RISC-V8

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

Image Credit: Advanced Engine Research P32 T 4.0 Litre V8 - 650 HP
http://mulsannescorner.com/aerturboImp1v8.html

Standards

- Open Container Initiative – Linux Foundation
  - Cloud Storage, Networking, and *Compute?*
    - T10 – SCSI, SATA, Fibre Channel, iSCSI, …
    - Ethernet – IEEE 802.3, IETF Internet Protocol, …
    - C – ISO/IEC 9899:2011, IEEE 754 Floating Point, …
    - POSIX.1-2008 – IEEE Std 1003.1™-2008, …

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

	RISC-V	x86-64
Design	RISC	CISC
Architecture	Load/Store	Register Memory
Registers	31	16
Bit width	64/32	64/32
Immediate width	20/12	64/32
Instruction sizes	2,4	1,2,3,4,5,6,7,8,9,…
Extension	Sign Extend	Zero / Merge
Control flow	Link Register	Stack

The problem

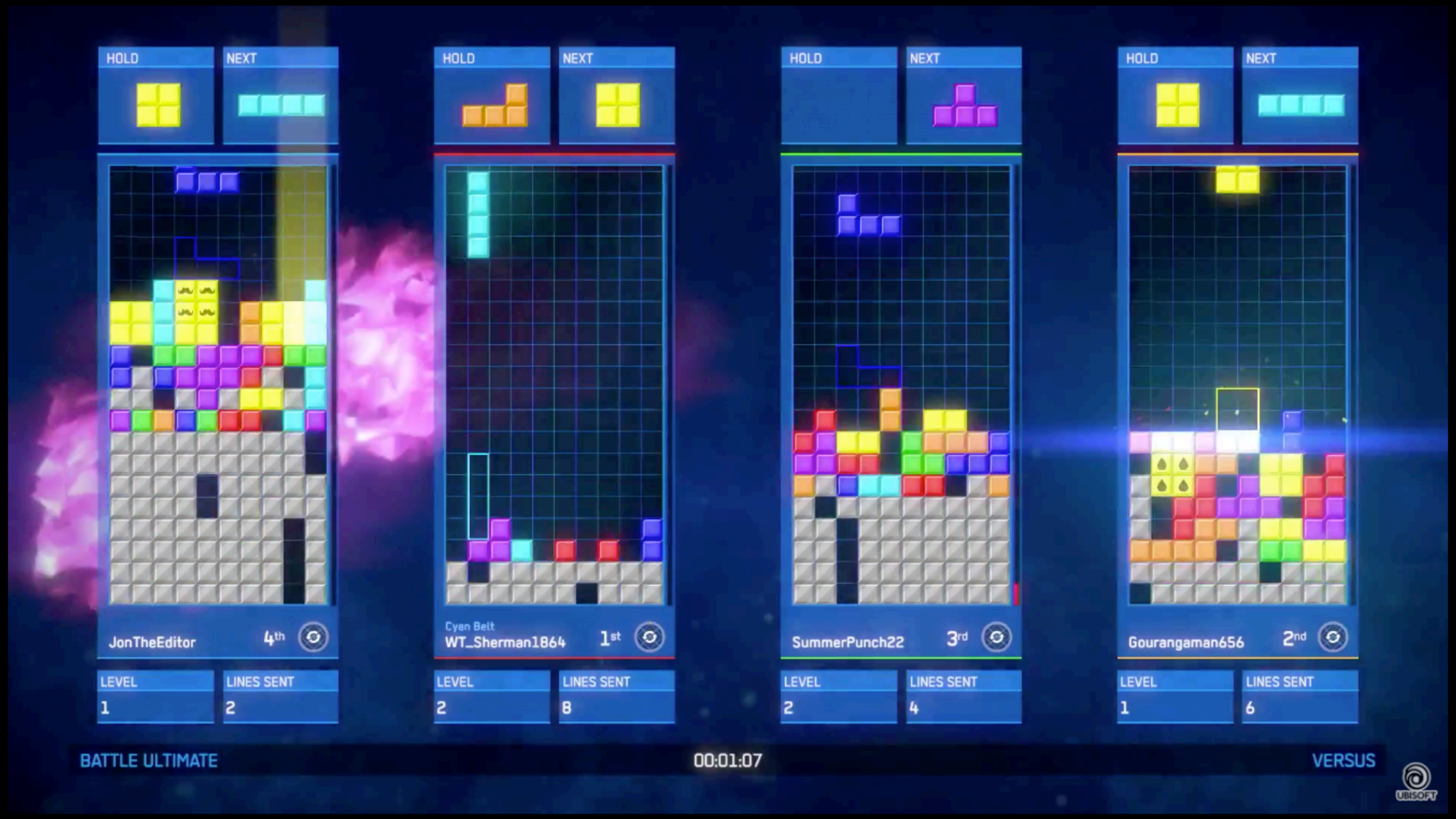
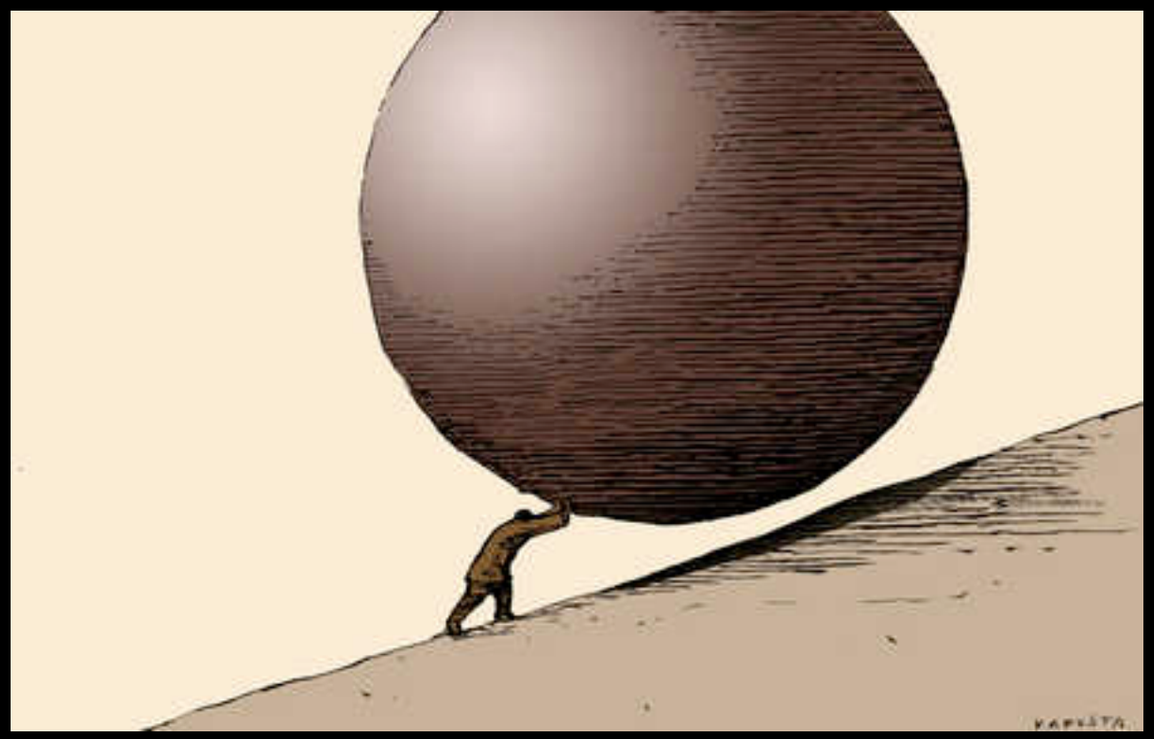


