rv8: a high performance RISC-V to x86 binary translator

Michael Clark
The rv8 contributors
michaeljclark@mac.com

Bruce Hoult
The rv8 contributors
bruce@hoult.org

ABSTRACT

Dynamic binary translation has a history of being used to ease transitions between CPU architectures[7], including micro-architectures. Modern x86 CPUs, while maintaining binary compatibility with their legacy CISC instruction set, have internal micro-architectures that resemble RISC. High performance x86 micro-architectures have long used a CISC decoder front-end to crack complex instructions into smaller micro-operations. Recently macro-op fusion [17][6] has been used to combine several instructions into one micro-op. Both techniques change the shape of the ISA to match the internal μop micro-architecture. Well-known binary translators also use micro-op internal representations to provide an indirection between the source and target ISAs as this makes the addition of new instruction sets much easier.

We present rv8, a high performance RISC-V simulation suite containing a RISC-V JIT (Just In Time) translation engine specifically targeting the x86-64 instruction set. Achieving the best possible mapping from a given source to target ISA pair requires a purpose designed internal representation. This paper explores a simple and fast translation from RISC-V to x86-64 that exploits knowledge of the geometries of the source and target ISAs, ABIs, and current x86-64 micro-architectures with a goal of producing a near optimal mapping from the 31 register source ISA to the 16 register target ISA. Techniques are explored that exploit the CISC encoding to increase instruction density, coalescing micro-ops into CISC instructions with the goal of generating the minimum number of micro-ops at the target micro-architectural level.

CCS CONCEPTS

• **Computer systems organization** → **Reduced instruction set computing**; **Complex instruction set computing**;

KEYWORDS

dynamic binary translation, RISC, CISC

Reference Format:
Michael Clark and Bruce Hoult. 2017. rv8: a high performance RISC-V to x86 binary translator. *First Workshop on Computer Architecture Research with RISC-V , Boston, MA, USA, October 2017 (CARRV 2017),* 7 pages.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CARRV 2017, October 2017, Boston, MA, USA
© 2017 Copyright held by the owner/author(s).

1 INTRODUCTION

RISC-V [20] is a modern, elegant and extensible open source instruction set architecture originally designed for computer architecture research. Similar to the Linux kernel two decades earlier in operating systems, RISC-V provides a standard and open base ISA and competitive reference hardware designs that anyone can freely use as-is or as a base for further innovation. This makes RISC-V uniquely suitable for the next phase of development in microprocessor hardware architectures much as the Linux kernel serves as the software foundation for embedded, mobile and server computing on a huge variety of CPU architectures and with a proliferation of operating systems such as Android, Tizen, Red Hat, Ubuntu, Fedora, Debian and many more.

Dynamic binary translation has frequently [7] [14] [2] [1] been used to provide binary compatibility for applications targeting legacy architectures during transitions to new architectures, however it can also be used to enable compatibility for binaries from newer architectures to allow for their execution on legacy hardware. Given x86 is the dominant architecture in cloud computing environments, dynamic binary translation provides a convenient means to enable RISC-V binary compatibility on existing hardware.

For binary translation to be acceptable as a mechanism to run RISC-V application images on legacy x86 hardware in the cloud, the performance must be similar to that of native code and there must be compelling advantages beyond performance, such as increased security [23]. While RISC-V is designed as an instruction set for hardware its features also make it an excellent platform abstraction for high performance virtualization. With this in mind, the goal of rv8 is to provide a binary translation platform that is able to achieve near native performance to enable secure RISC-V virtual machines in cloud computing environments.

Given Intel's and AMD's access to the latest process nodes[3], 4+ GHz clock speeds[22], superscalar execution, several dozen cores[15] and hundreds of GB of memory in a server, a near native speed RISC-V binary translator is likely to be the fastest RISC-V implementation and most practical build environment for things such as operating system distributions for some years to come.

2 PRINCIPLES

The rv8 binary translation engine has been designed as a hybrid interpreter and JIT compiler. The translator first interprets code and profiles it for hot paths [12]. Hot paths are then JIT translated to native code. The translation engine maintains a call stack to allow runtime inlining of hot functions. A jump target cache is used to accelerate returns and indirect calls through function pointers. The translator supports mixed binary translation and interpretation to handle instructions that do not yet have native translations. Currently, RISC-V 'IM' code is translated while 'AFD' is interpreted. The rv8 translator also supports RVC compressed code [19].

