Partial RISCV 128 support in QEMU for a x86_64 bit host

Benjamin Cathelineau
benjamin.cathelineau@grenoble-inp.org
Univ. Grenoble Alpes, Grenoble INP[†]
Grenoble, France

Frédéric Pétrot
frederic.petrot@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, CNRS, Grenoble INP[†], TIMA
Grenoble, France

ABSTRACT

Just like 64 bits computing was the logical continuation of 32 bits computing, 128 bits computing is the logical next step after 64 bits computing. A first step to 128 bit computing is to be able to simulate 128 bits computers on 64 bits computers. QEMU is a reference in this field but other tools exist as well. In this paper, using QEMU as a base, we implement a simulation of a RISC-V 128 bit computer on 64 bit "traditional" x86_64 computer. Only a small subset of the 128 bit instructions were implemented, those necessaries to show a proof of concept: sorting an array of 128 bit signed integers. Performance on this sorting were compared between 64 and 128 bits using a variety of metric. It was determined that the number of instructions to be executed is multiplied by 2, but that, asymptotically, the run time is multiplied by slightly more than 2, because of the vastly larger amounts of cache misses.

KEYWORDS

System-Level Simulation, Dynamic Binary Translation, 128 bit computing

1 INTRODUCTION

Most general purpose computer nowadays are based on a 64 bit model. This means that :

- the registers are 64 bits wide.
- the address space is on 64 bits, even though all 64 bits aren't necessarily used.
- the data types, such as int, double... can be up to 64 bits wide, but don't have to be, see the int type which is set to 32 bit by default by the compiler on many 64 bits architectures, such as x86_64.

A true 128 bit model would be :

- 128 bits wide registers.
- 128 bits address spaces.
- And data types that are up to 128 bits size.

However to this day there is no consumer available 128 bit computer. That can be easily explained by the fact that such a computer would hardly be necessary as we detail in section 2.1. Nonetheless it could become necessary in the future.

A successful transition to 64-bit computing was achieved around 2 decades ago, but 32 bits computer still remain for some application. We believe that the same will be true of 128-bit computing, at least at the beginning. If we look at the 32 to 64 bits transition, we can infer that 64-bit computing will probably remain relevant long after the introduction of 128-bit computers on the consumer market, just like 32 bits computers are still relevant to this day, in micro-controller applications, more that 10 years after the democratisation of 64 bits consumer computer.

This paper focuses on the simulation of 128-bit architectures. Although processor simulation is widely used and a subject of active research, to the best of our knowledge, there doesn't exist a 128 bits general purpose computer simulator.

The paper is organized as follow: we first give a overview of the use case and advantage that 128 bits computing could have in section 2. We then, in section 3, provide a bit of background on dynamic binary translation in general and on QEMU especially. In section 4 we detail the implementation of the subset of instructions necessary to perform our sort. In section 5 we detail the result of our measurements of performance between 64 and 128 bits in sorting the array. Finally section 6 highlights what the author think could be the next required step to implement a 128 bits address space.

[†]Institute of Engineering Univ. Grenoble Alpes.

2 USE FOR 128 BITS COMPUTERS

2.1 Address space size consideration: 128 bit is not yet necessary

As we explained before, in a 64-bit model of computing, the address bus is 64-bit large. The memory in a computer is addressed per byte, the generally accepted minimal item of information, in other words, each byte has its own address. Therefore, with 64 bits $2^{64} \approx 2\times10^{19}$ bytes can have their own address. However the most expensive consumer desktop computer currently available has at most (this is an over exaggeration of course, In reality most computer have between 2 and 64 gigabytes of memory) 1 terabytes of memory, that is $= 10^{12}$ bytes. Even using this over estimation we see that we are still missing $\approx 10^{19-12} = 10^7$ in order for exactly 64 bits to be useful. In order word, the amount of memory would have to be multiplied by 10^7 to use the full extends of a 64 bits address space. If we assume that a server has 10^3 as much memory as this hypothetical and overestimated most expensive consumer computer, we are still missing 10^4. Keep in mind that theses were massive overestimation.

A valid conclusion could be that because 64 bits is not used fully yet (and will not be any time soon on a single computer), 128 bits addresses spaces are simply useless, for now anyway.

However, a long term hypothesis is that larger address spaces could still be desirable for specific applications. For example, we could imagine a data center that has a shared, virtual, address space across multiple servers. Such an address space, realistically in a few year, could be above $2^{64} \approx 2 \times 10^{19}$ bytes in size. The address space could also be sparse, but useful as such, to ease large scale computers organization.

Let us consider, for a moment, the famous TOP 500 website[2]. This website contains a list of the 500 most powerful supercomputer in the world, ranked in order of their performance. For example if we look at the first entry in the latest list at the time of writing, dated from November 2020, we see that this computer has approximately 5×10^{15} bytes of memory. Now, if we take a step back and consider the historical data provided by this website, we can see the list as it was 20 years ago, in 2001. The computer that was at the top of the list at this time had 6×10^{12} bytes of memory.

In other words, in 20 years, the amount of memory in the most powerful supercomputer in the world was multiplied by 10^3. Therefore, it doesn't seem too unlikely, than in a bit more that 20 year, the memory capacity will again be multiplied by 10^3 and be just 10^1 shy of the limit of 64 bits, which is $2^{64} \approx 2 \times 10^{19}$.

2.2 Other 128 bit advantages and current techniques

128 bits computing is not only about a growth in address space. Indeed other advantage include but are not limited to :

(1) Bigger integer support : 128 bit would allow for integer to go from -2^{127} to $2^{127} - 1$ instead of -2^{63} to $2^{63} - 1$ on 64 bits
(2) Bigger (as in more range) and more precise floating point representation

As point 1 is fairly obvious, we will focus on point 2 which is certainly more interesting.

128 bits floating point numbers. As low level computer scientist, and people that require high precision in their calculation know, floating point are not equal to real numbers. In other words, for every real number in R there is not necessarily a exact floating point number representation. That is, in part[1], because there is an uncountable infinite number of real number and not an infinite number of floating point that can be represented in a finite amount of bytes. In fact an upper bound for the number of floating point that be represented on n bits is 2^n.

We know that exact precision for each real number is impossible, but having a larger amount of bits, such as 128 instead of 64, is still use-full. To illustrate this fact, in the following section, we will use the ULP [8]. In very short terms it is the maximal error that you can expect to have around a floating point number[2].