Image Credit: Ubisoft Tetris Ultimate®
https://www.ubisoft.com/en-au/game/tetris-ultimate/

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

ninety-ninety 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

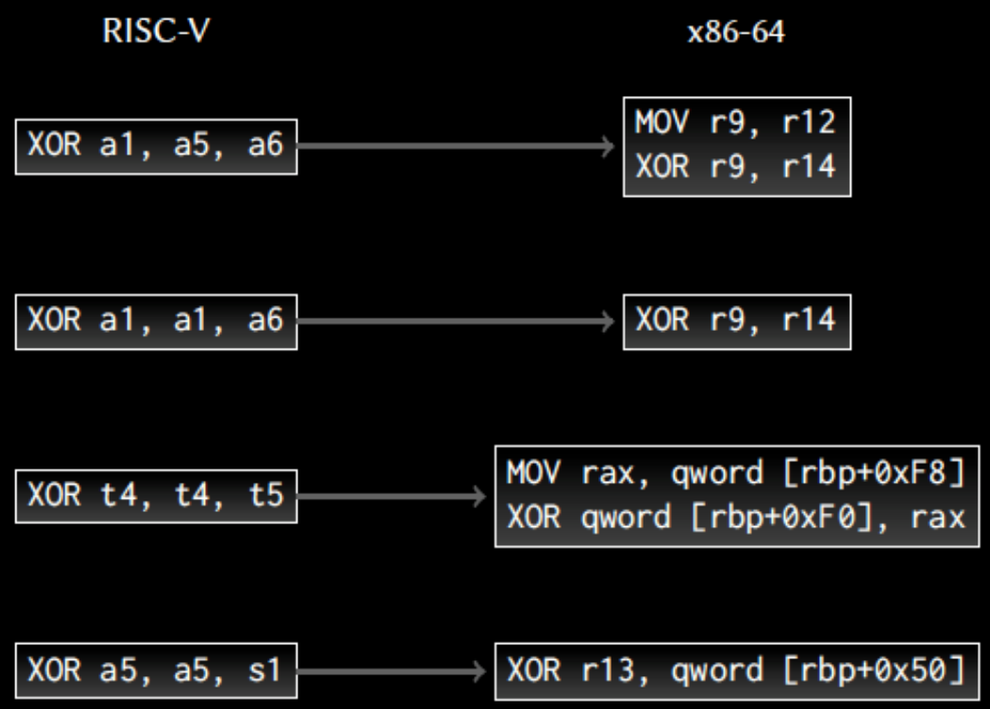
Image Credit: eMotimo http://emotimo.com/

