jonathanpwang chore(fmt): wrap comments and format code comments (#1575)

e4e180c · 3 weeks ago

298 lines (261 loc) · 10.5 KB

```rust
use std::{
    array,
    borrow::{Borrow, BorrowMut},
};

use openvm_circuit::arch::{
    AdapterAirContext, AdapterRuntimeContext, ImmInstruction, Result, VmAdapterInterface,
    VmCoreAir, VmCoreChip,
};
use openvm_circuit_primitives::bitwise_op_lookup::{
    BitwiseOperationLookupBus, SharedBitwiseOperationLookupChip,
};
use openvm_circuit_primitives_derive::AlignedBorrow;
use openvm_instructions::{instruction::Instruction, program::PC_BITS, LocalOpcode};
use openvm_rv32im_transpiler::Rv32AuipcOpcode::{self, *};
use openvm_stark_backend::{
    interaction::InteractionBuilder,
    p3_air::{AirBuilder, BaseAir},
    p3_field::{Field, FieldAlgebra, PrimeField32},
    rap::BaseAirWithPublicValues,
};
use serde::{Deserialize, Serialize};

use crate::adapters::{RV32_CELL_BITS, RV32_REGISTER_NUM_LIMBS};

const RV32_LIMB_MAX: u32 = (1 << RV32_CELL_BITS) - 1;

#[repr(C)]
#[derive(Debug, Clone, AlignedBorrow)]
pub struct Rv32AuipcCoreCols<T> {
    pub is_valid: T,
    // The limbs of the immediate except the least significant limb since it is always 0
    pub imm_limbs: [T; RV32_REGISTER_NUM_LIMBS - 1],
    // The limbs of the PC except the most significant and the least significant limbs
    pub pc_limbs: [T; RV32_REGISTER_NUM_LIMBS - 2],
```

```rust
        pub pc_limbs: [T; RV32_REGISTER_NUM_LIMBS - 2],
        pub rd_data: [T; RV32_REGISTER_NUM_LIMBS],
    }

    #[derive(Debug, Clone)]
    pub struct Rv32AuipcCoreAir {
        pub bus: BitwiseOperationLookupBus,
    }

    impl<F: Field> BaseAir<F> for Rv32AuipcCoreAir {
        fn width(&self) -> usize {
            Rv32AuipcCoreCols::<F>::width()
        }
    }

    impl<F: Field> BaseAirWithPublicValues<F> for Rv32AuipcCoreAir {}

    impl<AB, I> VmCoreAir<AB, I> for Rv32AuipcCoreAir
    where
        AB: InteractionBuilder,
        I: VmAdapterInterface<AB::Expr>,
        I::Reads: From<[[AB::Expr; 0]; 0]>,
        I::Writes: From<[[AB::Expr; RV32_REGISTER_NUM_LIMBS]; 1]>,
        I::ProcessedInstruction: From<ImmInstruction<AB::Expr>>,
    {
        fn eval(
            &self,
            builder: &mut AB,
            local_core: &[AB::Var],
            from_pc: AB::Var,
        ) -> AdapterAirContext<AB::Expr, I> {
            let cols: &Rv32AuipcCoreCols<AB::Var> = (*local_core).borrow();

            let Rv32AuipcCoreCols {
                is_valid,
                imm_limbs,
                pc_limbs,
                rd_data,
            } = *cols;
            builder.assert_bool(is_valid);

            // We want to constrain rd = pc + imm (i32 add) where:
            // - rd_data represents limbs of rd
            // - pc_limbs are limbs of pc except the most and least significant limbs
            // - imm_limbs are limbs of imm except the least significant limb

            // We know that rd_data[0] is equal to the least significant limb of PC
            // Thus, the intermediate value will be equal to PC without its most significant limb:
            let intermed_val = rd_data[0]
                + pc_limbs
                    .iter()
                    .enumerate()
                    .fold(AB::Expr::ZERO, |acc, (i, &val)| {
```

```
                acc + val * AB::Expr::from_canonical_u32(1 << ((i + 1) * RV32_CELL_BITS))
            });

        // Compute the most significant limb of PC
        let pc_msl = (from_pc - intermed_val)
            * AB::F::from_canonical_usize(1 << (RV32_CELL_BITS * (RV32_REGISTER_NUM_LIMBS - 1)))
                .inverse();

        // The vector pc_limbs contains the actual limbs of PC in little endian order
        let pc_limbs = [rd_data[0]]
            .iter()
            .chain(pc_limbs.iter())
            .map(|x| (*x).into())