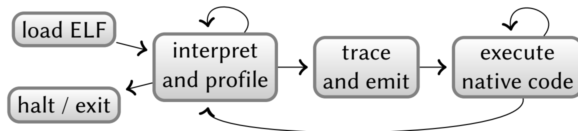

Figure 1: Principle of operation

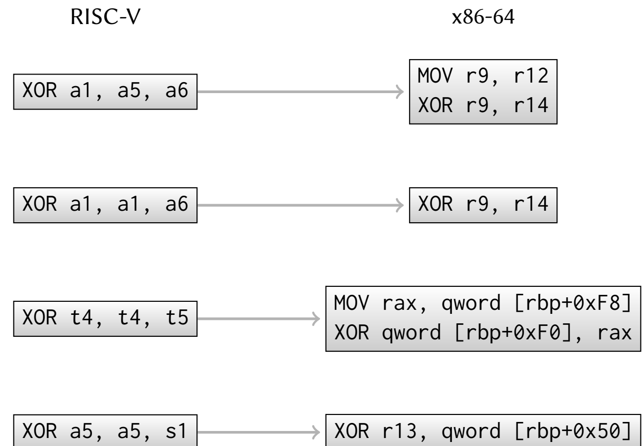

Figure 2: Complex memory operands

3 OPTIMISATIONS

The rv8 binary translator performs JIT translation of RISC-V code to x86-64 code. This is a challenging problem because the shapes of the instruction set architectures differ in a number of ways. RISC-V is a 31 register load/store reduced instruction set architecture while x86-64 is a 16 register complex instruction set architecture where loads and stores can be embedded in most instructions using indirect indexed memory operands. The challenge for a high performance JIT translator is to achieve as dense as possible mapping from RISC-V to x86-64 micro-ops via the legacy CISC decoder.

3.1 Register allocation

If the emulated ISA has significantly fewer registers than the host ISA – for example emulating x86 on a typical RISC – then register allocation is trivial. Just permanently assign each emulated register to a host register. In other cases translators normally either store all emulated registers in a memory array, or run a register allocation algorithm similar to a conventional compiler, treating source code registers like high level language variables or intermediate language virtual registers. The former approach is easy to write and translates quickly, but the translated code runs slowly. The latter approach slows translation but the generated code can run quickly – except possibly at control flow joins (phi nodes); and a number of move instructions may be required to resolve different register allocations on different execution paths.

rv8 approaches the register set size problem by making the observation that register usage is not evenly distributed, but is heavily biased to the ABI function call argument registers, stack pointer, and function return address register. We statically allocate each RISC-V register to either one of twelve fixed host x86-64 registers or to memory. Dynamic execution profiles were captured from long running programs (such as the Linux kernel) to find the dominant register set. The dominant set appears to be relatively stable. Refer to Figure 1 for the static allocation currently used.

The ‘C’ compressed instruction extension choice of eight registers is based on the same observation, though in that case it is the static distribution of register usage that matters, not dynamic. The difference is small. If future compilers are taught to favour the RVC registers to get better code compression our translator speed will also benefit.

RISC-V registers that are not mapped to an x86-64 register are accessed indirectly using the `rbp` register. e.g. `qword [rbp+0xF8]` would be used to access the RISC-V `t4` register.

Frequently accessed registers that are not in the statically allocated set will reside in L1 cache so accesses to spilled registers are relatively fast (approximately 1 nanosecond or 3 cycles latency), nevertheless in cases where the compiler allocates several hot registers outside of the statically allocated register set, substantial performance can be lost.

Registers that are allocated in memory is one place where the translator can take advantage of x86’s complex memory operands, allowing it to embed references to spilled registers directly within instructions rather than issuing separate loads and stores. This helps the translator maintain instruction density which lowers instruction cache pressure and helps improve throughput.

It is too late for the translator to rearrange stack memory for optimal stack spills as memory accesses must be translated precisely. RISC-V programs are optimised for 31 registers so stack spills are much less frequent. A future version of the translator may use a dynamic register allocator with live range splitting or potentially single static assignment [10] [4] [13]. This is discussed in Section 5.

