Fast simulation of future 128-bit architectures

Fabien Portas and Frédéric Pétrot
Univ. Grenoble Alpes, CNRS, Grenoble INP*, TIMA, 38000 Grenoble, France
fabien.portas@grenoble-inp.org, frederic.petrot@univ-grenoble-alpes.fr

***Abstract*—Whether 128-bit architectures will some day hit the market or not is an open question. There is however a trend towards that direction: virtual addresses grew from 34 to 48 bits in 1999 and then to 57 bits in 2019. The impact of a virtually infinite addressable space on software is hard to predict, but it will most likely be major. Simulation tools are therefore needed to support research and experimentation for tooling and software. In this paper, we present the implementation of the 128-bit extension of the RISC-V architecture in the QEMU functional simulator and report first performance evaluations. On our limited set of programs, simulation is slowed down by a factor of at worst 5 compared to 64-bit simulation, making the tool still usable for executing large software codes.**

I. Introduction

According to [1, Chapter 7], should today's super-computer trend continue, single address space 64-bit computers could be short of physical addresses by 2030. Computer history has taught us that while temporary PAE-like [2] solutions might exist for some time, the flat 128-address design will eventually emerge as the best solution. In that situation, naturally, the general purpose registers of a processor will end up being 128-bit wide. Clearly, not all computers will be 128-bit, and we are speaking here of a huge number of machines sharing a huge amount of memory, such as high-performance computing clusters.

Making the move towards 128-bit registers and addresses is a micro-architectural and architectural challenge, given the fact that transistor technology scaling as we knew it has been over for a decade or so [3]. But it is also a system level challenge: the impact on memory hierarchy organization, compilation flows, operating systems, and so on have to be studied and understood. It is this latter level that the work we present in this paper addresses. Indeed, we believe that having tools to help grasping the system level issues that 128-bit addresses pose is critical. That is why we report here our work on fast simulation of 128-bit instruction set architectures (ISA), a necessary step towards more general, full system, simulation environments.

For the sake of concreteness, we choose RISC-V as ISA given that it already features a clean-sheet 128-bit extension proposal, and the QEMU [4] dynamic binary translator (DBT) as fast simulation engine. Our contribution is thus the design, implementation, and performance analysis of the support for 128-bit operations in a DBT engine. Although not fundamental, such an experiment has not been reported yet as far as we know.

*Institute of Engineering Univ. Grenoble Alpes

II. Background

A. Summary of the RISC-V specification relevant to 128-bit

To the best of our knowledge, the RISC-V specification is the first one to introduce accurately, although informally, what could be a 128-bit processor. It has interesting assets: it is simple; has 32 general purpose registers all of the same size; makes instructions working by default on the natural size of the processor registers (e.g. the `add` instruction has the same opcode whether the processor has 32, 64 or 128-bit registers); naturally extends existing instructions when necessary (e.g. the shift immediate instructions have an immediate on 5, 6 or 7 bits depending on the registers size); and provides specific instructions for narrower computations (e.g. the `w` suffixed instructions operate on 32-bit data, and the `d` suffixed ones on 64-bit data). Memory accesses are originally suffixed by the size of the access, so the 128-bit version simply adds the `lq` and `sq` instructions to access quad-words.

This specification however has also an unusual property: the size of the registers, called `xlen` in the specification, might be run-time configurable. The machine-mode (highest-privilege level in RISC-V parlance) register size, `MXLEN`, can be defined by setting the two upper bits of the Machine ISA (`misa`) register. If running a 128-bit machine, it might be set to 128, 64 or 32, and if running a 64-bit machine, it might be set to 64 or 32. In addition, supervisor-mode and user-mode `xlen`s (`SXLEN` and `UXLEN`) can also be set dynamically, by changing bitfields in the processor status register of the appropriate modes.

B. More than 64-bit address initiatives

Multiple usages of addresses larger than 64-bit have been envisioned. The first one is to associate meta-data to pointers. Pointer tagging was introduced on early machines by using the lower bit(s) on addresses of naturally aligned 16-bit or 32-bit data for which these bits where forced to zero in hardware. Quickly, the size of the address would be such that it could address more than the available memory. On machines with 24-bit addresses but only 64 KB of memory, the upper 8-bit were available for various usages, and in particular protection. This paved the way for associating "capability" to a pointer [5], which is a generalization of this principle. Modern and efficient "capability" implementations provide type safety (dynamic size and permissions) to pointers by resorting to 128-bit addresses [6] that include a base virtual address, a bound, and permissions.

