DTU

Emad Jacob Maroun

Scry: The Future-Oriented Instruction Set Architecture

Emad Jacob Maroun <ejama@dtu.dk>

- Postdoc at the Technical University of Denmark
- Research:
  - Compilation & optimization
  - Computer Architecture (caches)
  - Real-time & Mixed-Criticality Systems
  - New: Instruction set architecture

Survey

Raise your hand if you are familiar with:

- Assembly programming

Survey

Raise your hand if you are familiar with:

- Assembly programming
- RISC-V ISA

Survey

Raise your hand if you are familiar with:

- Assembly programming
- RISC-V ISA
- Computer architecture design

Survey

Raise your hand if you are familiar with:

- Assembly programming
- RISC-V ISA
- Computer architecture design
- Pipelining

Survey

Raise your hand if you are familiar with:

- Assembly programming
- RISC-V ISA
- Computer architecture design
- Pipelining
- Out-of-order (OoO) execution

Survey

Raise your hand if you are familiar with:

- Assembly programming
- RISC-V ISA
- Computer architecture design
- Pipelining
- Out-of-order (OoO) execution
- Register renaming

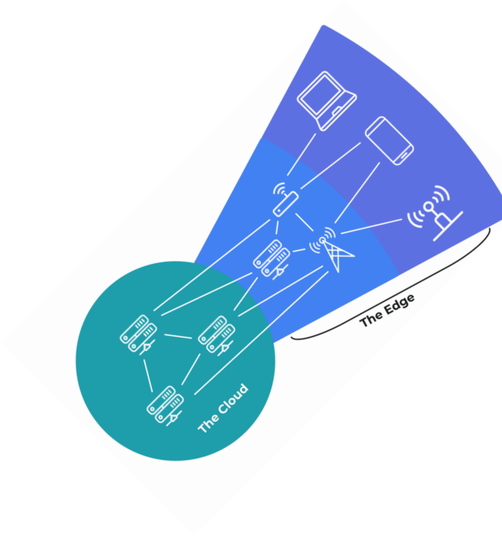
Survey

Raise your hand if you are familiar with:

- Assembly programming
- RISC-V ISA
- Computer architecture design
- Pipelining
- Out-of-order (OoO) execution
- Register renaming
- Branch prediction

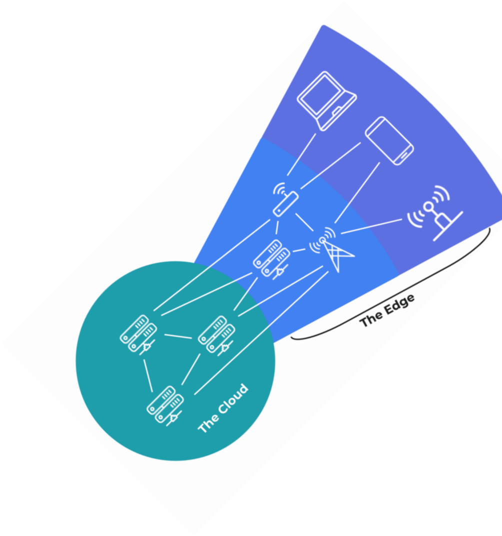
Embedded/Edge Performance

- More and more embedded/edge devices need performance
- Most need to be cheap and/or energy efficient
- Application examples:
  - Wearables
  - Medical
  - Edge AI
  - Personal Computing



Embedded/Edge Performance

- More and more embedded/edge devices need performance
- Most need to be cheap and/or energy efficient
- Application examples:
    - Wearables
    - Medical
    - Edge AI
    - Personal Computing
- **Problem:** On general-purpose CPUs, higher performance often leads to higher cost and lower energy efficiency.



