



# Virtual PC 4.0

## Emulator Architecture

**Eric Traut**

**Chief Technology Officer  
Virtual PC Architect  
Connectix Corp.**





# Virtual PC 4.0

## Development Process

- Virtual PC Background
- 4.0 Architecture
- Internal Tools
- Localization



# Virtual PC Background

- Mac-hosted PC emulator
- Started Virtual PC 1.0 in late 1995, released in early 1997
- Recently hit total sales of 1 million
- Original design favored compatibility and speed over portability and maintainability



# Customer Feedback

- Customers overwhelmingly indicated that *performance* was the number one feature they wanted to see in the next version.
- From Virtual PC 1.0 through 3.1, we were able to increase performance by about 25%. Increasing it further was going to require something drastic.



# Other Goals

- Cross-platform
- More maintainable
- Multiple virtual machine instantiation
- Better memory management
- Easier to localize



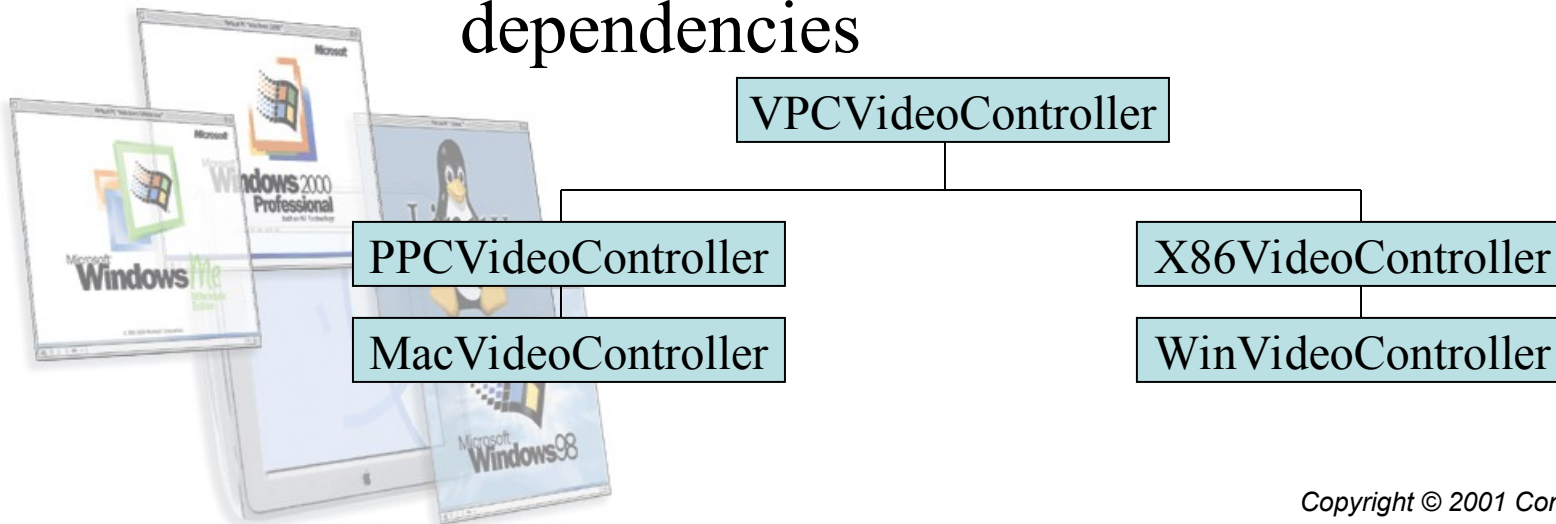
# The Old Architecture

- Each virtual “device” was represented by a blob of C code
- State was statically or globally allocated
- Mac-specific assumptions were interspersed throughout the code



# New Architecture

- Each virtual “device” implemented as C++ object that can be instantiated multiple times
- Class hierarchy used to abstract host processor and host operating system dependencies



