RISC-V8

a high-performance RISC-V to x86-64 binary translator
Standards

- Open Container Initiative – Linux Foundation
  - Cloud Storage, Networking, and *Compute*?
- T10 – SCSI, SATA, Fibre Channel, iSCSI, …
- Ethernet – IEEE 802.3, IETF Internet Protocol, …
Why

- Open Container Initiative – Linux Foundation
  - Cloud Storage, Networking, and Compute
- Standardized Container Execution Environment
  - a full stack standard requires a standardized ISA
  - x86-64 – encumbered de facto standard ISA
  - RISC-V – simple and elegant open standard ISA
How

- Develop RISC-V cloud execution environment
- Linux/POSIX User mode simulator
  - Container execution environment
- Linux/POSIX Full system emulator
  - Virtual machine execution environment
- Requires sufficient performance to be competitive
When

- Today
  - x86-64/AMD64 dominates commodity cloud
- 202x
  - RISC-V container execution environment
  - ~1X order of magnitude performance ratio
  - Piggy-back on x86-64/AMD64 commodity cloud
Step One

- Machine generated interpreter
- Encoding/decoding derived from riscv-opcodes
- Translator needs return to interpreter
Step Two

- Write a binary translator

- ...
## RISC-V vs x86-64

<table>
<thead>
<tr>
<th></th>
<th>RISC-V</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design</strong></td>
<td>RISC</td>
<td>CISC</td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td>Load/Store</td>
<td>Register Memory</td>
</tr>
<tr>
<td><strong>Registers</strong></td>
<td>31</td>
<td>16</td>
</tr>
<tr>
<td><strong>Bit width</strong></td>
<td>64/32</td>
<td>64/32</td>
</tr>
<tr>
<td><strong>Immediate width</strong></td>
<td>20/12</td>
<td>64/32</td>
</tr>
<tr>
<td><strong>Instruction sizes</strong></td>
<td>2,4</td>
<td>1,2,3,4,5,6,7,8,9,...</td>
</tr>
<tr>
<td><strong>Extension</strong></td>
<td>Sign Extend</td>
<td>Zero / Merge</td>
</tr>
<tr>
<td><strong>Control flow</strong></td>
<td>Link Register</td>
<td>Stack</td>
</tr>
</tbody>
</table>
The problem
Goals

- RISC-V front-end to a modern micro-architecture
- Explore (m:n IR) mapping from RISC-V to x86-64
- Map RISC-V ops to µops via legacy x86 decoder
- Maintain highest possible instruction density
- Target: ~1X order of magnitude performance
- Production quality RISC-V Execution Environment
ninetyninety rule

The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent.
Optimisations

- Statistically optimised static register allocation
- CISC Memory operands for spilled registers
- Inline caching of hot functions
- Indirect branch acceleration
- Macro-op fusion
- Branch tail dynamic linking
speed
More optimisations ...

- Phase II – dynamic register allocation
- Redundant sign extension elimination
- Lifting constants from registers to immediate
  - Constant propagation, reaching def analysis
- Lifting complex bit manipulation intrinsics
- De-optimisation metadata for correctness
CISC vs RISC

- x86-64
  - 2 operands, destructive
  - implicit dependencies
  - flexible but complex memory operands

- RISC-V
  - 3 operands, explicit dependencies, load, store
Register allocation

- rv8 currently uses a static allocation
  - Similar to the set of registers accessible by the RVC compressed ISA
- Access spilled registers using memory operands, 3 cycle latency for L1 access
- Investigating dynamic register allocation
  - Lift RISC-V machine code to SSA form
Register allocation

- ALU ops on native registers – ~1 cycle
- ALU ops on memory operands in L1 – ~3 cycles
- Register Allocation (Memory, Static, Dynamic)
  - Static allocation performance depends on the compiler’s choice of registers i.e. RVC
  - Benchmarks where static allocation is near optimal have close to native performance
Register allocation

```plaintext
jit-regalloc 0x00000000000127fa-0x00000000001281c

0x0000000000127fa  lbu  a4, 0(a1)
0x0000000000127fe  beq  a4, zero, pc + 48
0x000000000012800  lui  a5, 4096
0x000000000012802  addi  a5, a5, -1648
0x000000000012806  add  a5, a5, sp
0x000000000012808  slli  a2, a4, 2
0x00000000001280c  add  a2, a2, a5
0x00000000001280e  lw  a3, -1152(a2)
0x000000000012812  addi  a5, zero, 0
0x000000000012814  addi  a6, a3, 1
0x000000000012818  sw  a6, -1152(a2)
0x00000000001281c  andi  a2, a3, 1
0x000000000012820  slli  a5, a5, 1
0x000000000012822  addi  a4, a4, -1
0x000000000012824  or  a5, a5, a2
0x000000000012826  sral  a3, a3, 1
0x000000000012828  bne  a4, zero, pc - 12

1. a5  26.9% [7 ]
2. a2  23.1% [6 ]
3. a4  19.2% [5 ]
4. a3  15.4% [4 ]
5. a6  7.7% [2 ]
6. a1  3.8% [1 ]
7. sp  3.8% [1 ]
8. ra  0.0% [0 ]
9. t0  0.0% [0 ]
10. t1  0.0% [0 ]
11. a0  0.0% [0 ]
12. a7  0.0% [0 ]
```
Register allocation
Translator temporaries