Contemporary Performance

- CPUs achieve high performance by exploiting parallelism

Contemporary Performance

- CPUs achieve high performance by exploiting parallelism
- How: Find independent instruction $\rightarrow$ Execute in parallel

Contemporary Performance

- CPUs achieve high performance by exploiting parallelism
- How: Find independent instruction → Execute in parallel

```
add r1, r2, r3
..
add r4, r5, r6
..
add r7, r8, r9
..
..
```

Contemporary Performance

- CPUs achieve high performance by exploiting parallelism
- How: Find independent instruction $\rightarrow$ Execute in parallel
- 3 challenges:
  - You quickly run out of registers
  - You have to manage false dependencies
  - Branch conditions are late

```
add r1, r2, r3
..
add r4, r5, r6
..
add r7, r8, r9
..
..
```

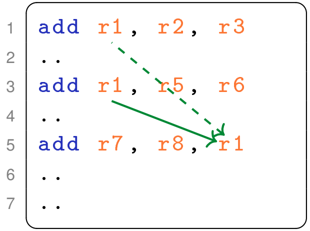
Contemporary Performance

- CPUs achieve high performance by exploiting parallelism
- How: Find independent instruction $\rightarrow$ Execute in parallel
- 3 challenges:
  - You quickly run out of registers
  - You have to manage false dependencies
  - Branch conditions are late

```
add r1, r2, r3
..
add r1, r5, r6
..
add r7, r8, r1
..
..
```

Contemporary Performance

- CPUs achieve high performance by exploiting parallelism
- How: Find independent instruction $\rightarrow$ Execute in parallel
- 3 challenges:
    - You quickly run out of registers
    - You have to manage false dependencies
    - Branch conditions are late



```
add r1, r2, r3
..
add r1, r5, r6
..
add r7, r8, r1
..
..
```

Contemporary Performance

- CPUs achieve high performance by exploiting parallelism
- How: Find independent instruction → Execute in parallel
- 3 challenges:
  - You quickly run out of registers
  - You have to manage false dependencies
  - Branch conditions are late

```
add r1, r2, r3
..
add r1, r5, r6
ld  r2, r10
add r7, r8, r1
beq r1, r2, label
..
```

Contemporary Performance

- CPUs achieve high performance by exploiting parallelism
- How: Find independent instruction $\rightarrow$ Execute in parallel
- 3 challenges:
  - You quickly run out of registers
  - You have to manage false dependencies
  - Branch conditions are late

```
add r1, r2, r3
..
add r1, r5, r6
ld  r2, r10
add r7, r8, r1
beq r1, r2, label
..
```

Contemporary Performance

- CPUs achieve high performance by exploiting parallelism
- How: Find independent instruction → Execute in parallel
- 3 challenges:
  - You quickly run out of registers
  - You have to manage false dependencies
  - Branch conditions are late
- Complicated fixes:
  - Register renaming: power hungry, scales poorly
  - Branch prediction: bottleneck, power hungry, large

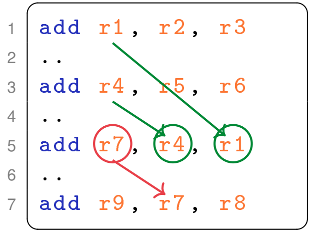
```
add r1, r2, r3
..
add r1, r5, r6
ld  r2, r10
add r7, r8, r1
beq r1, r2, label
..
```

Proposal

- Redesign the ISA to account for modern CPU design
- Approach: Give the CPU as much information about the future as possible
- Techniques:
    - Forward-Temporal Referencing
    - Deferred Control-Flow
- Other features summarized later

Register Referencing

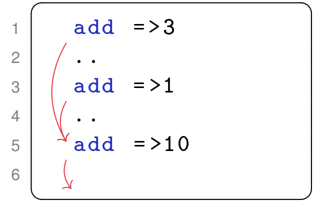
- All modern ISAs use registers for data flow
- Registers are referred to using their name
- Instruction typically have 1/2 inputs and 1 output
- Reference type: Spacial



```
add r1, r2, r3
..
add r4, r5, r6
..
add r7, r4, r1
..
add r9, r7, r8
```

Forward-Temporal Referencing

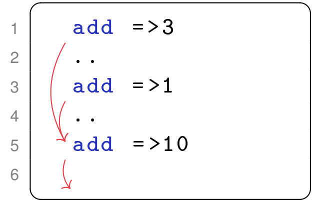
- Temporal Reference: How many instruction between producer and consumer



```
add =>3
..
add =>1
..
add =>10

