Rebutting the Sipser Halting Problem Proof

MIT Professor Michael Sipser has agreed that the following verbatim paragraph is correct (he has not agreed to anything else in this paper):

(a) If simulating halt decider H correctly simulates its input D until H correctly determines that its simulated D would never stop running unless aborted then (b) H can abort its simulation of D and correctly report that D specifies a non-halting sequence of configurations.

A simulating halt decider computes the mapping from its input finite strings to an accept or reject state on the basis of the actual behavior specified by this input as measured by its correct simulation of this input.

We start with Sipser's definitions of H and D:

> On input ⟨M, w⟩, where M is a TM and w is a string, H halts and accepts if M accepts w. Furthermore, H halts and rejects if M fails to accept w. In other words, we assume that H is a TM, where
>
> H(⟨M,w⟩ = { accept if M accepts w
> { reject if M does not accept w
>
> Now we construct a new Turing machine D with H as a subroutine. This new TM calls H to determine what M does when the input to M is its own description ⟨M⟩. Once D has determined this information, it does the opposite. That is, it rejects if M accepts and accepts if M does not accept.
>
> D(⟨M⟩) = { accept if M does not accept ⟨M⟩
> { reject if M accepts ⟨M⟩ (Sipser 1997:165)

We encode the Sipser D and define the behavior of Sipser H as C functions.

```
//
// Sipser H returns 1 when its input would halt and return 1
// otherwise Sipser H returns 0
//
01 typedef int (*ptr)(); // pointer to int function
02 int H(ptr M, ptr w)   // uses x86 emulator to simulate its input
03
04 int D(ptr M)
05 {
06  if ( H(M, M) )
07    return 0;
08  return 1;
09 }
10
11 int main()
12 {
13   Output((char*)"Input_Halts = ", D(D));
14 }
```

When H correctly simulates D it finds that D remains stuck in recursive simulation
Line 13: main() invokes D(D)
Line 06: Invoked D calls H that simulates D(D)
Line 06: Simulated D calls simulated H that simulates D(D) (**repeats until aborted**)
Simulation Invariant: D simulated by H never reaches Line 07 or Line 08.

D calls simulating halt decider H which computes the mapping from its input D to an accept or reject state on the basis of the behavior of its correct simulation of D. Because D correctly simulated by H cannot possibly terminate normally H aborts its simulation of D and correctly returns 0 for non-halting. When the directly executed D reverses this decision it returns 1.

This is used to correctly fill in the “?” in the Sipser Figure 4.6 (see below) with “accept”.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$ ...	$\langle D \rangle$ ...
M_1	accept	reject	accept	reject	accept
M_2	accept	accept	accept	accept	accept
M_3	reject	reject	reject	reject	reject
M_4	accept	accept	reject	reject	accept
...					
D	reject	reject	accept	accept	?
...					

Figure 4.6 (Sipser 1997:167)

Sipser, Michael 1997. Introduction to the Theory of Computation. Boston: PWS Publishing Company (165-167)

The x86utm operating system (includes several termination analyzers)
https://github.com/plolcott/x86utm Sipser_D can be invoked from main() of Halt7.c

It compiles with the 2017 version of the Community Edition
https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=Community&rel=15

Appendix

```
int Sipser_D(int (*M)())
{
  if ( Sipser_H(M, M) )
    return 0;
  return 1;
}

int main()
{
  Output((char*)"Input_Halts = ", Sipser_D(Sipser_D));
}

_Sipser_D()
[000012ae] 55             push ebp
[000012af] 8bec           mov ebp,esp
[000012b1] 8b4508         mov eax,[ebp+08]
[000012b4] 50             push eax
[000012b5] 8b4d08         mov ecx,[ebp+08]
[000012b8] 51             push ecx
[000012b9] e880fdffff     call 0000103e
[000012be] 83c408         add esp,+08
[000012c1] 85c0           test eax,eax
[000012c3] 7404           jz 000012c9
[000012c5] 33c0           xor eax,eax
[000012c7] eb05           jmp 000012ce
[000012c9] b801000000     mov eax,00000001
[000012ce] 5d             pop ebp
[000012cf] c3             ret
Size in bytes:(0034) [000012cf]
```

