ISA Shootout:
Comparing RISC-V, ARM, and x86 on SPECInt 2006
(or: How to make a high-performance RISC-V
processor using macro-op fusion)
Christopher Celio, Krste Asanovic, David Patterson
2016 July
celio@eecs.berkeley.edu
The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-op Fusion for RISC-V

full data is available as a tech report

https://arxiv.org/abs/1607.02318
Instruction Set Architecture (ISA)
Instruction Set Architecture (ISA)

- interface between the SW and the HW
Instruction Set Architecture (ISA)

- interface between the SW and the HW
- the "language" the processor speaks
Instruction Set Architecture (ISA)

- interface between the SW and the HW
- the "language" the processor speaks
- ISA =/= processor
Instruction Set Architecture (ISA)

- Interface between the SW and the HW
- The "language" the processor speaks
- ISA $\neq$ processor

Software

Architecture: RV64G

Micro-arch: rocket processor

BOOM processor

bzip2.cc
Instruction Set Architecture (ISA)

- interface between the SW and the HW
- the "language" the processor speaks
- ISA \(\neq\) processor

Software

Architecture

micro-arch

bzip2.cc

rocket processor

BOOM processor

Intel

UC Berkeley

AMD64 (x86-64)

ARMv7 (32-bit)

ARMv8 (64-bit)

rv64g (general-purpose)

rv64gc (compressed extension)
RISC vs CISC: Conventional Wisdom
RISC vs CISC: Conventional Wisdom

- CISC ISAs are more expressive, denser than RISC ISAs
RISC vs CISC: Conventional Wisdom

- CISC ISAs are more expressive, denser than RISC ISAs
- RISC ISAs map well to high-performance pipelines
RISC vs CISC: Conventional Wisdom

- CISC ISAs are more expressive, denser than RISC ISAs
- RISC ISAs map well to high-performance pipelines
- CISC instructions can be translated to RISC micro-ops
RISC vs CISC: Conventional Wisdom

- CISC ISAs are more expressive, denser than RISC ISAs
- RISC ISAs map well to high-performance pipelines
- CISC instructions can be translated to RISC micro-ops
RISC vs CISC: Conventional Wisdom

- CISC ISAs are more expressive, denser than RISC ISAs
- RISC ISAs map well to high-performance pipelines
- CISC instructions can be translated to RISC micro-ops
RISC vs CISC: Conventional Wisdom

- CISC ISAs are more expressive, denser than RISC ISAs
- RISC ISAs map well to high-performance pipelines
- CISC instructions can be translated to RISC micro-ops

My claim

a well-designed RISC ISA can be very competitive with CISC ISAs
Name that ARMv7 Instruction!

LDMIAEQ SP!, { R4-R7, PC }
Name that ARMv7 Instruction!

LDMIAEQ SP!, { R4-R7, PC }

- load multiple, increment-address
Name that ARMv7 Instruction!

LDMIAEQ SP!, { R4-R7, PC }

- load multiple, increment-address
- writes to 7 registers from 6 loads
Name that ARMv7 Instruction!

LDMIAEQ SP!, { R4-R7, PC }

- load multiple, increment-address
- writes to 7 registers from 6 loads
- only executes if EQ condition code is set
Name that ARMV7 Instruction!

LDMIAEQ SP!, { R4-R7, PC }

- load multiple, increment-address
- writes to 7 registers from 6 loads
- only executes if EQ condition code is set
- writes to the PC!
Name that ARMv7 Instruction!

LDMIAEQ SP!, { R4-R7, PC }

- load multiple, increment-address
- writes to 7 registers from 6 loads
- only executes if EQ condition code is set
- writes to the PC!
- idiom for "stack pop and return from a function call"
# The x86 Registers