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



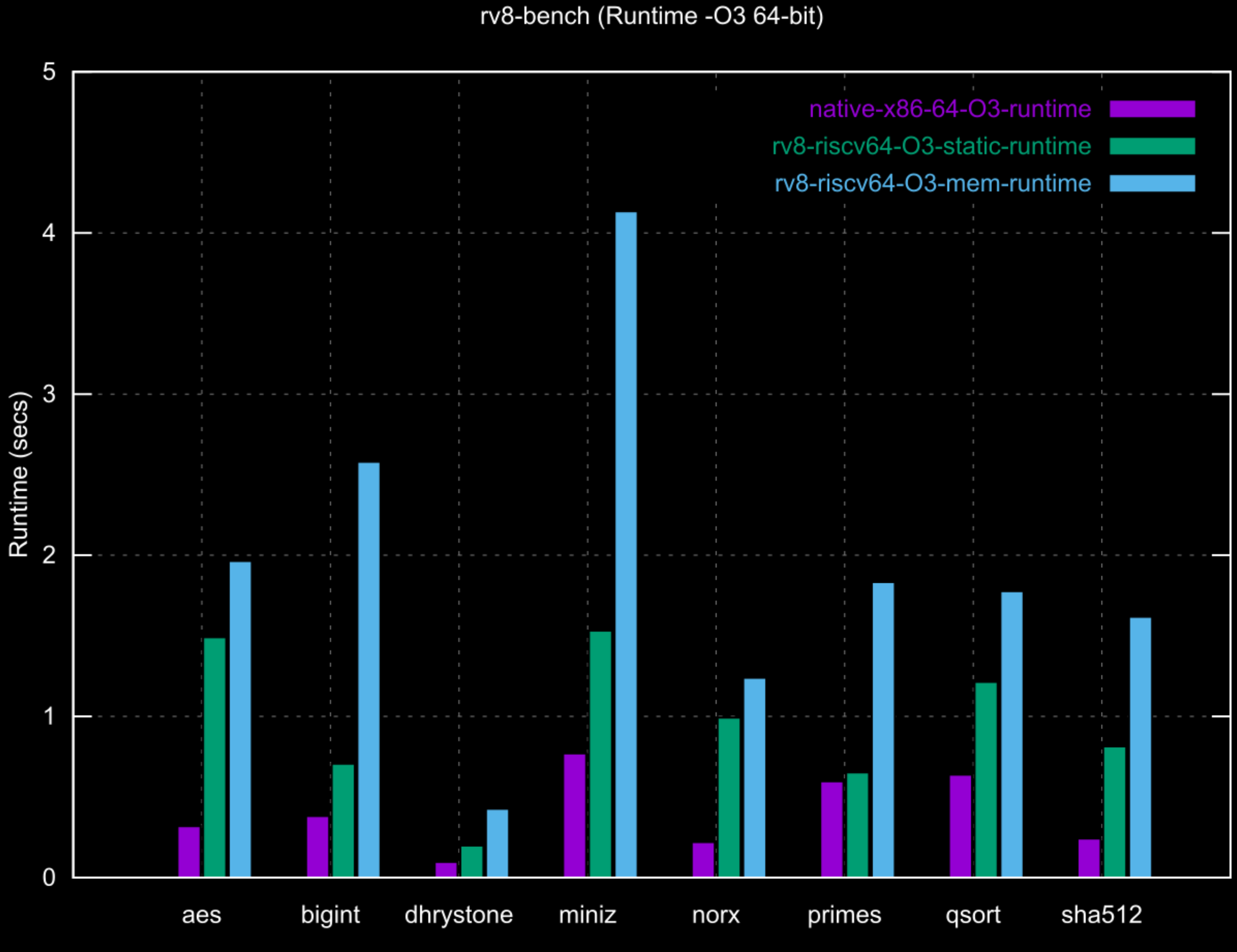
Register allocation

- 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

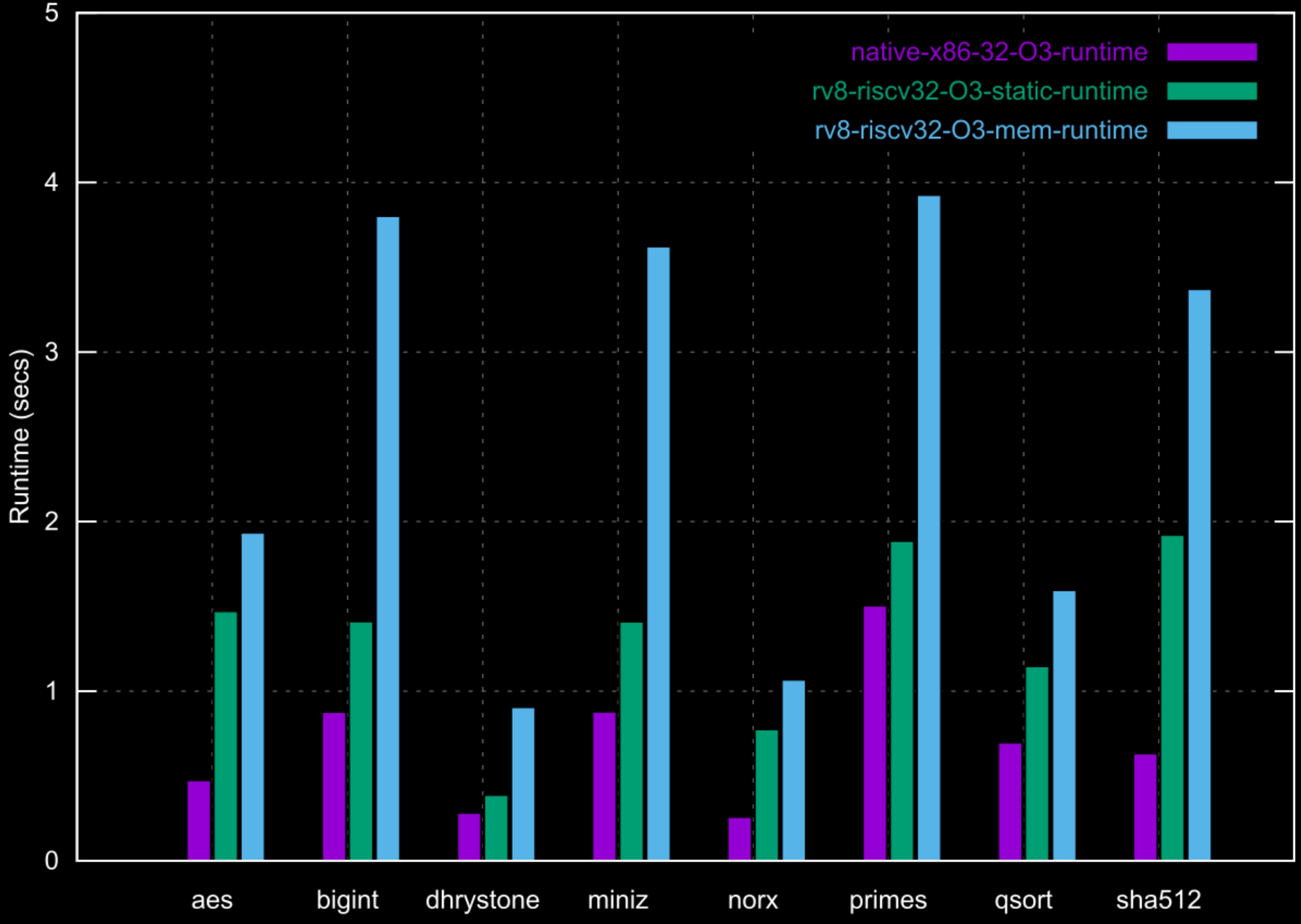
RISC-V	x86-64	Spill slot
zero		
ra	rdx	[rbp + 16]
sp	rbx	[rbp + 24]
gp		[rbp + 32]
tp		[rbp + 40]
t0	rsi	[rbp + 48]
t1	rdi	[rbp + 56]
t2		[rbp + 64]
s0		[rbp + 72]
s1		[rbp + 80]
a0	r8	[rbp + 88]
a1	r9	[rbp + 96]
a2	r10	[rbp + 104]
a3	r11	[rbp + 112]
a4	r12	[rbp + 120]
a5	r13	[rbp + 128]
a6	r14	[rbp + 136]
a7	r15	[rbp + 144]
s2		[rbp + 152]
s3		[rbp + 160]
s4		[rbp + 168]
s5		[rbp + 176]
s6		[rbp + 184]
s7		[rbp + 192]
s8		[rbp + 200]
s9		[rbp + 208]
s10		[rbp + 216]
s11		[rbp + 224]
t3		[rbp + 232]
t4		[rbp + 240]
t5		[rbp + 248]
t6		[rbp + 256]