# Processor & MMU Emulation

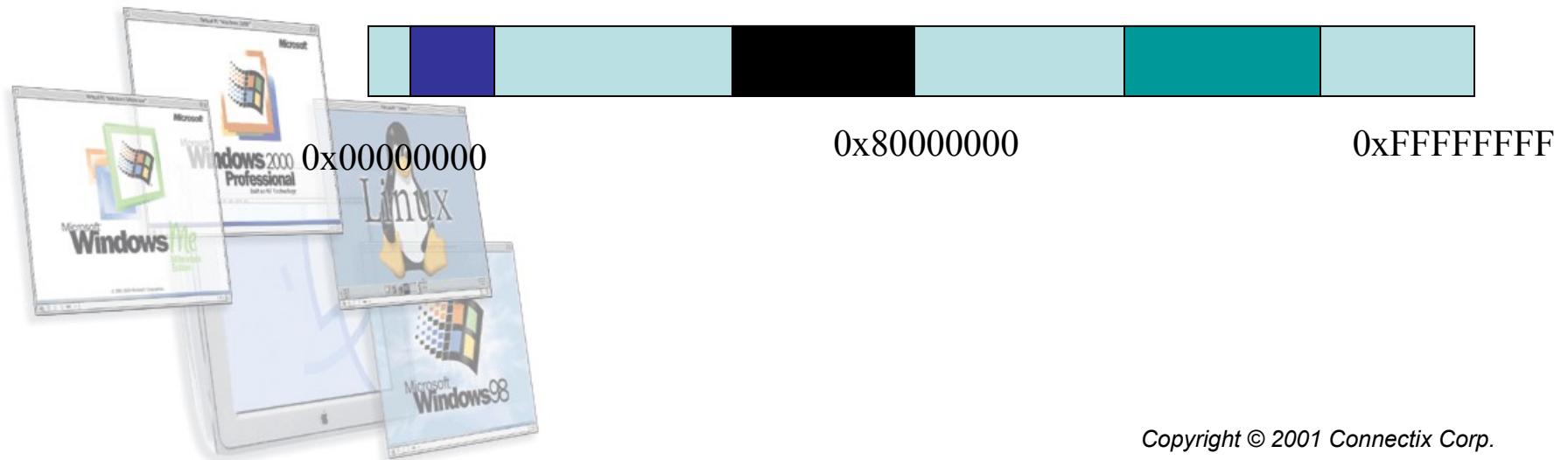
- In Virtual PC, about 80% of time is typically spent emulating guest x86 instructions. This is the biggest area for performance gains.
- Challenges in emulating x86:
  - MMU (32-bit address space)
  - Segmentation
  - Heavy dependency on memory accesses
  - Correct emulation of exceptions





# MMU Emulation

- Emulating a 32-bit processor on a 32-bit processor
- Don't have control of guest OS memory map



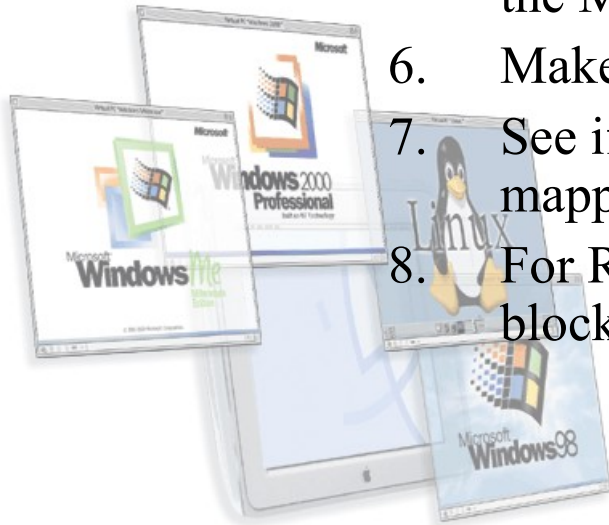
# Segmentation in x86

- Every x86 memory access (including instruction fetches) are checked against an upper and lower bound defined by one of six segment registers.
  - Accesses that go beyond the limits need to produce a GPF (general protection fault) for correct program behavior.
- Every segment also has a 32-bit base that needs to be added to the offset.
  - Linear address = segment base + offset



# Simple (but slow) implementation

1. Fetch segment base and segment bounds
2. Calculate effective address (which produces the segment offset)
3. Check offset against bounds of segment, generate GPF if out of bounds
4. Add offset to base to calculate linear address
5. Convert linear address to physical address by traversing the MMU page tables
6. Make sure access doesn't cross page boundary
7. See if physical address accesses RAM or memory-mapped I/O
8. For RAM accesses, use physical address to index into block of memory that represents logical RAM



# Power PC Example

- PUSH EAX
  - [2] lwz rTemp3,SSLimit(rData)
  - [0] subi rTemp2,rESP,4
  - [1] cmpl cr7,rTemp2,rTemp3
  - [2] lwz rTemp4,SSBase(rData)
  - [0] bgt cr7,GenerateGPF
  - [1] add rTemp1,rTemp2,rTemp4
  - [15] bl ConvertLogToPhys
  - [1] stwbrx rEAX,rTemp1,rRAMBase
  - [0] subi rESP,rESP,4
  - Total cycles: 22 (versus 1)
  - (500MHz PPC  $\Rightarrow$  23MHz x86)