3.2 CISC vs RISC

rv8 makes use of CISC memory operands to access registers residing in the memory backed register spill area. The complex memory operands end up being cracked into micro-ops in the CISC pipeline, however their use helps increase instruction density, which increases performance due to better use of instruction cache. Figure 2 shows various embeddings for resident and spilled registers.

There are many combinations of instruction expansions depending on whether a register is in memory or whether there are two or three operands. A three operand RISC-V instruction is translated into a move and a destructive two operand x86-64 instruction. Temporary registers are used if both operands are memory backed.

Future versions of the translator may perform passes to coalesce loads, ALU ops and stores into ALU ops with memory operands. This is discussed in Section 5.

3.3 Translator registers

The rv8 translator needs to use several host registers to point to translator internal structures and for use as temporary registers during emulation of some instructions.

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]

Table 1: Static register allocation

- Store instructions require the use of two temporary registers if both register operands are in the register spill area.
- x86 variable offset shift instructions require the shift amount to be in the `cl` register.
- x86 instructions such as `imul` and `idiv` require the use of two temporary registers as they take implicit operands and clobber `rax` and `rdx`.

The registers `rax` and `rcx` were chosen over `rax` and `rdx` to optimise for variable offset shift instructions as these are more frequent and more latency sensitive than multiplies and divides. The translator uses `rbp` to point at the register spill area and `rsp` points at the host stack.

Four x86 host registers are reserved leaving twelve registers available for mapping directly to RISC-V registers. See Table 2 for details on translator reserved registers.

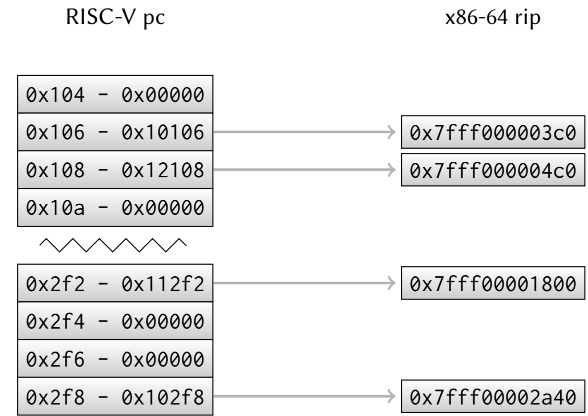

Figure 3: L1 translation cache

3.4 Prologue and epilogue

The interpreter is normal C code that can use any host register, and so always uses the memory array to store RISC-V registers. Therefore an assembly-language prologue loads RISC-V registers from memory to their assigned host registers when transitioning from the interpreter to native code and an epilogue saves RISC-V registers from host registers back to memory when transitioning from native code to the interpreter.

The emitted epilogue and prologue is relatively large so the JIT translator works to minimize the number of transistions from native code back to the translator. Nested loops run completely in native code without any returns to the interpreter.

3.5 Indirect branch acceleration

Indirect unconditional branches, such as calls through function pointers, cannot be statically translated as the target address of their host code is not known at the time of translation. rv8 maintains a hashtable mapping guest program addresses to translated host native code addresses. A full lookup is relatively slow because it requires saving caller-save registers and calling into C++ code.

To accelerate indirect calls through function pointers, a small assembly stub looks up the target address in a sparse 1024 entry direct mapped cache, and falls back to a slow call into the interpreter if the RISC-V target address is not found. The cache is then updated so that later indirect calls to the same RISC-V target address can be accelerated [18].

The direct mapped translation cache is indexed by `bits[10:1]` of the guest address as shown in Figure 3. Bit zero can be ignored because RISC-V instructions must start on a 2-byte boundary.