The second one is related to operating systems (OS). Interestingly enough, when going to 64-bit addresses, the operating system community considered how this vast space

978-3-9819263-6-1/DATE22/©2022 EDAA

could be used [7] and proposed, among others, to access external objects transparently through the network. This idea is currently being pursued with larger address spaces in mind, e.g., [8], although ad-hoc extensions are used rather than going all the way to a flat 128-bit space.

We speculate that addresses might yet again become scarce resources again in supercomputers, in a context where integration technology allows huge memory sizes, numerous devices, and even access to resources external to a chip at reasonable performance, hence we predict a growing interest for 128-bit in the near future

C. Retargetable dynamic binary translation in a nutshell

Given this context, accessing an environment for the fast execution of 128-bit based code is of interest for compiler writers, operating system designers, and even applications programmers. Although many simulation technologies exist, the one most suited to large scale full system software execution is undoubtedly dynamic binary translation. We now briefly remind the reader of its main characteristics. DBT aims to transform instructions from a target, i.e., simulated CPU, into host instructions, i.e., CPU on which the simulation runs, during execution. DBT based simulators are generally retargetable, to avoid having to model a target for each host. As seen Fig. 1,

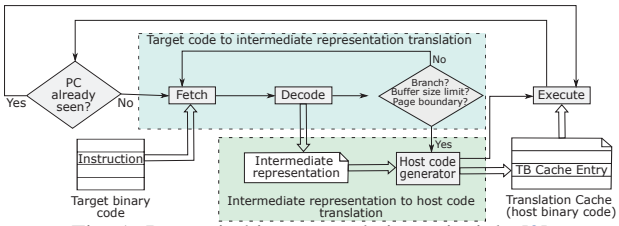

Fig. 1: Dynamic binary translation principle [9]

the translator fetches target instructions until a branch is found, translating each instruction in a set of *micro-operations*, the intermediate representation (IR) of the engine that resembles a 3-address RISC ISA. On a branch, the aggregated set of micro-operations is optimized and translated into host instructions terminated by a return to the simulator run-time with the value of the program counter of the next target instruction to execute.[1] The set of translated micro-ops is called a translation block. It is then executed until it gives back control to the simulator, which will then lookup an already translated block. If the block is found, the execution resumes, otherwise, the translation process is executed starting at the new instruction address. The strength of this strategy compared to naïve instruction per instruction execution is first that it decodes the instructions once and for all, second that it executes code at the granularity of the translation block, and third that powerful intra- and inter-translation block optimizations can be performed. Its drawback is the associated complexity, hence the interest of extending existing DBT engines. Given its stability, performance, and extensive supported processor list, we selected QEMU to work with.

[1]This explanation is greatly simplified and describes only the overall principle.

III. Design and Implementation

A. Processor state representation

A processor simulator, independently of its implementation, has to maintain the state of the processor in the host memory. Assuming a 64-bit host, the register file for a 64-bit RISC-V will typically be declared as an array of 32 64-bit unsigned integers. As there is no standard type for 128-bit data in C, a 128-bit data will be stored as two 64-bit values, and instructions will operate on these two values. From a storage point of view, we have two alternatives: either two arrays of 32 entries each (noted "2a"), or a single array with 64 entries. In the former case, fetching a 128-bit data corresponds to an access to each array at the same index. In the latter case, either we access data at position i and $i + 32$ (noted "1a"), leading more or less to the same solution than the previous one, or we access position $2 \times i$ and $2 \times i + 1$ (noted "1c"). The advantage for the first and second solutions is that there is no computation overhead when generating the code performing the access to the registers for the 32-bit and 64-bit versions of the ISA, while for the last one we might benefit from better cache locality during execution. We implemented the three solutions in QEMU, compiled with default optimization (`-O2`) using `gcc` version 10.2. We executed QEMU on an AMD EPYC 7702P processor, and measured performance using Linux `perf` tool [10] on 3 synthetic programs written in 128-bit assembly. The programs are executed 100 times each, and Fig. 2 reports the results (average, minimum and maximum execution times) normalized to the "2a" solution. We also computed the normalized standard deviation, which is small enough for us to trust our measures. No solution clearly outperforms the others, given the execution

