

HOME

{ 2008 12 12 }

PAGES

About Cog
About this blog
Building a Cog
Development Image
Cog Projects
Collaborators
Compiling the VM
Downloads
Eliot Miranda
On-line Papers and
Presentations

CATEGORIES

Cog
Spur

SEARCH

Find

Simulate Out Of The Bochs

When I worked in Paris for a short time I was hugely amused to hear the colloquialism "Quesque c'est ta boîte?", "What's your job?" or "what's your company?", which translates literally as "What's your box"? I know we all love working but occasionally "box" describes one's lot perfectly, hein?

Anyway, if I'm to implement the Cog JIT in Smalltalk and use the Slang translation-to-C route to generate the C source of the production VM I need a way of emulating the JIT's target processor from within Smalltalk. Using the real processor ties the development platform to one's real hardware and potentially leaves one open to random crashes of the entire IDE if one generates incorrect code. Communicating through a ptrace-like interface to another process would mean the existing InterpreterSimulator's memory, a large Bitmap instance, would have to be replaced and mapped onto the remote process, and still one is tied to the available hardware. A software simulator is much more attractive.

Searching for such a beast I stumbled on [Bochs](#), a C++ implementation of an entire x86-based PC that is capable of running Windows or Linux and emulating a sound blaster, and more. At first I thought of trying to translate the code into Smalltalk but wiser heads than mine, specifically my colleague Josh Gargus, pointed out that this would be a bad idea (complex, error-prone and time consuming) and that I'd be better off using the actual C++ code, wrapping it in some kind of primitive interface.

The first thing I needed was an interface to Bochs' C++ cpu object and that means Alien. [Alien](#) is a minimal [FFI](#) I wrote for the Newspeak team at Cadence, under [the direction of Gilad Bracha](#). The part I need for Bochs is the external data modelling code. To quote from the class comment:

Aliens represent ABI (C language) data. They can hold data directly in their bytes or indirectly by pointing to data on the C heap. Alien instances are at least 5 bytes in length. The first 4 bytes of an Alien hold the size, as a signed integer, of the datum the instance is a proxy for. If the size is positive then the Alien is "direct" and the actual datum resides in the object itself, starting at the 5th byte. If the size is negative then the proxy is "indirect", is at least 8 bytes in length and the second 4 bytes hold the address of the datum, which is assumed to be on the C heap. Any attempt to access data beyond the size will fail. If the size is zero then the Alien is a pointer, the second 4 bytes hold a pointer, as for "indirect" Aliens, and accessing primitives

indirect through the pointer to access data, but no bounds checking is performed.

When Aliens are used as parameters in FFI calls then all are "passed by value", so that e.g. a 4 byte direct alien will have its 4 bytes of data passed, and a 12-byte indirect alien will have the 12 bytes its address references passed. Pointer aliens will have their 4 byte pointer passed. So indirect and pointer aliens are equivalent for accessing data but different when passed as parameters, indirect Aliens passing the data and pointer Aliens passing the pointer.

