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

Cornell University
Computer Systems Laboratory
Motivation
Motivation

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

Performance

Interpretive: 1-10 MIPS (1-10 days)
Motivation

Performance
Interpretive: 1-10 MIPS (1-10 days)
Typical DBT: 100s MIPS (1-3 hours)
Motivation

Performance

Interpretive: 1-10 MIPS (1-10 days)
Typical DBT: 100s MIPS (1-3 hours)
QEMU DBT: 1000 MIPS (0.5 hours)
## 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)

### Diagram

- Software
  - Instruction-Set Simulator
- Hardware
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)
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)

Vendor

Proprietary ISA

Software

Instruction-Set Simulator

Hardware
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)

Software
- Instruction-Set Simulator

Hardware

RISC-V Foundation

RISC-V

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

Performance
Performance

Productivity

Architectural Description Language

Instruction Set Interpreter in C with DBT
Productivity

Architectural Description Language

Performance

Instruction Set Interpreter in C with DBT
Performance

Productivity

Architectural Description Language

[SimIt-ARM2006] [Wagstaff2013]

Instruction Set Interpreter in C with DBT


Productivity

Architectural Description Language

[SimIt-ARM2006]
[Wagstaff2013]

Performance

Instruction Set Interpreter in C with DBT

Dynamic Language Interpreter in C with JIT Compiler

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

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

Similar productivity-performance challenges for building high-performance interpreters of dynamic languages. (e.g. JavaScript, Python)
Architectural Description Language → SimIt-ARM2006 [Wagstaff2013] → Instruction Set Interpreter in C with DBT

Dynamic-Language Interpreter in RPython → 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

[SimIt-ARM2006] [Wagstaff2013]

Instruction Set Interpreter in C with DBT

Dynamic-Language Interpreter in RPython

RPython Translation Toolchain

Dynamic Language Interpreter in C with JIT Compiler

Meta-Tracing JIT:
makes JIT generation generic across languages

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

Productivity  ➞  Performance

Architectural Description Language

RPython Translation Toolchain

JIT ≈ DBT

Instruction Set Interpreter in C with DBT
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

```python
encodings = [
    # ...
    ['xori', 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx100xxxxx0010011'],
    ['ori', 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx110xxxxx0010011'],
    ['andi', 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx111xxxxx0010011'],
    ['slli', '000000xxxxxxxxxxxx001xxxxx0010011'],
    ['srli', '000000xxxxxxxxxxxx101xxxxx0010011'],
    ['srai', '010000xxxxxxxxxxxx101xxxxx0010011'],
    ['add', '0000000xxxxxxxxxxxx000xxxxx0110011'],
    ['sub', '0100000xxxxxxxxxxxx000xxxxx0110011'],
    ['sll', '0000000xxxxxxxxxxxx001xxxxx0110011'],
    # ...
]
```
Pydgin Architecture Description Language

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

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
The RPython Translation Toolchain

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

- **Pydgin Framework**
  - State
    - Encoding
    - Semantics
  - Pydgin for RISC-V: A Fast and Productive Instruction-Set Simulator

- **Debug on Python Interpreter**
  - 100 KIPS

- **RPython Translation Toolchain**
The RPython Translation Toolchain

Pydgin Framework

- Encoding
- Semantics

State

Debug on Python Interpreter

100 KIPS

Pydgin Interpretive Simulator

10 MIPS

RPython Source

- Type Inference
- Optimization
- Code Generation
- Compilation

Compiled Interpreter
The RPython Translation Toolchain

RPython Source

- Type Inference
- Optimization
- JIT Generator
- Code Generation
- Compilation
- Compiled Interpreter with JIT

RPython Translation Toolchain

- Debug on Python Interpreter
- 100 KIPS
- Pydgin for RISC-V: A Fast and Productive Instruction-Set Simulator

Pydgin Interpretive Simulator

- 10 MIPS

Pydgin Framework

- Encoding
- Semantics
- State
The RPython Translation Toolchain

- State
- Encoding
- Semantics

Pydgin Framework

Debug on Python Interpreter

- Pydgin Interpretive Simulator
- Pydgin DBT Simulator
  - 100 KIPS
  - 10 MIPS
  - <10 MIPS

RPython Source

- Type Inference
- Optimization
- Code Generation
- Compilation
- JIT Generator

Compiled Interpreter with JIT

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

Pydgin Interpretive Simulator
10 MIPS

Pydgin DBT Simulator
100+ MIPS

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!

23X improvement over no annotations

SPECINT2006 on ARM
Pydgin Performance

MIPS

<table>
<thead>
<tr>
<th>Pydgin</th>
<th>Pydgin+DBT</th>
<th>Spike</th>
</tr>
</thead>
<tbody>
<tr>
<td>401.bzip2</td>
<td>429.mcf</td>
<td>445.gobmk</td>
</tr>
<tr>
<td>456.hmmr</td>
<td>458.sjeng</td>
<td>462.libquantum</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>471.omnetpp</td>
<td>473.astar</td>
</tr>
</tbody>
</table>
Pydgin Performance

![Pydgin Performance Chart]

- **Pydgin**
- **Pydgin+DBT**
- **Spike**

MIPS

- 401.bzip2
- 429.mcf
- 445.gobmk
- 456.hmmer
- 458.sjeng
- 462.libquantum
- 464.h264ref
- 471.omnetpp
- 473.astar

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

The graph compares the performance of Pydgin, Pydgin+DBT, and Spike in terms of MIPS (Million Instructions Per Second) for various benchmarks. The benchmarks include:
- 401.bzip2
- 429.mcf
- 445.gobmk
- 456.hmmer
- 458.sjeng
- 462.libquantum
- 464.h264ref
- 471.omnetpp
- 473.astar

Pydgin+DBT shows significantly higher MIPS compared to Pydgin and Spike, particularly for the 458.sjeng benchmark.
Spike is an interpretive simulator with some advanced DBT features:

- Caching decoded instructions
- PC-indexed dispatch
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
RISC-V encourages ISA extensions.
RISC-V encourages ISA extensions.