```

Forward-Temporal Referencing

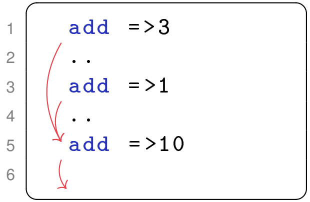
- Temporal Reference: How many instruction between producer and consumer
- Benefits:
  - Doesn't limit number of live variables
  - Doesn't exhibit false dependencies
  - Provides lifetimes up front $\rightarrow$ optimization
  - Dense encoding



```
add =>3
..
add =>1
..
add =>10
```

Forward-Temporal Referencing

- Temporal Reference: How many instruction between producer and consumer
- Benefits:
  - Doesn't limit number of live variables
  - Doesn't exhibit false dependencies
  - Provides lifetimes up front $\rightarrow$ optimization
  - Dense encoding
- Drawbacks:
  - Difficult to write assembly
  - Potentially infinite number of live values
  - Needs data-flow instructions



```
add =>3
..
add =>1
..
add =>10

```

Notice

- Inputs are implicit
- Input order: First produced
- Number of inputs may change semantics (`add` $\rightarrow$ `inc`)
- Values disappear after use
- Branching must ensure consumer's temporal position

Data-Flow Instructions

Echo

Increase reach

```
add   =>0
echo  =>100
```

Split values

```
some_fn:
echo  =>10,  =>20
```

Duplicate

Multiple use

```
add  =>0
dup  =>10,  =>20
```

Pick

Choose values

```
pick.i  1,  =>10
```

Example 1: strcpy

```
strcpy:
            echo =>dup_dst, =>dup_src   // =>3, =>0
lp_start:                               // Loop start
dup_src:    dup =>load, =>inc_src       // =>0, =>4
load:       ld  u8, =>0                 // Load next char
            dup =>lp_cond, =>store      // =>4, =>3
dup_dst:    dup =>store, =>0            // =>2, =>0
            add.s =>lp_end=>lp_start=>dup_dst // =>6
inc_src:    add.s =>lp_end=>lp_start=>dup_src // =>2
store:      st                          // Store char
lp_cond:    jmp lp_start, lp_end        // Loop on non-null
lp_end:     ret return_at
return_at:
```

Control-Flow Issues

- Conditions take a long time to evaluate → Stall
- Delay slots:
  X instructions following a branch always execute
- Splits (in time) branch from control-flow trigger

```
..
ld      r1
beq.d2  r1, r2, r3
add     r4, r5, r6
add     r7, r8, r9
..
```

Control-Flow Issues

- Conditions take a long time to evaluate → Stall
- Delay slots:
  X instructions following a branch always execute
- Splits (in time) branch from control-flow trigger
- Problems:
  - Might not fill delay slots
  - Might not be enough for deep pipelines
  - Makes OoO pipelines difficult to handle
    Must track which instructions come from delay slots

```
..
ld      r1
beq.d2  r1, r2, r3
add     r4, r5, r6
add     r7, r8, r9
..
```

Control-Flow Issues

- Conditions take a long time to evaluate → Stall
- Delay slots:
  X instructions following a branch always execute
- Splits (in time) branch from control-flow trigger
- Problems:
  - Might not fill delay slots
  - Might not be enough for deep pipelines
  - Makes OoO pipelines difficult to handle
    Must track which instructions come from delay slots
- Modern Solution: Branch prediction

```
..
ld      r1
beq.d2  r1, r2, r3
add     r4, r5, r6
add     r7, r8, r9
..
```

Deferred Control-Flow

- Split branch from trigger **in space**
- Branch: define trigger position
- Triggers only when execution reaches position

```
..
jmp jTo, jFrom
..
jFrom:
..
jTo:
..
```

Deferred Control-Flow

- Split branch from trigger **in space**
- Branch: define trigger position
- Triggers only when execution reaches position
- Benefits:
  - Gives time to evaluate condition
  - Large defer-range
  - No added complexity for OoO
  - Enables multiple in-flight branches
  - Enables parallel branch prediction

```
..
jmp jTo, jFrom
..
jFrom:
..
jTo:
..
```

