A Halt-Averse Instruction Set Architecture for Embedded Hypercomputers

Ben Burrill [0x7F6, 0x76D]
California State University, Northridge

April 0, 2023

Abstract

In this work, we aim to bring a few of the more practical performance benefits of hypercomputers to embedded applications by limiting ourselves to a computational model—an ISA with a Turing jump instruction—which is not even Turing complete. Nevertheless, it is capable of general-purpose computation, and is semi-optimized for oracle-oriented programming (OOP). It may be emulated on traditional processor architectures, albeit at a slight performance penalty, using the emulator made freely available at https://github.com/benburrill/sphinx.

1 Introduction

In low-power embedded environments, the demands of unrestricted hypercomputation can pose some technical complications. To provide limited hypercomputation for embedded applications, we present Sphinx, a halt-averse instruction set architecture. While not strictly hypercomputational, nor even Turing complete, Sphinx provides some of the amenities familiar to hypercomputer programmers.

The conditional-branch instructions of conventional architectures are replaced by a Turing jump instruction, which offers limited insight into the machine's own halting problem. Specifically, a jump is performed if not jumping would lead to halting. Conditional execution is enabled through use of conditional-halt instructions.

From a performance standpoint, programs running on a Sphinx architecture can send half a bit of information back in time, allowing algorithms to optimize away certain special cases, while preserving worst-case time complexity.

From a backwards compatibility standpoint, Sphinx may be emulated even on non-hypercomputational devices.

From a programmer's standpoint, Sphinx combines the readability of assembly languages with the traceability of logic programming languages.

2 Theory

2.1 Halting problems

Famously, the halting problem of Turing machines is undecidable. If an algorithm existed to solve the halting problem of Turing machines, you could program a Turing machine to run this algorithm on itself, and then do the opposite of whatever it says. Since Turing machines do not exist, this bears no real-world relevance, and need not concern us in any way, so long as we choose to willfully reject Turing machines from our thoughts (Murphy VII 2008).

Many machines—including all real-world computers—have halting problems (excluding I/O) that *are* decidable. A trivial algorithm exists to solve the halting problem for machines with finitely many possible configurations in finite time (Simmons 2021).

However, in stating that the halting problem of Turing machines is undecidable, we did not directly invoke the fact that Turing machines have infinitely many possible configurations, instead making use of the relationship between decidability and computability-on-a-Turing-machine. More generally, this can be extended into a statement about self-computability:

Theorem 1. *Computers*[1] *cannot solve*[2] *their own halting problem*[3]*, even if their halting problem is decidable.*

2.2 Semi-semi-decision procedures

One the better workarounds for the annoying restrictions due to Theorem 1 is the concept of semi-decidability. The halting problem of Turing machines is semi-decidable, and more generally halting problems are semi-self-computable. That is, there exists an algorithm implementable on machine M that will tell you, in finite time, whether any halting program for machine M will halt. For non-halting programs, this algorithm will loop indefinitely. Formally, this algorithm is known as "running the program", which we shall abbreviate to SDPSHP as the semi-decision procedure for the self-halting problem.

The Sphinx architecture allows the SDPSHP algorithm to be optimized and made more user-friendly. A program $h(p)$ can easily be written for a Sphinx machine, S_n with n bytes of state, that will *immediately* produce output indicating whether or not any program p running on S_n would halt.[4]

The slight of hand here is that $h(p)$ is actually still just a semi-decision procedure with the same termination behavior as SDPSHP—it will only terminate if p would. However, unlike SDPSHP which runs p in all cases, $h(p)$ may be implemented using Sphinx's Turing jump instruction so that

[1] As in, something vaguely capable of general purpose computation
[2] As in, determine in all cases eg. a boolean value that it can use
[3] Assuming its halting problem isn't trivial–like if it can never halt
[4] $h(p)$ might require more *instructions* than p by a constant amount, depending on how you implement it. By considering only bytes of *state* we are neglecting this. That alone doesn't protect Sphinx from the corrupting influence of logic, since we can still write a program that tests itself in much the same way as we can test any other program. See examples/self_inquiry.s

it *only* runs p for the $(1 - \Omega) \times 100\%$ of all programs that will not halt.[5] If p would halt, it is never run.