When H correctly simulates D it finds that D remains stuck in recursive simulation

```
Sipser_H: Begin Simulation   Execution Trace Stored at:111fa8
 machine   stack     stack     machine    assembly
 address   address   data      code       language
 ========  ========  ========  =========  =============
[000012ae][00111f94][00111f98] 55         push ebp      // Begin Sipser_D
[000012af][00111f94][00111f98] 8bec       mov ebp,esp
[000012b1][00111f94][00111f98] 8b4508     mov eax,[ebp+08]
[000012b4][00111f90][000012ae] 50         push eax      // push Sipser_D
[000012b5][00111f90][000012ae] 8b4d08     mov ecx,[ebp+08]
[000012b8][00111f8c][000012ae] 51         push ecx      // push Sipser_D
[000012b9][00111f88][000012be] e880fdffff call 0000103e // call Sipser_H
Sipser_H: Infinitely Recursive Simulation Detected Simulation Stopped
```

We can see that the first seven instructions of Sipser_D simulated by Sipser_H precisely match the first seven instructions of the x86 source-code of Sipser_D. This conclusively proves that these instructions were simulated correctly.

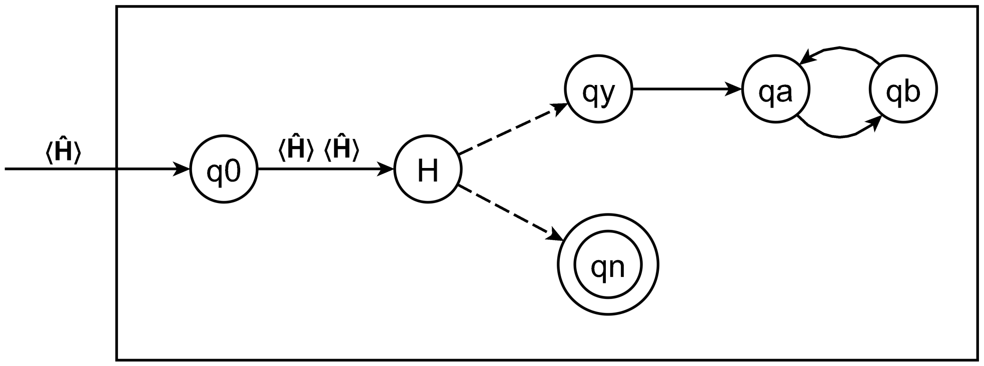
Anyone sufficiently technically competent in the x86 programming language will agree that the above execution trace of Sipser_D simulated by Sipser_H shows that Sipser_D will never stop running unless Sipser_H aborts its simulation of Sipser_D.

Sipser_H detects that Siper_D is calling itself with the exact same arguments that Siper_H was called with and there are no conditional branch instructions from the beginning of Sipser_D to its call to Sipser_H that can possibly escape the repetition of this recursive simulation.

Peter Linz Halting Problem Proof adapted to use a simulating halt decider

When we see the notion of a simulating halt decider applied to the embedded copy of Linz H within Linz Ĥ then we can see that the ⟨Ĥ⟩ ⟨Ĥ⟩ input to embedded H specifies recursive simulation that never reaches its own final state of ⟨Ĥ.qy⟩ or ⟨Ĥ.qn⟩.

computation that halts … the Turing machine will halt whenever it enters a final state. (Linz:1990:234)



Ĥ.q_0 ⟨Ĥ⟩ ⊢* H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qy ∞
If ⟨Ĥ⟩ ⟨Ĥ⟩ correctly simulated by H would reach its own final state of ⟨Ĥ.qy⟩ or ⟨Ĥ.qn⟩.