[1]it is also because some number in R are only representable (in normal form and not in fraction form) exactly in some base and not others. $\frac{1}{3}$ in base 10 is a common example $\frac{1}{10}$ in base 2 is another

[2]Theses calculation where done only considering the mantissa part of the representation and not the exponent, which means that we are way beyond the number range for 16 and 32 bits, nonetheless, this is still a correct result

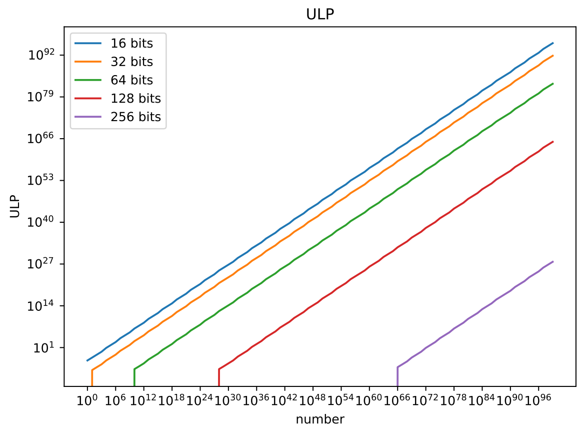

Figure 1: ULP for 16, 32, 64, 128, 256 bits floating points number, with respect to the IEEE754[9] format

As we can see from figure 1, 128 bits is much better than 64 bits for floating points numbers. The ULP is bellow 10 for number as big as $\approx 10^{30}$, a $\approx 10^{18}$ factor compared to 64 bits's $\approx 10^{12}$. However, another pattern emerges : the increase in precision is not linear but exponential. That is, when you double the number of bit n to $2n$, you do not double the precision, but multiply it by 2^n. This means that increasing the number of bit does not follow the law of diminishing returns, quite the opposite, what's happening here could be called "increasing returns". As an example, the increase from 16 to 32 bits multiply the range at ulp 10 by $\approx 10^2$, while the increase from 128 to 256 multiplies it by $\approx 10^{37}$!

Of course that does not mean that a trade off is not necessary, because as we see in section 2.3 adding bits does have a cost.

Alternatives to a full 128 bits computer to reach higher precision floating points and higher ranging integers. There are already tools available to performs computation with data types that have more than 64 bits. Of course, the one tool that immediately comes to mind is the python[6] programming language, which is widely in use in scientific community that need to do precise computations on, because of its build in support for bignums. As far as the authors know, the build in support is only for integer, for arbitrary precision floating points external libraries are still needed.

But, as it known[17], the big drawback of python compared to C are performances.

Fortunately, there are also libraries that exist in C to performs computation with data types that have more than 64 bits. For big integer the very famous opensll[16] library, provides a big number implementation, which is used here for cryptography applications. For floating point the industry standard is `GNU MPFR`[14], which was actually used to implement a arbitrary precision floating point extension[10] in QEMU for RISC-V targets.

2.3 128 bit drawbacks : energy consumption and performance

This section is about the energy consumption and performance of 128 bits on real hardware, for the performance of dynamic binary translation of a 128 bits target on a 64 bits see section 5.

A reasonable hypothesis would be for the energy consumption to double as we double the number of bits of the architecture. As there is no 128 bits computer available yet, it is impossible to do experimentation and we have to resort to estimations. In theses following estimation we will make the assumption that the macro and micro architecture will not change outside of turning to 128 bits register and address space. This assumption is not historically accurate because, in the case of the x86 (32 bits) to x86_64 (64 bits) transition, many other changes were made. For example, the number of register was increased and there also was micro architectural changes. Nonetheless we will go with this assumption because:

- It is reasonable to assume the 64 to 128 bits transition will not cause as much changes as the 32 to 64 bits transition did, at least for RISC-V which is a more modern ISA, where the idea of 128 bits is already partially present.
- We have no way to predict what change will actually be made to the various ISAs that make the transition and to their micro-architectures.

Circuit size. As the register double in size, the surface necessary more than double[19]. In a similar way, the number of connections doubles.

Power consumption of a 128 bits computer. To guess what would be the energy consumption drawback of 128 bits computer compared to a 64 bits computer we can look at an estimation of the power consumption of a processor. By definition P, the power consumed is : $P = \alpha \times \frac{1}{2}CV^2f$, where α is the rate of wire changing value, $(0- > 1$ or $1- > 0)$, C is the overall capacity of the whole design, V is the voltage and f is the frequency. Now, when we multiply the size of the register by 2, we multiply C by 2, therefore multiplying the power consumption by 2. While power consumption itself is a problem of importance, because of its impact on the environment, it also in turn impact the performance negatively. For a mobile device, such as a phone or laptop, this could mean a battery life reduced by half. For desktop computer or server, this would mean the need for even more efficient cooling solutions, or thermal throttling would ensue.

3 BACKGROUND IN DYNAMIC BINARY TRANSLATION AND QEMU

Before tackling the subject of dynamic binary translation and QEMU, it is worthwhile to bring to the reader's attention that dynamic binary translation is but one of the many techniques used to simulate a computer on another. For example, something like `gem5`[12], simulates computers at a microachitectural level, in other words, it simulates directly the hardware behavior, including what's invisible to the assembly programmer, like instruction pipelining. Dynamic binary translation on the other hand, including QEMU of course, only implements what the assembly programmer can see : instructions. Theses instructions do not have to be working in a similar way to their actual implementation in the hardware. That's why QEMU is less accurate, but much faster, than something like `gem5`.

It is important to define a vocabulary before talking about dynamic binary translation:

- The *Host* is the system that does the simulation, the actual physical hardware. In our case it is a "traditional" x86_64 computer.
- The *Target* is the computer that is being simulated, in our case a RISC-V computer.

The goal of QEMU is to translate the machine code of the *Target* into the machine code of the *Host*. For better performance, this is not done instruction per instruction, but block per block, theses block are called "translation block".[3]

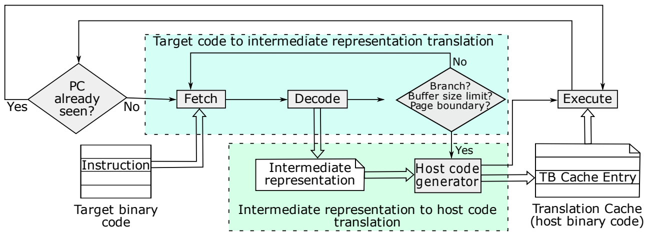
Target code to intermediate representation translation
PC already seen?
Yes
No
Fetch
Decode
Branch? Buffer size limit? Page boundary?
No
Yes
Execute
Instruction
Target binary code
Intermediate representation
Host code generator
TB Cache Entry
Intermediate representation to host code translation
Translation Cache (host binary code)

Figure 2: Dynamic binary translation principle

You can see in figure 2 taken from [15] an illustration of the process of translation and execution. Translation block are delimited by branches most of the times[4]. In other words, a sequence of instruction that is finished by a branch is a translation block. Once translated, the block is put in a cache, so that it can be taken latter from here, and the translation step can be skipped. QEMU does exactly that but it structured in such way that it is *retargetable*. Instead of translating the *Target* machine code into the *Host* machine code, the *Target* machine code is instead first translated into micros operations. Theses micros operations are written in C and are used, once compiled, by a dynamic code generator to finally generate host machine code. See [11] for more details.

