Static Simulator

Jiaqi Si

CUHK(SZ)

2025-07-25

Outline

Outline

1. Simulator Structure

1.1 Main Three Parts

The simulator is mainly divided into three main parts: the fetch unit, the execution section, and the register.

Among these, the fetch unit is relatively simple and only needs to queue instructions, as we do not handle jump statements.

2. Execution Section

2.1 Execution Section

The execution section consists of two main types of units:

- **Function Units**: Handle computational operations
- **Memory Units**: Handle load/store operations

Both unit types share a common buffer architecture design.

2.2 Function Unit Types

The simulator supports multiple function unit types:

- **Vector Units**: VectorAlu, VectorMul, VectorDiv, VectorSlide
- **Float Units**: FloatAlu, FloatMul, FloatDiv
- **Integer Units**: IntegerAlu, IntegerDiv

Each function unit operates independently and can process different instruction types concurrently.

2.3 Memory Unit Structure

The Load/Store Unit (LSU) handles memory operations:

- **Multiple Ports**: Separate read and write ports
- **Port-based Processing**: Each port can handle one memory operation at a time
- **Direction Support**: Read (load) and Write (store) operations

```
pub enum Direction {
    Read,   // Load operations
    Write   // Store operations
}
```

2.4 Buffer Architecture Overview

Each execution unit (both function and memory) uses a **BufferPair** structure:

```
pub struct BufferPair {
    pub input_buffer: InputBuffer,
    pub result_buffer: ResultBuffer,
    pub current_instruction: Option<Inst>,
    pub owner: BufferOwnerType,
}
```

This design provides:

- **Input buffering** for operand data
- **Output buffering** for results
- **Instruction tracking** for current operation

2.5 Input Buffer Structure

The Input Buffer manages multiple resource types:

```
pub struct InputBuffer {
    pub resource: Vec<Resource>
}

pub struct Resource {
    pub resource_type: ResourceType,
    pub target_size: u32,    // Total bytes needed
    pub current_size: u32,   // Current bytes available
}
```

Resource Types:

- `Register(RegisterType)`: Vector/Scalar/Float registers

2.5 Input Buffer Structure

- `Memory`: Memory data

2.6 Output Buffer Structure

The Result Buffer stores computation results:

```
pub struct ResultBuffer {
    pub destination: Option<EnhancedResource>
}

pub struct EnhancedResource {
    pub resource_type: ResourceType,
    pub target_size: u32,       // Total bytes to produce
    pub current_size: u32,      // Current bytes stored
    pub consumed_bytes: u32     // Bytes already consumed
}
```

This tracks both **production** and **consumption** of results.

2.7 Buffer Event System

The buffer system uses an event-driven model:

Producer Events: Add data to buffers

```
pub struct ProducerEvent {
    pub resource_index: usize,
    pub append_length: u32,
}
```

Consumer Events: Read data from buffers

```
pub struct ConsumerEvent {
    pub maximum_consume_length: u32,
}
```

2.8 Function Unit Processing Flow

1. **Input Stage**: Operands arrive in Input Buffer
2. **Event Generation**: EventGenerator creates processing events
3. **Execution**: Function unit processes data, remove an event from the queue and change the state of the Result Buffer
4. **Output Stage**: Results stored in Result Buffer
5. **Consumption**: Register file reads results

```
pub struct EventGenerator {
    func_inst: FuncInst,
    cycle_per_event: u32,
    bytes_per_event: u32,
    total_bytes: u32,
```

2.8 Function Unit Processing Flow

```
        processed_bytes: u32,
    }
```

2.9 Memory Unit Processing Flow

1. **Port Allocation**: Find available read/write port
2. **Data Waiting**: Check if data is available in Input Buffer
3. **Data Transfer**: Move data between memory and buffers
4. **Port Release**: Free port for next operation

```
pub struct MemoryPortEventGenerator {
    index: usize,
    bytes_per_cycle: u32,
    raw_inst: MemInst,
    total_bytes: u32,
    current_pos: u32,
}
```

2.10 Latency Calculation

Each unit type has configurable latency:

```
pub fn calc_func_cycle(inst: &FuncInst) -> u32 {
    match inst.get_key_type() {
        FunctionUnitKeyType::IntegerAlu =>
config.function_units.interger_alu.latency,
        FunctionUnitKeyType::VectorAlu => /* depends on
float/int */,
        FunctionUnitKeyType::VectorDiv => /* depends on
float/int */,
        // ... other unit types
    }
}
```

Latencies are defined in the configuration file and can be customized.