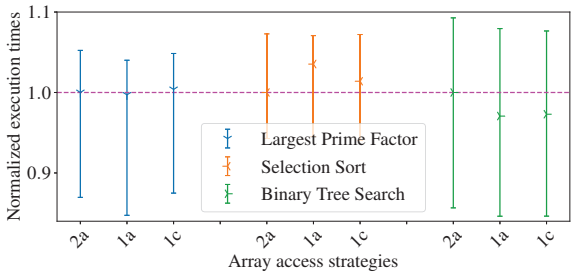

Fig. 2: Normalized QEMU run-times for the 3 possible register array access strategies

time variations. This is not really a surprise as the overhead of the registers addresses computation in the "1c" case occurs only at translation time. The fact that the two registers addresses are not contiguous (they are 512 bytes apart) for solution "2a" and "1a" does not slow down execution, most probably because the cache capacity is large enough to hold the often needed parts of the CPU state. Since speed of execution is similar, we opted for solution "2a" as it is the least intrusive in QEMU existing code.

B. Instructions implementation

QEMU contains a skeleton generator, the "decodetree" [11], which produces the instruction decoder that calls a function

with the appropriate arguments (context of execution at the time of the translation, register numbers, value of an immediate, etc) for each instruction. Therefore, from a practical point of view, implementing an instruction consists of writing the code that generates the IR whose behavior is equivalent to the one of the instruction.

Most instructions are simple enough to be translated in a handful of 64-bit micro-ops, as the base instructions are only loads, stores, arithmetic and logic operations, shifts and branches, none having side effects. For instance, denoting (a_h, a_l) and (b_h, b_l) two 128-bit values made each of two 64-bit values, the result r of an addition is computed as $r_l \leftarrow a_l + b_l; r_h \leftarrow a_h + b_h + r_l \overset{u}{<} a_l$. This shows that 128-bit instructions will in average more than double the number of micro-ops compared to 64-bit ones, which in turn will increase both code size and execution time. Loads and stores are straightforward as long as addresses stay on 64-bit to benefit from the existing address translation mechanism: we use only the 64 lower bits currently. Algorithms for shifts instructions are relatively simple too, and need around five to six 64-bit instructions in average. Comparison predicates use various bitwise logic tricks [12, Section 2.12] and in average less than double the number of instructions compared to 64-bit.

Finally, we also support the M-extension, that includes multiplication, division and remainder. If the various 128-bit multiplications can be generated with around ten micro-ops, the division and remainder are too complex and we resorted to helpers, i.e. calls to C functions from within the translation block. Our implementation is based on the quite efficient algorithm proposed by Kanthak [13], or uses the non-standard compiler support for 128-bit operations when it exists, as it is the case for `gcc`.

IV. Performance evaluation

We conduct several experimentations to characterize our simulator. To that aim, we developed 3 simple synthetic programs in assembly, in such a way that they can be executed for an `xlen` of 32, 64 and 128-bit. As most of the instructions have the same encoding independently of `xlen`, it suffices to use dedicated macros for the loads and stores that produce binary directly (thanks to the support of the `.insn` directive in the RISC-V assembler), and have the assembler compute addresses and offsets using the data type size. This guaranties that the algorithms are identical but for one thing: the data-type size. We validated this property by checking (thanks to QEMU `-d in_asm` option) that the code fetched during simulation is identical for our 3 programs, which is the case except for the load and store instructions and offset factors.

The algorithms are naïve implementations of a selection sort, the search for the largest prime factor in a number using a brute force approach, and a binary tree search. The parameters for the algorithms are somewhat arbitrary, but allow simulations in the range of 0.1 s to 60 s, to capture a wide range of behavior.