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

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

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

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

[riscv] declare virtualizers to keep rythym happy
[riscv] do some arith magic to calculate mult instructions
[riscv] some usability improvements to test-all script
Pydgin RISC-V Development

2015-06-19 Berkin Ilbeysi
[riscv] bootstrap differently for non-tests
2015-06-19 Derek Lockhart
[riscv] add floating-point register file, cleanup RiscV RF init
2015-06-19 Derek Lockhart
[riscv] add xlen/flen state, also csr and fcsr reg state
2015-06-19 Berkin Ilbeysi
[riscv] impl syscall emul wrappers
2015-06-19 Berkin Ilbeysi
[syscall-tests] compile syscall tests for riscv

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

12 / 16
Pydgin RISC-V Development

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

2015-06-21 Derek Lockhart
[riscv] add fdiv_s, fdiv_d, fsqrt_s, fsqrt_d insts

2015-06-21 Derek Lockhart
[riscv] add fnadd, fnmadd, fnmsub, fnmsub .s and .d insts

2015-06-21 Derek Lockhart
[riscv] don't import fp if fp not enabled

2015-06-21 Derek Lockhart
[riscv] hacky impl of f{min,max},{s,d}, doesn't support NaN correctly

2015-06-21 BerkIn Ilbeiy
[riscv] minor fixes to allow translation W/o fp

2015-06-21 BerkIn Ilbeiy
[riscv] impl tsgn, tsgnj, tsgnjx insns for .s and .d

2015-06-21 BerkIn Ilbeiy
[riscv] impl fsw and fsd instructions

2015-06-21 BerkIn Ilbeiy
[riscv] impl tr.w and sc.w instrs: all asm tests pass!

2015-06-21 BerkIn Ilbeiy
[riscv] impl tr_d and sc_d instrs, untested

2015-06-21 BerkIn Ilbeiy
[riscv] fix ld/st/jump bugs that show up in dense mem

2015-06-21 BerkIn Ilbeiy
[riscv] trim_64 fp ld/st ops as well

2015-06-21 BerkIn Ilbeiy
[riscv] map fsd and fdl to nop if fp not enabled

2015-06-21 BerkIn Ilbeiy
[riscv] start the stack from end of memory

2015-06-21 BerkIn Ilbeiy
[riscv] order tests

2015-06-21 BerkIn Ilbeiy
[riscv] grow mem for some spec bnarks

2015-06-21 BerkIn Ilbeiy
[scripts] added riscv as a translation target
Pydgin RISC-V Development

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

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

100+ MIPS simulator after 9 days of development!
Pydgin Extensibility

```
# greatest common divisor semantics

def execute_gcd( s, inst ):
```

```python
codings = [
    # ...
    ['andi', 'xxxxxxxxxxxxxxxxx111xxxxx0010011'],
    ['slli', '000000xxxxxxxxxx01xxxxx0010011'],
    ['srli', '000000xxxxxxxxx101xxxxx0010011'],
    ['srai', '010000xxxxxxxxx101xxxxx0010011'],
    ['add', '000000xxxxxxxxx000xxxxx0110011'],
    # ...
    ['custom2', 'xxxxxxxxxxxxxxxxxxxxxx000xxxxx1011011'],
    # ...
]
```
Pydgin Extensibility

```
encodings = [
    # ...
    ['andi', 'xxxxxxxxxxxxxxxxxxxx111xxxxx0010011'],
    ['sll', '00000xxxxxxxxxxxxxxxxxxx01xxxx0010011'],
    ['srli', '000000xxxxxxxxxxxx101xxxxx0010011'],
    ['sra', '010000xxxxxxxxxxxx101xxxxx0010011'],
    ['add', '000000xxxxxxxxxxxx000xxxx0110011'],
    # ...
    ['gcd', 'xxxxxxxxxxxxxxxxxxxx000xxxxx1011011'],
    # ...
]
```

# greatest common divisor semantics

def execute_gcd(s, inst):

# State

Pydgin Framework + JIT Annot.

Encoding

Semantics

Debug on Python Interpreter

RPython Translation Toolchain

Pydgin Interpretive Simulator

Pydgin DBT Simulator

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

```python
encodings = [
    # ...
    ['andi', 'xxxxxxxxxxxxxxxxxx111xxxxx0010011'],
    ['slli', '000000xxxxxxxxxxxx001xxxxx0010011'],
    ['srli', '000000xxxxxxxxxxxx001xxxxx0010011'],
    ['srai', '010000xxxxxxxxxxxx001xxxxx0010011'],
    ['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
```
Pydgin Instrumentation

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

# count misaligned stores
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

# record and count all executed loops
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
```
# 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 )
    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]:
        s.pc = trim_xlen( s.pc + inst.sb_imm )
    else:
        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]:
        s.pc = trim_xlen( s.pc + inst.sb_imm )
    else:
        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
Pydgin in Our Research Group

• Statistics for software-defined regions
• Data-structure specialization experimentation
Pydgin in Our Research Group

- Statistics for software-defined regions
- Data-structure specialization experimentation
- Control- and memory- divergence for SIMD
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
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

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

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

[Pydgin](https://github.com/cornell-brg/pydgin)

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