Once I had the Alien framework in the VM I needed to generate an interface to the Bochs cpu. Here's the program that does that:

```
-----8<----- printcpu.c -----8<-----
/*
% g++ -I.. -I../cpu -I../instrument/stubs -Wno-
invalid-offsetof @ -o #
*/

#include <stddef.h>
#include <bochs.h>
#define NEED_CPU_REG_SHORTCUTS
#include <cpu.h>

static char buf[10];

char *
lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        buf[i] = tolower(s[i]);
    buf[i] = 0;
    return buf;
}

int
main()
{
#define stoffsetof(type,field)
(offsetof(type,field)+1)
#define print(r,n) \
printf("!BochsIA32Alien methodsFor: 'accessing'
stamp: 'eem %d/%d/%d %d:%d'!\r"
"%s\r\t^self unsignedLongAt: %d! !\r",
m,d,y,h,i, lower(#r), \
stoffsetof(BX_CPU_C,gen_reg[n].dword.ern));\
printf("!BochsIA32Alien methodsFor: 'accessing'
stamp: 'eem %d/%d/%d %d:%d'!\r"
"%s: anUnsignedInteger\r\t^self
unsignedLongAt: %d put: anUnsignedInteger! !\r",
m,d,y,h,i, lower(#r), \
stoffsetof(BX_CPU_C,gen_reg[n].dword.ern));\

    time_t nowsecs = time(0);
    struct tm now = *localtime(&nowsecs);
    int m = now.tm_mon + 1; /* strange but true
*/

    int d = now.tm_mday;
```

```

        int y = now.tm_year + 1900;
        int h = now.tm_hour;
        int i = now.tm_min;

        printf("\nHello world!!!\n!\r");
        printf("!BochsIA32Alien class methodsFor:
'instance creation' stamp: 'eem %d/%d/%d %d:%d'!\r"
        "dataSize\r\t^%d! !\r", m,d,y,h,i,
sizeof(BX_CPU_C));

        printf("!BochsIA32Alien methodsFor:
'accessing' stamp: 'eem %d/%d/%d %d:%d'!\r"
        "eflags\r\t^self unsignedLongAt: %d! !\r",
m,d,y,h,i,
stoffsetof(BX_CPU_C,eflags));

        print(EAX,BX_32BIT_REG_EAX);
        print(EBX,BX_32BIT_REG_EBX);
        print(ECX,BX_32BIT_REG_ECX);
        print(EDX,BX_32BIT_REG_EDX);
        print(ESP,BX_32BIT_REG_ESP);
        print(EBP,BX_32BIT_REG_EBP);
        print(ESI,BX_32BIT_REG_ESI);
        print(EDI,BX_32BIT_REG_EDI);
        print(EIP,BX_32BIT_REG_EIP);

        return 0;
}
-----8<----- printcpu.c -----8<-----

```

xc is an old unix program written by David MacKenzie from 1989 that I love. You say

```
xc printcpu.c
```

and it evaluates

```
g++ -l. -l../cpu -l../instrument/stubs -Wno-invalid-offsetof
printcpu.c -o printcpu
grabing the instructions from the comment and the front of printcpu.c.
Say xc again and it remembers which file you last ran. No need for a
makefile for simple one-file programs. Neat.
```

printcpu.c then produces output looking like

```

-----8<----- BochsIA32Alien.st -----8<-----
"Hello world!!!"
!BochsIA32Alien class methodsFor: 'instance creation'
stamp: 'eem 12/11/2008 18:46'!
dataSize
^18000! !
!BochsIA32Alien methodsFor: 'accessing' stamp: 'eem
12/11/2008 18:46'!
eflags
^self unsignedLongAt: 513! !
!BochsIA32Alien methodsFor: 'accessing' stamp: 'eem
12/11/2008 18:46'!
eax
^self unsignedLongAt: 469! !
!BochsIA32Alien methodsFor: 'accessing' stamp: 'eem
12/11/2008 18:46'!
eax: anUnsignedInteger
^self unsignedLongAt: 469 put: anUnsignedInteger! !

```

!BochsIA32Alien methodsFor: 'accessing' stamp: 'eem
12/11/2008 18:46'!

ebx

^self unsignedLongAt: 481! !

!BochsIA32Alien methodsFor: 'accessing' stamp: 'eem
12/11/2008 18:46'!

ebx: anUnsignedInteger

^self unsignedLongAt: 481 put: anUnsignedInteger! !

etc...