# Using Registers & Traps

- PUSH EAX

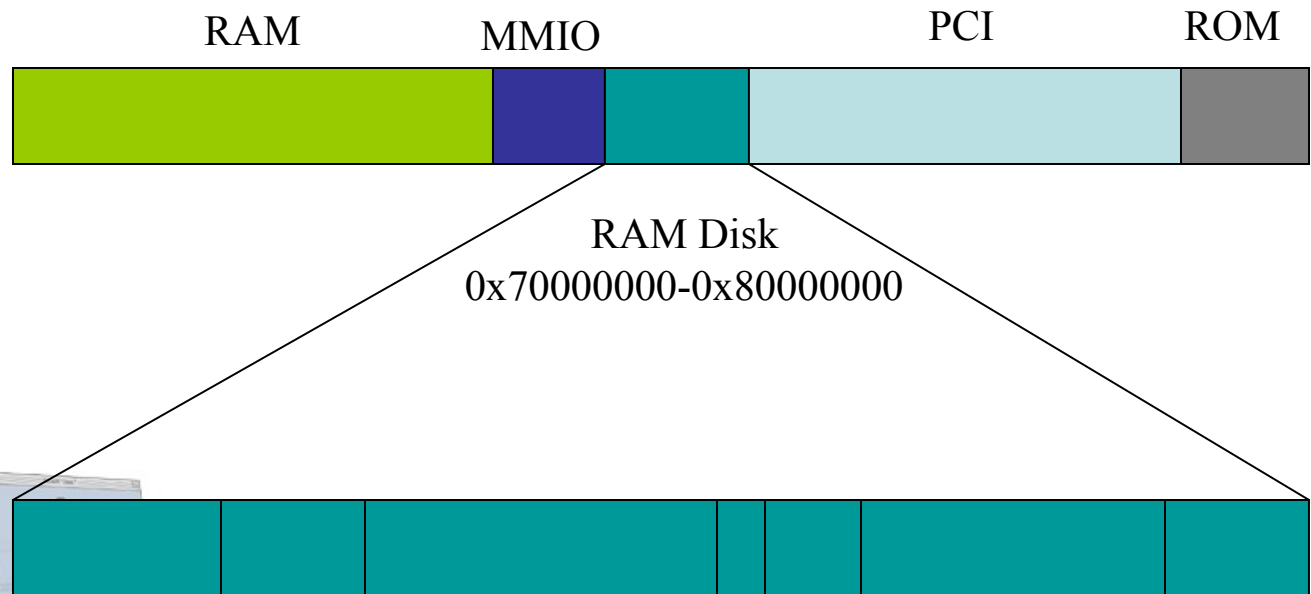
- [1] subi rTemp2, rESP, 4
- [1] tlgt rTemp2, rSSLimit
- [1] add rTemp1, rTemp2, rSSBase
- [15] bl ConvertLogToPhys
- [1] stwbrx rEAX, rTemp1, rRAMBase
- [0] subi rESP, rESP, 4

- Total cycles: 20 (versus 1)
- (500MHz PPC  $\Rightarrow$  25MHz x86)



# Using the MMU

- The classic Mac memory map



Broken up into contiguous guest address space blocks



# Using the Host MMU

- PUSH EAX

- [1] subi rTemp2, rESP, 4
- [1] add rTemp3, rTemp2, rSSAdjBase
- [1] tlgt rTemp3, rSSAdjLimit
- [1] stwbrx rEAX, rTemp3, rSSChBase
- [0] subi rESP, rESP, 4

- Total cycles: 4 (versus 1)
- (500MHz PPC  $\Rightarrow$  125MHz x86)



# Rearchitecting

- Use the entire 32-bit address space
- Take advantage of the fact that most new Windows code uses wide-open segments
- Take advantage of PowerPC “little endian mode”





# Assuming 32-bit address space and wide-open segments

- PUSH EAX

- [1] subi rTemp2, rESP, 4
- [1] stwbrx rEAX, r0, rTemp2
- [0] subi rESP, rESP, 4
- Total cycles: 2 (versus 1)
- (500MHz PPC  $\Rightarrow$  250MHz x86)



# Using Little Endian Mode

- PUSH EAX

- [1] subi rTemp2, rESP, 4
- [1] stwx rEAX, r0, rTemp2
- [0] subi rESP, rESP, 4

and finally...

