15:11 X86 is Turing-Complete without Data Fetches

by Chris Domas

One might expect that to compute, we must first somehow access data. Even the most primitive Turing tarpits generally provide some type of load and store operation. It may come as a surprise, then, that most modern architectures are Turing-complete without reading data at all!

We begin with the (somewhat uninspiring) observation that the effect of any traditional data fetch can be accomplished with a pure instruction fetch instead.

```
  data:
    .dword    0xdeadc0de
    mov       eax, [data]
```

That fetch in pure code would be a move sourced from an immediate value.

```
    mov       eax, 0xdeadc0de
```

With this, let us then model memory as an array of "fetch cells," which load data through instruction fetches alone.

```
  cell_0:
    mov       eax, 0xdeadc0de
    jmp       esi
  cell_1:
    mov       eax, 0xfeedface
    jmp       esi
  cell_2:
    mov       eax, 0xcafed00d
    jmp       esi
```

So to read a memory cell, without a data fetch, we'll `jmp` to these cells after saving a return address. By using a `jmp`, rather than a traditional function call, we can avoid the indirect data fetches from the stack that occur during a `ret`.

```
  mov       esi, mret                 load return address
  jmp       cell_2                    load cell 2
mret:                                 return
```

A data write, then, could simply modify the immediate used in the read instruction.

```
    mov       [cell_1+1], 0xc0ffee    set cell 1
```

Of course, for a proof of concept, we should actually compute something, without reading data. As is typical in this situation, the BrainFuck language is an ideal candidate for implementation — our fetch cells can be easily adapted to fit the BF memory model.

Reads from the BF memory space are performed through a `jmp` to the BF data cell, which loads an immediate, and jumps back. Writes to the BF memory space are executed as self modifying code, overwriting the immediate value loaded by the data cell. To satisfy our "no data fetch" requirement, we should implement the BrainFuck interpreter without a stack. The I/O BF instructions (`.` and `,`), which use an `int 0x80`, will, at some point, use data reads of course, but this is merely a result of the Linux implementation of I/O.

First, let us create some macros to help with the simulated data fetches:

```
%macro simcall 1
  mov       esi, %%retsim
  jmp       %1
%%retsim:
%endmacro

%macro simfetch 2
  mov       edi, %2
  shl       edi, 3
  add       edi, %1
  mov       esi, %%retsim
  jmp       edi
%%retsim:
%endmacro

%macro simwrite 2
  mov       edi, %2
  shl       edi, 3
  add       edi, %1+1
  mov       [edi], eax
%%retsim:
%endmacro
```

Next, we'll compose the skeleton of a basic BF interpreter:

```
_start:
.execute:
  simcall   fetch_ip
  simfetch  program, eax

  cmp       al, 0
  je        .exit
  cmp       al, '>'
  je        .increment_dp
  cmp       al, '<'
  je        .decrement_dp
  cmp       al, '+'
  je        .increment_data
  cmp       al, '-'
  je        .decrement_data
  cmp       al, '['
  je        .forward
  cmp       al, ']'
  je        .backward
  jmp       done
```

Then, we'll implement each BF instruction without data fetches.

```
.increment_dp:
  simcall   fetch_dp
  inc       eax
  mov       [dp], eax
  jmp       .done



.decrement_dp:
  simcall   fetch_dp
  dec       eax
  mov       [dp], eax
  jmp       .done



.increment_data:
  simcall   fetch_dp
  mov       edx, eax
  simfetch  data, edx
  inc       eax
  simwrite  data, edx
  jmp       .done



.decrement_data:
  simcall   fetch_dp
  mov       edx, eax
  simfetch  data, edx
  dec       eax
  simwrite  data, edx
  jmp       .done



.forward:
  simcall   fetch_dp
  simfetch  data, eax
  cmp       al, 0
  jne       .done
  mov       ecx, 1



.forward.seek:
  simcall   fetch_ip
  inc       eax
  mov       [ip], eax
  simfetch  program, eax
  cmp       al, ']'
  je        .forward.seek.dec
  cmp       al, '['
  je        .forward.seek.inc
  jmp       .forward.seek
.forward.seek.inc:
  inc       ecx
  jmp       .forward.seek
.forward.seek.dec:
  dec       ecx
  cmp       ecx, 0
  je        .done
  jmp       .forward.seek
```

```
.backward:
  simcall   fetch_dp
  simfetch  data, eax
  cmp       al, 0
  je        .done
  mov       ecx, 1
.backward.seek:
  simcall   fetch_ip
  dec       eax
  mov       [ip], eax
  simfetch  program, eax
  cmp       al, '['
  je        .backward.seek.dec
  cmp       al, ']'
  je        .backward.seek.inc
  jmp       backward.seek
.backward.seek.inc:
  inc       ecx
  jmp       .backward.seek
.backward.seek.dec:
  dec       ecx
  cmp       ecx, 0
  je        .done
  jmp       .backward.seek

.done:
  simcall   fetch_ip
  inc       eax
  mov       [ip], eax
  jmp       .execute

.exit:
  mov       eax, 1
  mov       ebx, 0
  int       0x80
```

Finally, let us construct the unusual memory tape and system state. In its data-fetchless form, it looks like this.

```
fetch_ip:
  db       0xb8                     mov eax, xxxxxxxx
ip:
  dd       0
  jmp      esi
fetch_dp:
  db       0xb8                     mov eax, xxxxxxxx
dp:
  dd       0
  jmp      esi
data:
  times    30000 \
  db       0xb8, 0, 0, 0,           mov eax, xxxxxxxx, jmp
           0, 0xff, 0xe6, 0x90      esi, nop
program:
  times    30000 \
  db       0xb8, 0, 0, 0,           mov eax, xxxxxxxx, jmp
           0, 0xff, 0xe6, 0x90      esi, nop
```

For brevity, we've omitted the I/O functionality from this description, but the complete interpreter source code is available.[43]

And behold! a functioning Turing machine on x86, capable of execution without ever touching the data read pipeline. Practical applications are nonexistent.

[43] `git clone https://github.com/xoreaxeaxeax/tiresias || unzip pocorgtfo15.pdf tiresias.zip`