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

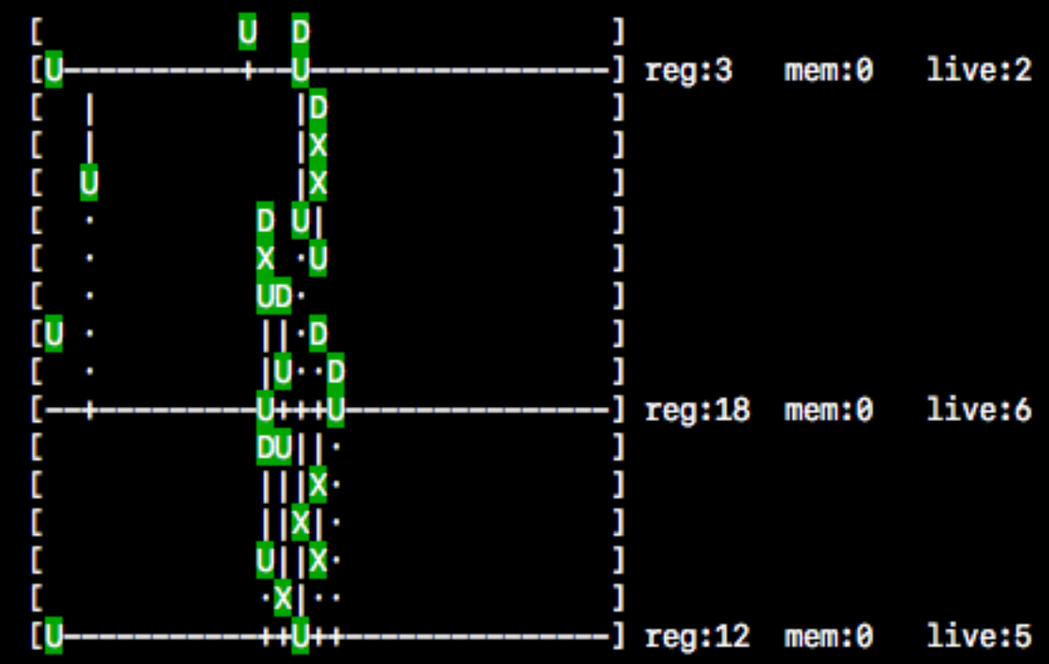

rv8-bench (Runtime -O3 64-bit)
native-x86-64-O3-runtime
rv8-riscv64-O3-static-runtime
rv8-riscv64-O3-mem-runtime
Runtime (secs)
0
1
2
3
4
5
aes
bigint
dhrystone
miniz
norx
primes
qsort
sha512

rv8-bench (Runtime -O3 32-bit)



Register allocation

```
jit-regalloc 0x00000000000127fa-0x000000000001281c

    0x00000000000127fa    lbu         a4, 0(a1)
    0x00000000000127fe    beq         a4, zero, pc + 48
    0x0000000000012800    lui         a5, 4096
    0x0000000000012802    addi        a5, a5, -1648
    0x0000000000012806    add         a5, a5, sp
    0x0000000000012808    slli        a2, a4, 2
    0x000000000001280c    add         a2, a2, a5
    0x000000000001280e    lw          a3, -1152(a2)
    0x0000000000012812    addi        a5, zero, 0
    0x0000000000012814    addi        a6, a3, 1
    0x0000000000012818    sw          a6, -1152(a2)
    0x000000000001281c    andi        a2, a3, 1
    0x0000000000012820    slli        a5, a5, 1
    0x0000000000012822    addi        a4, a4, -1
    0x0000000000012824    or          a5, a5, a2
    0x0000000000012826    srli        a3, a3, 1
    0x0000000000012828    bne         a4, zero, pc - 12
```