- [1] stwu rEAX, -4(rESP)



- Total cycles: 1 (versus 1)
- (500MHz PPC  $\Rightarrow$  500MHz x86)

# Floating Point Example

- `FLD DS:[EAX]`

- `[1] twgt rEAX, rDSLlimit`
- `[1] lwbrx rTemp1, rDSBase, rEAX`
- `[0] addi rTemp2, rEAX, 4`
- `[1] lwbrx rTemp2, rDSBase, rTemp2`
- `[1] stw rTemp1, Swap+4(rData)`
- `[1] stw rTemp2, Swap+0(rData)`
- `[11] lfd fp0, Swap+0(rData)`



- Total cycles: 16 (versus 1)  
- (500MHz PPC  $\Rightarrow$  31MHz x86)

# Floating Point Example

- `FLD DS:[EAX]`
  - `[1] lfdx fp0,rDSBase,rEAX`
  - Total cycles: 1 (versus 1)
  - (500MHz PPC  $\Rightarrow$  500MHz x86)



# “Pseudo little endian”

- The PowerPC doesn't really implement little endian mode.
- It presents a “facade” that makes the software believe it's running in little endian mode.



# “Pseudo Little Endian”

- Example:

```
unsigned long * ptr = NULL;
```

```
ptr[0] = 0x00112233;
```

```
ptr[1] = 0x44556677;
```

Memory address:      0      1      2      3      4      5      6      7

Big Endian:

00	11	22	33	44	55	66	77
----	----	----	----	----	----	----	----

Little Endian:

33	22	11	00	77	66	55	44
----	----	----	----	----	----	----	----

Pseudo Little Endian:

44	55	66	77	00	11	22	33
----	----	----	----	----	----	----	----



# “Pseudo Little Endian”

- Crossing 8-byte boundaries gets weird
- Example:

```
unsigned char * ptr = NULL;  
(long *)ptr[6] = 0x44556677;
```

Pseudo Little Endian:

0	1	2	3	4	5	6	7
66	77						
8	9	A	B	C	D	E	F
						44	55



# Working with Pseudo Little Endian

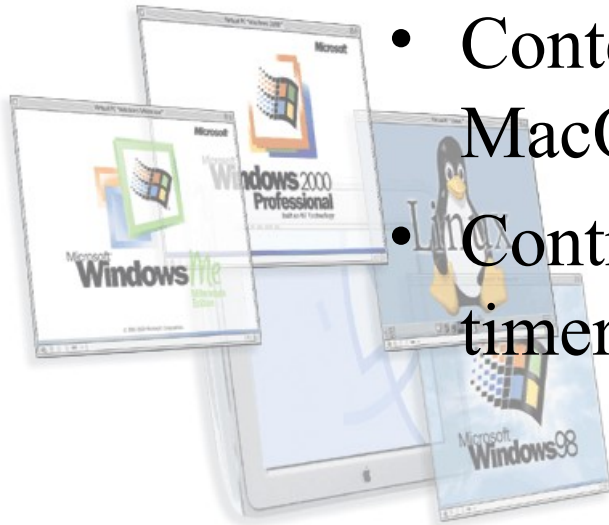
- All data shared between big endian context and little endian context needs to be “sizzled” (i.e. converted from big endian to pseudo little endian).
- All share data structures need to be defined twice.