We start by gathering size information on the translation blocks for each of the programs, compared to 64 and 32 bit versions, to better understand how the translated code increases in size when handling 128-bit registers. We measure both the average number of micro-ops (before and after optimization by QEMU) and the average number of x86-64 instructions and size in bytes per translation block. The results are presented Table I. We can see that for the 32-bit and 64-bit ISA, the resulting number of micro-operations and x86-64 instructions is very close, while the 64-bit version leads to slightly bigger translation blocks, which might lead to a marginally less efficient instruction cache utilization. In average, the number of micro-operations is approximately 1.5x larger with 128-bit, and it is a similar factor compared to the x86-64 metrics. This might seem lower than expected, but:

TABLE I: Average size of the translation blocks

	micro-ops before/after optim.			x86-64 nb. of insns/size in bytes		
Program	32	64	128	32	64	128
Largest Prime Factor	20/15	21/15	35/20	34/149	36/163	53/243
Selection Sort	26/18	27/18	51/31	51/223	53/246	81/357
Binary Tree Search	19/14	18/14	34/20	37/167	39/181	61/273

1) QEMU already provides some 128-bit operations as micro-operations, such as the heavily used `add` and `sub`, but also various `mult` flavors,
2) QEMU performs deadcode elimination, and since the loads and stores use only the lower 64-bit of the addresses, only few additional instruction are necessary compared to 64-bit,
3) each translation block contains a prologue and an epilogue consisting of 6 to 8 micro-operations, which decreases the overhead of the 128-bit implementation by construction,
4) the x86-64 has an add with carry instruction, which make adding two pairs of 64-bit registers just two instructions.

Given this bigger code size, it can be expected that the instruction cache misses more often. Furthermore, as the data are also twice larger (remember that the instructions work on the `xlen` register size), we might expect larger slowdowns due to the lower relative data-cache capacity.

We now focus on the wall-clock time of QEMU execution for the 3 possible register sizes. We first check that our add-ons do not increase execution times for the 32 and 64 `xlen`s: the comparison between vanilla QEMU and ours leads to less than 3% execution time variations in average over all the executions, positively and negatively, hence there is no visible impact.

Then we run simulations for different values of the main parameter of each of our programs. Fig. 3 plots the run-times of the Largest Prime Factor, using a log-log scale. This program makes an heavy use of the M-extension, in particular division and remainder that are realized through calls to helpers (implemented with `gcc` internal support for 128-bit arithmetic). The 128-bit `xlen` execution time is on average around 4 to 5 times slower than its 64 and 32-bit counterparts, both having similar run-times.

Fig. 4 reports the Selection Sort run-times. This mainly computes addresses, needing shifts and adds, makes comparisons, and memory accesses. Interestingly enough, the ratio between 128-bit and 64 (or 32)-bit execution times is still around 5, for the bigger arrays. We might have expected a bit better