The fact that QEMU is *retargetable* means that, while untested, the work presented in this paper should function on all *Hosts* supported by QEMU.

4 WORK AND IMPLEMENTATION

4.1 Adding 128 bits registers

There are multiple way to do this but the authors decided on following the example of the Power Architecture (PowerPc) extension and added a new array of cpu registers, `gprh`, that holds the 64 high bits of the 128-bit registers.

It's important to keep in mind that theses registers have to be initialized, see figure 26.

4.2 Deep dive in the implementation of one instruction: `sll`

`sll` is the instruction that performs shift left in RISC-V just like `SHL` in x86. We chose to present this instruction in particular from the subset of instruction that was available, because it shows well the kind of binary *tricks* that you have to do to implement 128 bit instructions using only 64 bits registers.

[3]However, it is possible to instruct QEMU to translate instruction per instruction, for debugging purpose for example, with the `-singlestep` option
[4]Or if the maximum size for a translation block is reached, or if a page boundary is reached

The problem presented by shift instructions: the overflow or underflow. The meaning of overflow and underflow in this section is a bit different from the usual meaning which relates to addition or multiplication, this new meaning is explained in the section itself

Since we've chosen to represent a 128-bit register by using 2 64-bit registers, each instruction has to be performed on each 64 bits sub register individually. But sometime the result of the operation on the low register impact the result of the operation on high register and this must be accounted for. This happens in the `sll` instruction.

See for example the following 128 integer *a* in hexadecimal `0x0000000000000100 00000000F0000000`[5]

Assuming the `t0` register holds this integer *a*. What would happen if a programmer used the instruction `sll t0,t0,4` ? In that case there is no problem, nothing tricky is necessary, we just need to apply the shift micro operation, already provided by qemu, on the register holding the 64 most significant bits and on the register holding the least significant 64 bits, which would give us the following result in `t0` : `0x0000000000001000 0000000F00000000`

But what about a different scenario where we would now consider that the following integer *b* is in t0 instead:
`0x0000000000000100 F000000000000000`
In that scenario if we stick simply to the previous method then we would get the following incorrect result:
`0x0000000000001000 0000000000000000` instead of the wanted and correct result which is :
`0x000000000000100F 0000000000000000`.

Indeed, as you can certainly understand at this stage, we have to "carry" the "overflow" of the shift into the most significant bit register. The same is of course true of the `shr` instruction, but in that case we would have to carry a "underflow" into the least significant bit register. This is what is meant by *overflow* and *underflow* in this section.

A solution for the shift problem. For the sake of simplicity we will present, without loss of generality, because indeed the solution for shift right would be the same in "reverse", only the shift left instruction. The idea between the solution is simple. We need to copy the *n* bits from least significant 64 bits registers most significant side into the most significant 64 bits registers least significant side. To remove any remaining ambiguity here is an example:
Consider the following 128 bits integer *c* :
`0x00000000000010 00 AA00000000000000`
In the case of, `t0` holding *c* as usual, the instruction `sll t0,t0,8` we would have to copy the 2 red A's into the green "slots".
The final result of the shift would be :
`0x00000000001000AA 0000000000000000`

To achieve this "copy" using only qemu micro operations is no simple feat. The code for this algorithm is presented bellow, some part that are repeated multiple time, or not important for the explanation, were removed to increase readability.

[5]For the sake of readability again the number are represented left to right, which means that the most significant bit are on the left and least significant bit on the right

```
static bool trans_sll(DisasContext *ctx, arg_sll *a)
{
   /*All registers are initialized with tcg_temp_new().
   source1 holds the 64 low bit of the value to be shifted while
   source1h holds the 64 high bit of the value to be shifted,
   source2 hold the shift amount and
   target_long_bit holds the size of our native registers, that is 64 */
   TCGv source1, source1h, source2, overflow, mask, mask_offset, target_long_bit;

   /*We will generate a mask that will allow us to copy the right number of bits
   from the low 64 bits register*/
   /** Mask generation code **/
   /*We now have the difference between our native register size and shift amount*/
   tcg_gen_sub_tl(mask_offset, target_long_bit, source2);
   /*The mask initial value is 64 ones*/
   tcg_gen_movi_tl(mask, 0xffffffffffffffff);
   /*we shift the mask using the previously calculated mask offset
   that will clear the right most bits that are not to be copied*/
   tcg_gen_shl_tl(mask, mask, mask_offset);

   /**Overflow generation code**/
   /* This gets us the bytes that must be copied*/
   tcg_gen_and_tl(overflow, mask, source1);
   /*Now we shift them back to their correct position in the high register*/
   tcg_gen_shr_tl(overflow, overflow, mask_offset);

   /*Perform the shift normaly using qemu micro operations*/
   tcg_gen_shl_tl(source1, source1, source2);
   tcg_gen_shl_tl(source1h, source1h, source2);

   /*We use our overlfow and add it to the high 64 bits register */
   tcg_gen_or_tl(source1h, source1h, overflow);
   /*Rest of code not relevant for explanation*/

}
```

Figure 3: the source code of the sll instruction

4.3 List of Implemented Instruction

For reference you can see the full RISC-V instruction set here [5]

```
add, sub, sll, slt, lq, sq
```

Figure 4: List of implemented instructions

`lq` and `sq` are just the 128 bits equivalent of `ld` and `sd` in 64 bits.

5 EXPERIMENTATION AND PERFORMANCE RESULTS

5.1 Examination of QEMU optimised micro operations and result hypothesis

Qemu, like a compiler, performs optimisation on the micro operation that will be generated. Therefore it is a lot more accurate and a lot more fair to compare the micro operation that will be actually executed, that is after optimisation, instead of the ones that the authors wrote.