Some RISC-V indirect branches can be interpreted as direct branches which makes their translation more efficient. In particular the RISC-V `CALL` and `TAIL` macros for relative far unconditional branches. The `CALL` and `TAIL` macro expansions include the jump and link register instruction. The `CALL` macro expands to:

```
1: auipc ra,   %pcrel_hi(sym)
   jalr  ra,   %pcrel_lo(1b)(ra)
```

The jump and link register instruction would normally be considered an indirect branch, however the relative immediate value

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

Table 2: Translator reserved registers

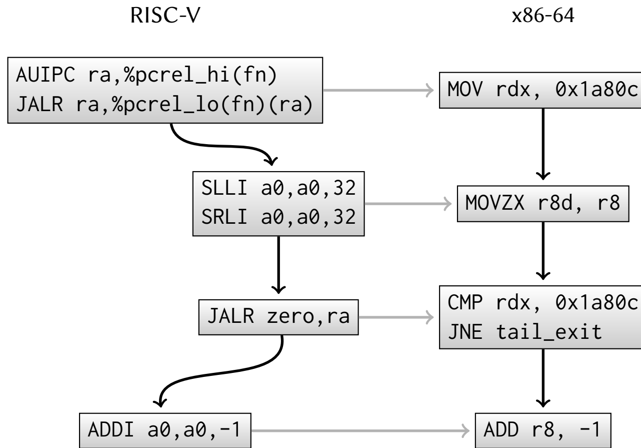

Figure 4: Inline caching and macro-op fusion

for the call instruction sequence is fully specified and can be interpreted as a direct branch using macro-op fusion. rv8 includes an optimisation to translate these sequences of instructions as direct branches using macro-op fusion, as described in Section 3.7 and Table 3.

3.6 Inline caching

The translator inlines hot functions using profile information captured during the interpretation phase. Procedure call control flow is followed during tracing and procedure calls and returns are inlined within the emitted native code [5].

Returns are another form of indirect unconditional branch, and also make use of the translation cache, however in the case where a procedure call is inlined, the indirect return can be elided and replaced with a comparison of the program counter with the expected return address. See Figure 4 for an example.

The translator maintains a return address stack to keep track of procedure returns. Upon reaching an inlined procedure RET instruction (`jalr zero, ra`), the link register (`ra` in RISC-V, `rdx` in the x86 translation) is compared against the caller's known return address and if it matches, control flow continues along the return path. In the case that the function is not inlined, the translation cache is used to lookup the address of the translated code. An inlined subroutine call needs to test the return address due to context switching and functions such as `setjmp` and `longjmp`.

3.7 Macro-op fusion

The rv8 translator implements an optimisation known as macro-op fusion [17][6] [9] whereby specific patterns of adjacent instructions are translated into a shorter sequence of host instructions. The macro-op fusion pattern matcher has potential to increase performance with the addition of common patterns. Table 3 shows macro-op fusion patterns currently implemented by rv8.

rv8 currently only macro-op fuses instruction sequences where the destination register of an earlier RISC-V instruction is overwriten by a later instruction in the sequence. As the same restriction also applies to hardware macro-op fusion implementations, and macro-op fusion is likely to be a key technique in high performance RISC-V processors, we expect future compiler code generation to be tweaked to maximize fusion opportunities.

A technique known as deoptimisation [11] can be employed to allow elision of writes to temporary registers in macro-op fusion patterns assuming the translator sees the register overwritten within its translation window. Deoptimisation requires that the optimised translation has an accompanying deoptimisation sequence to fill in elided register values, and this is played back in the case of a fault (device or debug interrupt) so that the visible machine state precisely matches that which the ISA dictates. rv8 does not presently implement deoptimisation, however it may be necessary to allow more sophisticated optimisations.

3.8 Branch tail dynamic linking

The translator performs lazy translation of the source program during tracing. When it reaches a branch, the translator can only link both sides of the branch if there is an existing native code translation for the not taken side of the branch.

To accelerate branch tail exits, the translator emits a relative branch to a trampoline which is used to return the guest program counter to the tracer main loop. The tracer adds the branch to a table of branch fixup addresses indexed by target guest address. If the branch direction becomes hot, once it has been translated, all relative branches that point to the branch tail trampolines will be relinked to branch directly to the translated native code.

3.9 Sign extension versus zero extension

rv8 has to make sure that all 32-bit operations on registers have their result sign extended instead of zero-extended. The normal behaviour of 32-bit operations on x86-64 is to *zero fill* bits 32 to bit 63 whereas RISC-V requires implementations to *sign extend* bit 31 to bit 63 [21].