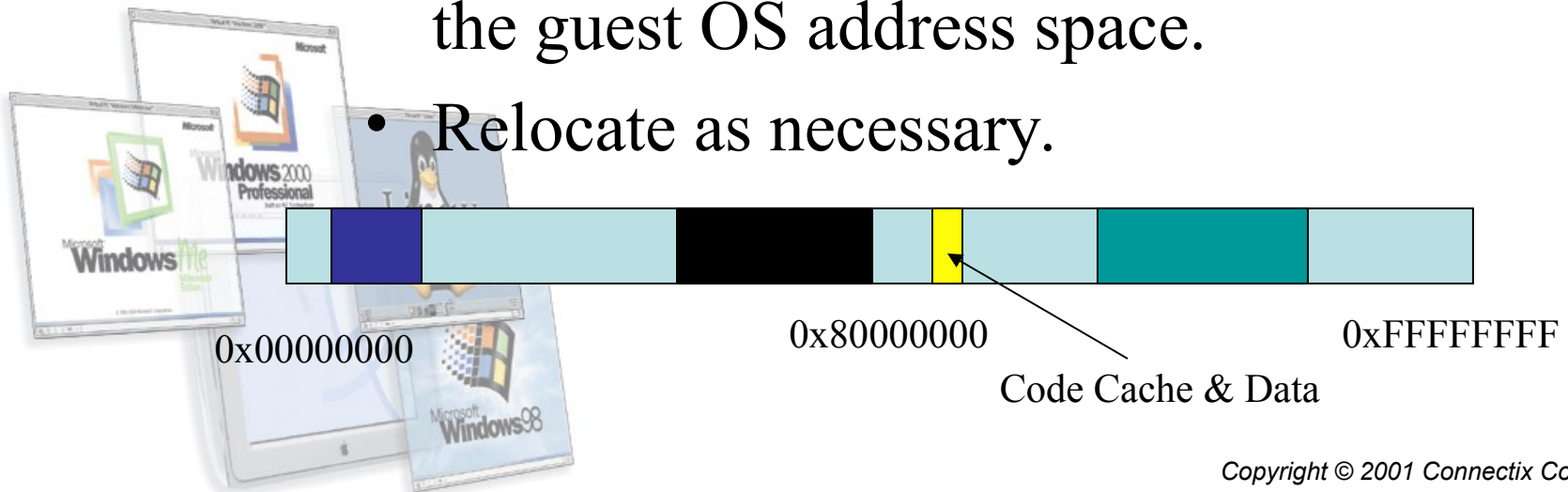
# Where can we find a 32-bit address space?

- Create a second address space and processor state separate from that of the MacOS.
- Code running within this “alternate” context runs in little endian mode.
- Context switcher steals control from MacOS kernel “for a while”.
- Control is returned on external interrupt, timer interrupt or explicitly.



# Where to store the code cache?

- The emulator still requires space for data and code. If the entire 32-bit address space is under control of the guest operating system, where does this go?
- Solution: Find an unoccupied “hole” in the guest OS address space.
- Relocate as necessary.



# Alternate Context Coroutine

- Implemented as a *coroutine*
  - Like a thread in that it has its own state
  - Like a subroutine in that it is called, and returns control back to the caller on certain events



# Improving the compiler

- Original x86-to-PowerPC translator was written in PowerPC assembly.
- New version is written in C++, and is about 1/2 the speed.
  - Code is translated on a basic-block basis
  - Compiler performs true code flow analysis to remove redundant condition code calculation
  - Limited register allocation performed



# Handling Exceptions

- If you take an exception (e.g. a page fault or GPF) in the middle of a block of PowerPC code, what x86 instruction does that correspond to?
  - Reverse map tables