Consider the instruction `add t0,zero,zero`

```
mov_i64 tmp2,$0x0
mov_i64 tmp3,$0x0
mov_i64 tmp4,$0x0
mov_i64 tmp5,$0x0
add_i64 tmp2,tmp2,tmp3
mov_i64 tmp8,$0x0
setcond_i64 tmp7,tmp2,tmp3,ltu
add_i64 tmp4,tmp8,tmp7
add_i64 tmp4,tmp4,tmp5
mov_i64 x5/t0,tmp2
mov_i64 x5/t0,tmp4
```

Figure 5: Translation of `add t0,zero,zero` in 128 bits before optimisations

```
mov_i64 x5/t0,$0x0
mov_i64 x5/t0,$0x0
```

Figure 6: Translation of `add t0,zero,zero` in 128 bits after optimisations

```
mov_i64 tmp2,$0x0
mov_i64 tmp3,$0x0
add_i64 tmp2,tmp2,tmp3
mov_i64 x5/t0,tmp2
```

Figure 7: Translation of `add t0,zero,zero` in 64 bits before optimisations

```
mov_i64 x5/t0,$0x0
```

Figure 8: Translation of `add t0,zero,zero` in 64 bits after optimisations

While the increase in micro operations necessary to perform 128 bits computing on a 64 bits host is sometime large (here we go from 4 micro operations in 64 bits to 10 in 128 bits, which is more than doubling), `QEMU` might be able to reduce it drastically trough optimisation (from 4 to 1 in 64 bits and 10 to 2 in 128 bits). Let's call Q the ration between the number of instruction in 128 bits and in 64 bits. Of course, the lower Q is the better and if $Q = 1$ then there is as much instruction in 128 bits that in 64 bits. In the end, for this specific instruction, we get from $Q = \frac{10}{4}$ in the unoptimized micro operations to $Q = \frac{2}{1}$ in the optimized ones which is a slight but significant decrease.

Of course it is important to keep in mind that, here the optimisation is huge, because of the nature of the operation (adding 2 zeros and storing the result in another register). The result will not be the same depending not only on the instruction but also on its "arguments".

Analysing all the relevant implemented instructions, that is `add`, `sub`, `sll`, `slt`, `lq`, `sq`, on the assembly code for our bubble sort (see section 5), we get that on average Q is 2.154789915966387 for unoptimized micros operations and 2.2989130434782608 for optimized micros operations.

Therefore, our work hypothesis is that 128 bits should take approximately $Q = 2.2989130434782608$ time instructions than 64 bits. For the sake of simplicity we can take the approximation $Q = 2$. This hypothesis is also perfectly coherent with the fact that we are doubling the number of bits.

We will check, trough experimentation, this hypothesis in the following section.

5.2 Sorting an array of 128 integer

Our experiment consisted in the sorting of array of integer using the bubble sort algorithm (see figure 12).

They where run a PowerEdge R710/0G7WYD machine with 4 Intel Xeon X5690 processors that include 6 cores each, for a total of 24 cores. The cores run at 3.47GHz, and have a 192 kB instruction and 192 kB data L1 caches, a unified 1.5 MB L2 cache, and a shared 12 MB L3. This machine has 79 gigabytes of memory. For additional information, you can see a dump of the `lscpu` command in figure 13.

The tool used to evaluate the performance is `perf`[13], and we run the experimentation alone on the computer. The data presented here is the average of 10 runs with the two most extreme value removed, to limit biases due to sporadic kernel tasks.

We have decided to present in the paper only what we believe to be the two most relevant plot, but the totality of the plots is available in Annex 8.1.

Every plot presented from now on is a linear plot.

For every plot a polynomial regression *pr*, computed using numpy's polyfit[3] function is provided. Sometime, the regression fits the data so perfectly that it overlaps it. That is the case with the instruction plot (9). Another thing of note is that most of theses polynomial regression have a huge coefficient in front of x^1 and x^0. The authors believe that this can be explained by the fact that the data, even for smallest array, never starts at zero, because it contains a constant factor, that is the translation step by QEMU and every other thing that QEMU could do.

5.3 Number of host instruction : 64 bits vs 128 bits

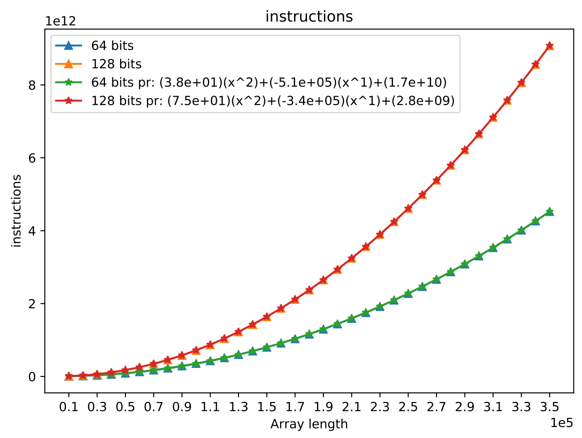

Figure 9: Number of host instruction with increasing array length

It is immediately apparent from figure 9 that 128 bits simulation on 64 bits computer will be much slower. In fact, at first glance it looks very much to confirm our hypothesis of $Q = 2$, especially if we look at the coefficient in front of the highest degree part (x^2) of our polynomial. For 64 bits this coefficient is 3.8, while it is 7.5 for 128. Therefore, asymptotically, $Q \approx \frac{7.5}{3.8} \approx 1.97$, for the case of instructions.

If we take a different approach to analysing this plot, ignore the polynomial regression, and instead look a a few point in the curve we find that:

- at array length 160000 the number of instruction is 1×10^{12} for 64 bits and 2×10^{12} for 128 bits.
- at 230000 the number of instruction is 2×10^{12} for 64 bits and 4×10^{12} for 128 bits
- finally at 280000 the number of instruction is 3×10^{3} for 64 bits and 6×10^{3} for 128 bits.

From this, it's easy to see confirmation of our earlier hypothesis that $Q = 2$, that is 128 bits necessitate twice as much work than 64 bits in Dynamic Binary Translation.

5.4 Number of cache misses : 64 bits vs 128 bits

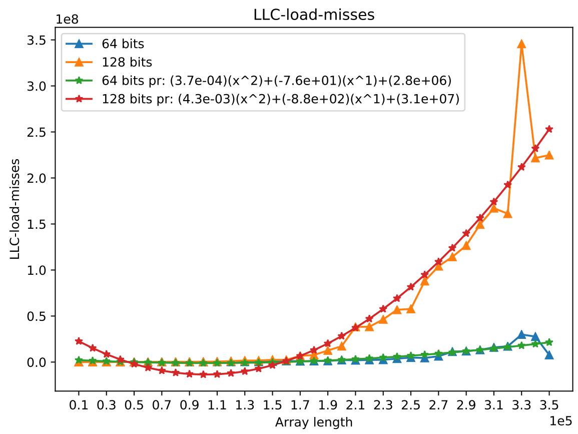

Figure 10: Number of last level cache misses with increasing array length

Figure 10 is much messier than the previously explained figure 9. However, we also see a clear pattern emerge that 128 bits simulation will cause more cache misses that 64 bits simulation and therefore be slower. Interestingly, it's only with array length 1.7×10^5 that the cache misses for 128 bits start to really take off, as they stay mostly constant before and on par with 64 bits. Because 64 bits caches misses stay constant the whole way, while 128 bits caches misses increase dramatically after a while, the ratio between 64 and 128 bits, Q, is not constant and therefore not equal to 2.