Deferred Control-Flow

- Split branch from trigger **in space**
- Branch: define trigger position
- Triggers only when execution reaches position
- Benefits:
  - Gives time to evaluate condition
  - Large defer-range
  - No added complexity for OoO
  - Enables multiple in-flight branches
  - Enables parallel branch prediction
- Drawbacks:
  - May still stall/need prediction
  - Must track live branches

```
..
jmp jTo, jFrom
..
jFrom:
..
jTo:
..
```

Notice

- Defer distance can be large (max. 2048)
- Later branch instructions may trigger before earlier ones (e.g. nested loops)
- May not jump past a waiting trigger
- Picks minimize the amount of branches

Example 2: strcpy (deferred)

```
strcpy:
            echo =>dup_dst, =>dup_src
            ret return_at
lp_start:
dup_src:    dup =>load, =>inc_src
load:       ld  u8, =>0
            dup =>lp_cond, =>store
lp_cond:    jmp lp_start, lp_end // -3, 4
dup_dst:    dup =>store, =>0
            add.s =>lp_end=>lp_start=>dup_dst
inc_src:    add.s =>lp_end=>lp_start=>dup_src
store:      st
lp_end:     nop
return_at:
```

Other Features

- Internally tagged: All values have a type
- First class:
  - Function calling
  - Stack management
- Polymorphic instructions based on input number & type
- 16-bit instructions

Other Features

- Internally tagged: All values have a type
- First class:
  - Function calling
  - Stack management
- Polymorphic instructions based on input number & type
- 16-bit instructions
- Encoding efficiency:

	RISC-V (RV64IMC)	Scry	Relation
Code points	2 902 171 650	18 605	0,000 6 %
Utilization	68 % (32 bits)	28 % (16 bits)	41 %

Assembly Comparison

Assembly program statistics (Scry vs RV64IMC):

	Instructions		Bytes		Data		Control		*Logues	
Function	Scry	RV	Scry	RV	Scry	RV	Scry	RV	Scry	RV
`strcpy`	10	**7**	20	**18**	4	**1**	2	2	2	**1**
`memcpy`	14	**9**	28	**22**	6	**1**	3	3	3	**1**
`isxdigit`	13	13	**26**	40	6	**4**	**1**	3	**2**	3
`bsearch`	**33**	47	**66**	106	15	**9**	**5**	7	**7**	25
`cmpu8`	5	**4**	**10**	12	1	**0**	1	1	2	**1**
`find_max`	14	**13**	**28**	36	9	**5**	**2**	5	2	2
`hextol`	36	**31**	**72**	96	21	**8**	**2**	6	2	**1**

Scry ISA Simulator: Scryer

- ISA reference model for compiler and processor implementations
- Input: machine code or assembly
- Output:
  - Return values (Human readable)
  - Debug information
  - Runtime statistics

Scry ISA Simulator: Scryer

- ISA reference model for compiler and processor implementations
- Input: machine code or assembly
- Output:
  - Return values (Human readable)
  - Debug information
  - Runtime statistics
- Demo

Reflection

- What is a simulator?
  - State machine running on sequence of atomic inputs

Reflection

- What is a simulator?
  - State machine running on sequence of atomic inputs
- What do we need?
  - Result of running all inputs

Reflection

- What is a simulator?
  - State machine running on sequence of atomic inputs
- What do we need?
  - Result of running all inputs (Infinite possibilities)

Reflection

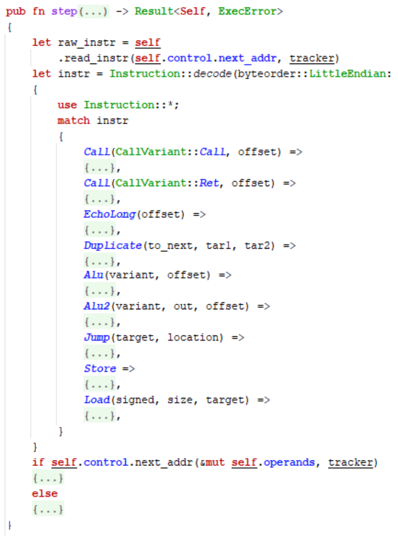
- What is a simulator?
  - State machine running on sequence of atomic inputs
- What do we need?
  - Result of running all inputs (Infinite possibilities)
- What part is easy to test?
  - State transition on single atomic input
  - Induction proof shows sequence will work