<table>
<thead>
<tr>
<th>x86-64</th>
<th>Translator register purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbp</td>
<td>pointer to the register spill area and the L1 translation cache</td>
</tr>
<tr>
<td>rsp</td>
<td>pointer to the host stack for procedure calls</td>
</tr>
<tr>
<td>rax</td>
<td>general purpose translator temporary register</td>
</tr>
<tr>
<td>rcx</td>
<td>general purpose translator temporary register</td>
</tr>
</tbody>
</table>

- MOV rax, [rbp + n] pattern for read-modify-write on spilled registers requires 1 temporary register
- Loads and stores require 2 temporary registers for operations on spilled registers
- Complex operations such as MULH[S][U], DIV[U], REM[U] require 2 temporary registers
• Loads, stores and complex ops require 2 registers for operations on spilled registers

• Register allocator can assign temporary registers between loads, stores and complex ops

• Pipelined instructions (short live spans)

```
# 0x0000000000017e26    add         s2, s2, a0
# 0x00000000000017e26    lw          s2, 48(s2)
mov eax, dword [rbp+0x4C]
add eax, r8
movsxd eax, dword [eax+0x30]
mov dworsd [rbp+0x4C], eax
```
Immediate operands

- RISC-V uses registers for constants >12 bits
- x86-64 can encode 32-bit immediate operands and 64-bit with MOVABS
- Reduce register pressure
- Requires de-optimisation techniques to reify elided register values
Sign extension

- \( \text{0xffffffff80000000 vs 0x0000000080000000} \)

- Current implementation performs eager sign extension using MOVsx after all 32-bit operations

- This has a significant performance impact on 64-bit code that uses a lot of 32-bit operations

- Analysis pass can elide sign extension if the result is used in another 32-bit operation
  - or sign preserving operations (and/or/xor/not)
Indirect branches

- Indirect calls and returns require looking up trace cache to find address of translated code
  - RET, JALR ra, t0
- Avoid expensive returns to the interpreter
- Small assembly stub contains fast path that performs direct mapped trace cache lookup
Inline caching

- Inlining of hot functions
- Maintain pc histogram during interpretation
- Maintain call stack during translation
- Compare return address on return
  - Handle setjmp/longjmp and context switching
### Macro-op fusion

<table>
<thead>
<tr>
<th>Macro-op fusion pattern</th>
<th>Macro-op fusion expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUIPC r1, imm20; ADDI r1, r1, imm12</td>
<td>PC-relative address is resolved using a single MOV instruction. Target address is constructed using a single MOV instruction. Target address register write is elided.</td>
</tr>
<tr>
<td>AUIPC r1, imm20; JALR ra, imm12(r1)</td>
<td>Fused into single MOV with an immediate addressing mode</td>
</tr>
<tr>
<td>AUIPC ra, imm20; JALR ra, imm12(ra)</td>
<td>Fused into single MOV with an immediate addressing mode</td>
</tr>
<tr>
<td>AUIPC r1, imm20; LW r1, imm12(r1)</td>
<td>Fused into a single MOVZX instruction.</td>
</tr>
<tr>
<td>AUIPC r1, imm20; LD r1, imm12(r1)</td>
<td>Fused into 32-bit zero extending ADD instruction.</td>
</tr>
<tr>
<td>SLLI r1, r1, 32; SRLI r1, r1, 32</td>
<td>Fused into 64-bit ROR with one residual SHL or SHR temporary</td>
</tr>
<tr>
<td>ADDIW r1, r1, imm12; SLLI r1, r1, 32; SRLI r1, r1, 32</td>
<td>Fused into 32-bit ROR with one residual SHL or SHR temporary</td>
</tr>
<tr>
<td>SRLI r2, r1, imm12; SLLI r3, r1, (64-impact12); OR r2, r2, r3</td>
<td></td>
</tr>
<tr>
<td>SRLIW r2, r1, imm12; SLLIW r3, r1, (32-impact12); OR r2, r2, r3</td>
<td></td>
</tr>
</tbody>
</table>