            .chain([pc_msl])
            .collect::<Vec<AB::Expr>>();

        let mut carry: [AB::Expr; RV32_REGISTER_NUM_LIMBS] = array::from_fn(|_| AB::Expr::ZERO);
        let carry_divide = AB::F::from_canonical_usize(1 << RV32_CELL_BITS).inverse();

        // Don't need to constrain the least significant limb of the addition
        // since we already know that rd_data[0] = pc_limbs[0] and the least significant limb of
        // is 0 Note: imm_limbs doesn't include the least significant limb so imm_limbs[i -
        // 1] means the i-th limb of imm
        for i in 1..RV32_REGISTER_NUM_LIMBS {
            carry[i] = AB::Expr::from(carry_divide)
                * (pc_limbs[i].clone() + imm_limbs[i - 1] - rd_data[i] + carry[i - 1].clone());
            builder.when(is_valid).assert_bool(carry[i].clone());
        }

```



```
                .eval(builder, is_valid);
        }

        // The immediate and PC limbs need range checking to ensure they're within [0,
        // 2^RV32_CELL_BITS) Since we range check two items at a time, doing this way helps
        // efficiently divide the limbs into groups of 2 Note: range checking the limbs of
        // immediate and PC separately would result in additional range checks       since
        // they both have odd number of limbs that need to be range checked
        let mut need_range_check: Vec<AB::Expr> = Vec::new();
        for limb in imm_limbs {
            need_range_check.push(limb.into());
        }

        // pc_limbs[0] is already range checked through rd_data[0]
        for (i, limb) in pc_limbs.iter().skip(1).enumerate() {
            if i == pc_limbs.len() - 1 {
                // Range check the most significant limb of pc to be in [0,
                // 2^{PC_BITS-(RV32_REGISTER_NUM_LIMBS-1)*RV32_CELL_BITS})
                need_range_check.push(
                    (*limb).clone()
```

```
                            * AB::Expr::from_canonical_usize(
                                1 << (pc_limbs.len() * RV32_CELL_BITS - PC_BITS),
                            ),
                    );
                } else {
                    need_range_check.push((*limb).clone());
                }
            }

            // need_range_check contains (RV32_REGISTER_NUM_LIMBS - 1) elements from imm_limbs
            // and (RV32_REGISTER_NUM_LIMBS - 1) elements from pc_limbs
            // Hence, is of even length 2*RV32_REGISTER_NUM_LIMBS - 2
            assert_eq!(need_range_check.len() % 2, 0);
            for pair in need_range_check.chunks_exact(2) {
                self.bus
                    .send_range(pair[0].clone(), pair[1].clone())
                    .eval(builder, is_valid);
            }

            let imm = imm_limbs
                .iter()
                .enumerate()
                .fold(AB::Expr::ZERO, |acc, (i, &val)| {
                    acc + val * AB::Expr::from_canonical_u32(1 << (i * RV32_CELL_BITS))
                });
            let expected_opcode = VmCoreAir::<AB, I>::opcode_to_global_expr(self, AUIPC);
            AdapterAirContext {
                to_pc: None,
                reads: [].into(),
                writes: [rd_data.map(|x| x.into())].into(),
                instruction: ImmInstruction {
                    is_valid: is_valid.into(),
                    opcode: expected_opcode,
                    immediate: imm,
                }
                .into(),
            }
        }

        fn start_offset(&self) -> usize {
            Rv32AuipcOpcode::CLASS_OFFSET
        }
    }

    #[repr(C)]
    #[derive(Debug, Clone, Serialize, Deserialize)]
    pub struct Rv32AuipcCoreRecord<F> {
        pub imm_limbs: [F; RV32_REGISTER_NUM_LIMBS - 1],
        pub pc_limbs: [F; RV32_REGISTER_NUM_LIMBS - 2],
        pub rd_data: [F; RV32_REGISTER_NUM_LIMBS],
    }