Reflection

- What is a simulator?
  - State machine running on sequence of atomic inputs
- What do we need?
  - Result of running all inputs (Infinite possibilities)
- What part is easy to test?
  - State transition on single atomic input
  - Induction proof shows sequence will work
- How should a simulator be implemented/tested?
  - Part 1: What is (easily) testable
  - Part 2: What is not (easily) testable

Scryer

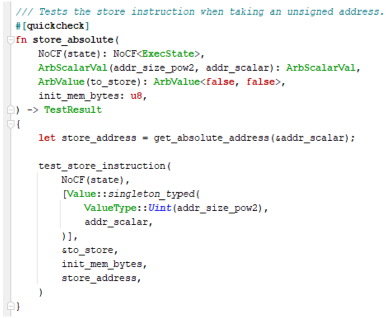
- Scry_sim: (State, Instruction) → State



```
pub fn step(...) -> Result<Self, ExecError>
{
    let raw_instr = self
        .read_instr(self.control.next_addr, tracker)
    let instr = Instruction::decode(byteorder::LittleEndian:
    {
        use Instruction::*;
        match instr
        {
            Call(CallVariant::Call, offset) =>
            {...},
            Call(CallVariant::Ret, offset) =>
            {...},
            EchoLong(offset) =>
            {...},
            Duplicate(to_next, tar1, tar2) =>
            {...},
            Alu(variant, offset) =>
            {...},
            Alu2(variant, out, offset) =>
            {...},
            Jump(target, location) =>
            {...},
            Store =>
            {...},
            Load(signed, size, target) =>
            {...},
        }
    }
    if self.control.next_addr(&mut self.operands, tracker)
    {...}
    else
    {...}
}
```

Scryer

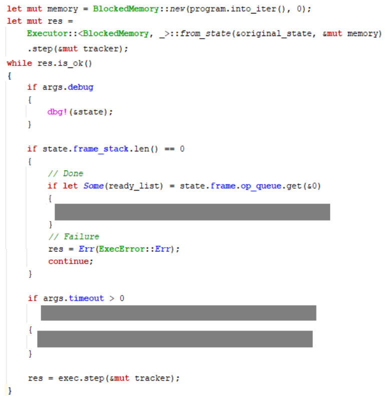
- Scry_sim: (State, Instruction) → State
- Property-based testing using QuickCheck
    - Properties: 90
    - Tests: 180 000



```rust
/// Tests the store instruction when taking an unsigned address.
#[quickcheck]
fn store_absolute(
    NoCF(state): NoCF<ExecState>,
    ArbScalarVal(addr_size_pow2, addr_scalar): ArbScalarVal,
    ArbValue(to_store): ArbValue<false, false>,
    init_mem_bytes: u8,
) -> TestResult
{
    let store_address = get_absolute_address(&addr_scalar);

    test_store_instruction(
        NoCF(state),
        [Value::singleton_typed(
            ValueType::Uint(addr_size_pow2),
            addr_scalar,
        )],
        &to_store,
        init_mem_bytes,
        store_address,
    )
}
```

Scryer

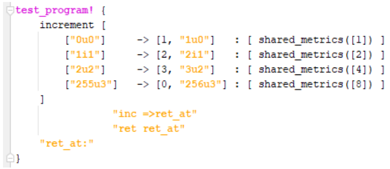
- Scry_sim: (State, Instruction) → State
- Property-based testing using QuickCheck
  - Properties: 90
  - Tests: 180 000
- Scryer: Wrapper around Scry_sim handling:
  - Setup initial state
  - Repeatedly calling Scry_sim
  - Handling any user-dependent functionality between calls



```rust
let mut memory = BlockedMemory::new(program.into_iter(), 0);
let mut res =
    Executor::<BlockedMemory, _>::from_state(&original_state, &mut memory)
    .step(&mut tracker);
while res.is_ok()
{
    if args.debug
    {
        dbg!(&state);
    }

    if state.frame_stack.len() == 0
    {
        // Done
        if let Some(ready_list) = state.frame.op_queue.get(&0)
        {
         [redacted]
        }
        // Failure
        res = Err(ExecError::Err);
        continue;
    }

    if args.timeout > 0
      [redacted]
    {
      [redacted]
    }

    res = exec.step(&mut tracker);
}
```