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

Table 3: Macro-op fusion patterns and expansions

One potential optimisation is lazy sign extension. It may be possible in a future version of the JIT translation engine to elide redundant sign extension operations, however it is important that the register state precisely matches the ISA semantics before executing any instructions that may cause faults e.g. loads and stores.

3.10 Bit manipulation intrinsics

The benchmarks contain compression algorithms, digest algorithms and cryptographic ciphers which can take advantage of bit manipulation instructions such as rotate and bswap. Present day compilers detect rotate and byte swap bitwise logical operations by matching intermediate representation patterns that can be lowered directly to bit manipulation instructions such as ROR, ROL, BSWAP on x86-64.

Intermediate representation detection of bit manipulation intrinsics has the benefit of accelerating code that does not use inline assembly or compiler builtin functions. RISC-V currently lacks bit manipulation instructions however there are proposals to add them in the 'B' extension. The following is a typical byte swap pattern.

- 32-bit integer byteswap pattern

```
((rs1 >> 24) & 0x000000ff) |
((rs1 << 8 ) & 0x00ff0000) |
((rs1 >> 8 ) & 0x0000ff00) |
((rs1 << 24) & 0xff000000)
```

rv8 implements rotate macro-op fusion which can translate two shift instructions and one 'or' instruction with the correct offsets into one shift and one rotate. The rotate macro-op fusion needs to preserve the residual temporary register side effects so that the register file contents are precisely matched, as it can't easily prove the residual temporary register is not later used. Deoptimisation would be required to elide the temporary register write.

- 32-bit rotate right or left pattern

```
(rs1 >> shamt) | (rs1 << (32 - shamt))
```

- 64-bit rotate right or left pattern

```
(rs1 >> shamt) | (rs1 << (64 - shamt))
```

4 BENCHMARKS

To assess results of the rv8 binary translation strategies, a suite of computationally intensive benchmarks [8] were run comparing rv8 to QEMU riscv64 and QEMU aarch64. The rv8 benchmark suite includes the AES cipher, dhrystone v1.1, miniz compression and decompression, the NORX cipher, prime number generation, qsort and the SHA-512 digest algorithm.

This table summarises the performance ratio geomeans:

optimisation level	qemu aarch64	qemu riscv64	rv8 riscv64	native x86-64
-O3	4.68	4.57	2.62	1.00
-Os	4.62	5.20	2.47	1.00

See benchmark runtime bar charts and tables in Figure 5, Figure 6, Table 7 and Table 8, performance ratios in Table 9 and Table 10, and finally millions of instructions per second for emulated RISC-V and native x86-64 in Table 11 and Table 12.

5 CONCLUSION AND FUTURE WORK

The translation strategy employed by rv8 appears to have borne fruit when compared to other open source dynamic binary translators. Translation is simple and fast and the generated code is good enough for the overall performance to be on on average approximately 75% faster than the leading competitor. On some benchmarks we are approximately 30% slower than native x86-64 code. However we believe there is still a lot of remaining optimization potential in our approach that has not yet been realized.

rv8 implements a user mode simulator and an interpreted full system emulator modelling the RISC-V privileged ISA with an MMU, interrupts, MMIO devices and privilege levels. The current version of the rv8 JIT translator is presently only applied to user mode simulation so future work will explore applying dynamic binary translation to the full system emulator with hardware accelerated MMU using shadow paging. Future work may include:

- Exploring dynamic register allocation algorithms.
- Coalescing loads and stores into complex memory operands.
- Register write elision for pipelined operations with short live spans, to make better use of translator temporary registers.
- Hardware accelerated MMU using shadow paging.

ACKNOWLEDGMENTS

The authors would like to thank Petr Kobalicek for providing the excellent AsmJit [16] library used for rv8's x86-64 back-end support.