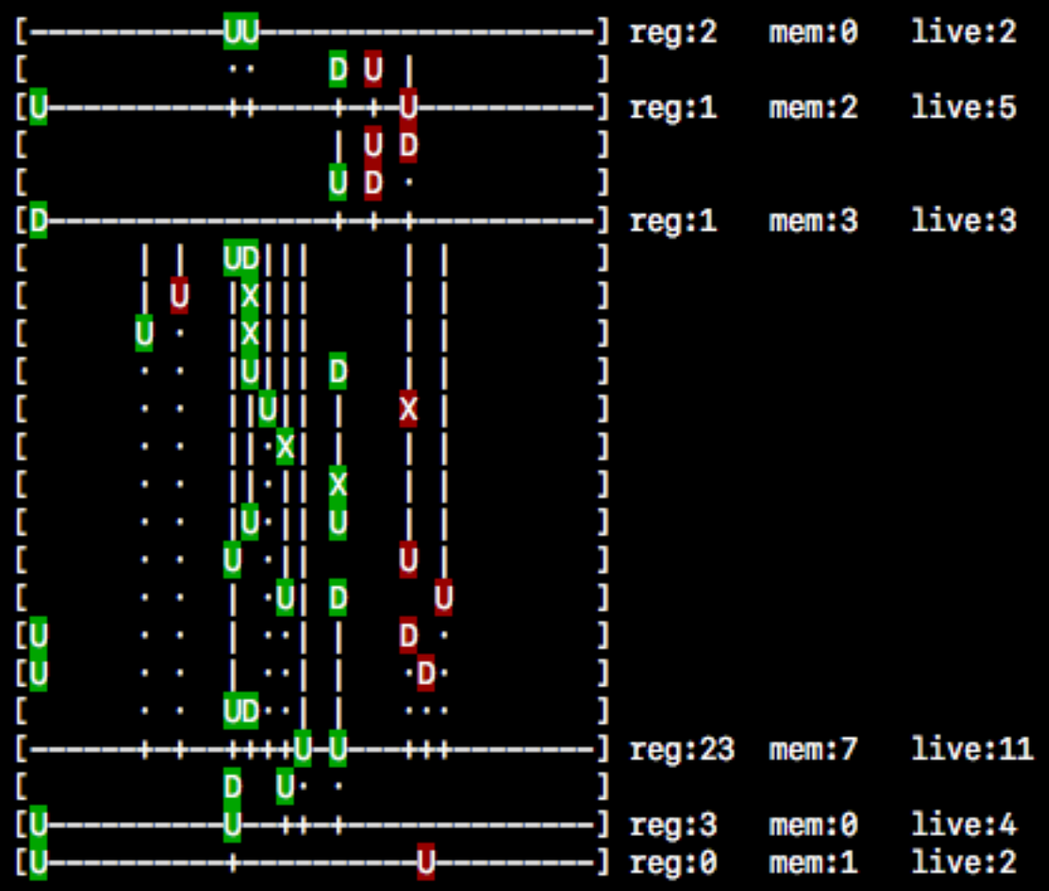


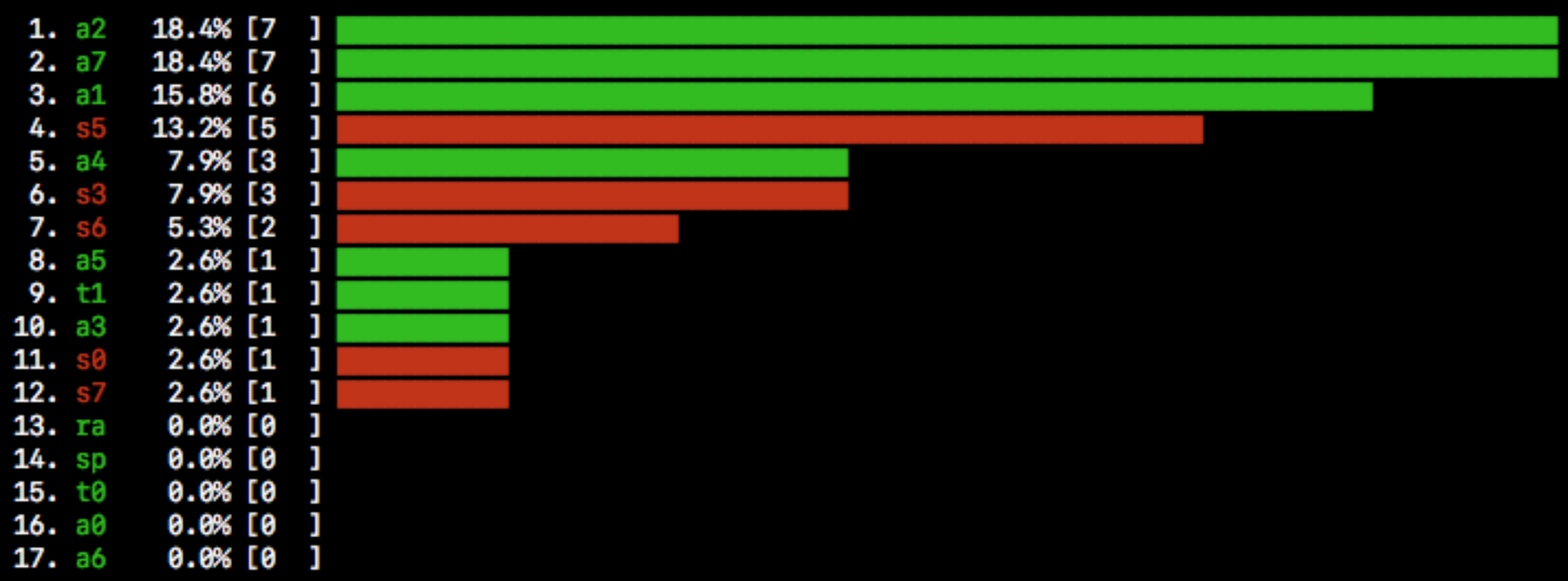

```
 1. a5   26.9% [7  ]
 2. a2   23.1% [6  ]
 3. a4   19.2% [5  ]
 4. a3   15.4% [4  ]
 5. a6    7.7% [2  ]
 6. a1    3.8% [1  ]
 7. sp    3.8% [1  ]
 8. ra    0.0% [0  ]
 9. t0    0.0% [0  ]
10. t1    0.0% [0  ]
11. a0    0.0% [0  ]
12. a7    0.0% [0  ]
```