- AUIPC+JALR is interpreted as a direct jump and link with target address temporary side effect
- AUIPC+{LW,LD} are fused into a single MOV
- SLLI r1,r1,32; SRLI r1,r1,32 is fused into MOVZX
Branch tail linking

- Current JIT gathers stats during interpretation and performs translation during runtime tracing
- Not taken sides of branches cause “tail exits”
- Tails branch to program counter trampolines
- When branch is taken enough times it is lazily dynamically linked to translated native code
- Incremental profile guided dynamic linking
Bit manipulation

- Lift simple patterns such as rotate and bit test
  - ROL, ROR, BT
- Lift more complex patterns such as byte swap
  - BSWAP, MOVBE
- Requires de-optimisation to elide temporaries
- Ultimately we need the ‘B’ Extension
De-optimisation

- Elide writes to temporary registers
  - AUIPC+JALR (CALL), ROT, ROL, BSWAP
- Elide register writes for 32-bit immediate operands
  - LI t0, 0xff000000
- Safety and correctness – Reaching definition analysis
- Interrupts – metadata to reconstruct elided values
Benchmarks
Benchmarks

- i386, x86-64, riscv32, riscv64, arm32, aarch64
- Host: Intel Core i7-5557U (3.1GHz, 4MB cache)
- Benchmarks run 20 times and best result is taken
- Position independent executables (-fPIE)
- x86-64 μops measured with
  - perf stat -e cycles, instructions, r1b1, r10e, r2c2, r1c2
## Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes</td>
<td>crypto</td>
<td>encrypt, decrypt and compare 30MiB of data</td>
</tr>
<tr>
<td>bigint</td>
<td>numeric</td>
<td>compute $23^{111121}$ and count base 10 digits</td>
</tr>
<tr>
<td>dhrystone</td>
<td>synthetic</td>
<td>well known synthetic integer workload</td>
</tr>
<tr>
<td>miniz</td>
<td>compression</td>
<td>compress, decompress and compare 8MiB of data</td>
</tr>
<tr>
<td>norx</td>
<td>crypto</td>
<td>encrypt, decrypt and compare 30MiB of data</td>
</tr>
<tr>
<td>primes</td>
<td>numeric</td>
<td>calculate largest prime number below 3333333333</td>
</tr>
<tr>
<td>qsort</td>
<td>sorting</td>
<td>sort array containing 50 million items</td>
</tr>
<tr>
<td>sha512</td>
<td>digest</td>
<td>calculate SHA-512 hash of 64MiB of data</td>
</tr>
</tbody>
</table>
# Ratios (-O3, 64-bit)

<table>
<thead>
<tr>
<th>Program</th>
<th>qemu-aarch64</th>
<th>qemu-riscv64</th>
<th>rv8-riscv64</th>
<th>native-x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes</td>
<td>4.12</td>
<td>6.76</td>
<td>4.68</td>
<td>1.00</td>
</tr>
<tr>
<td>bigint</td>
<td>3.62</td>
<td>2.83</td>
<td>1.85</td>
<td>1.00</td>
</tr>
<tr>
<td>dhrystone</td>
<td>9.96</td>
<td>5.87</td>
<td>2.03</td>
<td>1.00</td>
</tr>
<tr>
<td>miniz</td>
<td>3.46</td>
<td>2.86</td>
<td>1.99</td>
<td>1.00</td>
</tr>
<tr>
<td>norx</td>
<td>2.73</td>
<td>5.33</td>
<td>4.51</td>
<td>1.00</td>
</tr>
<tr>
<td>primes</td>
<td>3.49</td>
<td>2.11</td>
<td>1.09</td>
<td>1.00</td>
</tr>
<tr>
<td>qsort</td>
<td>11.55</td>
<td>7.46</td>
<td>1.90</td>
<td>1.00</td>
</tr>
<tr>
<td>sha512</td>
<td>2.66</td>
<td>5.13</td>
<td>3.36</td>
<td>1.00</td>
</tr>
<tr>
<td>(Geomean)</td>
<td>4.44</td>
<td>4.39</td>
<td>2.40</td>
<td>1.00</td>
</tr>
</tbody>
</table>
rv8-bench (Runtime -O3 32-bit)

- native-x86-32-O3-runtime
- rv8-riscv32-O3-runtime
- qemu-riscv32-O3-runtime
- qemu-arm32-O3-runtime

Runtime (secs)

- aes
- bigint
- dhrystone
- miniz
- norx
- primes
- qsort
- sha512
## Ratios (-O3, 32-bit)

