Pydgin for RISC-V: A Fast and Productive Instruction-Set-Set Simulator

Berkin Ilbeyi

In collaboration with Derek Lockhart (Google), and Christopher Batten

3rd RISC-V Workshop, Jan 2016
Motivation

Productivity
- Develop
- Extend
- Instrument

Performance
- Interpretive: 1-10 MIPS (1-10 days)
- Typical DBT: 100s MIPS (1-3 hours)
- QEMU DBT: 1000 MIPS (0.5 hours)
Architectural Description Language

Instruction Set Interpreter in C with DBT

[SimIt-ARM2006] [Wagstaff2013]

Key Insight:

Similar productivity-performance challenges for building high-performance interpreters of dynamic languages.
(e.g. JavaScript, Python)
Productivity  

Architectural Description Language

[SimIt-ARM2006]  
[Wagstaff2013]

Instruction Set Interpreter in C with DBT

Performance

Dynamic-Language Interpreter in RPython

Meta-Tracing JIT: makes JIT generation generic across languages

RPython Translation Toolchain

Dynamic Language Interpreter in C with JIT Compiler

Pydgin for RISC-V: A Fast and Productive Instruction-Set Simulator
Performance

Productivity

Architectural Description Language

Instruction Set Interpreter in C with DBT

Pydgin

RPython Translation Toolchain

JIT $\approx$ DBT

Pydgin for RISC-V: A Fast and Productive Instruction-Set Simulator
Pydgin Architecture Description Language

Architectural State
Instruction Encoding
Instruction Semantics
class State( object ):
    def __init__( self, memory, reset_addr=0x400 ):
        self.pc = reset_addr
        self.rf = RiscVRegisterFile()
        self.mem = memory

        # optional state if floating point is enabled
        if ENABLE_FP:
            self.fp = RiscVFPRegisterFile()
            self.fcsr = 0
Pydgin Architecture Description Language

Instruction Encoding

encodings = [
    # ...
    ['xori', 'xxxxxxxxxxxxxxxxxxxxx100xxxxx0010011'],
    ['ori', 'xxxxxxxxxxxxxxxxxxxxx110xxxxx0010011'],
    ['andi', 'xxxxxxxxxxxxxxxxxxxxx111xxxxx0010011'],
    ['slli', '000000xxxxxxxxxxx001xxxxx0010011'],
    ['srli', '000000xxxxxxxxxxx101xxxxx0010011'],
    ['srai', '010000xxxxxxxxxxx101xxxxx0010011'],
    ['add', '0000000xxxxxxxxxx000xxxxx0110011'],
    ['sub', '0100000xxxxxxxxxx000xxxxx0110011'],
    ['sll', '0000000xxxxxxxxxx001xxxxx0110011'],
    # ...
]
### Instruction Semantics

```python
def execute_addi(s, inst):
    s.rf[inst.rd] = s.rf[inst.rs1] + inst.i_imm
    s.pc += 4

def execute_sw(s, inst):
    addr = trim_xlen(s.rf[inst.rs1] + inst.s_imm)
    s.mem.write(addr, 4, trim_32(s.rf[inst.rs2]))
    s.pc += 4

def execute_beq(s, inst):
    if s.rf[inst.rs1] == s.rf[inst.rs2]:
        s.pc = trim_xlen(s.pc + inst.sb_imm)
    else:
        s.pc += 4
```
def instruction_set_interpreter( memory ):
    state = State( memory )

    while True:

        pc = state.fetch_pc()

        inst = memory[ pc ]  # fetch
        execute = decode( inst )  # decode
        execute( state, inst )  # execute
Pydgin Framework

Interpreter Loop

```python
def instruction_set_interpreter( memory ):
    state = State( memory )
    while True:
        pc = state.fetch_pc()
        inst = memory[ pc ]  # fetch
        execute = decode( inst )  # decode
        execute( state, inst )  # execute
```

Debug on Python Interpreter

100 KIPS
The RPython Translation Toolchain

- **State**
  - Encoding
  - Semantics

**Pydgin Framework**

**Debug on Python Interpreter**
- 100 KIPS

**RPython Source**
- Type Inference
- Optimization
- Code Generation
- Compilation

**Compiled Interpreter**
The RPython Translation Toolchain

- RPython Source
  - Type Inference
  - Optimization
  - Code Generation
  - Compilation
  - Compiled Interpreter

- Pydgin Interpretive Simulator
  - 10 MIPS

- Pydgin for RISC-V: A Fast and Productive Instruction-Set Simulator
  - 100 KIPS