Register allocation

```
jit-regalloc 0x0000000000012e44-0x0000000000012e44

    0x0000000000012e44    lbu         a1, 0(a4)                 [          D  U                ]
    0x0000000000012e48    beq         a1, zero, pc - 96         [U---------U--+----------------] reg:3   mem:0   live:2
    0x0000000000012e4a    beq         s6, zero, pc + 54         [U---------+---------U--------] reg:0   mem:1   live:2
    0x0000000000012e80    beq         a1, a2, pc + 1322         [----------UU--------+---------] reg:2   mem:0   live:3
    0x00000000000133aa    addi        s5, s5, 1                 [ |        ..       X         ]
    0x00000000000133ac    addi        s6, zero, 0               [U|        ..       |D        ]
    0x00000000000133ae    bne         s5, ra, pc - 1398         [-U---------++--------U+--------] reg:1   mem:4   live:5
    0x0000000000012e38    addi        a4, a4, 1                 [ .        |  X|     . |        ]
    0x0000000000012e3a    sub         a7, a4, s7                [ .        |  U| D   . U        ]
    0x0000000000012e3e    addi        a2, a1, 0                 [ .        UD .| |   . .        ]
    0x0000000000012e40    bgeu        a7, a5, pc + 154          [-+---------++-+U-U---+-+--------] reg:8   mem:1   live:8

 1. a1    22.2% [4  ]
 2. a4    16.7% [3  ]
 3. s6    11.1% [2  ]
 4. a2    11.1% [2  ]
 5. s5    11.1% [2  ]
 6. a7    11.1% [2  ]
 7. a5     5.6% [1  ]
 8. ra     5.6% [1  ]
 9. s7     5.6% [1  ]
10. sp     0.0% [0  ]
11. t0     0.0% [0  ]
12. t1     0.0% [0  ]
13. a0     0.0% [0  ]
14. a3     0.0% [0  ]
15. a6     0.0% [0  ]
```

```
jit-regalloc 0x0000000000012e80-0x0000000000012e80

    0x0000000000012e80    beq         a1, a2, pc + 1322
    0x0000000000012e84    addi        a7, s3, 1
    0x0000000000012e88    beq         s5, zero, pc + 1356
    0x0000000000133d4     addi        s5, s3, 0
    0x0000000000133d6     addi        s3, a7, 0
    0x0000000000133d8     jal         zero, pc - 1318
    0x0000000000012eb2    slli        a2, a1, 1
    0x0000000000012eb6    add         a2, a2, s0
    0x0000000000012eb8    add         a2, a2, t1
    0x0000000000012eba    lhu         a7, 1554(a2)
    0x0000000000012ebe    add         s5, s5, a3
    0x0000000000012ec0    addi        a4, a4, 1
    0x0000000000012ec2    addi        a7, a7, 1
    0x0000000000012ec4    sh          a7, 1554(a2)
    0x0000000000012ec8    sb          a1, 1472(s5)
    0x0000000000012ecc    sub         a7, a4, s7
    0x0000000000012ed0    addi        s5, zero, 0
    0x0000000000012ed2    addi        s6, zero, 0
    0x0000000000012ed4    addi        a2, a1, 0
    0x0000000000012ed6    bltu        a7, a5, pc - 146
    0x0000000000012e44    lbu         a1, 0(a4)
    0x0000000000012e48    beq         a1, zero, pc - 96
    0x0000000000012e4a    beq         s6, zero, pc + 54
```





Translator temporaries

x86-64	Translator register purpose
rbp	pointer to the register spill area and the L1 translation cache
rsp	pointer to the host stack for procedure calls
rax	general purpose translator temporary register
rcx	general purpose translator temporary register

- 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

Translator temporaries

- 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
# 0x0000000000017e26    lw            s2, 48(s2)
mov eax, dword [rbp+0x4C]
add eax, r8
movsxd eax, dword [eax+0x30]
mov dword [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

```
1  unsigned bswap(unsigned p)
2  {
3      return  ((p >> 24) & 0x000000ff) |
4              ((p << 8 ) & 0x00ff0000) |
5              ((p >> 8 ) & 0x0000ff00) |
6              ((p << 24) & 0xff000000);
7  }
```

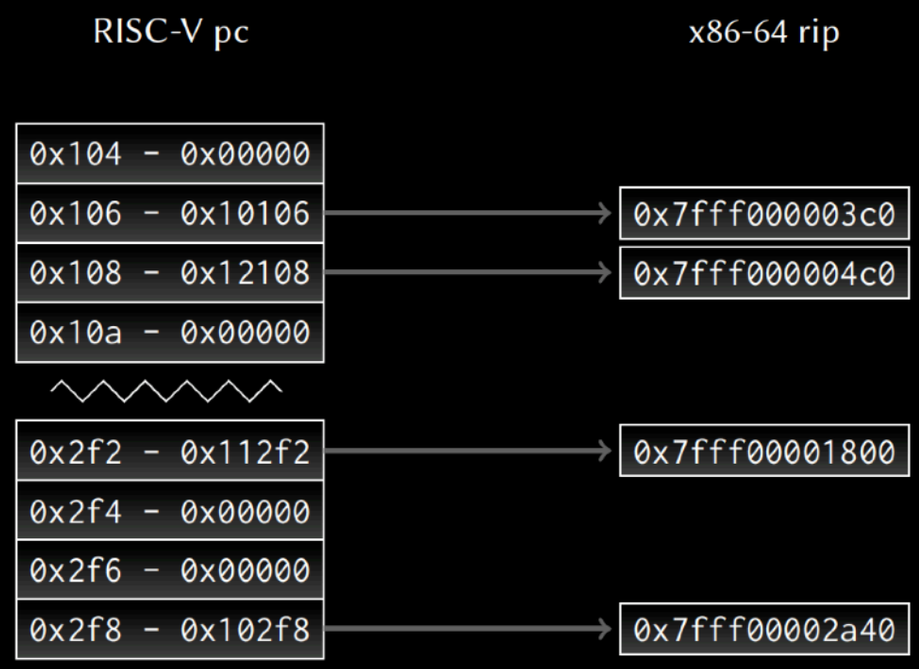
```
1  bswap(unsigned int):
2    slliw a3,a0,24
3    srliw a5,a0,24
4    slliw a4,a0,8
5    or a5,a5,a3
6    li a3,16711680
7    and a4,a4,a3
8    or a5,a5,a4
9    li a4,65536
10   addi a4,a4,-256
11   srliw a0,a0,8
12   and a0,a0,a4
13   or a0,a5,a0
14   sext.w a0,a0
15   ret
```

