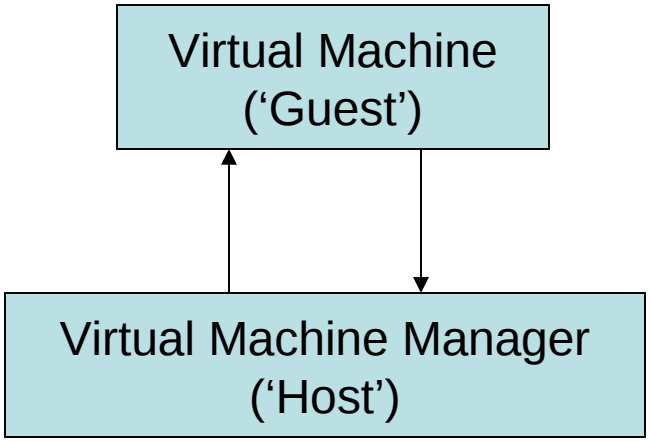
Our planned VMX demo

Implementation-code for our
‘Guest’ VM and ‘Host’ VMM

Quick review

- We plan to demonstrate use of Intel's new VMX instructions in a 'simple' context:
  - One 'Virtual-8086 mode' guest-task (the VM)
  - A '64-bit mode' host-task (the VM Manager)



Guest ‘resources’

- We identified the needed data-structures and procedures to support our ‘Guest’ VM:
  - Task State Segment (with permission-bitmap)
  - Global Descriptor Table (for TSS and LDT)
  - Local Descriptor Table (code, data, vram, flat)
  - Page-Directory Table (for one ‘4-MB’ frame)
  - Two stacks (for use in ring3 and in ring0)
  - Code for ‘guest_task’ (also ‘guest_isrGPF’)

Testing our ‘Guest’

- We created ‘vmxstep1.s’ to implement our guest’s data-structures and its procedures
- We also wrote ‘trystep1.s’ as a testbed for resources in our ‘vmxstep1.s’ component
- Two files should be separately assembled, then linked to create the test ‘executable’

```
$  as  vmxstep1.s  -o vmxstep1.o
$  as  trystep1.s  -o trystep1.o
$  ld  trystep1.o  vmxstep1.o  -T ldscript  -o trystep1.b
$  dd  if=trystep1.b  of=/dev/sda4  seek=1
```

What will our ‘Host’ do?

- Our VMM will need to take these actions:
  - Initially enter VMX mode (using ‘**vmxon**’)
  - Clear our guest’s VMCS (using ‘**vmclear**’)
  - Load the guest-pointer (using ‘**vmptrld**’)
  - Write VMCS parameters (using ‘**vmwrite**’)
  - Launch the guest-task (using ‘**vmlaunch**’)
  - Read guest-exit information (using ‘**vmread**’)
  - Maybe reenter the guest (using ‘**vmresume**’)
  - Eventually leave VMX mode (using ‘**vmxoff**’)

Host’s data-resources

- We need supporting VMM data-structures:
  - 4-level page-tables (required for 64-bit code)
  - Task-State Segment (no permission-bitmap)
  - Global Descriptor Table (code, data, TSS)
  - Interrupt Descriptor Table (for GP faults)
  - One stack-region (for VMM’s ‘ring0’ code)
  - Virtual Machine Control Structures (two)
  - Collection of variables (VMCS parameters)

‘vmxstep2.s’