<table>
<thead>
<tr>
<th>ZMM0</th>
<th>YMM0</th>
<th>XMM0</th>
<th>ZMM1</th>
<th>YMM1</th>
<th>XMM1</th>
<th>ST(0)</th>
<th>MM0</th>
<th>ST(1)</th>
<th>MM1</th>
<th>AL</th>
<th>AX</th>
<th>EAX</th>
<th>RAX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM2</td>
<td>YMM2</td>
<td>XMM2</td>
<td>ZMM3</td>
<td>YMM3</td>
<td>XMM3</td>
<td>ST(2)</td>
<td>MM2</td>
<td>ST(3)</td>
<td>MM3</td>
<td>BL</td>
<td>BH</td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td>ZMM4</td>
<td>YMM4</td>
<td>XMM4</td>
<td>ZMM5</td>
<td>YMM5</td>
<td>XMM5</td>
<td>ST(4)</td>
<td>MM4</td>
<td>ST(5)</td>
<td>MM5</td>
<td>CL</td>
<td>CH</td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td>ZMM6</td>
<td>YMM6</td>
<td>XMM6</td>
<td>ZMM7</td>
<td>YMM7</td>
<td>XMM7</td>
<td>ST(6)</td>
<td>MM6</td>
<td>ST(7)</td>
<td>MM7</td>
<td>DL</td>
<td>DH</td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>ZMM8</td>
<td>YMM8</td>
<td>XMM8</td>
<td>ZMM9</td>
<td>YMM9</td>
<td>XMM9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SI</td>
<td>SP</td>
<td>LP</td>
<td>ESP</td>
</tr>
<tr>
<td>ZMM10</td>
<td>YMM10</td>
<td>XMM10</td>
<td>ZMM11</td>
<td>YMM11</td>
<td>XMM11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RI</td>
<td>RSP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM12</td>
<td>YMM12</td>
<td>XMM12</td>
<td>ZMM13</td>
<td>YMM13</td>
<td>XMM13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BP</td>
<td>BPS</td>
<td>RBP</td>
<td></td>
</tr>
<tr>
<td>ZMM14</td>
<td>YMM14</td>
<td>XMM14</td>
<td>ZMM15</td>
<td>YMM15</td>
<td>XMM15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DI</td>
<td>EDI</td>
<td>RDI</td>
<td></td>
</tr>
<tr>
<td>ZMM16</td>
<td>ZMM17</td>
<td>ZMM18</td>
<td>ZMM19</td>
<td>ZMM20</td>
<td>ZMM21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SI</td>
<td>ESP</td>
<td>ESP</td>
<td></td>
</tr>
<tr>
<td>ZMM22</td>
<td>ZMM23</td>
<td>ZMM24</td>
<td>ZMM25</td>
<td>ZMM26</td>
<td>ZMM27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM30</td>
<td>ZMM31</td>
<td>ZMM28</td>
<td>ZMM29</td>
<td>ZMM30</td>
<td>ZMM31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **CW** (Control Word)
- **FP_IP** (Floating-Point Instructions)
- **FP_DP** (Floating-Point Data Processing)
- **FP_CS** (Floating-Point Control State)
- **ES** (Segment Register)
- **FS** (Segment Register)
- **GS** (Segment Register)
- **GDTR** (Global Descriptor Table Register)
- **TR** (Task Register)
- **IDTR** (Interrupt Descriptor Table Register)
- **DR0** (Debug Register 0)
- **DR1** (Debug Register 1)
- **DR2** (Debug Register 2)
- **DR3** (Debug Register 3)
- **DR4** (Debug Register 4)
- **DR5** (Debug Register 5)
- **DXCR** (Debug Exception Control Register)

## 256-bit Register
- + 80-bit Register
- + 64-bit Register
- + 16-bit Register
- 8-bit Register
- 512-bit Register
- 128-bit Register
- 32-bit Register

---

https://en.wikipedia.org/wiki/File:Table_of_x86_Registers_svg.svg

by Immae (Creative Commons Attribution-Share Alike 3.0 Unported)
My new favorite x86 instruction

- **vzerouppers** AVX instruction
  - zero upper 128-bits of YMM registers

```plaintext
IF (64-bit mode)
YMM0[VLMAX-1:128] ← 0
YMM1[VLMAX-1:128] ← 0
YMM2[VLMAX-1:128] ← 0
YMM3[VLMAX-1:128] ← 0
YMM4[VLMAX-1:128] ← 0
YMM5[VLMAX-1:128] ← 0
YMM6[VLMAX-1:128] ← 0
YMM7[VLMAX-1:128] ← 0
YMM8[VLMAX-1:128] ← 0
YMM9[VLMAX-1:128] ← 0
YMM10[VLMAX-1:128] ← 0
YMM11[VLMAX-1:128] ← 0
YMM12[VLMAX-1:128] ← 0
YMM13[VLMAX-1:128] ← 0
YMM14[VLMAX-1:128] ← 0
YMM15[VLMAX-1:128] ← 0
ELSE
YMM0[VLMAX-1:128] ← 0
YMM1[VLMAX-1:128] ← 0
YMM2[VLMAX-1:128] ← 0
YMM3[VLMAX-1:128] ← 0
YMM4[VLMAX-1:128] ← 0
YMM5[VLMAX-1:128] ← 0
YMM6[VLMAX-1:128] ← 0
YMM7[VLMAX-1:128] ← 0
YMM8[VLMAX-1:128] ← 0
YMM9[VLMAX-1:128] ← 0
YMM10[VLMAX-1:128] ← 0
YMM11[VLMAX-1:128] ← 0
YMM12[VLMAX-1:128] ← 0
YMM13[VLMAX-1:128] ← 0
YMM14[VLMAX-1:128] ← 0
YMM15[VLMAX-1:128] ← 0
FI
```
BOOM Processor

http://ucb-bar.github.io/riscv-boom

2-wide BOOM (16kB/16kB) 1.2mm² @ 45nm
Iron Law of Performance

\[
\text{Performance (secs/program)} = \frac{\text{Cycles}}{\text{Insts}} \times \frac{\text{seconds}}{\text{Cycles}} \times \frac{\text{Insts}}{\text{Program}}
\]
Iron Law of Performance

\[
\text{Performance (secs/program)} = \frac{\text{Insts} \times \text{Cycles}}{\text{Insts} \times \text{Cycles}} \times \frac{\text{Insts}}{\text{Program}}
\]

processor (\(\mu\text{arch}\))
Iron Law of Performance

Performance (secs/program) = \[
\begin{array}{c}
\text{Cycles} \\
\text{Insts}
\end{array} * \begin{array}{c}
\text{seconds} \\
\text{Cycles}
\end{array} * \begin{array}{c}
\text{Insts} \\
\text{Program}
\end{array}
\]

processor (μarch)

process technology
Iron Law of Performance

Performance
(secs/program) = \frac{\text{Cycles}}{\text{Insts}} \times \frac{\text{seconds}}{\text{Cycles}} \times \frac{\text{Insts}}{\text{Program}}

processor (\mu arch)