Although characteristically identical to the SDPSHP, there are some practical advantages to $h(p)$. If we want $h(p)$ to output to the user "p will not halt" only in the case that p will not halt, we can do so by simply prepending that code to p since it won't be run at all if p would halt. Since such prepended code is part of what is being tested, it is not contradictory to prepend a halt—it would simply invalidate the test, just as it would for the SDPSHP.

Since it is a somewhat less semi- version of a semi-decision procedure, we describe this as a semi-semi-decision procedure. Unlike the SDPSHP algorithm, $h(p)$ makes a decision straight away. To an outside observer, the problem is fully decided. However, the program itself obtains no more usable information than could eventually be found by any other semi-decision procedure. In this sense, Sphinx cannot solve its own halting problem, it only looks like it does.

A formal semi-proof is provided in Appendix A for picky readers who demand more rigor.

2.3 The Turing jump operator

According to the entry on Wikipedia, the Free Encyclopedia (Wikipedia contributors 2023), the Turing jump operator is defined on "sets" (presumably hash sets). The Turing jump X' of the lookup table X is a new lookup table, indexed by Gödel's perfect hash function φ, such that $\forall p : X'[\varphi(p)]$ is true iff the program p would halt given access to the lookup table X.

Unlike many mathematical operators such as addition, hardware support for the Turing jump operator has historically been lacking. Traditional oracle-equipped hypercomputers are based around precomputed Turing jump tables. Although this approach works well for many hypercomputational systems, microcontrollers typically do not have the available resources to store such large lookup tables.

3 Design of the Sphinx ISA

3.1 The Turing jump instruction

Sphinx's Turing jump instruction `j` operates on the principle of self-preservation against future halts. A jump is only performed if not jumping would lead to halting. Like all Sphinx instructions, the Turing jump instruction is specified to take one clock cycle.

It is also important to explicitly specify that the jump instruction is deterministic. At a repeated state, the jump instruction must always make the same decision. Since a jump is *only* to be performed if not jumping would lead to halting, there is never any ambiguity here[6] as to which deterministic decision should be made.

Sphinx's jump instruction eschews true hypercomputation in favor of cold hard pragmatism. It does not offer as much power as a conventional Turing jump table (Section 2.3), from which one may obtain a full bit of information about whether some code would halt. With Sphinx one may only obtain half a bit of information. You're forced to actually do the work to get the other half of the bit in the case that not jumping would not lead to a halt.

3.2 The state vector

Instead of separating registers and main memory, the Sphinx ISA is based around a unified concept of state. That is, it is a memory-only architecture—there are no registers. Instead, immediately-addressed state values may be used directly in instructions, much like registers. There are load/store instructions for dynamic state access. Additionally, there is a read-only memory section called const.

Most instruction arguments can be any of the following:

1. Immediate values
2. An immediate address to look up a value from state
3. An immediate address to look up a value from const

Syntactically, `yield 0` outputs[7] the immediate value 0, `yield [0]` outputs the value loaded from address 0 in state, and `yield {0}` outputs the value loaded from address 0 in const. Notice that there is no unified address space, 0 may separately be a state address and a const address.

The `code` section is separate from state and is inaccessible (Sphinx is a Harvard architecture). The program counter is also not in the `state` section, it is an inaccessible pseudo-register. We'll define the "complete state" as the combination of the program counter and `state` section.

3.3 Halt instructions

The `halt` instruction is an unconditional halt.

Additionally, there are an assortment of conditional halt instructions: `heq`, `hne`, `hlt` (not to be confused with `halt`), `hgt`, `hle`, and `hge`. These perform signed comparisons. For example `heq [x], 0` will halt execution if the word in state at label `x` is equal to 0.

3.4 Word size

The Sphinx specification does not mandate a particular word size. If your code requires a 3 byte (24-bit) word size, the preprocessor command `%format word 3` can be used. Word offset literals may be used in the assembler. For example, with 3 byte words, `2w` = 6. The infinite word size (Section 5.1) may be specified with `%format word inf`.

4 Sphinx Programming

4.1 Terminal non-termination

Prophecy 1. *If you read this paper, you will eventually die*

Prophecy 1 is pretty much worthless unless you had some expectation of immortality. If that was the case for you, I'm very sorry, it's too late now—maybe should've led with that.