Sign extension

- `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)

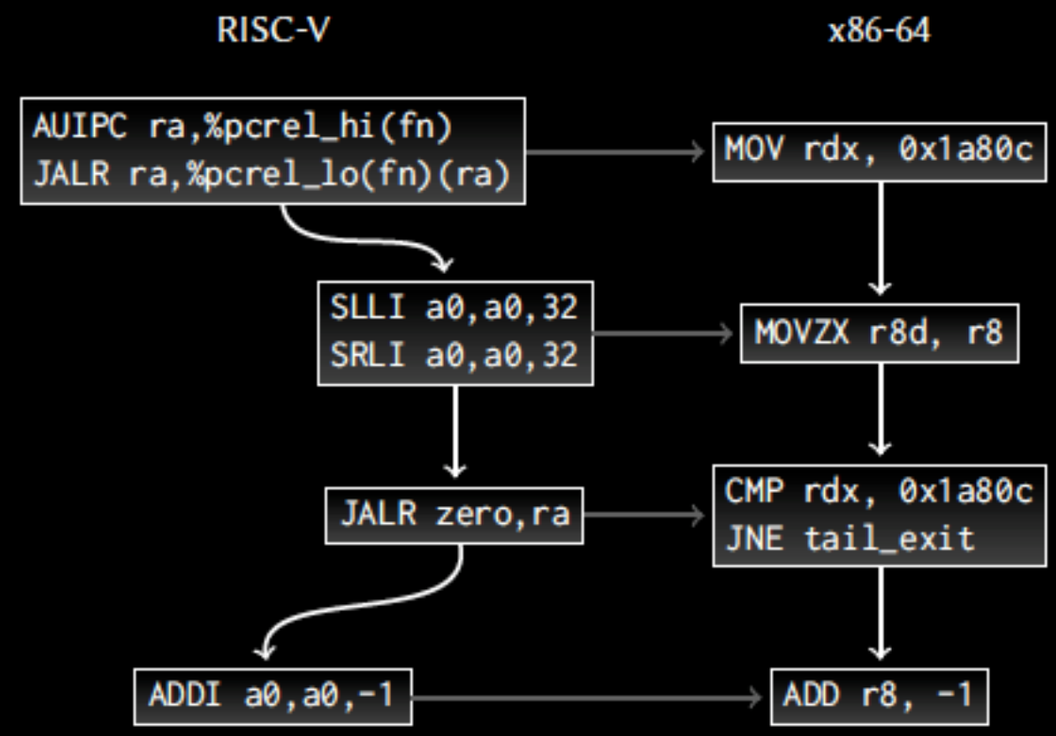
Indirect branches

- 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

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

- 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

Image Credit: Advanced Engine Research P32 T 4.0 Litre V8 - 650 HP
http://mulsannescorner.com/aerturboImp1v8.html

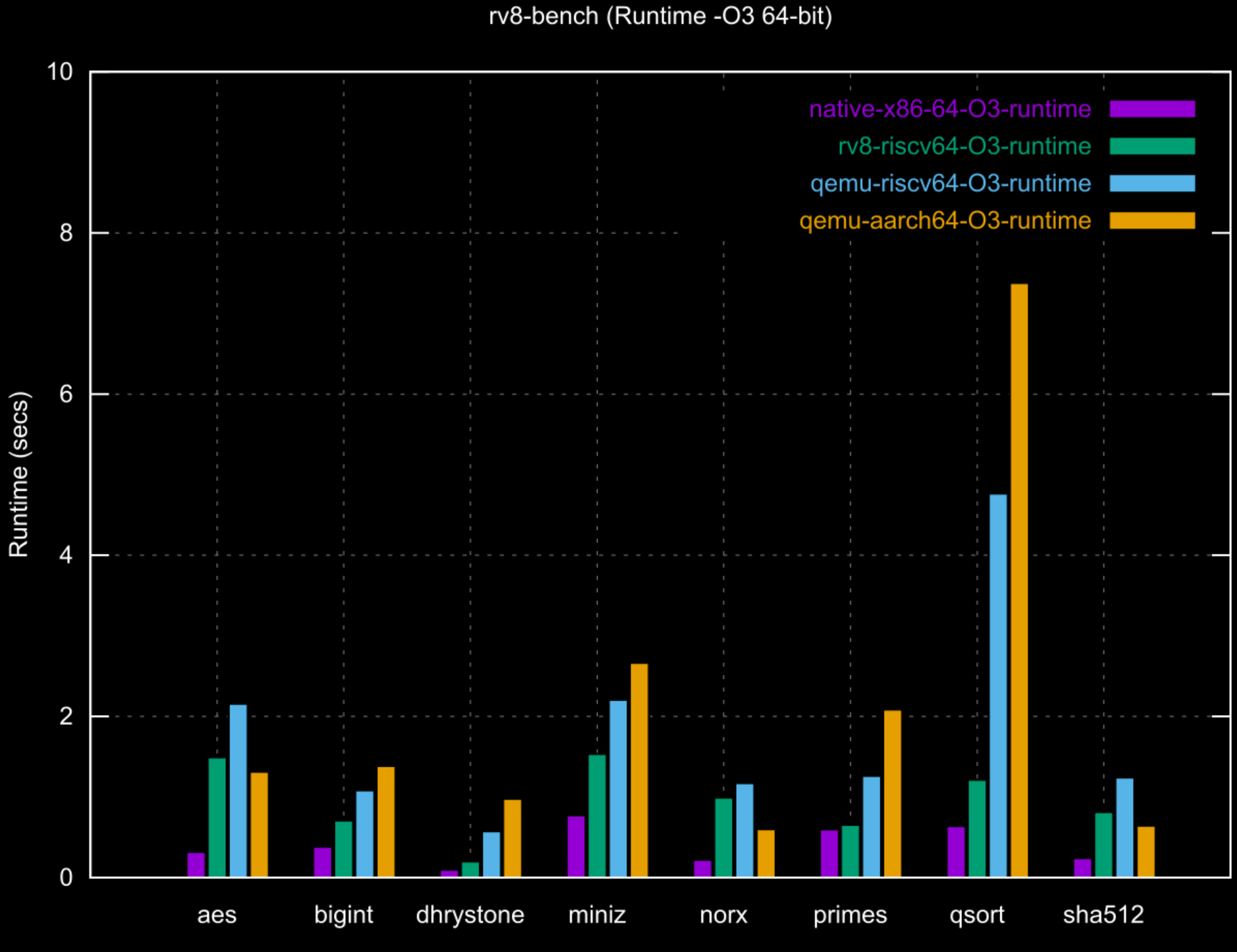
Image Credit: 2013 Chevrolet Equinox 3.0 Litre V6 - 182 HP
http://www.motortrend.com/cars/chevrolet/equinox/2013/