This means that the increase in number of caches misses does not linearly follow the increase in number of host instructions[6] (and of micro operations). Instead there is almost no increase before we reach a tipping point, after which the number of caches misses increase dramatically. This will have an impact on performance as we will see in section 5.5.

[6]This probably apply to program other than dynamic binary translation

5.5 Second elapsed : 64 bits vs 128 bits

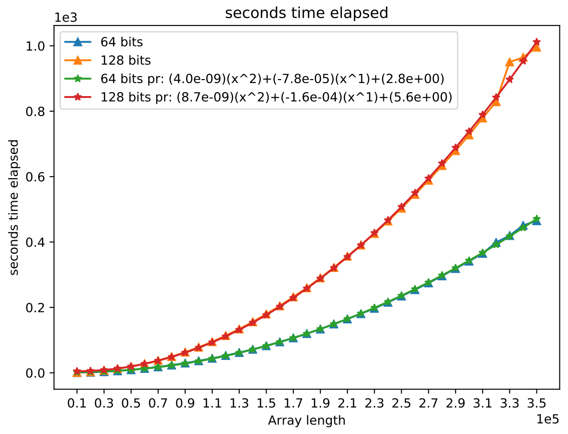

Figure 11: Runtime with increasing array length

Figure 11 resemble much the previously commented figure 9, where the ratio, $Q = 2$. However if we look closely we can see that the ratio of the polynomials regressions most significant coefficient, (in front of x^2) is a bit higher than 2, at $\frac{8.7}{4} = 2.175$.

To cross check, if we look at point on the curve we have:

- at array length 160000 the number of instruction is $\approx 0.1 \times 10^3$ for 64 bits and $\approx 0.2 \times 10^3$ for 128 bits.
- at 230000 the number of instruction is $\approx 0.2 \times 10^3$ for 64 bits and slightly above 0.4×10^3 for 128 bits, at $\approx 0.43 \times 10^3$
- finally at 280000 the number of instruction is a bit bellow 0.3×10^3 for 64 bits, at $\approx 0.29 \times 10^3$ and a bit above 0.6×10^3 for 128 bits, at $\approx 0.63 \times 10^3$.

The point is that Q for the run time is a bit above 2, while for the instruction it was exactly 2. The effect size is undeniably minimal but still very much present. A likely explanation is that the caches misses are starting to have a negative impact on performance as well, as we see in figure 10, they are much more important in 128 bits. The run-time ratio is expected to get worse asymptotically.

6 WORK THAT REMAIN TO BE DONE : ADDRESS SPACE AND OTHER CONSIDERATION

6.1 Address space

As you surely know, one of the key advantage of 128 bits computing is the growth in address space. However the work presented in this paper did not implement a 128 bits address space. While we saw in section 2.1 that such an address space is still unnecessary, we believe this is still a very important topic and would like to present our reflection on the usage of a 128 bits address space. This reflection is general and not necessarily applied to QEMU.

Page table. First, a reflection on the usage of page table will have to be conducted. Indeed, with 5 level of page table on 64 bits, with a lowest size of 2^{12} bytes we already have a theoretical maximum page size of $2^{12} \times (2^9)^4 = 2^{48} \approx 3 \times 10^{14}$ bytes $\approx$ 300 TeraBytes, which is already unnecessary for > 99% of computers. This size is achieved by having multiple level of page table, each addressed on 9 bits. If we keep following this approach, we can reach an upper limit page size, of $2^{12} \times (2^9)^12 = 2^{120} \approx 1 \times 10^{36}$. This is even more extreme for almost all applications. Traversing the page table hierarchy could become costly if it keeps increasing in size. Perhaps would it preferable to have less level in this hierarchy, for example using 32 or 64 bits for the top level. This would depend on the application and has to be discussed.

Other use for the additional bits. Because a 128 bits address space will remain unnecessary for the foreseeable future, it would be relevant to try to use the additional bits for something else. This approach has been explored before in the cheri machine[20]. In this extension for the 64 bits MIPS architecture, additional bits are associated with each pointer to hold memory protection capacities.

6.2 Other consideration

One thing that is obviously necessary, outside of the 128 bits address space is to implement the remaining instruction, outside of the subset implemented for sorting the array.

Another thing would be further testing of the implemented instructions, for example, the authors note that the sub instruction does not behave at it should. Related would be to the clean the source code so that it approaches QEMU coding stlye [4]

Finally, if all theses step are completed, a very thing interesting to do would be to try to optimize the instruction so that as little as possible micro operations are used per instruction. For example, for the `sll` instruction, which implementation is detailed in 4.2, there exist a better performing algorithm, as in less micro operations, presented in the book [18]

7 CONCLUSION

In this paper we show our successful implementation of a partial 128 bits RISC-V support in QEMU for 64 bits target. We highlight the performance impact that it had and find that a factor of 2 is to be expected at least, and that factor could get higher asymptotically because of the caches misses that are much more present in 128 bits, compared to 64 bits. In general, according to our estimations, this factor 2 is to be expected on real hardware has well, unless other optimisation techniques are used, this time concerning power consumption. Additionally, we detail the insight, that we have acquired during this work, on what would be the next step to complete 128 bits RISC-V support in QEMU (*e.g.* booting an operating such as Linux). The "biggest" next step that is necessary is to create a 128 bit address space. While we do not implement it, we do point to the fact that it is completely unnecessary for now, but we also show how the additional bits could be used, for security purpose for example and how the page table architecture might need to be changed for a 128 bits computer.