-----8<----- BochsIA32Alien.st -----8<-----

which files straight into Squeak once I've defined the class:

```
Alien variableByteSubclass: #BochsIA32Alien
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Cog-Processors'
```

With a little bit of fiddling pairing away the bits of Bochs I don't need I've got down to just the CPU. Here's the plugin that interfaces to it:

```
SmartSyntaxInterpreterPlugin subclass: #BochsIA32Plugin
instanceVariableNames: ''
classVariableNames: 'BaseHeaderSize BytesPerOop'
poolDictionaries: ''
category: 'Cog-ProcessorPlugins'
```

BochsIA32Plugin methods for primitives

primitiveNewCPU

| cpu |

self var: #cpu type: 'void *'.

self primitive: 'primitiveNewCPU' parameters: #().

cpu := self cCode: 'newcpu()' inSmalltalk: [0].

cpu = 0 ifTrue:

[^interpreterProxy primitiveFail].

interpreterProxy

pop: 1

thenPush: (interpreterProxy positive32BitIntegerFor:
(self

cCoerceSimple: cpu

to:

'unsigned long'))

"cpuAlien <BochsIA32Alien>" primitiveSingleStepIn: memory "
<Bitmap|ByteArray|WordArray>"

"Single-step the cpu using the argument as the memory."

| cpuAlien cpu maybeErr |

self var: #cpu type: 'void *'.

cpuAlien := self primitive: 'primitiveSingleStepIn'

parameters: #(WordsOrBytes)

receiver: #Oop.

(cpu := self startOfData: cpuAlien) = 0 ifTrue:

[^interpreterProxy primitiveFailFor: PrimErrBadReceiver].

maybeErr := self singleStep: cpu

In: memory

Size: (interpreterProxy byteSizeOf: memory

cPtrAsOop).

maybeErr ~= 0 ifTrue:

```
[^interpreterProxy primitiveFailFor: PrimErrInappropriate].  
^cpuAlien
```

BochsIA32Plugin methods for alien support

sizeField: rcvr

"Answer the first field of rcvr which is assumed to be an Alien of at least 8 bytes"

self inline: true.

^self longAt: rcvr + BaseHeaderSize

startOfData: rcvr "<Alien oop> ^<Integer>"

"Answer the start of rcvr's data. For direct aliens this is the address of

the second field. For indirect and pointer aliens it is what the second field points to."

self inline: true.

^(self sizeField: rcvr) > 0

ifTrue: [rcvr + BaseHeaderSize + BytesPerOop]

ifFalse: [self longAt: rcvr + BaseHeaderSize +

BytesPerOop]

BochsIA32Alien hooks up to this code via

BochsIA32Alien class methods for instance creation

new

^self atAddress: self primitiveNewCPU

BochsIA32Alien class methods for primitives

primitiveNewCPU

"Answer the address of a new Bochs C++ class
bx_cpu_c/BX_CPU_C x86 CPU emulator instance."

<primitive: 'primitiveNewCPU' module: 'BochsIA32Plugin'>

^self primitiveFailed

BochsIA32Alien methods for execution

singleStepIn: aMemory

| result |

result := self primitiveSingleStepIn: aMemory.

result ~~ self ifTrue:

[self error: 'eek!']

BochsIA32Alien methods for primitives

primitiveSingleStepIn: memoryArray "<Bitmap|ByteArray>"

"Single-step the receiver using the argument as the store."

<primitive: 'primitiveSingleStepIn' module: 'BochsIA32Plugin'>

^self primitiveFailed

where **atAddress:** is part of Alien's standard instance creation facilities.

And lo and behold the following test passes, yeah!!

BochsIA32AlienTests methods for tests

testCPUID

| vendorString |

self processor

eip: 0;

eax: 0. "get vendor identification string"

self processor singleStepIn: (ByteArray with: 16r0F with: 16rA2

with: 16r90) "cpuid;nop".

self assert: self processor eip = 2.

```

        self assert: self processor eax ~= 0.
        vendorString := (ByteArray new: 12)
                                longAt: 1 put: self processor ebx
bigEndian: false;
                                longAt: 5 put: self processor edx
bigEndian: false;
                                longAt: 9 put: self processor ecx
bigEndian: false;
                                asString.
        self assert: (vendorString = 'GenuineIntel'
                                or: [vendorString = 'AuthenticAMD'])

```

BochsIA32AlienTests methods for accessing processor

```

processor ifNil:
    [processor := BochsIA32Alien new].
^processor

```

I am a happy chappie!