Process technology

ISA (arch)
Goal
Goal

- Measure RISC-V gcc's current code generation quality
Goal

- Measure RISC-V gcc's current code generation quality
- Given a **fixed** ISA...
Goal

- Measure RISC-V gcc's current code generation quality
- Given a **fixed** ISA...
  - what can the **compiler** do to improve performance?
Goal

- Measure RISC-V gcc's current code generation quality
- Given a fixed ISA...
  - what can the compiler do to improve performance?
  - what can the programmer do to improve performance?
Goal

- Measure RISC-V gcc's current code generation quality
- Given a fixed ISA...
  - what can the compiler do to improve performance?
  - what can the programmer do to improve performance?
  - what can the micro-architect do to improve performance?
Non-goals
Non-goals

- Lobby for more instructions (CISC or otherwise)
Non-goals

- Lobby for more instructions (CISC or otherwise)
  - instructions/cycle, cycle time, area/power costs, verification costs, time-to-market, compiler target-ability...
Non-goals

- Lobby for more instructions (CISC or otherwise)
  - instructions/cycle, cycle time, area/power costs, verification costs, time-to-market, compiler target-ability...
- make claims of relative ISA merits
Non-goals

- Lobby for more instructions (CISC or otherwise)
  - instructions/cycle, cycle time, area/power costs, verification costs, time-to-market, compiler target-ability...
- make claims of relative ISA merits
Non-goals

- Lobby for more instructions (CISC or otherwise)
  - instructions/cycle, cycle time, area/power costs, verification costs, time-to-market, compiler target-ability...

- make claims of relative ISA merits
  - none of this matters if application is cache missing or spinning on user input!
Non-goals

- Lobby for more instructions (CISC or otherwise)
  - instructions/cycle, cycle time, area/power costs, verification costs, time-to-market, compiler target-ability...
- make claims of relative ISA merits
  - none of this matters if application is cache missing or spinning on user input!
  - dynamic instruction count can be misleading...
Micro-ops

instructions (ISA)

micro-ops (μarch)

rep movs

ld ...
st ...
add...
CISC ISAs and Micro-ops

Performance (secs/program) = \( \frac{\text{Cycles}}{\text{Insts}} \times \frac{\text{seconds}}{\text{Cycles}} \times \frac{\text{Insts}}{\text{Program}} \)
CISC ISAs and Micro-ops

- **Intel x86**
  - `rep movs' (repeat move, aka, "the memcpy instruction")
    - repeatedly copy C words from address SI to address DI
  - **Sources**
    - 3 (implicit) register operands
    - EFLAGS register
  - **Side-effects**
    - C loads
    - C stores
    - writebacks to SI and DI registers

\[
\text{Performance (secs/program)} = \frac{\text{Cycles}}{\text{Insts}} \times \frac{\text{seconds}}{\text{Cycles}} \times \frac{\text{Insts}}{\text{Program}}
\]
CISC ISAs and Micro-ops

Performance (secs/program) = \[
\text{Cycles} \times \frac{\text{seconds}}{\text{Insts}} \times \frac{\text{Insts}}{\text{Cycles}} \times \frac{\text{Program}}{\text{Insts}}
\]

- **Intel x86**
  - `rep movs' (repeat move, aka, "the memcpy instruction")
    - repeatedly copy C words from address SI to address DI
  - **Sources**
    - 3 (implicit) register operands
    - EFLAGS register
  - **Side-effects**
    - C loads
    - C stores
    - writebacks to SI and DI registers

- **Micro-ops!**
  - x86 decoder generates RISC-like micro-ops to perform CISC instructions
  - micro-ops map well to processor pipelines
  - **Advantages:** fewer instructions/program, fewer dynamic instruction bytes
  - **Disadvantages:** complex! (ex: how do you do precise exceptions?)
**CISC ISAs and Micro-ops**

### Intel x86
- `rep movs` (repeat move, aka, "the memcpy instruction")
  - repeatedly copy C words from address SI to address DI
- **Sources**
  - 3 (implicit) register operands
  - EFLAGS register
- **Side-effects**
  - C loads
  - C stores
  - writebacks to SI and DI registers

### Micro-ops!
- x86 decoder generates RISC-like micro-ops to perform CISC instructions
- micro-ops map well to processor pipelines
- **Advantages:** fewer instructions/program, fewer dynamic instruction bytes
- **Disadvantages:** complex! (ex: how do you do precise exceptions?)