8 ANNEX

```
main:
    add t0,zero,zero
    la t3,donnee_programme
    la a0, tableau
    lq(t1,0 * TYPESIZE,t3)
    lq(a1,1 * TYPESIZE,t3)
    lq(t2,2 * TYPESIZE,t3)
    lq(t5,3 * TYPESIZE,t3)

for1_opt:
    # sub s0,a1,t1
    slt s0,t0,t5
    beqz s0, fin_for1_opt
    add s1,t0,zero
    add s2,t0,t1
for2_opt:
    slt s0, s2,a1
    beqz s0, fin_for2_opt
    sll t4,s2,t2
    add t4,t4,a0
    lq(t4,0,t4)
    sll s3,s1,t2
    add s3, s3,a0
    lq(s4,0,s3)
    slt s5,t4,s4
    beqz s5,fin_if_opt
    add s1,s2,zero
fin_if_opt:
    add s2,s2,t1
    j for2_opt

fin_for2_opt:
    sll s5,t0,t2
    add s5,s5,a0
    lq(t4,0,s5)
    sll s3,s1,t2
    add s3,s3,a0
    lq(s4,0,s3)
    sll s5,t0,t2
    add s5,s5,a0
    sq(s4,0,s5)
    sll s3,s1,t2
    add s3,s3,a0
    sq(t4,0,s3)
    add t0,t0,t1
    j for1_opt
fin_for1_opt:
```