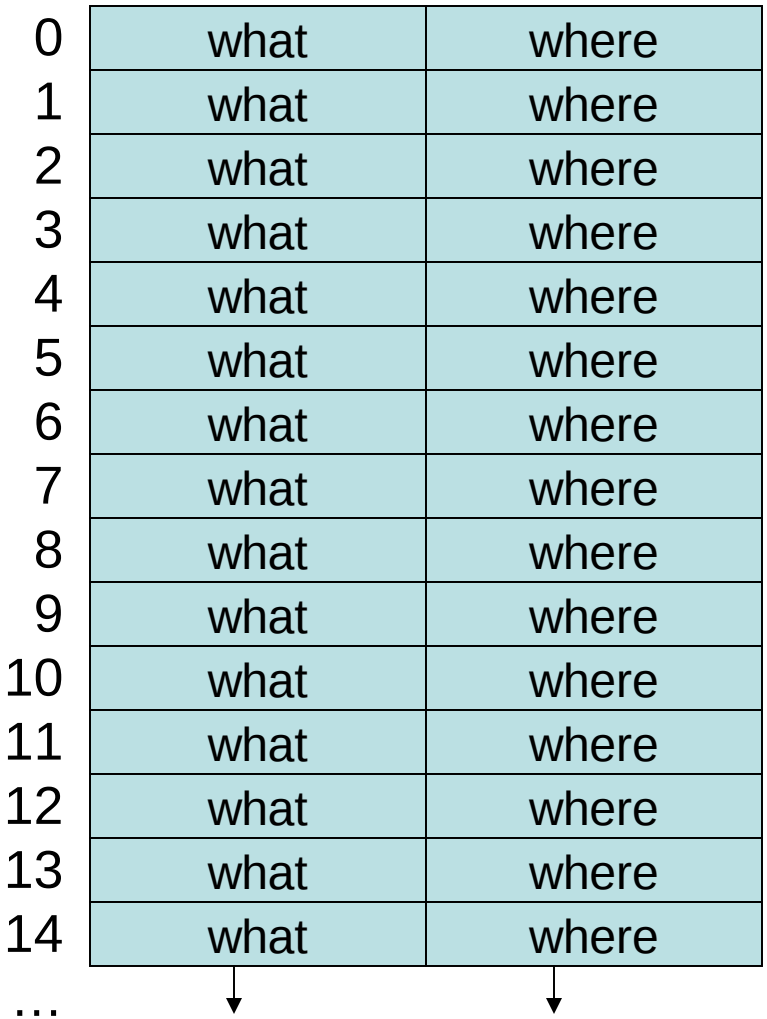
- We created a preliminary implementation for our ‘Host’ data-structures, and part of its executable code
- Some ‘external’ data-items are referenced
- Our demo-program’s ‘Control’ component remains to be designed and implemented
- The required CPU initializations still have to be considered (and then implemented)

The ‘machine’ array

- We contemplate doing initialization of our guest’s Virtual Machine Control Structure within a single program-loop by our host
- That loop refers to an array of parameters that we’ve named ‘machine[]’ – its entries will need to be initialized
- We foresee that our VMM will do reading of many diagnostic parameters in a loop, using an array that we’ve names ‘results’

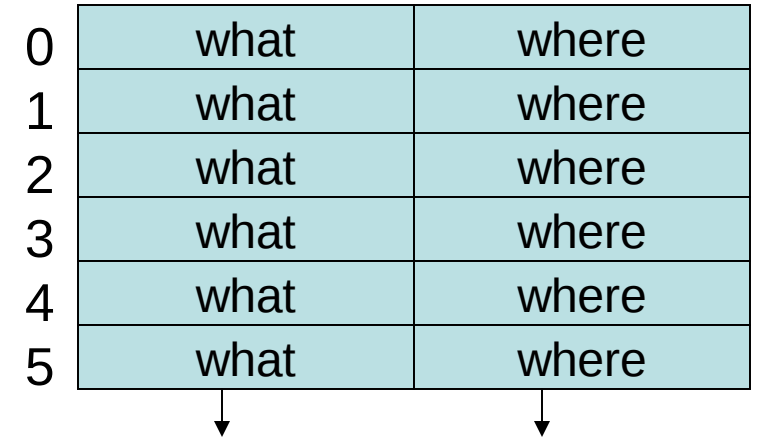
The array-formats

machine[]:



The ‘what’ fields contain encodings for parameter-elements in a VMCS and the ‘where’ fields hold pointers to variables in our program where the parameter-values will be stored

results[]:



CPU initializations

- As with Intel’s EM64T features, which are not automatically enabled at startup-time, the VTX features also need to be ‘enabled’
- In a few cases, these will requires some additional background to be understood
- In other cases, they will merely require setting some extra bits in familiar registers

CPU has VMX support?