---

**Performance (secs/program)**

\[
\text{Performance} = \frac{\text{Cycles}}{\text{Insts}} \times \frac{\text{seconds}}{\text{Cycles}} \times \frac{\text{Insts}}{\text{Program}}
\]
instructions (ISA)

micro-ops (μarch)

Micro-op generation

rep movs

ld ... st ... add...

Macro-op Fusion

cmp

jne

bne
Macro-op Fusion
Macro-op Fusion

- RISC-V
  - register-register magnitude compare and branch instruction
  - a single 4-byte instruction
Macro-op Fusion

- **RISC-V**
  - register-register magnitude compare and branch instruction
  - a single 4-byte instruction

- **ARM, x86**
  - **compare** and **branch-on-outcome** are two instructions!
    - compare sets a condition flag
    - branch-on-condition-flag
Macro-op Fusion

- **RISC-V**
  - register-register magnitude compare and branch instruction
  - a single 4-byte instruction

- **ARM, x86**
  - **compare** and **branch-on-outcome** are two instructions!
    - compare sets a condition flag
    - branch-on-condition-flag

**Solution:** lie to your decoder!

tell it "cmp,bne" is a single 8-byte instruction
Micro-ops and Macro-op Fusion

**Micro-ops generation**
- rep movs
- ld ...
- st ...
- add...

**Macro-op Fusion**
- cmp
- jne
- bne
ISA Shootout!

- Compare 6 ISAs using SPECInt 2006
  - RISC-V
    - RV64G
    - RV64GC (with compressed ISA extension)
  - ARM
    - ARMv7 (32-bit)
    - ARMv8 (64-bit)
  - x86
    - ia32 (32-bit)
    - x86-64 (64-bit)

- Measurements
  - instruction counts (and micro-op counts for x86-64)
  - dynamic instruction bytes
Methodology: SPECint
Methodology: SPECint

- 12 benchmarks in CINT2006
  - 35 workloads written in C or C++ (--reference)
  - ~20 trillion instructions total
- workstation workloads
  - lots of data, lots of compute
  - little data generation
  - no idle periods
Methodology: SPECint

- 12 benchmarks in CINT2006
  - 35 workloads written in C or C++ (--reference)
  - ~20 trillion instructions total
  - *workstation* workloads
    - lots of data, lots of compute
    - little data generation
    - no idle periods

- RISC-V
  - requires libc *(riscv64-unknown-linux-gnu*)
  - run on Linux
  - [https://github.com/ccelio/Speckle](https://github.com/ccelio/Speckle)
    - useful for generating portable SPEC directories
Methodology: Compilation

- gcc 5.3.0 -static -O3
- lots of tricks to make SPEC go faster (I won't be using)
  - purpose is not drag-racing
Vector/SIMD
Vector/SIMD

- only want to compare **scalar** ISAs
- only want to compare scalar ISAs
- it's really hard to completely remove Intel's SSE from your binary
- only want to compare **scalar** ISAs
- it's really hard to completely remove Intel's SSE from your binary
- it's really hard to generate SSE for code you actually care about!
- only want to compare **scalar** ISAs
- it's really hard to completely remove Intel's SSE from your binary
- it's really hard to generate SSE for code you actually care about!
- `gcc -march=native -mtune=native -O3` is used
Methodology: Data Collection

- ARM, x86
  - run on native hardware (Cortex-A15, Cortex-A53, Intel Sandy Bridge Xeon)
  - *perf* to read hardware counters
  - use Intel's *Pin* tool to build a PC histogram generator for x86

- RISC-V
  - spike -g --disk=spec.bin bbl vmlinux
  - side-channel process snapshots rdinstret ("instructions retired")
  - "spike -g" captures a PC histogram
Dynamic Instructions
(Normalized to x86-64)

Total Dynamic Instructions

- Benchmarks
  - 400.perlbench
  - 401.bzip2
  - 403.gcc
  - 429.mcf
  - 445.gobmk
  - 456.hmmer
  - 458.sjeng
  - 462.libquantum
  - 464.h264ref
  - 471.omnetpp
  - 473.astar
  - 483.xalancbmk
  - Geomean

- Normalized Instructions
  - x86-64
  - ia32
Dynamic Instructions
(Normalized to x86-64)