A little more work figuring out that the Bochs emulation is accurate and boots the processor into real mode with 16-bit default operand sizes for the CS and SS segments and that I need to put it into protected 32-bit mode with 32-bit default operand sizes and the following works, yay!

BochsIA32AlienTests methods for tests

```

testNfib4
    "self new testNfib4"
    self runNFib: 4.
    self assert: self processor eip = self nfib size.
    self assert: self processor eax = 4 benchFib

```

BochsIA32AlienTests methods for execution

```

runNFib: n
    "Run nfib with the argument. Answer the result."
    | memory finalSP |
    memory := ByteArray new: 4096 withAll: self processor
nopOpcode.
    finalSP := memory size - 4. "Stop when we return to the nop
following nfib"
    memory
        replaceFrom: 1 to: self nfib size with: self nfib asByteArray
startingAt: 1;
        longAt: 4093 put: n bigEndian: false; "argument n"
        longAt: 4089 put: self nfib size bigEndian: false. "return
address"
    self processor
        eip: 0;
        esp: (memory size - 8). "Room for return address and
argument n"
    [self processor singleStepIn: memory.
    self processor esp ~= finalSP] whileTrue.
    ^self processor eax

```

BochsIA32AlienTests methods for accessing nfib

```

nfib
    "long fib(long n) { return n <= 1 ? 1 : fib(n-1) + fib(n-2) + 1; }
as compiled by Microsoft Visual C++ V6 (12.00.8804) cl /O2 /Fc"


```

```

"| bat nfib ip |
bat := BochslA32AlienTests new.
nfib := bat nfib asByteArray.
ip := 0.
20 timesRepeat:
    [bat processor disassembleInstructionAt: ip In: nfib into:
        [:da :len]
        Transcript nextPutAll: da; cr; flush.
        ip := ip + len]]
^#("00000" 16r56                                "push
esi"
        "00001" 16r8B 16r74 16r24 16r08            "mov esi,
DWORD PTR _n$[esp]"
        "00005" 16r83 16rFE 16r01                  "cmp esi,
1"
        "00008" 16r7F 16r07                          "jg
SHORT $L528"
        "0000a" 16rB8 16r01 16r00 16r00 16r00      "mov eax,
1"
        "0000f" 16r5E                                "pop
esi"
        "00010" 16rC3                                "ret 0"
"
$L528:"
        "00011" 16r8D 16r46 16rFE                    "lea eax,
DWORD PTR [esi-2]"
        "00014" 16r57                                "push
edi"
        "00015" 16r50                                "push
eax"
        "00016" 16rE8 16rE5 16rFF 16rFF 16rFF      "call _fib"
        "0001b" 16r4E                                "dec
esi"
        "0001c" 16r8B 16rF8                          "mov
edi, eax"
        "0001e" 16r56                                "push
esi"
        "0001f" 16rE8 16rDC 16rFF 16rFF 16rFF      "call _fib"
        "00024" 16r83 16rC4 16r08                    "add esp,
8"
        "00027" 16r8D 16r44 16r07 16r01              "lea eax,
DWORD PTR [edi+eax+1]"
        "0002b" 16r5F                                "pop
edi"
        "0002c" 16r5E                                "pop
esi"
        "0002d" 16rC3                                "ret
0")

```

I'm ready to start work on the JIT. No more machine-level debugging for quite a while I hope 😊

	Send article as PDF	<input type="text" value="Enter email address"/>	<input type="button" value="Send"/>
---	---------------------	--	-------------------------------------

Posted by admin on Friday, December 12th, 2008, at 8:14 pm, and
filed under [Cog](#).

Follow any responses to this entry with the [RSS 2.0](#) feed.