<table>
<thead>
<tr>
<th>Program</th>
<th>qemu-arm32</th>
<th>qemu-riscv32</th>
<th>rv8-riscv32</th>
<th>native-x86-32</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes</td>
<td>3.56</td>
<td>3.97</td>
<td>3.09</td>
<td>1.00</td>
</tr>
<tr>
<td>bigint</td>
<td>3.39</td>
<td>1.56</td>
<td>1.61</td>
<td>1.00</td>
</tr>
<tr>
<td>dhrystone</td>
<td>4.13</td>
<td>3.91</td>
<td>1.37</td>
<td>1.00</td>
</tr>
<tr>
<td>miniz</td>
<td>3.40</td>
<td>2.47</td>
<td>1.60</td>
<td>1.00</td>
</tr>
<tr>
<td>norx</td>
<td>2.98</td>
<td>2.96</td>
<td>3.00</td>
<td>1.00</td>
</tr>
<tr>
<td>primes</td>
<td>2.79</td>
<td>1.55</td>
<td>1.25</td>
<td>1.00</td>
</tr>
<tr>
<td>qsort</td>
<td>12.04</td>
<td>6.54</td>
<td>1.65</td>
<td>1.00</td>
</tr>
<tr>
<td>sha512</td>
<td>6.04</td>
<td>4.60</td>
<td>3.04</td>
<td>1.00</td>
</tr>
<tr>
<td>(Geomean)</td>
<td>4.23</td>
<td>3.09</td>
<td>1.95</td>
<td>1.00</td>
</tr>
</tbody>
</table>
# MIPs (-O3, 64-bit)

<table>
<thead>
<tr>
<th>Program</th>
<th>qemu-riscv64</th>
<th>rv8-riscv64</th>
<th>native-x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes</td>
<td>2414</td>
<td>3489</td>
<td>11035</td>
</tr>
<tr>
<td>bigint</td>
<td>3738</td>
<td>5720</td>
<td>10557</td>
</tr>
<tr>
<td>dhrystone</td>
<td>1843</td>
<td>5327</td>
<td>8369</td>
</tr>
<tr>
<td>miniz</td>
<td>2625</td>
<td>3778</td>
<td>5530</td>
</tr>
<tr>
<td>norx</td>
<td>2223</td>
<td>2628</td>
<td>9112</td>
</tr>
<tr>
<td>primes</td>
<td>2438</td>
<td>4712</td>
<td>6100</td>
</tr>
<tr>
<td>qsort</td>
<td>644</td>
<td>2527</td>
<td>5780</td>
</tr>
<tr>
<td>sha512</td>
<td>2982</td>
<td>4550</td>
<td>12177</td>
</tr>
<tr>
<td>(Geomean)</td>
<td>2149</td>
<td>3932</td>
<td>8232</td>
</tr>
</tbody>
</table>
### MIPs (-O3, 32-bit)

<table>
<thead>
<tr>
<th>Program</th>
<th>qemu-riscv32</th>
<th>rv8-riscv32</th>
<th>native-x86-32</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes</td>
<td>2442</td>
<td>3137</td>
<td>9634</td>
</tr>
<tr>
<td>bigint</td>
<td>3964</td>
<td>3837</td>
<td>9780</td>
</tr>
<tr>
<td>dhrystone</td>
<td>1998</td>
<td>5696</td>
<td>3747</td>
</tr>
<tr>
<td>miniz</td>
<td>2195</td>
<td>3384</td>
<td>4988</td>
</tr>
<tr>
<td>norx</td>
<td>2824</td>
<td>2791</td>
<td>9146</td>
</tr>
<tr>
<td>primes</td>
<td>3039</td>
<td>3773</td>
<td>6368</td>
</tr>
<tr>
<td>qsort</td>
<td>671</td>
<td>2667</td>
<td>6259</td>
</tr>
<tr>
<td>sha512</td>
<td>2773</td>
<td>4200</td>
<td>11074</td>
</tr>
<tr>
<td>(Geomean)</td>
<td>2259</td>
<td>3586</td>
<td>7186</td>
</tr>
</tbody>
</table>
• Benchmark Runtimes
• Compiler Optimisation Level Comparison
• Instructions Per Second (MIPS)
• Dynamic Retired Micro-ops
• Static Executable File Sizes
• Dynamic Register Usage Histograms
• Dynamic Instruction Usage Histograms
Instruction counting

- JIT accelerated ("instret")
- Update counters at basic block boundaries
- Quickly gather dynamic instruction counts
- ~20% performance hit for ("instret")
  - Maintain billions of Instructions per second
- Potential to accelerate other performance counters
  - e.g. dynamic retired instruction bytes
Future work

• Native performance full system emulation

• Dynamic register allocation

• Coalescing op + loads and stores and indexed load stores into complex memory operands

• De-optimisation – register write elision for constants and macro-op fusion side effects

• Hardware accelerated MMU using shadow paging