Total Dynamic Instructions

<table>
<thead>
<tr>
<th>benchmarks</th>
<th>x86-64</th>
<th>ia32</th>
<th>ARMv7</th>
<th>ARMv8</th>
</tr>
</thead>
<tbody>
<tr>
<td>401.bzip2</td>
<td>1.01</td>
<td>1.02</td>
<td>1.03</td>
<td>1.04</td>
</tr>
<tr>
<td>403.gcc</td>
<td>0.99</td>
<td>1.01</td>
<td>1.02</td>
<td>1.03</td>
</tr>
<tr>
<td>429.mcf</td>
<td>1.00</td>
<td>1.01</td>
<td>1.02</td>
<td>1.03</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>0.98</td>
<td>1.00</td>
<td>1.01</td>
<td>1.02</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>0.97</td>
<td>1.00</td>
<td>1.01</td>
<td>1.02</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>0.96</td>
<td>1.00</td>
<td>1.01</td>
<td>1.02</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>0.95</td>
<td>1.00</td>
<td>1.01</td>
<td>1.02</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>0.94</td>
<td>1.00</td>
<td>1.01</td>
<td>1.02</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>0.93</td>
<td>1.00</td>
<td>1.01</td>
<td>1.02</td>
</tr>
<tr>
<td>473.astar</td>
<td>0.92</td>
<td>1.00</td>
<td>1.01</td>
<td>1.02</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>0.91</td>
<td>1.00</td>
<td>1.01</td>
<td>1.02</td>
</tr>
<tr>
<td>geomean</td>
<td>0.90</td>
<td>1.00</td>
<td>1.01</td>
<td>1.02</td>
</tr>
</tbody>
</table>
Dynamic Instructions
(Normalized to x86-64)
Dynamic Instructions
(Normalized to x86-64)

Total Dynamic Instructions

- memset
- memcpy
Dynamic Instructions
(Normalized to x86-64)

Total Dynamic Instructions

-(memset)
-(memcpy)
Summary: Dynamic Instructions

Geometric Mean

- x86-64 micro-ops
- x86-64
- ia32
- ARMv7
- ARMv8
- RV64G

Dynamic Instructions (normalized to x86-64)
- RV64G is 16% **more** instructions than x86-64
- RV64G is 16\% \textbf{more} instructions than x86-64
- RV64G is 4\% \textbf{fewer} instructions than ARMv7
- RV64G is 16% more instructions than x86-64
- RV64G is 4% fewer instructions than ARMv7
- RV64G is same number of x86-64 retired micro-ops
What about Dynamic Instruction Bytes?

- Let's also compare **RV64GC**
  - Compressed ISA Extension (RVC)
  - adds 2-byte instructions
  - assembler-aware only!
  - code generation is identical to RV64G

- use histograms from Pin and Spike + objdumps to compute bytes fetched for x86-64, **RV64GC**
RV64GC wins on 9 out of 12 benchmarks!
2 of those 3 use memset, memcpyp
Data Summary

- **Instruction Counts**
  - RV64G is 16% more instructions than x86-64
  - RV64G is 4% fewer instructions than ARMv7
  - RV64G is same number of x86-64 retired micro-ops

- **Dynamic Bytes**
  - RV64G is 23% more instruction bytes than x86-64
Instruction Counts
- RV64G is 16% more instructions than x86-64
- RV64G is 4% fewer instructions than ARMv7
- RV64G is same number of x86-64 retired micro-ops

Dynamic Bytes
- RV64G is 23% more instruction bytes than x86-64
- RV64GC is 28% fewer instructions bytes than ARMv7
Data Summary

- **Instruction Counts**
  - RV64G is 16% more instructions than x86-64
  - RV64G is 4% fewer instructions than ARMv7
  - RV64G is same number of x86-64 retired micro-ops

- **Dynamic Bytes**
  - RV64G is 23% more instruction bytes than x86-64
  - RV64GC is 28% fewer instructions bytes than ARMv7
  - RV64GC is 8% fewer instruction bytes than x86-64
Data Summary

- **Instruction Counts**
  - RV64G is 16% **more** instructions than x86-64
  - RV64G is 4% **fewer** instructions than ARMv7
  - RV64G is **same number** of x86-64 retired micro-ops

- **Dynamic Bytes**
  - RV64G is 23% **more** instruction **bytes** than x86-64
  - RV64GC is 28% **fewer** instructions **bytes** than ARMv7
  - RV64GC is 8% **fewer** instruction **bytes** than x86-64

- **Code density**
Data Summary

- Instruction Counts
  - RV64G is 16% more instructions than x86-64
  - RV64G is 4% fewer instructions than ARMv7
  - RV64G is same number of x86-64 retired micro-ops

- Dynamic Bytes
  - RV64G is 23% more instruction bytes than x86-64
  - RV64GC is 28% fewer instructions bytes than ARMv7
  - RV64GC is 8% fewer instruction bytes than x86-64

- Code density
  - x86-64 averages 3.7 bytes / instruction
Data Summary

- **Instruction Counts**
  - RV64G is 16% **more** instructions than x86-64
  - RV64G is 4% **fewer** instructions than ARMv7
  - RV64G is **same number** of x86-64 retired micro-ops

- **Dynamic Bytes**
  - RV64G is 23% **more** instruction **bytes** than x86-64
  - RV64GC is 28% **fewer** instructions **bytes** than ARMv7
  - RV64GC is 8% **fewer** instruction **bytes** than x86-64

- **Code density**
  - x86-64 averages **3.7 bytes / instruction**
  - RV64GC averages **3.0 bytes / instruction**
Why is RISC-V 16% more instructions?
The most common idioms: reading from arrays!!!
The most common idioms: reading from arrays!!!

```
RISC-V
# array[r_offset]
slli a5,a5,0x2
add  a5,s9,a5
lw   a5,0(a5)
```
The most common idioms: reading from arrays!!