    pub struct Rv32AuipcCoreChip {
```

```rust
    pub air: Rv32AuipcCoreAir,
    pub bitwise_lookup_chip: SharedBitwiseOperationLookupChip<RV32_CELL_BITS>,
}

impl Rv32AuipcCoreChip {
    pub fn new(bitwise_lookup_chip: SharedBitwiseOperationLookupChip<RV32_CELL_BITS>) -> Self {
        Self {
            air: Rv32AuipcCoreAir {
                bus: bitwise_lookup_chip.bus(),
            },
            bitwise_lookup_chip,
        }
    }
}

impl<F: PrimeField32, I: VmAdapterInterface<F>> VmCoreChip<F, I> for Rv32AuipcCoreChip
where
    I::Writes: From<[[F; RV32_REGISTER_NUM_LIMBS]; 1]>,
{
    type Record = Rv32AuipcCoreRecord<F>;
    type Air = Rv32AuipcCoreAir;

    #[allow(clippy::type_complexity)]
    fn execute_instruction(
        &self,
        instruction: &Instruction<F>,
        from_pc: u32,
        _reads: I::Reads,
    ) -> Result<(AdapterRuntimeContext<F, I>, Self::Record)> {
        let local_opcode = Rv32AuipcOpcode::from_usize(
            instruction
                .opcode
                .local_opcode_idx(Rv32AuipcOpcode::CLASS_OFFSET),
        );
        let imm = instruction.c.as_canonical_u32();
        let rd_data = run_auipc(local_opcode, from_pc, imm);
        let rd_data_field = rd_data.map(F::from_canonical_u32);

        let output = AdapterRuntimeContext::without_pc([rd_data_field]);

        let imm_limbs = array::from_fn(|i| (imm >> (i * RV32_CELL_BITS)) & RV32_LIMB_MAX);
        let pc_limbs: [u32; RV32_REGISTER_NUM_LIMBS] =
            array::from_fn(|i| (from_pc >> (i * RV32_CELL_BITS)) & RV32_LIMB_MAX);

        for i in 0..(RV32_REGISTER_NUM_LIMBS / 2) {
            self.bitwise_lookup_chip
                .request_range(rd_data[i * 2], rd_data[i * 2 + 1]);
        }

        let mut need_range_check: Vec<u32> = Vec::new();
        for limb in imm_limbs {
            need_range_check.push(limb);
        }
```

```
247
248             for (i, limb) in pc_limbs.iter().skip(1).enumerate() {
249                 if i == pc_limbs.len() - 1 {
250                     need_range_check.push((*limb) << (pc_limbs.len() * RV32_CELL_BITS - PC_BITS));
251                 } else {
252                     need_range_check.push(*limb);
253                 }
254             }
255
256             for pair in need_range_check.chunks(2) {
257                 self.bitwise_lookup_chip.request_range(pair[0], pair[1]);
258             }
259
260             Ok((
261                 output,
262                 Self::Record {
263                     imm_limbs: imm_limbs.map(F::from_canonical_u32),
264                     pc_limbs: array::from_fn(|i| F::from_canonical_u32(pc_limbs[i + 1])),
265                     rd_data: rd_data.map(F::from_canonical_u32),
266                 },
267             ))
268         }
269
270         fn get_opcode_name(&self, opcode: usize) -> String {
271             format!(
272                 "{:?}",
273                 Rv32AuipcOpcode::from_usize(opcode - Rv32AuipcOpcode::CLASS_OFFSET)
274             )
275         }
276
277         fn generate_trace_row(&self, row_slice: &mut [F], record: Self::Record) {
278             let core_cols: &mut Rv32AuipcCoreCols<F> = row_slice.borrow_mut();
279             core_cols.imm_limbs = record.imm_limbs;
280             core_cols.pc_limbs = record.pc_limbs;
281             core_cols.rd_data = record.rd_data;
282             core_cols.is_valid = F::ONE;
283         }
284
285         fn air(&self) -> &Self::Air {
286             &self.air
287         }
288     }
289
290     // returns rd_data
291     pub(super) fn run_auipc(
292         _opcode: Rv32AuipcOpcode,
293         pc: u32,
294         imm: u32,
295     ) -> [u32; RV32_REGISTER_NUM_LIMBS] {
296         let rd = pc.wrapping_add(imm << RV32_CELL_BITS);
297         array::from_fn(|i| (rd >> (RV32_CELL_BITS * i)) & RV32_LIMB_MAX)
298     }
```