- Your software can confirm that your CPU implements the VMX instruction-set: use the CPUID instruction with EAX=1, and verify that bit #5 in register ECX is ‘1’

```
# using CPUID to check for Virtualization Technology support

        mov     $1, %eax
        cpuid
        bt      $5, %ecx
        jnc     no_vmx
        jmp     ok_vmx
```

Feature Control Register

This is a Model-Specific Register having register-index 0x0000003A

Bit 2: ENABLE

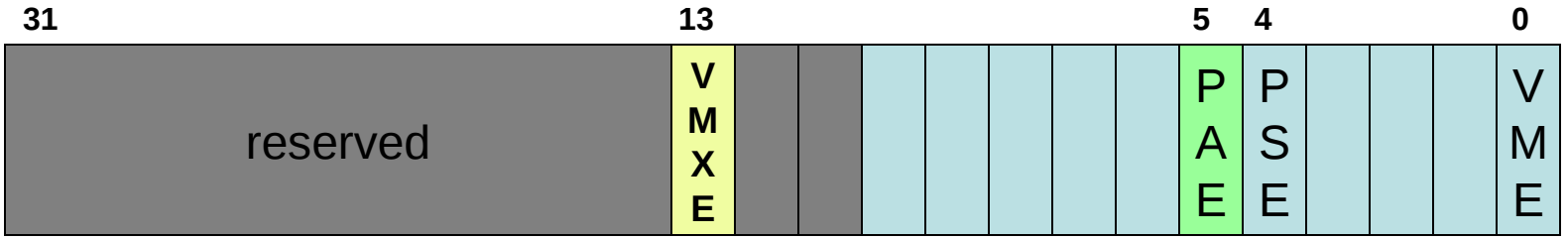
When this bit is 0, any attempt to execute the VMXON instruction causes a General Protection Exception (i.e., VMX will be unavailable)

Bit 0: LOCK

When this bit is 1, then the Feature Control Register's value will be 'locked' (i.e., any attempt to execute WRMSR to modify the contents of this register will cause a General Protection Exception) until power-up.

NOTE: On our 'anchor' machines the ROM-BIOS start-up code will initialize and 'lock' this register (based on an option our SysAdmin selected during SETUP).

Control Register CR4



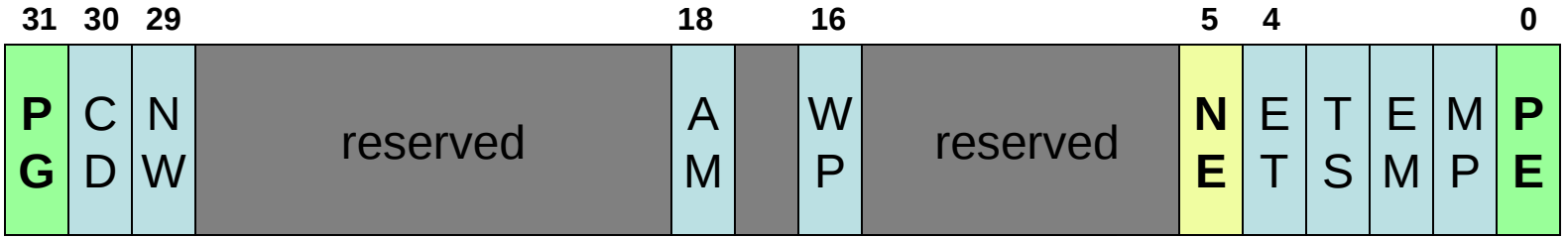
VMXE (Virtual Machine Extensions Enabled)
1=yes, 0=no (default is no)

PAE (Page-Addressing Extensions)
1 = enabled, 0 = disabled
(must be enabled for 64-bit mode)