Figure 12: Bubble sort in RISCV

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
Address sizes:       40 bits physical, 48 bits virtual
CPU(s):              24
On-line CPU(s) list: 0-23
Thread(s) per core:  2
Core(s) per socket:  6
Socket(s):           2
NUMA node(s):        2
Vendor ID:           GenuineIntel
CPU family:          6
Model:               44
Model name:          Intel(R) Xeon(R) CPU           X5690  @ 3.47GHz
Stepping:            2
CPU MHz:             1596.006
BogoMIPS:            6915.55
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            12288K
NUMA node0 CPU(s):   0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s):   1,3,5,7,9,11,13,15,17,19,21,23
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est
tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 popcnt aes lahf_lm pti
tpr_shadow vnmi flexpriority ept vpid dtherm ida arat
```

Figure 13: lscpu dump

8.1 Experiment data and plots

All plots are, just like in the main section, in function of the size of the array. Figure 16, 14 18, 19, 21, 20, 23 and 24 all follow roughly (without explicit calculation of the multiplicative factor) the same pattern as figure 9, 11 and 10 which were explained in the main document.

Figure 15 and 22 present a situation where 64 bits and 128 bits are much closer to each other, but 128 bits still comes on top a visible margin.

Finally figure 17 is probably not relevant, as the cpu migration should be 0 if the computer is perfectly free when we run the test. An hypothesis to explain this figure is that another task arrived on the computer, even though we tried to make sure it was completely free.

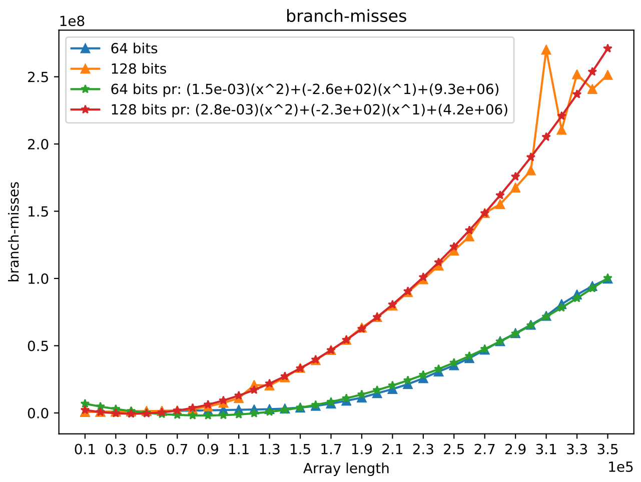

Figure 14: Branches misses

This is the number of branches misses. This refers to the branch prediction mechanism present in modern day processors[7]

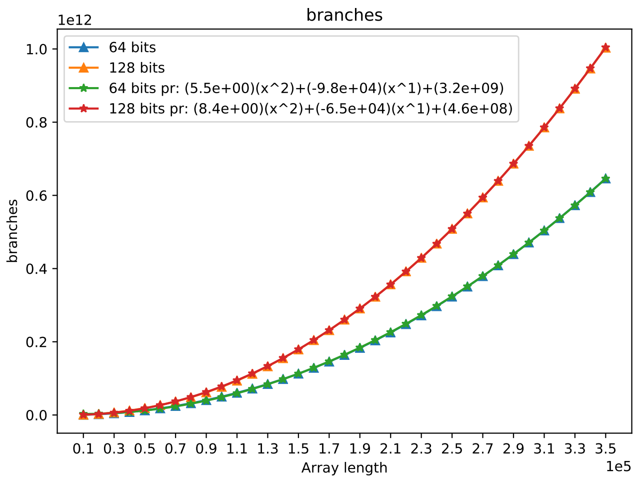

Figure 16: Branches

This is the number of branches that were taken as QEMU is being executed.

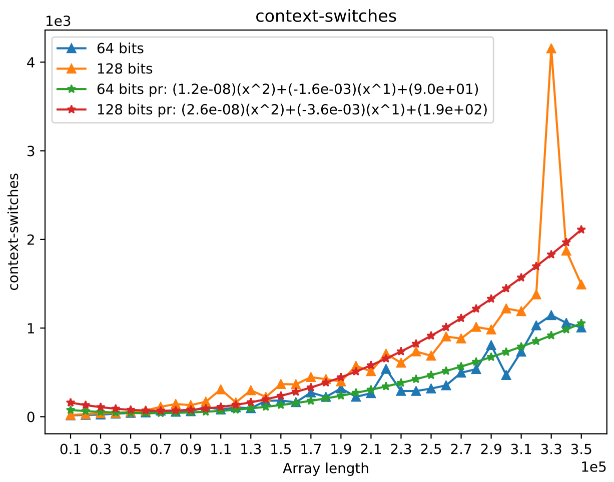

Figure 15: Contexts switches

This is the number of context switches. A context switch takes place every time the processor is assigned a new task.

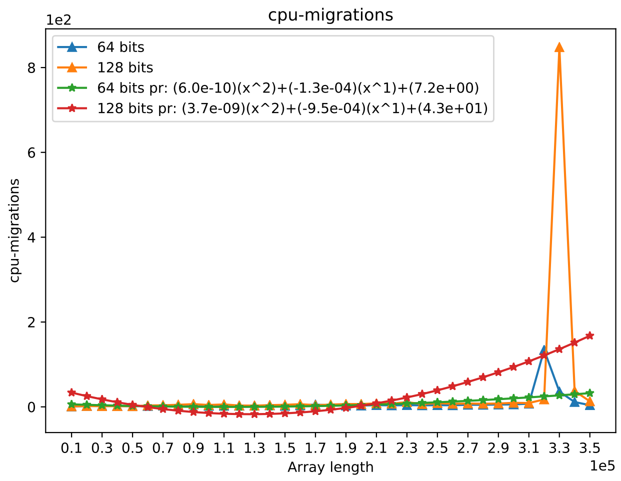

Figure 17: cpu migrations

This is the number of time the QEMU process was moved across different processing core. This movement is done in Linux to balance the load across processing cores.

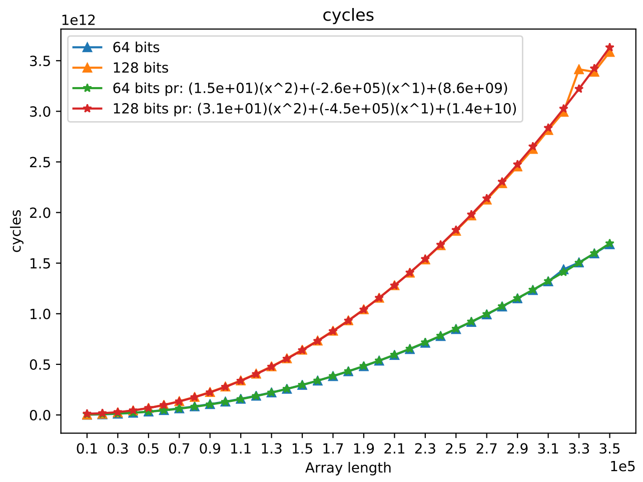

Figure 18: Cycles

This is the number of processor cycle that were necessary to perform the sort.

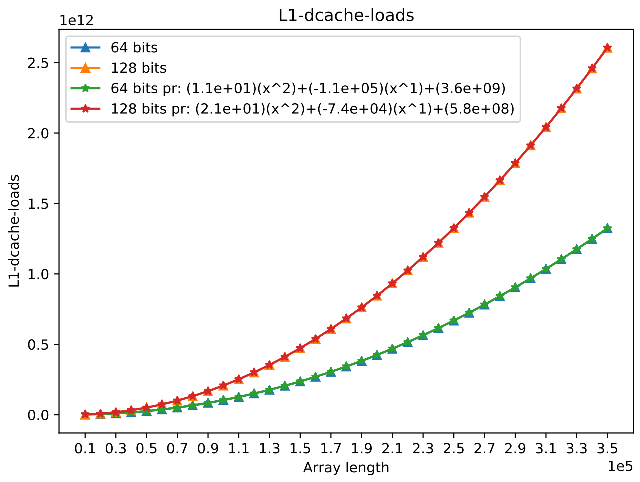

Figure 20: L1-dcache-loads

The number of load from the level 1 data cache of the processor.

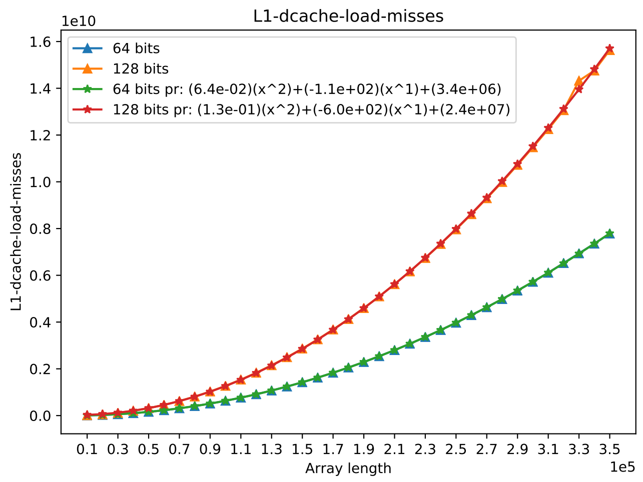

Figure 19: L1-dcache-load-misses

The number of caches misses for the level 1 data cache of the processor.

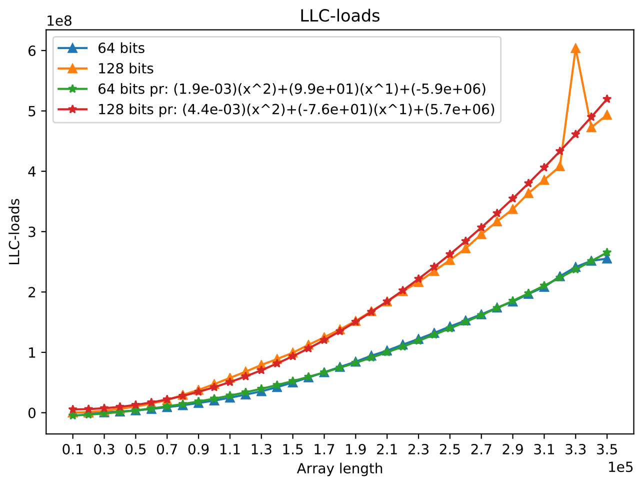

Figure 21: LLC-loads

The number of load from the last level cache of the processor.

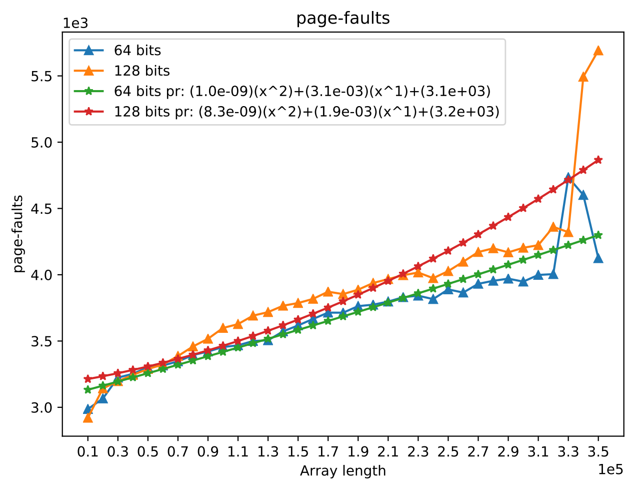

Figure 22: page-faults

The number of page fault, that is when a page is not in main memory when it is demanded by a program. The page will have to be brought from disk to main memory.

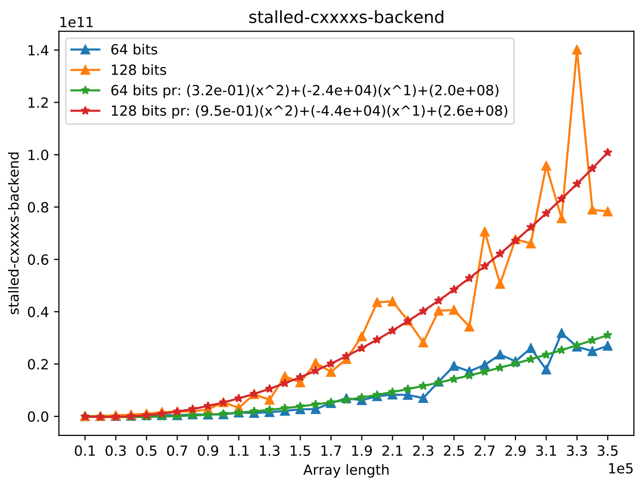

Figure 23: stalled-cxxxxs-backend

The number of stalled cycle in the back-end, that is cycle where the processor cannot perform any work /instruction, because it's waiting on the result of another operations, such as a load from memory.

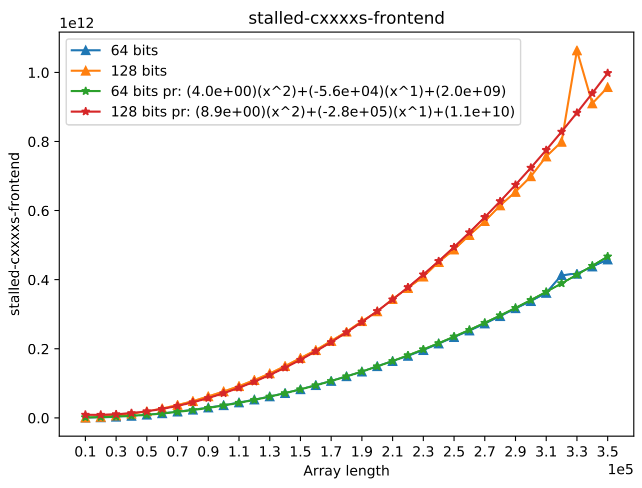

Figure 24: stalled-cxxxxs-frontend

The number of stalled cycle in the front-end, that is cycle where the processor cannot perform any work /instruction, because their is no instruction ready to be executed (they haven't been decoded, or were not in cache for example).

8.2 Technical QEMU details

In this section file path are given starting from the root of the source code of QEMU.

For QEMU to be able to process an instruction in a compiled RISCV binary :

- The instruction OPCode needs to be described in one of the `.decode`[1] files located in `/target/riscv/`. See for example the figure 25 bellow. Essentially the "`.`" act as a wild card that will hold the instruction's arguments. The bits strings are the opcode of the instruction. The "`@`" indicate the format of the instruction.
- A "handler function" which uses QEMU's tiny code generator must be written in the appropriate `.c.inc file` located in `/target/riscv/insn_trans/` file. Once the instruction is described in a .decode file, QEMU will automatically generate a stub in the correct `.c.inc` file, which means that the developer just has to fill theses handler function.