You can [post a comment](#).

Comments

1. [Carl Gundel](#) | 12-Dec-08 at 10:13 pm | [Permalink](#)

Cool. Good work. No coal in your stocking this Christmas. 😊

2. [Stephen Pair](#) | 14-Dec-08 at 6:12 am | [Permalink](#)

Nice, I had totally forgotten about Bochs (I could use that for another project I've been working on). But I still think you need to port it to Smalltalk. ;))

3. [Pete Cockerell](#) | 29-Dec-08 at 5:12 pm | [Permalink](#)

Interestingly I use the idiom exemplified by

```
for (i = 0; i < strlen(s); i++)
    buf[i] = tolower(s[i]);
```

as a “What’s wrong with this code?” interview question. The answer is of course that it exhibits highly avoidable $O(n^2)$ performance, since `strlen()`, which is $O(n)$, is being called n times. You can certainly argue that performance hardly matters in this case (and n is rather small), but seeing that idiom out there in the wild still causes me to gasp a little 😊

4. [Eliot Miranda](#) | 30-Dec-08 at 12:08 pm | [Permalink](#)

Pete,

Ouch!. Forgive me, I wasn’t thinking about the C, only about the Smalltalk output. But I shouldn’t be displaying bad examples all the same.

But this is interesting. The equivalent Smalltalk code is simply

```
aString asLowercase
```

but in C I had to write a function to do it. There was no `strtolower` function to hand in the C library, a library routine that would be tried and tested and free of errors. The Smalltalk code is something as simple as

```
asLowercase
^self collect: [:cl c asLowercase]
```

whereas the C code has – count ‘em – 9 lines! So C forces one to write more code, providing much more opportunity for mistakes, and makes it far harder to debug and correct (low level debuggers, no edit-and-continue).

I make mistakes. I’m currently working on a JIT written in Smalltalk that will be transliterated into C. Were I in C I would be making snail-like progress but in Smalltalk I feel like I’m flying along. Programming systems should be designed for humans, with all

their failings. I put down my $O(n^2)$ blunder to the paucity of the C development ecosystem and yet another example of why trying to implement a faster Smalltalk VM makes sense 😊

That said, next time I'll be sure to write

```
char *
lower(char *s)
{
    int i = strlen(s);
    buf[i] = 0;
    while (--i >= 0)
        buf[i] = tolower(s[i]);
    return buf;
}
```



5. **Pete Cockerell** | 30-Dec-08 at 1:04 pm | [Permalink](#)

Thanks, Eliot 😊 Of course, you're one of the lucky ones, still being able to pursue the ST dream, while most of us have to think down to the level of C or (in my case) Java on a daily basis. Your example got me wondering, though: are current ST implementations Unicode-aware? Writing `Character#asLowerCase` would be quite interesting in a Unicode world...

Sorry, bit off-topic!

6. **Eliot Miranda** | 30-Dec-08 at 7:09 pm | [Permalink](#)

Pete,

Ouch ^2. Yes I am lucky. At least know that I'm hugely grateful to be able to follow the ST dream. But well spotted that Unicode makes things more complicated. Yes, most current Smalltalk implementations are Unicode savvy, and in fact Squeak's `String>>asLowerCase` implementation looks like

```
asLowerCase
    "Answer a String made up from the receiver whose characters are all lowercase."
    ^ self copy asString translateToLowerCase
translateToLowerCase
    "Translate all characters to lowercase, in place"
    self translateWith: LowercasingTable
```

Which looks a little over-complicated to me. But its not my problem 😊

Somebody give Pete a job ... doing Smalltalk!

Post a Comment

Your email is *never* published nor shared. Required fields are marked *

Name *

Email *

Website

Message



Post

« [MECHANISED
MODIFICATIONS AND
MISCELLANEOUS
MEASUREMENTS](#)

[THE IDÉE FIXE AND THE
PERFECTED PROFILER](#) »