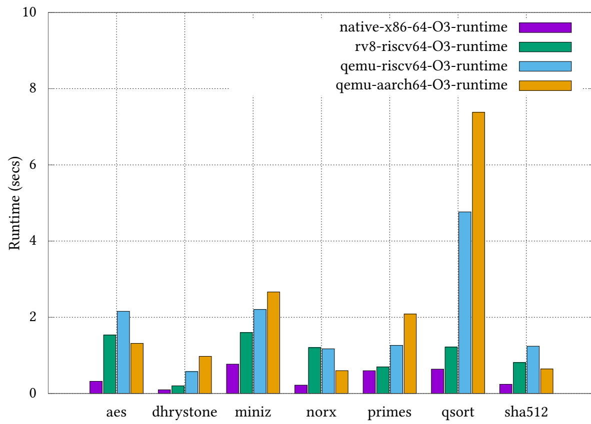

Figure 5: Runtimes 64-bit -O3

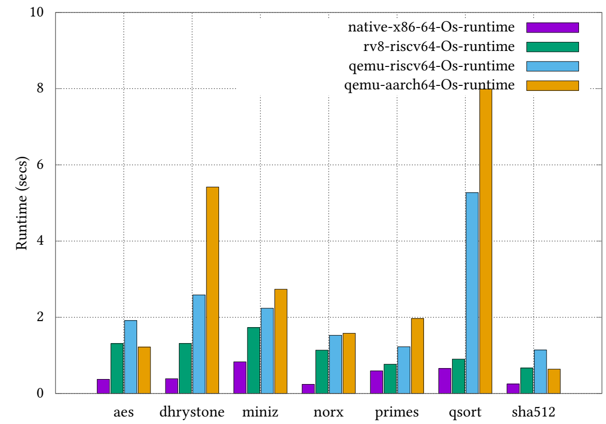

Figure 6: Runtimes 64-bit -Os

program	qemu aarch64	qemu riscv64	rv8 riscv64	native x86-64
aes	1.31	2.16	1.53	0.32
dhrystone	0.98	0.57	0.20	0.10
miniz	2.66	2.21	1.60	0.77
norx	0.60	1.17	1.20	0.22
primes	2.09	1.26	0.70	0.60
qsort	7.38	4.76	1.22	0.64
sha512	0.64	1.24	0.81	0.24
Sum	15.66	13.37	7.26	2.89

Figure 7: Benchmark Runtimes -O3 (seconds)

program	qemu aarch64	qemu riscv64	rv8 riscv64	native x86-64
aes	1.22	1.91	1.31	0.37
dhrystone	5.42	2.59	1.31	0.39
miniz	2.74	2.24	1.73	0.83
norx	1.58	1.53	1.14	0.24
primes	1.97	1.23	0.77	0.59
qsort	7.99	5.27	0.90	0.66
sha512	0.64	1.14	0.67	0.25
Sum	21.56	15.91	7.83	3.33

Figure 8: Benchmark Runtimes -Os (seconds)

program	qemu aarch64	qemu riscv64	rv8 riscv64	native x86-64
aes	4.12	6.76	4.81	1.00
dhrystone	9.96	5.87	2.05	1.00
miniz	3.46	2.86	2.07	1.00
norx	2.73	5.33	5.47	1.00
primes	3.49	2.11	1.17	1.00
qsort	11.55	7.46	1.91	1.00
sha512	2.66	5.13	3.36	1.00
Geomean	4.57	4.68	2.62	1.00

Figure 9: Performance Ratio -O3

program	qemu aarch64	qemu riscv64	rv8 riscv64	native x86-64
aes	3.29	5.16	3.53	1.00
dhrystone	13.97	6.66	3.38	1.00
miniz	3.30	2.70	2.08	1.00
norx	6.59	6.36	4.74	1.00
primes	3.31	2.07	1.29	1.00
qsort	12.20	8.05	1.38	1.00
sha512	2.56	4.58	2.69	1.00
Geomean	5.20	4.62	2.47	1.00

Figure 10: Performance Ratio -Os

program	qemu aarch64	qemu riscv64	rv8 riscv64	native x86-64
aes	-	2414	3395	11035
dhrystone	-	1843	5274	8369
miniz	-	2625	3622	5530
norx	-	2223	2167	9112
primes	-	2438	4421	6100
qsort	-	644	2518	5780
sha512	-	2982	4556	12177
Geomean	-	1985	3552	7945

Figure 11: Millions of Instructions Per Second -O3

program	qemu aarch64	qemu riscv64	rv8 riscv64	native x86-64
aes	-	2655	3879	10072
dhrystone	-	1250	2462	9073
miniz	-	2650	3427	5052
norx	-	1817	2439	8852
primes	-	2226	3555	6101
qsort	-	572	3340	6063
sha512	-	3269	5567	12206
Geomean	-	1821	3402	7855

Figure 12: Millions of Instructions Per Second -Os

REFERENCES

[1] Wikipedia authors. 1994. Mac 68k emulator: a Motorola 680x0 software emulator built into all versions of the classic Mac OS for PowerPC. https://en.wikipedia.org/wiki/Mac_68k_emulator. (1994).