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

Program	Category	Description
aes	crypto	encrypt, decrypt and compare 30MiB of data
bigint	numeric	compute 23 ^ 111121 and count base 10 digits
dhrystone	synthetic	well known synthetic integer workload
miniz	compression	compress, decompress and compare 8MiB of data
norx	crypto	encrypt, decrypt and compare 30MiB of data
primes	numeric	calculate largest prime number below 33333333
qsort	sorting	sort array containing 50 million items
sha512	digest	calculate SHA-512 hash of 64MiB of data


rv8-bench (Runtime -O3 64-bit)
Runtime (secs)
10
8
6
4
2
0
native-x86-64-O3-runtime
rv8-riscv64-O3-runtime
qemu-riscv64-O3-runtime
qemu-aarch64-O3-runtime
aes
bigint
dhrystone
miniz
norx
primes
qsort
sha512

Ratios (-O3, 64-bit)

Program	qemu-aarch64	qemu-riscv64	rv8-riscv64	native-x86-64
aes	4.12	6.76	4.68	1.00
bigint	3.62	2.83	1.85	1.00
dhrystone	9.96	5.87	2.03	1.00
miniz	3.46	2.86	1.99	1.00
norx	2.73	5.33	4.51	1.00
primes	3.49	2.11	1.09	1.00
qsort	11.55	7.46	1.90	1.00
sha512	2.66	5.13	3.36	1.00
(Geomean)	4.44	4.39	2.40	1.00

rv8-bench (Runtime -O3 32-bit)

Runtime (secs)

10
8
6
4
2
0

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

aes
bigint
dhrystone
miniz
norx
primes
qsort
sha512

Ratios (-O3, 32-bit)

Program	qemu-arm32	qemu-riscv32	rv8-riscv32	native-x86-32
aes	3.56	3.97	3.09	1.00
bigint	3.39	1.56	1.61	1.00
dhrystone	4.13	3.91	1.37	1.00
miniz	3.40	2.47	1.60	1.00
norx	2.98	2.96	3.00	1.00
primes	2.79	1.55	1.25	1.00
qsort	12.04	6.54	1.65	1.00
sha512	6.04	4.60	3.04	1.00
(Geomean)	4.23	3.09	1.95	1.00

rv8-bench (MIPS -O3 64-bit)

native-x86-64-O3-mips
rv8-riscv64-O3-mips
qemu-riscv64-O3-mips

MIPS (Millions of Instructions Per Second)

14000
12000
10000
8000
6000
4000
2000
0

aes
bigint
dhrystone
miniz
norx
primes
qsort
sha512

MIPs (-O3, 64-bit)

Program	qemu-riscv64	rv8-riscv64	native-x86-64
aes	2414	3489	11035
bigint	3738	5720	10557
dhrystone	1843	5327	8369
miniz	2625	3778	5530
norx	2223	2628	9112
primes	2438	4712	6100
qsort	644	2527	5780
sha512	2982	4550	12177
(Geomean)	2149	3932	8232

rv8-bench (MIPS -O3 32-bit)

MIPS (Millions of Instructions Per Second)

native-x86-32-O3-mips
rv8-riscv32-O3-mips
qemu-riscv32-O3-mips

0
2000
4000
6000
8000
10000
12000
14000

aes
bigint
dhrystone
miniz
norx
primes
qsort
sha512

MIPs (-O3, 32-bit)

Program	qemu-riscv32	rv8-riscv32	native-x86-32
aes	2442	3137	9634
bigint	3964	3837	9780
dhrystone	1998	5696	3747
miniz	2195	3384	4988
norx	2824	2791	9146
primes	3039	3773	6368
qsort	671	2667	6259
sha512	2773	4200	11074
(Geomean)	2259	3586	7186

rv8.io/bench

- 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

rv8-bench (Retired Operations -O3 64-bit)

native-x86-64-O3-instret
native-x86-64-O3-uops-executed
native-x86-64-O3-uops-issued
native-x86-64-O3-uops-retired-all
native-x86-64-O3-uops-retired-slots
rv-sim-riscv64-O3-instret

Retired Operations

1x10^10
8x10^9
6x10^9
4x10^9
2x10^9
0

aes
bigint
dhrystone
miniz
norx
primes
qsort
sha512

rv8-bench (Retired Operations -O3 32-bit)

Retired Operations

2x10^10
1.5x10^10
1x10^10
5x10^9
0

native-x86-32-O3-instret
native-x86-32-O3-uops-executed
native-x86-32-O3-uops-issued
native-x86-32-O3-uops-retired-all
native-x86-32-O3-uops-retired-slots
rv-sim-riscv32-O3-instret

aes
bigint
dhrystone
miniz
norx
primes
qsort
sha512

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