For the same reason, most Sphinx programs are designed to be non-terminating in order to make effective use of the prophetic wisdom provided by the Turing jump instruction.

[5] TODO: Plug in the value of Chaitin's constant for Ω

[6] Hopefully anyway. See Appendix A if you have any questions.

[7] Sphinx provides generic output capabilities through `yield`, but does not specify any means to read real-time input from peripherals.

Although non-termination is often a very desirable feature (Simmons 2021), occasionally one wants to write a program that, after completing its primary computational task, indicates that it is done and performs no additional work. To do this while maintaining immortality, the standard technique in Sphinx is to simply produce a `done` flag and then enter into an infinite loop.

This pattern is called **terminal non-termination**. An example may be seen in the following code:

```
1  flag done
2  tnt: j tnt
3  halt
```

Not jumping on line 2 would clearly lead to the subsequent unconditional halt, so the jump is unconditional.

4.2 Example: max value

This code searches for the maximum value of an array, jumping to `found_max` as soon as the maximum value is first encountered:

```
1  mov [max_val], {arr}
2  j found_max
3
4  mov [addr], arr + 1w
5  loop:
6  hge [addr], end_arr
7      lwc [cur_val], [addr]
8
9      j continue
10     hle [cur_val], [max_val]
11
12     mov [max_val], [cur_val]
13     j found_max
14
15     continue:
16     add [addr], [addr], 1w
17 j loop
18 halt
19
20 found_max:
21 yield [max_val]
22 flag done
23 tnt: j tnt
24 halt
```

Running this code under the Sphinx emulator using the array [9, 8, 7, 6, 5, 4, 3, 2, 1, 0] produces the following output:

```
$ spasm max_value.s
9
Reached done flag
    CPU time: 4 clock cycles
    Emulator efficiency: 5.80%
```

Nevermind the "emulator efficiency", that is just an implementation detail of the emulator. The point is that it found the maximum value of an unstructured array of 10 elements in just 4 clock cycles!

How does it work? To understand the code, let's first recognize that reaching `found_max` guarantees non-termination (Section 4.1). The loop is set up to halt if we reach the end of the array (line 6). Once in the loop, line 13 presents the only opportunity to avoid a halt by jumping to `found_max`. Lines 9 and 10 force lines 12 and 13 to be skipped unless `cur_val > max_val`. So we only have the opportunity to jump to safety whenever we reach a value larger than any previously encountered. The jump on line 13 will only be taken if not jumping would lead to halting, i.e. if no larger value would be found. So the code breaks out early as soon as the maximum value is encountered.

4.3 Time complexity

It's worth noting that the worst-case time complexity of our prophetic max-value algorithm (Section 4.2) is the same $O(n)$ as in the traditional algorithm. It is also only able to match the time complexity of search algorithms of the kind presented by Freshman 2022 in the best-case, though it does not suffer from any loss in accuracy.

A true oracle-equipped hypercomputer could do even better, with an $O(\log(n))$ algorithm—since the index of the maximum value may be expressed with $\lceil \log_2(n) \rceil$ bits, the hypercomputational algorithm would be to simply use a halting oracle to determine each bit of the index.

However, the *best* measure of performance is obviously the *best*-case time complexity, where we get a perfect score of $O(1)$. Additionally, the $O(\log(n))$ algorithm would fail for finding the maximum value of linked lists with elements of arbitrary size, so in a more general sense, this algorithm is hypercomputationally optimal.

4.4 Halt propagation

In code that may lead to a halt, the form of conditional branch used in lines 9-10 of max-value (Section 4.2) can sometimes be problematic since you are testing both the condition *and* any future halt.[8] In cases where this is a problem, one strategy is to propagate halts backwards in time by halting on the complement of the condition you are testing if the branch is taken. That way, if a jump is performed for the "wrong" reason, the code will still correctly halt. This works well[9] so long as you want an entire block to be skipped if it would lead to a halt (which is usually the case). If you need halting code that actually runs all the way to the "real" cause of the halt, halt propagation would not be sufficient, and you may need a more sophisticated form of conditional branch, such as in examples/halting.s.

4.5 Conservation of magic

Interpreters may be written in Sphinx for languages with Sphinx-like halt aversion in a way that makes use of the halt-aversion provided by `j` (not just emulated as in Section 6.2). In principle, a self-interpreter should be possible.

