rs**), which is a modified version of *CS:APP2e Web Aside ARCH:HCL: HCL Descriptions of Y86 Processors*.

1 Motivation

The parser of the HCL in CS:APP2e in the original Archlab is implemented with `bison` and `flex`. These tools are stable and powerful, but they are struggling to keep up with modern development practices, such as:

1. Lack of syntax highlighting and auto-completion in the editor for the HCL.
2. Lack of type checking and error reporting when writing HCL source code.
3. Lack of a good way to debug the Y86 processor described by the HCL.

Moreover, the C implementation of the simulator in the original Archlab triggers each pipeline stage in a fixed order (F-M-E-D-W), which is not flexible enough to support arbitrary HCL descriptions.

In order to provide a better lab experience with greater flexibility and modern development practices, we decided to reimplement the HCL parser and simulator in Rust. With the help of the VSCode extension `rust-analyzer`, we can now enjoy syntax highlighting, auto-completion, type checking, and error reporting when writing HCL source code. Moreover, a debugger and its corresponding VSCode extension are also provided to help debug the Y86 processor described by the HCL.

2 Rust Implementation

One's **HCL-rs** source code is written in some macro blocks in a Rust source file. These macros convert your descriptions into Rust source code that defines a simulator structure. Macro blocks are processed during the preprocessing stage (just like `#define` in C), thus Rust compiler and editor extensions are able to check the **HCL-rs** source during development and in build time.

You can browse the macro implementation in `sim_macro` for details.

3 Data Types

In general, any Rust data types can be used in **HCL-rs** source. But firstly since the operators in expressions are limited, and secondly according to the hardware defined in this Archlab, we restrict available data types to the following items:

3.1 Primitive Types

Rust Type	Corresponding C/C++ Type	Example Literals
`u8`	`unsigned char`	`1u8`
`u64`	`unsigned long long`	`2u64`
`bool`	`bool`	`true`, `false`

Bear in mind that there is no C-style numerical implicit type conversion in Rust. i. e. For `u64 a` and `u8 b`, the expression `a == b` is invalid.

3.2 Predefined Types

There are some other data structures. You can inspect their definition in `sim/src/isa.rs` (The syntax of Rust is similar to C/C++, so you can easily understand the code):

1. `ConditionCode`: Flags stored in the cc register.

   Its Rust definition is

   ```rust
   /// A data structure that simulates the condition codes.
   #[derive(Debug, Clone, Copy, Default)]
   #[cfg_attr(feature = "serde", derive(serde::Serialize))]
   pub struct ConditionCode {
       pub sf: bool,
       pub of: bool,
       pub zf: bool,
   }
   ```

   To access its fields, you can write `cc.sf`, `cc.of`, etc.

   We've exported a constant default value of it, which can be helpful when you want to store `ConditionCode` in pipeline registers. Its definition is

   ```rust
   /// Default value of the condition code.
   pub const CC_INIT: ConditionCode = ConditionCode {
       sf: false,
       of: false,
       zf: false,
   };
   ```

   In HCL macro block, you can directly use `CC_INIT` to refer to it.

2. `Stat`: Status of the pipeline register.

   ```rust
   /// Simulator State (at each stage), depending on the hardware design.
   #[derive(Debug, Clone, PartialEq, Eq, Copy)]
   #[cfg_attr(feature = "serde", derive(serde::Serialize))]
   pub enum Stat {
       /// Indicates that everything is fine.
       Aok = 0,
       /// Indicates that the stage is bubbled. A bubbled stage execute the NOP
       /// instruction. Initially, all stages are in the bubble state.
       Bub = 1,
       /// The halt state. This state is assigned when the instruction fetcher
       /// reads the halt instruction. (If your architecture lacks a
       /// instruction fetcher, there should be some other way to specify the
       /// halt state in HCL.)
       Hlt = 2,
       /// This state is assigned when the instruction memory or data memory is
       /// accessed with an invalid address.
       Adr = 3,
       /// This state is assigned when the instruction fetcher reads an invalid
       /// instruction code.
       Ins = 4,
   }
   ```

This enum can be seen as a set of type-guarded constant values. For example, you can directly use `Aok`, `Bub` in your HCL source code. You may write `Aok == Bub` as a Boolean expression, but you cannot write `Aok == 1` since the operands do not belong to the same type.