RISC-V

# array[r_offset]
slli a5,a5,0x2
add a5,s9,a5
lw a5,0(a5)

x86-64

# array[r_offset]
mov 0x0(%r13,%rcx,4),%ecx
# c = mem[r13 + c*4 + 0x0]
The most common idioms: reading from arrays!!

**RISC-V**

`# array[r_offset]`

`slli a5,a5,0x2`

`add a5,s9,a5`

`lw a5,0(a5)`

**x86-64**

`# array[r_offset]`

`mov 0x0(%r13,%rcx,4),%ecx`

`# c = mem[r13 + c*4 + 0x0]`

**bzip2...**

```java
n = ((Int32)block[ptr[unHi]+d]) - med;
```
The most common idioms: reading from arrays!!!

**RISC-V**

```assembly
# array[r_offset]
slli a5,a5,0x2
add a5,s9,a5
lw a5,0(a5)
```

**x86-64**

```assembly
# array[r_offset]
imul 0x0(%r13,%rcx,4),%ecx
# c = mem[r13 + c*4 + 0x0]
```

bzip2...

```
n = ((Int32)block[ptr[unHi]+d]) - med;
```

```assembly
lw a4,0(t4)
addw a5,s3,a4
slli a5,a5,0x20
srli a5,a5,0x20
add a5,s0,a5
lbu a5,0(a5)
subw a5,a5,t3
```
The most common idioms:
reading from arrays!!

RISC-V

# array[r_offset]
slli a5,a5,0x2
add a5,s9,a5
lw a5,0(a5)

x86-64

# array[r_offset]
mov 0x0(%r13,%rcx,4),%ecx
# c = mem[r13 + c*4 + 0x0]

bzip2...
n = ((Int32)block[ptr[unHi]+d]) - med;

lw a4,0(t4)
addw a5,s3,a4
slli a5,a5,0x20
srli a5,a5,0x20
add a5,s0,a5
lbu a5,0(a5)
subw a5,a5,t3

mov (%r10),%edx
lea (%r15,%rdx,1),%eax
movzbl (%r14,%rax,1),%eax
sub %r9d,%eax
Solution to a better RISC-V?
Solution to a better RISC-V?

- Add an indexed load instruction to match x86!
  - \( \text{rd} \gets \text{mem}(\text{rs1} + \text{rs2}) \)
  - Or...
  - \( \text{rd} \gets \text{mem}(\text{rs1} + (\text{rs2} \ll \text{shamt})) \)
    - shift-amount is built into opcode (0,1,2,3 for lb,lh,lw,ld)
Solution to a better RISC-V?

- Add an indexed load instruction to match x86!
  - \( \text{rd} \leftarrow \text{mem}(\text{rs1} + \text{rs2}) \)
  - Or...
  - \( \text{rd} \leftarrow \text{mem}(\text{rs1} + (\text{rs2} \ll \text{shamt})) \)
    - shift-amount is built into opcode (0,1,2,3 for lb,lh,lw,ld)

- trivial addition to any RISC pipeline!
Solution to a better RISC-V?

- Add an indexed load instruction to match x86!
  - \( \text{rd} \leftarrow \text{mem}(\text{rs1} + \text{rs2}) \)
  - Or...
  - \( \text{rd} \leftarrow \text{mem}(\text{rs1} + (\text{rs2} \ll \text{shamt})) \)
    - shift-amount is built into opcode (0,1,2,3 for lb,lh,lw,ld)

- trivial addition to any RISC pipeline!
- don't you want 5% less instructions?!
Solution to a better RISC-V?

- Add an indexed load instruction to match x86!
  - \( \text{rd} \leftarrow \text{mem}(\text{rs1} + \text{rs2}) \)
  - Or...
  - \( \text{rd} \leftarrow \text{mem}(\text{rs1} + (\text{rs2} \ll \text{shamt})) \)
    - shift-amount is built into opcode (0,1,2,3 for lb,lh,lw,ld)
- trivial addition to any RISC pipeline!
- don't you want 5% less instructions?!
Solution to a better RISC-V?

- Add an indexed load instruction to match x86!
  - \( rd \leftarrow \text{mem}(rs1 + rs2) \)
  - Or...
  - \( rd \leftarrow \text{mem}(rs1 + (rs2 \ll \text{shamt})) \)
    - shift-amount is built into opcode (0,1,2,3 for lb,lh,lw,ld)

- trivial addition to any RISC pipeline!
- don't you want 5% less instructions?!
Solution to a better RISC-V?

- Add an indexed load instruction to match x86!
  - \( \text{rd} \leftarrow \text{mem}(\text{rs1} + \text{rs2}) \)
  - Or...
  - \( \text{rd} \leftarrow \text{mem}(\text{rs1} + (\text{rs2} \ll \text{shamt})) \)
    - shift-amount is built into opcode (0,1,2,3 for lb,lh,lw,ld)

- trivial addition to any RISC pipeline!
- don't you want 5% less instructions?!