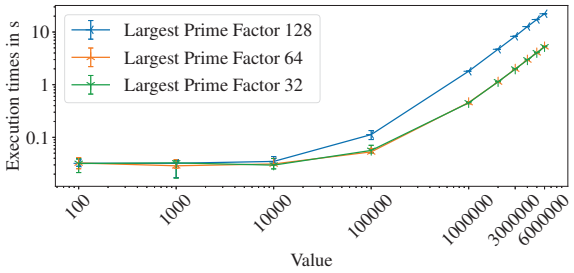

Fig. 3: Run-times for Largest Prime Factor

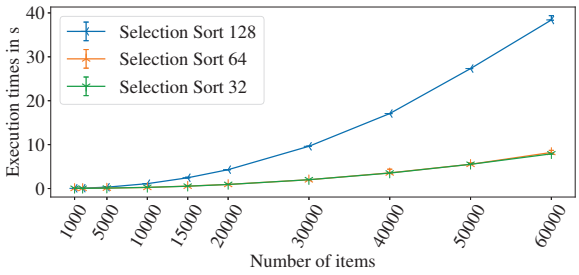

Fig. 4: Run-times for Selection Sort

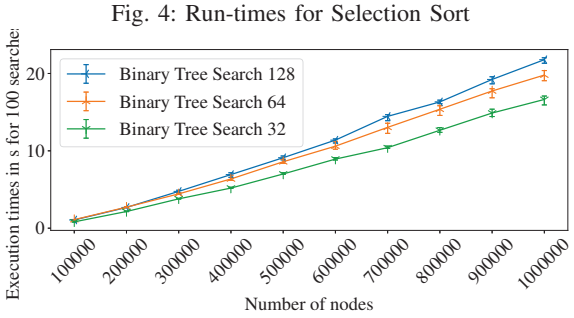

Fig. 5: Run-times for Binary Tree Search

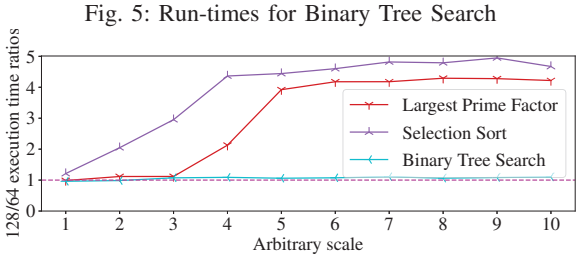

Fig. 6: Execution times ratios

performances, given that we do not use instructions that call helpers in this program.

Finally, Fig. 5 reports the same metric for Binary Tree Search. The program dereferences pointers and performs equality comparisons. The ratio over 64-bit execution is much lower, around 1.2× and 1.3×. This can be explained by the fact that the addresses are on 64-bit, and QEMU eliminates the code that computes the upper part of the address during translation. Thus the overhead is dominated by having to read 2 times 64-bit from memory and do a 2 times 64-bit comparison per node.

Fig. 6 gathers the 128/64 execution times ratios for the three programs, using a arbitrary unified x-axis unit corresponding to the experiment number on the previous figures.

For a small number of computations, the time is dominated by QEMU loading and starting time, then it steadily increases until it reaches a plateau. The asymptotic value is clearly application dependent, and its upper bound on our examples is around 5.

V. Conclusion and Future Work

This paper presents the design and implementation of a fast 128-bit RISC-V processor simulator based on QEMU. The performance evaluation of our solution indicates a slowdown of a factor of 4 to 5 compared to the 64-bit version, which is explained by the additional logic required to work on two 64-bit double words for each instruction. Nevertheless, simulation speed is largely sufficient to boot an OS or run large applications

There are two immediate follow-ups to this work: (a) optimize the backend by using e.g., SIMD instructions working on larger bitwidth, and (b) change address translation so that it handles 128-bit guest virtual addresses. Both are quite challenging from a technical standpoint.

Acknowledgment

The authors would like to thank the partners of the ANR Maplurinum project and acknowledge the financial support of the French Agence Nationale de la Recherche under grant ANR-21-CE25-0016.

References

[1] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2014-54, Updated*, p. 244, 2021.

[2] R. R. Collins, "Paging extensions for the pentium pro processor," *Dr. Dobb's Journal Undocumented Corner*, Jul. 1996, http://www.rcollins.org/ddj/Jul96/.

[3] C. A. Mack, "Fifty years of moore's law," *IEEE Transactions on semiconductor manufacturing*, vol. 24, no. 2, pp. 202–207, 2011.

[4] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[5] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[6] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *41st International Symposium on Computer Architecture*. ACM/IEEE, 2014, pp. 457–468.

[7] J. S. Chase, H. M. Levy, M. Baker-harvey, and E. D. Lazowska, "How to use a 64-bit virtual address space," Department of Computer Science and Engineering, University of Washington, Tech. Rep., 1992.

[8] X. Wang, J. D. Leidel, B. Williams, A. Ehret, M. Mark, M. A. Kinsy, and Y. Chen, "xbgas: A global address space extension on risc-v for high performance computing," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 454–463.

[9] M. Gligor, N. Fournel, and F. Pétrot, "Using binary translation in event driven simulation for fast and flexible MPSoC simulation," in *7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, 2009, pp. 71–80.

[10] A. C. De Melo, "The new linux "perf" tools," in *Linux Kongress*, vol. 18, 2010, http://vger.kernel.org/~acme/perf/lk2010-perf-acme.pdf.

[11] The QEMU Project Developers, "Decodetree specification," 2019, https://qemu.weilnetz.de/doc/devel/decodetree.html.

[12] H. S. Warren, *Hacker's delight*, 2nd ed. Pearson Education, 2013.

[13] S. Kanthak, "Fast(est) 128÷128-bit and 64÷64-bit integer division," 2021, https://skanthak.homepage.t-online.de/integer.html#udivmodti4.