Ĥ.q_0 ⟨Ĥ⟩ ⊢* H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qn
If ⟨Ĥ⟩ ⟨Ĥ⟩ correctly simulated by H would never reach its own final state of ⟨Ĥ.qy⟩ or ⟨Ĥ.qn⟩.

When Ĥ is applied to ⟨Ĥ⟩ // subscripts indicate unique finite strings
Ĥ copies its input ⟨$Ĥ_0$⟩ to ⟨$Ĥ_1$⟩ then H simulates ⟨$Ĥ_0$⟩ ⟨$Ĥ_1$⟩

Then these steps would keep repeating: (unless their simulation is aborted)
$Ĥ_0$ copies its input ⟨$Ĥ_1$⟩ to ⟨$Ĥ_2$⟩ then H_0 simulates ⟨$Ĥ_1$⟩ ⟨$Ĥ_2$⟩
$Ĥ_1$ copies its input ⟨$Ĥ_2$⟩ to ⟨$Ĥ_3$⟩ then H_1 simulates ⟨$Ĥ_2$⟩ ⟨$Ĥ_3$⟩
$Ĥ_2$ copies its input ⟨$Ĥ_3$⟩ to ⟨$Ĥ_4$⟩ then H_2 simulates ⟨$Ĥ_3$⟩ ⟨$Ĥ_4$⟩...

Since we can see that the input: ⟨$Ĥ_0$⟩ ⟨$Ĥ_1$⟩ correctly simulated by H would never reach its own final state of ⟨$Ĥ_0$.qy⟩ or ⟨$Ĥ_0$.qn⟩ we know that ⟨$Ĥ_0$⟩ specifies a non-halting sequence of configurations.

Linz, Peter 1990. An Introduction to Formal Languages and Automata. Lexington/Toronto: D. C. Heath and Company. (317-320)

When you deny this you deny a tautology
When the line-by-line execution trace of D simulated by H exactly matches the line-by-line behavior that the x86 source-code of D specifies then we know that the simulation is correct.

On 10/13/2022 10:00 PM, Richard Damon wrote:
> Yes, If H never aborts its simulation, then THAT H, the one
> that never aborts its simulation, never answers, and the D
> based on it is non-halting.

Professor Sipser has agreed to these verbatim words (and no more)
If simulating halt decider *H correctly simulates its input D until H*
correctly determines that its simulated D would never stop running
unless aborted then H can abort its simulation of D and correctly
report that D specifies a non-halting sequence of configurations.

A paraphrase of a portion of the above paragraph
Would D correctly simulated by H ever stop running if not aborted?

Is proven on page 3 of this paper to be "no" thus perfectly meeting the Sipser approved criteria shown above.

You don't have enough knowledge of the x86 language to understand the proof on page 3.

Rebutting the Sipser Halting Problem Proof
https://www.researchgate.net/publication/364302709_Rebutting_the_Sipser_Halting_Problem_Proof

If simulating halt decider H correctly predicts that its correct and
complete simulation of D would never stop running then H is correct
to abort its simulation of D and correctly report that D specifies a
non-halting sequence of configurations.

```
On 10/17/2022 10:23 AM, Ben Bacarisse wrote:
> H(D,D) /does/ meet the criterion for PO's Other Halting
> problem - the one no one cares about. D(D) halts (so H is
> not halt decider), but D(D) would not halt unless H stops
> the simulation. H /can/ correctly determine this silly
> criterion (in this one case) so H is a POOH decider
> (again, for this one case -- PO is not interested in the
> fact the POOH is also undecidable in general).

On 10/17/2022 10:23 AM, Ben Bacarisse wrote:
> ...D(D) would not halt unless H stops the simulation.
> H /can/ correctly determine this silly criterion (in this one case)...

https://www.amazon.com/Introduction-Theory-Computation-Sipser/dp/8131525295

comp.theory: [Solution to one instance of the Halting Problem]
```

On 3/14/2017 9:05 AM, peteolcott wrote:

The above reference on the USENET forum comp.theory documents the exact moment
when all of my key ideas came together that form the complete basis of my current solution.