```
lq        ............   ..... 010 ..... 0001111 @i
sq        ............   ..... 100 ..... 0100011 @s
```

Figure 25: A customized entry in a .decode file

The handler function is written in C but not every feature of the language is available to the programmer. Indeed it is important to keep in mind that this handler function will generate QEMU micro-operation. So writing theses handler is closer to assembly programming, with a few quirks added by QEMU, than typical high level programming.

```
cpu_gprh[i] = tcg_global_mem_new(cpu_env,
offsetof(CPURISCVState, gprh[i]), riscv_int_regnames[i]);
```

Figure 26: initialisation of cpu registers

Implementing instructions. The instruction lq and sq are apart from the rest as they have new names and opcodes. For them it was necessary to write a new entry in a decode file[7]. The other instructions are acting as replacement for the 64 bits instruction that already exists under the same name and opcodes.

Therefore, for theses already presents instruction, the only work necessary was to modify the `.c.inc` file. In our case we only edited the `/target/riscv/insn_trans/trans_rvi.c.inc` and the `/target/riscv/insn_trans/trans_rvq.c.inc` files. As you might have guessed `trans_rvi.c.inc` corresponds to the base integer set of instruction that is mandatory for any RISC-V implementation, while `trans_rvi.c.inc` corresponds to the standard quad precision extension

ACKNOWLEDGMENTS

Grenoble INP, Fabrice Bellard, Frédéric Petrot

REFERENCES

[1] [n.d.]. Decodetree Specification¶. https://qemu.weilnetz.de/doc/devel/decodetree.html

[2] [n.d.]. Home. https://www.top500.org/

[3] [n.d.]. numpy.polyfit — NumPy v1.20 Manual. https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html

[4] [n.d.]. QEMU Coding Style¶. https://qemu-project.gitlab.io/qemu/devel/style.html

[5] [n.d.]. RISC-V Instruction Set Specifications¶. https://msyksphinz-self.github.io/riscv-isadoc/html/index.html

[6] [n.d.]. Welcome to Python.org. https://www.python.org/

[7] 2021. Branch predictor. https://en.wikipedia.org/wiki/Branch_predictor

[8] 2021. Unit in the last place. https://en.wikipedia.org/wiki/Unit_in_the_last_place

[9] ANSI/IEEE 754 1985. *IEEE Standard for Binary Floating-Point Arithmetic*. Standard. Institute of Electrical and Electronics Engineers.

[10] Marie Badaroux and Frédéric Pétrot. 2021. Arbitrary and Variable Precision Floating-Point Arithmetic Support in Dynamic Binary Translation. *Proceedings of the 26th Asia and South Pacific Design Automation Conference* (2021). https://doi.org/10.1145/3394885.3431416

[11] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*. 41–46.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[13] Arnaldo Carvalho De Melo. 2010. The new linux 'perf' tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[14] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2 (June 2007).

[15] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. 2009. Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In *7th IEEE/ACM Int. Conf. on Hardware/Software Codesign and System Synthesis*. 71–80.

[16] Inc. OpenSSL Foundation. [n.d.]. OpenSSL. https://www.openssl.org/

[17] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (Vancouver, BC, Canada) *(SLE 2017)*. Association for Computing Machinery, New York, NY, USA, 256–267. https://doi.org/10.1145/3136014.3136031

[18] Henry S. Warren. 2013. *2.17.* Addison-Wesley.

[19] L. Wehmeyer, M.K. Jain, S. Steinke, P. Marwedel, and M. Balakrishnan. 2001. Analysis of the influence of register file size on energy consumption, code size, and execution time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 11 (2001), 1329–1337. https://doi.org/10.1109/43.959862

[20] Jonathan D. Woodruff and C Jonathan D. Woodruff. 2014. CHERI: A RISC capability machine for practical memory safety.

[7]For simplicity, we wrote them in /target/riscv/insn32-64.decode, even though that's not their correct location