Note that you are **NOT** forced to use this type for pipeline register status tracing. You can just use raw numerical values for that purpose. This data type is provided for code readability and debugger.

In the HCL macro block, you can directly use `Aok`, `Bub` to refer to these values.

4 HCL-rs Reference

This section describes the syntax and semantics of the **HCL-rs** language, which is written in the `sim_macro::hcl` macro block.

4.1 Comments

Comments in **HCL-rs** are similar to those in C. You can use `//` to start a line comment, or `/* ... */` to start a block comment.

However do not use `//!` or `///` (use `//<space>` or `//<space>!` instead) to start a doc comment, since it will be treated as Rust attributes.

4.2 Expressions

The **HCL-rs** parse your expression according to the following BNF rules:

```
<Expr>        ::= <LOrExpr>
<LOrExpr>     ::= <LAndExpr>
                | <LAndExpr> "||" <LOrExpr>
<LAndExpr>    ::= <RelExpr>
                | <RelExpr> "&&" <LAndExpr>
<RelExpr>     ::= <UnaryExpr>
                | <UnaryExpr> "==" <UnaryExpr>
                | <UnaryExpr> "!=" <UnaryExpr>
                | <UnaryExpr> "in" "{" (<LVal> ",")* <LVal> "}"
<UnaryExpr>   ::= <PrimaryExpr>
                | "!" <PrimaryExpr>
<PrimaryExpr> ::= <LVal>
                | <LitInt>
                | <LitBool>
                | "(" <Expr> ")"
<LVal>        ::= variable identifiers
<LitInt>      ::= numerical literal
<LitBool>     ::= "true"
                | "false"
```

The type of an expression is always `bool` as long as it includes at least one operator.

4.3 Combinatorial Logic Blocks

A *signal*, sometimes referred to as *intermediate signal*, is defined by its type and combinatorial logics in the following syntax:

```
<type t> <name> = [
  <cond_expr_1> : <case_expr_1 with type t>,
  <cond_expr_2> : <case_expr_2 with type t>,
  ...
];
```

The expression contains a series of cases, where each case i consists of a condition expression `cond_expr_i`, indicating whether this case should be selected, and a value expression `case_expr_i`, indicating the value resulting for this case. In evaluating a case expression, the condition expressions are conceptually evaluated in sequence. When one of them yields `true`, the value of the corresponding integer expression is returned as the case expression value. If no condition expression evaluates to `true`, then the value of the case expression is undefined. One good programming practice is to have the last condition expression be `true`, guaranteeing at least one matching case.

Note that you can create arbitrary signals in the HCL macro block, and all of them will be visible in the y86 debugger!

The following **HCL-rs** code snippet in `sim/src/architectures/builtin/seq_plus_std.rs` is taken as an example:

```
// What address should instruction be fetched at
u64 pc = [
    // Call.  Use instruction constant
    S.icode == CALL : S.valC;
    // Taken branch.  Use instruction constant
    S.icode == JX && S.cnd : S.valC;
    // Completion of RET instruction.  Use value from stack
    S.icode == RET : S.valM;
    // Default: Use incremented PC
    true : S.valP;
];
```

4.4 Set Device/PipeReg's Input Signal (HCL-rs feature)

Another drawback of the old archlab is that its HCL does not clearly show the relation between the hardware devices (pipeline register fields) and the intermediate signals. Here we provide a syntax to set the input signal of a device in the HCL code:

```
@set_input(device_identifier, {
  input_field_1, signal_1,
  input_field_2, signal_2,
  ...
});
```

The detailed list of input and output fields can be found in the hardware module. The device identifier is the short name of the device defined in the hardware module.