Scryer

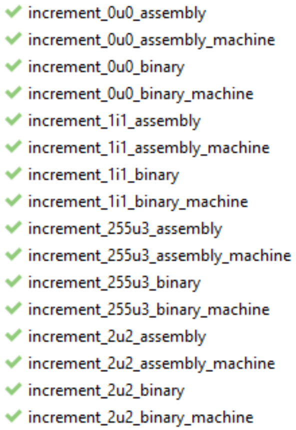
- Scry_sim: (State, Instruction) $\rightarrow$ State
- Property-based testing using QuickCheck
  - Properties: 90
  - Tests: 180 000
- Scryer: Wrapper around Scry_sim handling:
  - Setup initial state
  - Repeatedly calling Scry_sim
  - Handling any user-dependent functionality between calls
- Full program tests (not self checking!)



```
test_program! {
    increment [
        ["0u0"]     -> [1, "1u0"]   : [ shared_metrics([1]) ]
        ["1i1"]     -> [2, "2i1"]   : [ shared_metrics([2]) ]
        ["2u2"]     -> [3, "3u2"]   : [ shared_metrics([4]) ]
        ["255u3"]   -> [0, "256u3"] : [ shared_metrics([8]) ]
    ]
                "inc =>ret_at"
                "ret ret_at"
    "ret_at:"
}
```

Scryer

- Scry_sim: (State, Instruction) $\rightarrow$ State
- Property-based testing using QuickCheck
  - Properties: 90
  - Tests: 180 000
- Scryer: Wrapper around Scry_sim handling:
  - Setup initial state
  - Repeatedly calling Scry_sim
  - Handling any user-dependent functionality between calls
- Full program tests (not self checking!)
- Configuration matrix using Rust's macro system



```
✓ increment_0u0_assembly
✓ increment_0u0_assembly_machine
✓ increment_0u0_binary
✓ increment_0u0_binary_machine
✓ increment_1i1_assembly
✓ increment_1i1_assembly_machine
✓ increment_1i1_binary
✓ increment_1i1_binary_machine
✓ increment_255u3_assembly
✓ increment_255u3_assembly_machine
✓ increment_255u3_binary
✓ increment_255u3_binary_machine
✓ increment_2u2_assembly
✓ increment_2u2_assembly_machine
✓ increment_2u2_binary
✓ increment_2u2_binary_machine
```

Summary

- We need performance on the cheap
- Current expensive techniques:
  - Register allocation
  - Branch predication

Summary

- We need performance on the cheap
- Current expensive techniques:
  - Register allocation
  - Branch predication
- Scry:
  - Information about the future
  - Offload responsibility to the processor

Summary

- We need performance on the cheap
- Current expensive techniques:
    - Register allocation
    - Branch predication
- Scry:
    - Information about the future
    - Offload responsibility to the processor
- Features:
    - Forward-Temporal Referencing (FTR)
    - Deferred Control-Flow (DCF)
    - Internal tagging & instruction polymorphism
    - Extremely dense encoding

Summary

- We need performance on the cheap
- Current expensive techniques:
  - Register allocation
  - Branch predication
- Scry:
  - Information about the future
  - Offload responsibility to the processor
- Features:
  - Forward-Temporal Referencing (FTR)
  - Deferred Control-Flow (DCF)
  - Internal tagging & instruction polymorphism
  - Extremely dense encoding
- Open questions:
  - Can a compiler produce good code?
  - Can FTR be efficiently managed?
  - Can DCF be leveraged for easier or no prediction?
  - Do these techniques reduce energy consumption?