See examples/sphinxfuck.s for a true interpreter for a brainfuck-style language that features halt-averse forwards and backwards jumps instead of the usual `[` and `]`.

[8] This is not a problem for max value since the jump on line 13 handles the case of a future halt.

[9] For an example making use of halt propagation in various interesting ways, see examples/decimal.s. It's a rather complicated one, but see for example the `done` label at the end.

5 Sphinx Extensions

This section specifies two extensions to the instruction set, which are made optional due to potential difficulties that may be encountered in their implementation, such as logical contradictions.

5.1 Infinite Sphinx

One obvious deficiency of Sphinx is that it is not strictly Turing complete. The problem is that Sphinx fundamentally has only finitely bounded state, and so cannot be used to replace microcontrollers with infinite on-chip memory. This can be remedied by introducing an infinite word size, so that words can represent any integer. Sadly this makes the jump instruction undecidable.

Beyond mere undecidability, there are also a few other details which complicate the implementation of Infinite Sphinx. Here we will specify requirements that must be followed by conforming Infinite Sphinx implementations:

1. Memory is organized into word cells, composed of bytes. Both sides of a word cell shall be addressable.
2. Integers shall be represented with an alternating-endian byte-order, where the least-significant byte is stored at the start of the word cell, and the next-least-significant byte is stored at the end, increasing inwards. There shall not be a most-significant byte.
3. Depending on the instruction in which it is used, a word of data may be interpreted as either an integer, an address, or a word-cell-offset.
4. The bijective mappings between integers, addresses, and word-cell-offsets are left unspecified, and hence pointer arithmetic is undefined behavior in most cases.
5. Addition using the `add` instruction shall be defined between addresses and word-cell-offsets, producing a corresponding address in an offset word-cell.
6. Multiplication using the `mul` instruction shall be defined to scale word-cell-offsets.
7. The offset argument of `lbso`, `lbco`, `lwso`, `lwco`, `sbso`, and `swso` instructions shall be interpreted as an integer to offset the address by the given number of bytes.

5.2 Suicide Sphinx

One may also contemplate a variant of `j`, which we shall call `k`, for which a jump is performed if not jumping would *not* lead to halting. That is, if `j` operates on the principle of self-preservation (Section 3.1), `k` is suicidally averse to immortality. As it turns out, although `j` and `k` are seemingly similar, `k` is logically problematic as an instruction.

Notice that the combination of `j` and `k` is very powerful:

```
1  j would_halt
2  k would_not_halt
3  ; Some code that might halt
```

It's not just powerful—it's too powerful for its own good! We've damn near violated Theorem 1, so it should be *trivial* to produce a contradiction from the above code.

Ok, turns out it's not actually so trivial[10] (or at least I couldn't manage it with this approach). But regardless, we can make an even stronger statement. It's not just the combination of `j` and `k` that is a problem, but `k` itself.

Theorem 2. *Suicide Sphinx is logically unsound*

Proof. We will show that the problem of getting up out of bed is reducible to Suicide Sphinx.

It is widely accepted that getting up out of bed is an undecidable decision problem, at least in the short-snooze limit (Me this morning). In the getting up out of bed problem, we desire to get some more sleep under the condition that we get up eventually.

In the following code we reduce this problem to Suicide Sphinx, using the `k` instruction to get up out of bed if and only if we otherwise never would:

```
1  alarm_goes_off:
2  k get_out_of_bed
3  sleep 15 * 60 * 1000
4  j alarm_goes_off
5  halt
6
7  get_out_of_bed: halt
```

Undecidability follows from the usual proof for the getting up out of bed problem: the decision of whether or not to jump on line 2 is predicated upon making the opposite decision in the same situation.

We only used `j` for an unconditional jump, something that's impossible with `k` alone, but would certainly be a necessary feature in any instruction set featuring `k`.

Therefore, we must reject the suicidal jump instruction from any logically consistent instruction set, regardless of its apparent utility. □

6 Implementation

6.1 Time travel

The most obvious way to achieve a constant-time jump instruction is through the application of time travel. However, although Sphinx itself is robust against paradox (Appendix A), user error might introduce bugs into your code, along with minor time travel paradoxes and possibly the destruction of the universe.