Remember:
don't listen to Chris!
RVC+Macro-op Fusion To the Rescue!

```
add   a5, s9, a5
lw    a5, 0(a5)
```
RVC+Macro-op Fusion To the Rescue!

```
add    a5, s9, a5
lw     a5, 0(a5)
```

- add+load sequence is 8 bytes
add+load sequence is 8 bytes

Either...
  – make an indexed load instruction that is 4 bytes
RVC+Macro-op Fusion To the Rescue!

\[
\begin{align*}
\text{add} & \quad a5, s9, a5 \\
\text{lw} & \quad a5, 0(a5)
\end{align*}
\]

- add+load sequence is 8 bytes
- Either...
  - make an indexed load instruction that is \textbf{4 bytes}
- Or...
  - use RVC and get a two 2-byte instruction sequence (4 bytes total)!
  - lie to the decoder and tell it it has indexed loads!
RVC+Macro-op Fusion To the Rescue!

add a5, s9, a5
lw a5, 0(a5)

- add+load sequence is 8 bytes
- Either...
  - make an indexed load instruction that is 4 bytes
- Or...
  - use RVC and get a two 2-byte instruction sequence (4 bytes total)!
  - lie to the decoder and tell it it has indexed loads!
    We get indexed loads!
    and we didn't even change the ISA!
Proposed Macro-op Fusion Pairs

- **Load Effective Address**

  ```
  // &(array[offset])
  slli rd, rs1, {1,2,3}
  add rd, rd, rs2
  ```

- **Indexed Load**

  ```
  // rd = array[offset]
  add rd, rs1, rs2
  ld rd, 0(rd)
  ```

- **Clear Upper Word**

  ```
  // rd = rs1 & 0xffffffff
  slli rd, rs1, 0x20
  srli rd, rd, 0x20
  ```
- fusion provides **5.4%** fewer "effective" instructions for RV64
- fusion provides 5.4% fewer "effective" instructions for RV64
- RV64GC is 8% fewer bytes than x86-64!
fusion provides **5.4%** fewer "effective" instructions for RV64

**RV64GC is 8% fewer bytes** than x86-64!
- fusion provides 5.4% fewer "effective" instructions for RV64
- RV64GC is 8% fewer bytes than x86-64!
- fusion provides **5.4%** fewer "effective" instructions for RV64
- RV64GC is **8% fewer bytes** than x86-64!
- RV64GC+fusion executes **4.2% fewer ops** than x86-64!
Dynamic Instructions (Normalized to x86-64)

Total Dynamic Instructions

- x86-64 micro-ops
- x86-64
- ia32
- ARMv7
- ARMv8
- RV64G
- RV64GC Macro-ops

benchmarks:
- 400.perlbench
- 401.bzip2
- 403.gcc
- 429.mcf
- 445.gobmk
- 456.hmmer
- 458.sjeng
- 462.libquantum
- 464.h264ref
- 471.omnetpp
- 473..astar
- 483.xalancbmk
- geomean
What about ARMv8?
What about ARMv8?

- don't know micro-op count
What about ARMv8?

- don't know micro-op count
- but let's guess!
What about ARMv8?

- don't know micro-op count
- but let's guess!
- complex memory instructions
  - requires 2 write ports
    - load increment address (ldia)
    - load pair (lp)
  - requires 3 write ports
    - load pair increment address (lpdia)
What about ARMv8?

- don't know micro-op count
- but let's guess!
- complex memory instructions
  - requires 2 write ports
    - load increment address (ldia)
    - load pair (lp)
  - requires 3 write ports
    - load pair increment address (lpdia)
- modify QEMU to measure frequency
  - assume each micro-op == a single write-back
What about ARMv8?

- don't know micro-op count
- but let's guess!
- complex memory instructions
  - requires 2 write ports
    - load increment address (ldia)
    - load pair (lp)
  - requires 3 write ports
    - load pair increment address (lpdia)
- modify QEMU to measure frequency
  - assume each micro-op == a single write-back
- adds 4% to the effective instruction count
RV64GC is 9% more instructions than ARMv8
RV64GC is **9% more instructions** than ARMv8

RV64GC is **18% fewer bytes** than ARMv8
RV64GC is 9% more instructions than ARMv8
RV64GC is 18% fewer bytes than ARMv8
RV64GC+fusion executes same number as ARMv8 micro-ops
RV64GC is **9% more instructions** than ARMv8

RV64GC is **18% fewer bytes** than ARMv8

RV64GC+fusion executes **same number** as ARMv8 micro-ops
RV64GC is **9% more instructions** than ARMv8

RV64GC is **18% fewer bytes** than ARMv8

RV64GC+fusion executes **same number** as ARMv8 micro-ops
Fusion to the Extreme: making Rocket even faster!

- Fusion isn't just for superscalar, out-of-order cores
Fusion to the Extreme: making Rocket even faster!