- Pydgin Framework
  - Debug on Python Interpreter
  - 100 KIPS

- State
  - Encoding
  - Semantics
The RPython Translation Toolchain

- **State**
  - Encoding
  - Semantics

- **Pydgin Framework**
  - Debug on Python Interpreter
  - Pydgin Interpretive Simulator

- **RPython Source**
  - Type Inference
    - Optimization
    - Code Generation
    - Compilation
    - JIT Generator

- **Compiled Interpreter with JIT**
The RPython Translation Toolchain

- **Pydgin Framework**
  - **Type Inference**
  - **Optimization**
  - **Compilation**
  - **Code Generation**
  - **JIT Generator**
- **RPython Source**
- **Semantics**
- **Encoding**
- **Debug on Python Interpreter**
  - **Pydgin Interpretive Simulator** (100 KIPS)
  - **Pydgin DBT Simulator** (<10 MIPS)

- **Compiled Interpreter with JIT**

- **State**

Pydgin for RISC-V: A Fast and Productive Instruction-Set Simulator
JIT Annotations and Optimizations

Additional RPython JIT hints:
+ Elidable Instruction Fetch
+ Elidable Decode
+ Constant Promotion of PC and Memory
+ Word-Based Target Memory
+ Loop Unrolling in Instruction Semantics
+ Virtualizable PC and Statistics
+ Increased Trace Limit

Please see our ISPASS paper for more details!

SPECINT2006 on ARM

23X improvement over no annotations
Spike is an interpretive simulator with some advanced DBT features:

- Caching decoded instructions
- PC-indexed dispatch

RISC-V QEMU port was out-of-date at the time of our development
Pydgin Productivity

RISC-V encourages ISA extensions.

- Productive Development
- Productive Extensibility
- Productive Instrumentation
Pydgin RISC-V Development

Pydgin for RISC-V: A Fast and Productive Instruction-Set Simulator
Pydgin RISC-V Development

100+ MIPS simulator after 9 days of development!

Pydgin for RISC-V: A Fast and Productive Instruction-Set Simulator
Pydgin Extensibility

```python
encodings = [
    # ...,
    ['andi', 'xxxxxxxxxxxxxxxxxxxx111xxxxx0010011'],
    ['slli', '000000xxxxxxxxxxxx011xxxxx0010011'],
    ['srli', '000000xxxxxxxxxxxx101xxxxx0010011'],
    ['srai', '010000xxxxxxxxxxxx101xxxxx0010011'],
    ['add', '000000xxxxxxxxxxxx000xxxxx0110011'],
    # ...
    ['gcd', 'xxxxxxxxxxxxxxxxxxxx000xxxxx1011011'],
    # ...]

# greatest common divisor semantics
def execute_gcd( s, inst ):
    a, b = s.rf[inst.rs1], s.rf[inst.rs2]
    while b:
        a, b = b, a%b
    s.rf[inst.rd] = a
    s.pc += 4
```
# count number of adds

def execute_addi(s, inst):
    s.rf[inst.rd] = s.rf[inst.rs1] + inst.i_imm
    s.num_adds += 1
    s.pc += 4

# count misaligned stores

def execute_sw(s, inst):
    addr = trim_xlen(s.rf[inst.rs1] + inst.s_imm)
    if addr % 4 != 0:
        s.num_misaligned += 1
        s.mem.write(addr, 4, trim_32(s.rf[inst.rs2]))
    s.pc += 4

# record and count all executed loops

def execute_beq(s, inst):
    if s.rf[inst.rs1] == s.rf[inst.rs2]:
        old_pc = s.pc
        s.pc = trim_xlen(s.pc + inst.sb_imm)
        if s.pc <= old_pc: s.loops[(s.pc, old_pc)] += 1
    else:
        s.pc += 4
Pydgin in Our Research Group

- Statistics for software-defined regions
- Data-structure specialization experimentation
- Control- and memory-divergence for SIMD
- Basic Block Vector generation for SimPoint
- Analysis of JIT-enabled dynamic language interpreters
Conclusions

Pydgin leverages the RPython translation toolchain into high-performance, DBT Instruction Set Simulator.

Pydgin provides a succinct architecture description language within Python to give users a productive development, extension, and instrumentation experience.

Current State: RV64IMAFD (RV64G) Bare-Metal on 64-bit host

Thank you to our sponsors for their support:
NSF, DARPA, and donations from Intel Corporation and Synopsys, Inc.

https://github.com/cornell-brg/pydgin