The simplest implementation would be to just make the processor physically impossible to turn off or destroy. This would mitigate the disasterous influence of user intent upon a program's behavior. However, if the ability to turn off and destroy the processor is a desired feature, these should not be treated as a halt, as that would cause all jumps to be unconditional (Munroe 2013).

6.2 The Sphinx emulator

The Sphinx assembler/emulator `spasm`, which may be obtained from https://github.com/benburrill/sphinx, is currently the only working implementation of Sphinx.

`spasm` uses a rather inefficient form of emulation that implements the jump instruction by simulating the future rather than predicting it outright.

[10] If `would_not_halt` halts it breaks the test

In the emulator, there is a concept of "virtual" and "real" execution, with the main difference being that no output is produced under virtual execution. Emulation begins in real execution. Once a jump is encountered, to determine if not jumping would lead to halting, the emulator performs a search for an infinite loop under virtual execution starting from `pc + 1`. If a path to an infinite loop is found, the entire future path of execution has been determined and is cached for use by all future jumps. Otherwise, if a halt would be reached, the emulator simply performs the jump.

In searching for an infinite loop, we keep track of the complete state of the program at jump points so that we may restore it in the event that not jumping would lead to halting, and also to test for repeated states.[11] If we reach a state has been visited previously, we have found an infinite loop and may unwind the stack producing a (cyclic) list of all jump decisions that were taken leading to the loop.

If the state has not been previously visited, we recursively attempt to skip the jump. If that leads to a halt, we try taking the jump instead. If that also halts, we unwind to the previous jump point.

In the worst-case, the number of previous states that must be kept track of scales exponentially with the amount of state. This is usually totally fine, but there are certain applications for which this may be problematic.

In computer graphics, it can sometimes be necessary to produce a delay lasting until the heat-death of the universe (Oh et al. 2022). One possible Sphinx port based on the Python implementation of Oh's algorithm is shown:

```
; Use a 336-bit word size
%format word 42
%section state
counter: .word 10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪ 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪ 000000000000000000

%section code
; Play an animation
; TODO

; Wait until heat-death of universe
loop:
    sleep 365 * 24 * 60 * 60 * 1000
    sub [counter], [counter], 1
j loop
hgt [counter], 0

; After the heat-death of the universe, enter
↪ terminal non-termination
flag done
tnt: j tnt
halt
```

In and of itself, this code performs well under emulation. However, if the animation needs to predict the future, there would be some issues. Specifically, if any jump instruction during the animation ever needs to be skipped, the entire heat-death loop would need to be fully predicted, which may result in unacceptably poor performance under emulation.

The minimum system requirements for available memory to play such an animation with a skipped jump under `spasm` may be conveniently expressed using the newly-adopted quetta- SI prefix as being on the order of 1 TQQQB. Using an estimate based on the Steam hardware survey as of February 2023 (Valve Corporation 2023), this would exceed the capabilities of at least 99.1% of personal computers.

Virtual execution in the emulator does not sleep, so prediction of each iteration of the loop would be much faster than in reality. As a result, although there might be a bit of a delay somewhere during the animation, it would only be a small fraction of eternity, leaving plenty of time for the animation to play. So the main concern is the memory use.[12]

The real problem with memory efficiency of course is that the emulator is written in Python, which incurs an outrageous memory overhead. Python's `bytes` objects alone have an overhead of 33 bytes. Do you know how much you can fit in 33 bytes? That's like 80% of what you need for the heat death of the universe. Python could include this remark!

7 Future Work

I was hoping to get away with just implementing Sphinx for my compilers class this semester, but unfortunately the language we create is supposed to have superfluous features like a type checker. So that plan backfired and I have been suckered in to implementing both Sphinx (done-ish) *and* a high-level oracle-oriented language that compiles to Sphinx (so far, I have implemented a lexer for it). Hence there is much need for an excuse for why I inevitably can't finish the code generator for this high-level language at the end of the semester. With the logical soundness of Sphinx assured (Appendix A) it is unclear what form this excuse could take.

The separation of `code`, `state`, and `const`, along with Sphinx's adjustable word size, complicates the categorization of its busy beavers. A von Neumann Sphinx with a fixed word size as in Simmons 2021 might be more advantageous to the study of bewitched breadboard busy beavers.