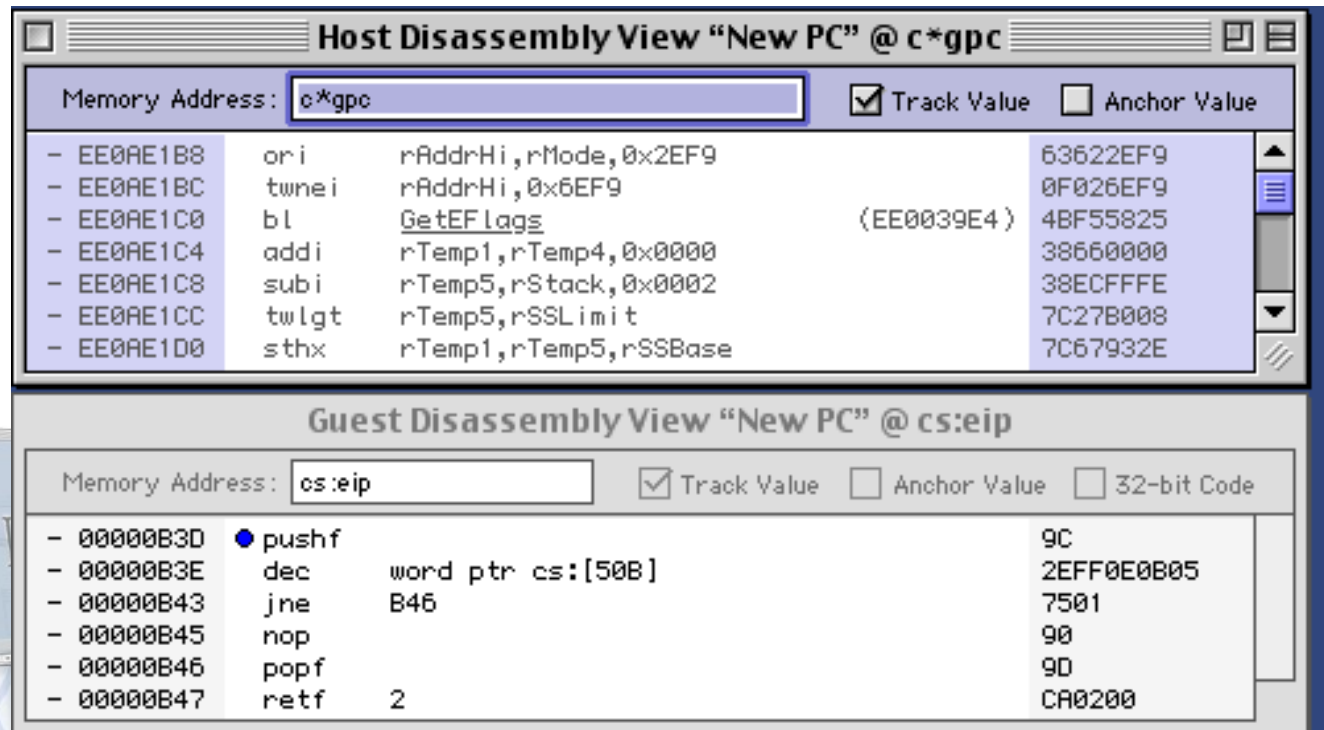
# Virtual PC on MacOS X

- Worked with Mach kernel engineers to implement “VMM” (virtual machine manager) feature into the kernel.
  - Implements alternate context coroutine mechanism
  - Provides page mapping features
  - Supports little endian mode



# Tools

- New debugging environment requires separate debuggers: one for the host, one for the guest



The screenshot displays two debugger windows side-by-side, illustrating the separate debugging environments for the host and guest.

**Host Disassembly View "New PC" @ c\*gpc**

Memory Address:  ☒ Track Value ☐ Anchor Value

- EE0AE1B8	ori	rAddrHi,rMode,0x2EF9	63622EF9
- EE0AE1BC	twnei	rAddrHi,0x6EF9	0F026EF9
- EE0AE1C0	bl	GetEFLags (EE0039E4)	4BF55825
- EE0AE1C4	addi	rTemp1,rTemp4,0x0000	38660000
- EE0AE1C8	subi	rTemp5,rStack,0x0002	38ECFFFE
- EE0AE1CC	twlgt	rTemp5,rSSLimit	7C27B008
- EE0AE1D0	sthx	rTemp1,rTemp5,rSSBase	7C67932E

**Guest Disassembly View "New PC" @ cs:eip**

Memory Address:  ☒ Track Value ☐ Anchor Value ☐ 32-bit Code

- 00000B3D	• pushf	9C
- 00000B3E	dec word ptr cs:[50B]	2EFF0E0B05
- 00000B43	jne B46	7501
- 00000B45	nop	90
- 00000B46	popf	9D
- 00000B47	retf 2	CA0200

# Localization

- ZStrings
  - Got idea from Adobe Photoshop team
  - Implements general utility string class that supports a wide array of string manipulation methods
  - Make use of “named string templates” which are embedded within the source code





# Named String Templates

- Based on HTML/XML-style tags
- Uses HTML escape characters for high-ASCII characters
- Examples:

```
"<Z name=VPC/Menu/File/Title>File</Z>"
```

```
"<Z name=VPC/SaveDialog/Text>"
```

```
"Are you sure you want to save
```

```
&ldquo&replace00&rdquo&hellip<\Z>"
```



# Using Named String Templates

- Example:

```
const char * sFileMenuTitle = "<Z  
    name=VPC/Menu/File/Title>File</Z>";  
ZString menuTitle;  
menuTitle.GetNamedString(sFileMenuTitle);
```



# Extracting Named Strings

- Tool extracts all named string templates directly from the application binary, producing a text file that can be viewed in an HTML browser.
- This file is verified for correctness (spell checked, proofread, etc.) and then sent to translators.



# Building Override Dictionaries

- Once translated string file come back from the translator, another tool builds an “override dictionary” which is pasted into the application in the form of a resource.
- The application is able to load the correct override dictionary at runtime based on system language.



# ZString Source

- ZString source code, tools, and documentation being put into the public domain.
- You'll find it on the MacHack CD.



# Q & A

- Other potential topics of interest
  - Future of Virtual PC on MacOS
  - Virtual PC on other platforms
  - Emulating x86 on x86
  - Demo of ZString tools