[2] Sorav Bansal and Alex Aiken. 2008. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association, 177–192.

[3] Mark Bohr. 2014. 14 nm Process Technology: Opening New Horizons. In *Intel Developer Forum (IDF'14)*.

[4] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. 2013. Simple and Efficient Construction of Static Single Assignment Form. In *Proceedings of the 22Nd International Conference on Compiler Construction (CC'13)*. Springer-Verlag, Berlin, Heidelberg, 102–122. https://doi.org/10.1007/978-3-642-37051-9_6

[5] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '03)*. IEEE Computer Society, Washington, DC, USA, 265–275. http://dl.acm.org/citation.cfm?id=776261.776290

[6] Christopher Celio, Palmer Dabbelt, David A. Patterson, and Krste Asanovic. 2016. The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V. *CoRR* abs/1607.02318 (2016). http://arxiv.org/abs/1607.02318

[7] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. 1998. FX!32: A Profile-Directed Binary Translator. *IEEE Micro* 18, 2 (March 1998), 56–64. https://doi.org/10.1109/40.671403

[8] Michael Clark. 2017. rv8 benchmark results: Runtimes, Instructions Per Second, Retired Micro-ops, Executable File Sizes and Dynamic Register Usage. https://rv8.io/bench. (2017).

[9] Intel Corporation. 2006. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html. (2006). [Online; page 2-18].

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

[11] Evelyn Duesterwald. 2005. Design and engineering of a dynamic binary optimizer. *Proc. IEEE* 93, 2 (2005), 436–448.

[12] Evelyn Duesterwald and Vasanth Bala. 2000. Software Profiling for Hot Path Prediction: Less is More. *SIGPLAN Not.* 35, 11 (Nov. 2000), 202–211. https://doi.org/10.1145/356989.357008

[13] Sebastian Hack and Gerhard Goos. 2006. Optimal Register Allocation for SSA-form Programs in Polynomial Time. *Inf. Process. Lett.* 98, 4 (May 2006), 150–155. https://doi.org/10.1016/j.ipl.2006.01.008

[14] Paul Hohensee, Mathew Myszewski, David Reese, and Sun Microsysytems. 1996. Wabi CPU emulation. In *Proceedings Hot Chips VIII*.

[15] James Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann.

[16] Petr Kobalicek. 2017. AsmJit: A Complete x86/x64 JIT and Remote Assembler for C++. https://github.com/asmjit/asmjit. (2017).

[17] M.H. Lipasti, S. Hu, J.E. Smith, and I. Kim. 2006. An approach for implementing efficient superscalar CISC processors. *2006 IEEE 12th International Symposium on High Performance Computer Architecture (HPCA)* 00 (2006), 41–52. https://doi.org/10.1109/HPCA.2006.1598111

[18] Mathias Payer and Thomas R. Gross. 2010. Generating Low-overhead Dynamic Binary Translators. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR '10)*. ACM, New York, NY, USA, Article 22, 14 pages. https://doi.org/10.1145/1815695.1815724

[19] Andrew Waterman. 2011. Improving energy efficiency and reducing code size with RISC-V compressed. (2011).

[20] Andrew Waterman. 2016. *Design of the RISC-V Instruction Set Architecture*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html

[21] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. 2011. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* (2011).

[22] Sapumal Wijeratne, Nanda Siddaiah, Sanu Mathew, Mark Anders, Ram Krishnamurthy, Jeremy Anderson, Seung Hwang, Matthew Ernest, and Mark Nardin. 2006. A 9GHz 65nm Intel Pentium 4 processor integer execution core. In *Solid-State Circuits Conference, 2006. ISSCC 2006. Digest of Technical Papers. IEEE International*. IEEE, 353–365.

[23] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 79–93.