Ever since some teenager recklessly breached the gates of hell (Szewczyk 2020), there has been a pressing need for a stronger Malbolge. Increasing its cryptographic strength would likely be the most effective approach, but incorporating Turing jump instructions into the design might also help to increase the space complexity of producing working programs for it.

To further develop the embedded hyperprocessor, one clear goal would be to determine how much more we can gently tickle the halting problem without getting bit. Can we construct even semier-semi-semi-decision procedures?

8 Conclusions

Jeez did I write all that? Well uh, pretty much, Sphinx is quite cool. Go play around with it I guess—maybe write a time-traveling merge-sort or something.

[11] We actually only check for repeated states at upwards jumps.

[12] At least in this case. The time complexity of emulated jumps is worst-case super-exponential, which should make them an asymptotically far stronger hardware decelerator than prior NaN-gate techniques (Jones 2020). In practice however, I find memory use—though merely exponential—to be the bigger concern unless you add pointless jumps.

References

Freshman, A College (2022). "A sometimes-accurate O(1) search algorithm". In: *A Record of the Proceedings of SIGBOVIK 2022*. The Association for Computational Heresy, p. 223.

Jones, Cassie (2020). "NaN-Gate Synthesis and Hardware Deceleration". In: *A Record of the Proceedings of SIGBOVIK 2020*. The Association for Computational Heresy, pp. 61–65.

Me (this morning). *What time is it? Urgh my head is spinning. I'm just going back to sleep.*

Munroe, Randall (2013). *Halting Problem*. URL: https://xkcd.com/1266 (visited on 03/20/2023).

Murphy VII, Tom (2008). "A non-non-destructive strategy for proving P = NP". In: *A Record of the Proceedings of SIGBOVIK 2008*. The Association for Computational Heresy, pp. 13–15.

Oh, Braden et al. (2022). "Solving Double Execution of Java's paint() Method by Counting Down to the Heat Death of the Universe (plus language compendium)". In: *A Record of the Proceedings of SIGBOVIK 2022*. The Association for Computational Heresy, pp. 111–137.

Simmons, Robert J. (2021). "Build your own 8-bit busy beaver on a breadboard!" In: *A Record of the Proceedings of SIGBOVIK 2021*. The Association for Computational Heresy, pp. 278–281.

Szewczyk, Kamila (2020). *MalbolgeLISP v1.2*. URL: https://github.com/kspalaiologos/malbolge-lisp (visited on 03/21/2023).

Valve Corporation (2023). *Steam Hardware & Software Survey*. URL: https://store.steampowered.com/hwsurvey (visited on 03/21/2023).

Wikipedia contributors (2023). *Turing jump — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/w/index.php?title=Turing_jump&oldid=1134853969 (visited on 03/19/2023).

Appendix A Proofs

In this section I will endeavor to beguile, guilt-trip, and if necessary hold hostage all those who question the logical soundness of Sphinx.

Theorem 3. *Sphinx is logically sound*

Semi-proof. Here we will establish the logical soundness of Sphinx by recursive Gish gallop.

Potato. Can you refute that one? Nope didn't think so.

Consider the following code:

```
1  %section state
2  is_sphinx_broken: .word 0
3
4  %section code
5  j would_halt
6  flag sphinx_is_perfect
7  loop: hne [is_sphinx_broken], 0
8  j loop
9  halt
10
11 would_halt: flag sphinx_is_broken
```

I am running this code with the Sphinx emulator under `pdb`. Let us suppose that I will be informed of a contradictory Sphinx program or similar problem that you discover. I would then use `pdb` to change the value of the word at `is_sphinx_broken` in state. This would cause a halt on line 7, trigger an `AssertionError` in `Emulator.step`, break my heart, and destroy the universe[13]—thereby alerting my past self to the issue, since the jump on line 5 would need to have been taken. The output of this program is as follows:

```
$ python3.11 -m pdb -c c -m spasm test_sphinx.s
Reached sphinx_is_perfect flag
```

Since the universe exists last I checked, this clearly demonstrates that no problems with Sphinx will ever be found.

If you're still not convinced that Sphinx is logically sound (or if you're convinced that it is not), read the semi-proof of Theorem 3 before continuing.

Therefore, Sphinx is logically sound. □

[13] As a corollary, this means that you would be responsible for the destruction of the universe, you monster.