3. Register

3.1 RegisterFile Structure

The RegisterFile manages three types of registers:

```
pub struct RegisterFile {
    pub scalar_registers: [CommonRegister; 32],
    pub vector_registers: [VectorRegister; 32],
    pub float_registers: [CommonRegister; 32],
}
```

Register Types:

- **Scalar Registers**: 32-bit integer values
- **Vector Registers**: Variable-length vector data
- **Float Registers**: Floating-point values

3.2 Register Internal Structure

Each register maintains a task queue for managing read/write operations:

CommonRegister (for Scalar and Float):

```
pub struct CommonRegister {
    pub task_queue: VecDeque<RegisterTask>,
    pub current_index: usize,
}
```

VectorRegister:

```
pub struct VectorRegister {
    pub task_queue: VecDeque<RegisterTask>,
```

3.2 Register Internal Structure

```
    pub current_index: usize,
}
```

3.3 RegisterTask Structure

Each task represents a read or write operation:

```
pub struct RegisterTask {
    pub current_place: u32,      // Current processing
position
    pub resource_index: usize,   // Buffer resource index
    pub behavior: TaskBehavior,  // Read or Write
    pub unit_key: UnitKeyType,   // Associated execution unit
}

pub enum TaskBehavior {
    Read,   // Read from register
    Write,  // Write to register
}
```

3.4 Task Registration Process

When an instruction is issued, read/write events are registered:

Function Unit Instructions:

```
pub fn add_task(&mut self, inst: &FuncInst, unit_key:
FuncKey) {
    // Add read tasks for source registers
    for src_reg in inst.get_source_registers() {
        self.add_read_task(src_reg, unit_key);
    }

    // Add write task for destination register
    if let Some(dst_reg) = inst.get_destination_register() {
        self.add_write_task(dst_reg, unit_key);
```

3.4 Task Registration Process

```
        }
    }
```

3.5 Memory Unit Task Registration

Memory Instructions:

```
pub fn add_mem_task(&mut self, inst: &MemInst, unit_key:
MemKey) {
    match unit_key {
        MemKey::Load => {
            // Read address dependencies
            self.add_read_task(inst.get_address_register(),
unit_key);
            // Write loaded data to destination
            self.add_write_task(inst.get_data_register(),
unit_key);
        },
        MemKey::Store => {
```

3.5 Memory Unit Task Registration

```
                // Read address and data dependencies
                self.add_read_task(inst.get_address_register(),
unit_key);
                self.add_read_task(inst.get_data_register(),
unit_key);
            }
        }
}
```

3.6 RegisterTaskHandler Trait

All register types implement the RegisterTaskHandler trait:

```
pub trait RegisterTaskHandler {
    fn handle_one_task(&mut self, forward_bytes: u32,
update_length: u32)
        -> Option<BufferEvent>;
    fn handle_event_result(&mut self, result:
BufferEventResult);
    fn generate_event(&mut self) -> Option<BufferEvent>;
    fn task_queue(&self) -> &VecDeque<RegisterTask>;
    fn get_total_bytes(&self, task: &RegisterTask) -> u32;
}
```

3.7 Task Processing Flow

1. **Task Creation**: When instruction issued, tasks added to register queues
2. **Event Generation**: Tasks generate BufferEvents (Producer/Consumer)
3. **Data Transfer**: Events move data between registers and execution units
4. **Progress Tracking**: Tasks track current_place and completion status
5. **Task Completion**: Completed tasks removed from queue

3.8 Data Forwarding Mechanism

The simulation supports data forwarding for chaining instructions

```
fn handle_one_task(&mut self, forward_bytes: u32,
update_length: u32)
    -> Option<BufferEvent> {
    if let Some(task) = self.task_queue().front_mut() {
        match task.behavior {
            TaskBehavior::Read => {
                // Generate Consumer event
                Some(BufferEvent::Consumer(ConsumerEvent
{ ... }))
            },
            TaskBehavior::Write => {
                // Generate Producer event
```

3.8 Data Forwarding Mechanism

```
                    Some(BufferEvent::Producer(ProducerEvent
{ ... }))
                }
            }
        }
}
```

3.9 Chaining

The data that needs to be forwarded will be registered as a **Read Task** queued after a certain **Write Task**. In this way, subsequent instructions can obtain the results already calculated by the previous instruction through the Register's processing of events in each cycle.

4. Current Limitations

4.1 Current Limitations

Does not support mask instructions.

Since MACC instructions need to be split into microinstructions, they are also not supported.

Each instruction must wait until the Result Buffer is cleared before it can be issued. Therefore, for scalar instructions, the fetch unit will be stuck for two cycles (being modified).