To set the input signal of a pipeline register, you can use the following syntax:

```
@set_stage(stage_pipereg_short_name, {
  input_field_1: signal_1,
  input_field_2: signal_2,
  ...
});
```

4.5 Global Attributes (HCL-rs feature)

Inside the `sim_macro::hcl` macro block, there are a few lines of code that describe some settings of the architecture:

1. `#![hardware = some::rust::module]`

   This attribute specifies the CPU hardware devices set. Essentially it will import all items from the hardware module.

2. `#![program_counter = pc_name]`

   It specifies the program counter by an intermediate signal. This value is read by the debugger. Conventionally, when we create a breakpoint at a line of code, the debugger seems to stop before executing the line of code. But in this simulator, The breakpoint takes effect when the current cycle is executed (so the value of pc is calculated) and *before entering the next cycle.*

   Changing this value to other signals makes no difference to the simulation. *But it affects the behavior of the y86 debugger.*

3. `#![termination = signal_name]`

   Specify a *boolean* intermediate signal named `signal_name` to indicate whether the program should be terminated.

4. `#![stage_alias(A => a, B => b, ...)]`

   This attribute defines the identifiers for pipeline registers. For `F => f`, which means this pipeline register in the last cycle is denoted by `F`, and in this cycle is denoted by `f`, the identifier `f` should be the short name of the pipeline register defined in `crate::define_stages`, while `F` can actually be selected arbitrarily.

   For a practical instance, `D.valA` is the value at the start of the cycle (you should treat it as read-only), while `d.valA` is the value at the end of the cycle (you should assign to it using `@set_stage(d, { valA: signal })`).

4.6 Stage Divider (HCL-rs feature)

We provide a special divider syntax to separate different (semantic) stages in your HCL logic. You can use `:====: title :====:` (the bar on both sides can be arbitrarily long) to declare a section. This helps to organize your code and the information displayed by the debugger. It makes no difference in the simulation. That means it does not alter the evaluation order of the CPU cycle.

With the stage divider, your entire HCL code may look like this:

```
sim_macro::hcl! {

// some global attributes ...
#![hardware = some::rust::module]
#![program_counter = pc_name]
#![termination = signal_name]
#![stage_alias(F => f, D => d, E => e, M => m, W => w)]

:==============================: Fetch Stage :================================:

// signals defined in fetch stage ...

:============================: Decode and Stage :=============================:

// signals defined in the decode stage ...

:=============================: Execute Stage :===============================:

// signals defined in the execute stage ...

:==============================: Memory Stage :===============================:

// signals defined in memory stage ...
:============================: Write Back Stage :=============================:

// signals defined in the write back stage ...

:=======================: Pipeline Register Control :========================:

// control the pipeline register of each stage (stall or bubble) ...

}
```

5 Pipeline Register Declaration (HCL-rs feature)

In this simulator, a pipeline register can be regarded as a struct, with each field assigned a default value. All pipeline registers are defined in the `crate::define_stages` macro block. The syntax is as follows:

```
crate::define_stages! {
    StageName1 stage_short_name_1 {
        field_name_1: field_type_1 = default_value_1,
        field_name_2: field_type_2 = default_value_2,
        ...
    }
    StageName2 stage_short_name_2 {
        ...
    }
}
```

Note that each pipeline register has *two implicit input fields*: `stall` and `bubble`. These special inputs are used to control the pipeline register's behavior. Still, you can set them in the HCL code by `@set_stage(stage_short_name, { stall: signal1, bubble: signal2 })`.

The usage of the pipeline register is the same as an ordinary struct in C. You can access its fields by `stage_last_name.field_name`, where `stage_last_name` is defined in

```
#![stage_alias(stage_last_name => stage_short_name)]
```

Although accessing `stage_short_name.field_name` does not cause a compile-time error, it is meaningless as `stage_short_name` is provided for writing only.

6 Hardware Definitions

The hardware definition is written in Rust (which is not required to understand). You can find the hardware module in `sim/src/architectures/{hardware_seq.rs,hardware_pipe.rs}`. A hardware module defines the CPU hardware devices set, containing a list of devices (units), and their input and output fields, which are the only things one should know to implement a correct HCL logic.

To access the output of a device is as simple as `device_short_name.field_name`. To set the input of a device, you need to use the `@set_input` syntax.