- Fusion isn't just for superscalar, out-of-order cores
- Rocket is Berkeley's RV64G single-issue, 5-stage in-order core
  - [github.com/ucb-bar/rocket](https://github.com/ucb-bar/rocket)
Fusion to the Extreme: making Rocket even faster!

- Fusion isn't just for superscalar, out-of-order cores
- Rocket is Berkeley's RV64G single-issue, 5-stage in-order core
  - github.com/ucb-bar/rocket
- Change Rocket to...
  - fetch 8 bytes, not 4 bytes
  - if macro-op fusion not possible, store extra 4-bytes
  - if macro-op fusion possible, fetch next 8 bytes
Fusion to the Extreme: making Rocket even faster!

- Fusion isn't just for superscalar, out-of-order cores
- Rocket is Berkeley's RV64G single-issue, 5-stage in-order core
  - [github.com/ucb-bar/rocket](https://github.com/ucb-bar/rocket)
- Change Rocket to...
  - fetch 8 bytes, not 4 bytes
  - if macro-op fusion not possible, store extra 4-bytes
  - if macro-op fusion possible, fetch next 8 bytes
- Search for the following macro-op sequences...
  - indexed loads (add, load), load address (slli, add)
  - clear upper bits (slli, srli)
  - global loads (auipc, lw), far jumps (auipc, jr)
  - 32-bit immediates (lui/addi), (lui/ld)
  - 2-registers+imm arithmetic (add rd, rs1, imm; add rd, rd, rs2)
  - post-increments loads and stores (integer loads require 2nd RF write port)
  - load-pair/store-pair (ld/ld, st/st)
  - and more ...

![Rocket Architecture Diagram](image-url)
Fusion to the Extreme: making Rocket even faster!

- Fusion isn't just for superscalar, out-of-order cores
- Rocket is Berkeley's RV64G single-issue, 5-stage in-order core
  - github.com/ucb-bar/rocket
- Change Rocket to...
  - fetch 8 bytes, not 4 bytes
  - if macro-op fusion not possible, store extra 4-bytes
  - if macro-op fusion possible, fetch next 8 bytes
- Search for the following macro-op sequences...
  - indexed loads (add, load), load address (slli, add)
  - clear upper bits (slli, srli)
  - global loads (auipc, lw), far jumps (auipc, jr)
  - 32-bit immediates (lui/addi), (lui/ld)
  - 2-registers+imm arithmetic (add rd, rs1, imm; add rd, rd, rs2)
  - post-increments loads and stores (integer loads require 2nd RF write port)
  - load-pair/store-pair (ld/ld, st/st)
  - and more ...
- Result...
  - remove >5% dynamic instructions from the pipeline!
Macro-op Fusion Summary

- dynamic fetch bytes is the same
- the pipeline control & datapath is the same
- less pipeline resources, less issue window slots, fewer register file reads and writes
- ISA stays very simple
- only idiots who measure "instruction counts" might notice something looks odd...
Conclusion: for the compiler

- memset, memcpy are important functions
- heuristics on register usage is very important
  - stack popping and pushing show up in function-heavy codes
- idioms should be kept together
- better generation is possible if the compiler knows fusion is available
Conclusion: for the programmer

- avoid "uint32" when indexing arrays
  - 64-bit ABIs often have "signed registers"
  - use size_t

- avoid multi-dimensional arrays
  - use extra arithmetic, not loads, to compute addresses

- profile your code!
  - you'd be amazed at what simple transformations can make a difference

```c
for (i=0; i < my_table->size; i++)
  my_table->data[i] = 0
```

versus

```c
int sz = my_table->size;
for (i = 0; i < sz; i++)
  ...
```
Conclusion: for the architect

- RVC is helpful for high-performance applications
  - no performance loss!
  - lowers dynamic bytes fetched (and icache pressure)
- Overfetching is cheap (and gives your cache a rest)
- Macro-op fusion can lower resource usage, decrease latency, improve performance!
- Not all solutions require ISA changes
RISC can be denser!
- **RV64GC** is 28% fewer instructions **bytes** than ARMv7
- **RV64GC** is 18% fewer instructions **bytes** than ARMv8
- **RV64GC** is 8% fewer instruction **bytes** than x86-64

RISC can be faster!

keep it simple!
- extra complexity is felt by EVERYBODY
- let the micro-architect decide
- use macro-op fusion to specialize the processor
- many proposed instructions can be emulated by RVC +fusion!
Future work?

- This is just the beginning...
  - gcc 6.1 :'(
  - SPECfp
  - new languages...
  - new benchmarks...
  - new run-times...

- What new idioms show up in your code?
Questions?
Funding Acknowledgements

- Research partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung.

- Approved for public release; distribution is unlimited. The content of this presentation does not necessarily reflect the position or the policy of the US government and no official endorsement should be inferred.

- Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.
Cumulative distribution function for the 100 most frequent RISC-V instructions of each of the 35 SPECInt workloads. Each line corresponds to one of the 35 SPECInt workloads. A (*) marker denotes the start of a new contiguous instruction sequence (that ends with a taken branch).