PSE (Page-Size Extensions)
1 = enabled, 0 = disabled

VME
(Virtual-8086 Mode Extensions)
1=on, 0=off

Control Register CR0



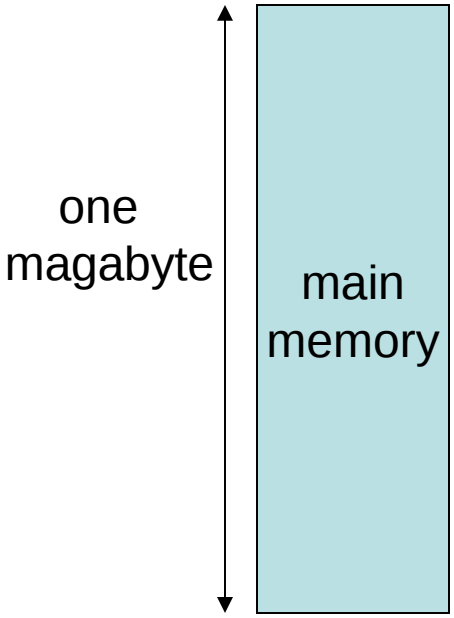
NE (Numeric Error)
1= native internal mechanism
0 = legacy PC-style mechanism
(must be ‘1’ for VMX operations)

PG (Paging): 1=enabled, 0=disabled
(must be ‘1’ for 64-bit mode)

PE (Protection): 1=enabled, 0=disabled
(must be ‘1’ for 64-bit mode)

The A20 address-line

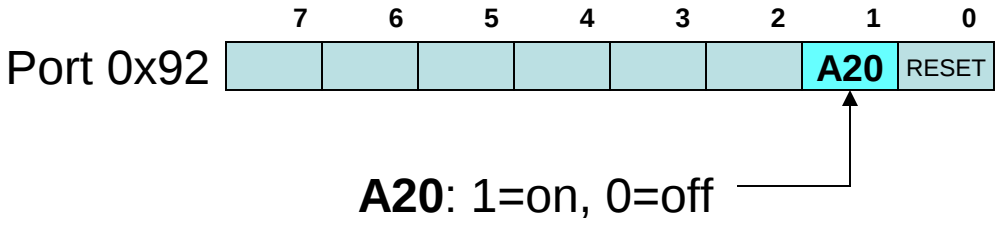
On contemporary platforms (with more than 20 address-bus lines), any faithful emulation of the 8086 processor's 'real-mode' addressing-scheme at startup requires 'forcing' an address-wraparound at the 1MB boundary, accomplished by turning off the A20 address-line



Special circuitry is available for turning "on" or "off" the function of the 21st address-line

VMX-operation requires that A20 must be "on"

The state of the A20-line can be controlled via software -- by toggling bit #1 at I/O-port 0x92



Turning “on” the A20-line

- Caution must be observed when turning “on” the A20 address-line via port 0x92 (since other bits affect vital operations!)

```
# this code-fragment turns the A20-line “on”
        in      $0x92, %al
        or      $0x02, %al
        out     %al, $0x92

# this code-fragment turns the A20-line “off”
        in      $0x92, %al
        and     $0xFD, %al
        out     %al, $0x92
```

VMCS

- Your ‘Host’ and ‘Guest’ each will need to access a page-aligned VMCS region that is initialized with the VMX version-ID in its first longword
- The value to use for that version-Identifier can be discovered from reading the VMX Capability MSRs (register-index 0x480)
- For our Core-2 Duo processors, it’s 7

Alignment and initialization

- Here is assembly language code you can add that will set up one VMCS region:

```
            .equ    ARENA, 0x10000          # program load-address

            .section .data
#-------------------------------------------------------------------------------
            .align  0x1000
vmcs1:      .long   0x00000007              # our VMX version-ID
            .zero   4092                    # zeros in rest of frame
#-------------------------------------------------------------------------------
region:     .quad   vmcs1 + ARENA           # physical address of vmcs1
#-------------------------------------------------------------------------------
```

In-class exercise #1

- Assemble the ‘vmxstep1.s’ and ‘trystep1.s’ files, then link them (using our ‘ldscript’) to get a binary-executable named ‘trystep1.b’
- Install ‘trystep1.b’ on the ‘/dev/sda4’ disk-partition of you assigned ‘anchor’ machine and verify that the guest-message can be received by ‘colby’ (via its serial cable)

In-class exercise #2

- Use our ‘newapp64.cpp’ utility to quickly make the assembly language source-code for a test-platform you can use to try executing the Intel ‘**vmxon**’ instruction:
  - You need a page-aligned 4K memory-region with first longword initialized to version-ID=7
  - You need a quadword-size variable holding the physical-address for that page-frame
  - You need to enable A20, set VMXE=1, NE=1
  - Place instruction ‘**vmxon region**’ in 64-bit code