How Bochs Works Under the Hood

2nd edition

By Stanislav Shwartsman

and Darek Mihoka

Jun 2012

Table of Contents

Overview

The Bochs virtual PC consists of many pieces of hardware. At a bare minimum there are always a CPU, a PIT (Programmable Interval Timer), a PIC (Programmable Interrupt Controller), a DMA controller, some memory (this includes both RAM and BIOS ROMs), a video card (usually VGA), a keyboard port (also handles the mouse), an RTC with battery backed NVRAM, and some extra motherboard circuitry.

There might also be a NE2K ethernet card, a PCI controller, a Sound Blaster 16, an IDE controller (+ harddisks/CDROM), a SCSI controller (+ harddisks), a floppy controller, an APIC ..

There may also be more than one CPU.

Most of these pieces of hardware have their own C++ class - and if bochs is configured to have more than one piece of a type of hardware, each will have its own object.

The pieces of hardware communicates over a couple of buses with each other - some of the things that the buses carry are reads and writes in memory space, reads and writes in I/O space, interrupt requests, interrupt acknowledges, DMA requests, DMA acknowledges, and NMI request/acknowledge. How that is simulated is explained later *FIXME*.

Other important pieces of the puzzle are: the options object (reads/writes configuration files, can be written to and queried while Bochs is running) and the GUI object. There are many different but compatible implementations of the GUI object, depending on whether you compile for X (Unix/Linux), Win32, Macintosh (two versions: one for Mac OS X and one for older OS's), Amiga, etc.

And then there is the supporting cast: debugger, config menu, panic handler, disassembler, tracer, instrumentation.

Weird macros and other mysteries

Bochs has many macros with inscrutable names. One might even go as far as to say that bochs is macro infested.

Some of them are gross speed hacks, to cover up the slow speed that C++ causes. Others paper over differences between the simulated PC configurations.

Many of the macros exhibit the same problem as C++ does: too much stuff happens behind the programmer's back. More explicitness would be a big win.

static methods hack

C++ methods have an invisible parameter called *this* pointer - otherwise the method wouldn't know which object to operate on. In many cases in Bochs, there will only ever be one object - so this flexibility is unnecessary. There is a hack that can be enabled by #defining BX_USE_CPU_SMF to 1 in config.h that makes most methods static, which means they have a "special relationship" with the class they are declared in but apart from that are normal C functions with no hidden parameters. Of course they still need access to the internals of an object, so the single object of their class has a globally visible name that these functions use. It is all hidden with macros.

Declaration of a class, from cpu/cpu.h:

```
...
#if BX_USE_CPU_SMF == 0
// normal member functions.  This can ONLY be used within BX_CPU_C classes.
// Anyone on the outside should use the BX_CPU macro (defined in bochs.h)
// instead.
#  define BX_CPU_THIS_PTR  this->
#  define BX_CPU_THIS      this
#  define BX_SMF
// with normal member functions, calling a member fn pointer looks like
// object->*(fnptr)(arg, ...);
// Since this is different from when SMF=1, encapsulate it in a macro.
#  define BX_CPU_CALL_METHOD(func, args) \
            (this->*((BxExecutePtr_tR) (func))) args
#  define BX_CPU_CALL_METHODR(func, args) \
            (this->*((BxResolvePtr_tR) (func))) args
#else
// static member functions.  With SMF, there is only one CPU by definition.
#  define BX_CPU_THIS_PTR  BX_CPU(0)->
#  define BX_CPU_THIS      BX_CPU(0)
#  define BX_SMF           static
#  define BX_CPU_CALL_METHOD(func, args) \
            ((BxExecutePtr_tR) (func)) args
#  define BX_CPU_CALL_METHODR(func, args) \
            ((BxResolvePtr_tR) (func)) args
#endif

...
class BX_CPU_C : public logfunctions {
public:
    BX_CPU_C(unsigned id=0);
   ~BX_CPU_C();

  BX_SMF void cpu_loop(void);
...
};
```

And cpu/cpu.cc:

```
...
#define NEED_CPU_REG_SHORTCUTS 1
#include "bochs.h"
#include "cpu.h"
#define LOG_THIS BX_CPU_THIS_PTR
...
void BX_CPU_C::cpu_loop(void)
{
#if BX_DEBUGGER
  BX_CPU_THIS_PTR break_point = 0;
  BX_CPU_THIS_PTR magic_break = 0;
  BX_CPU_THIS_PTR stop_reason = STOP_NO_REASON;
#endif

  if (setjmp(BX_CPU_THIS_PTR jmp_buf_env)) {
    // can get here only from exception function or VMEXIT
...
}
```

Ugly, isn't it? If we use static methods, methods prefixed with BX_SMF are declared `static` and references to fields inside the object, which are prefixed with BX_CPU_THIS_PTR, will use the globally visible object `bx_cpu`. If we don't use static methods, BX_SMF evaluates to nothing and BX_CPU_THIS_PTR becomes `this->`. Making it evaluate to nothing would be a lot cleaner, but then the scoping rules would change slightly between the two bochs configurations, which would be a load of bugs just waiting to happen.

Some classes use BX_SMF, others have their own version of the macro, like BX_VGA_SMF in the VGA emulation code.

CPU and memory objects in UP/SMP configurations

The CPU class is a special case of the above: if bochs is simulating a uni-processor machine then there is obviously only one bx_cpu_c object and the static methods trick can be used. If, on the other hand, bochs is simulating SMP machine then we can't use the trick. In a UP configuration, the CPU object is declared as `bx_cpu`. In an SMP configuration it will be an array of pointers to CPU objects (`bx_cpu_array[]`).

Access of a CPU object often goes through the `BX_CPU(x)` macro, which either ignores the parameter and evaluates to `&bx_cpu`, or evaluates to `bx_cpu_array[n]`, so the result will always be a pointer.

If static methods are used then BX_CPU_THIS_PTR evaluates to BX_CPU(0)->.

Ugly, isn't it? May be yes, but this simple optimization is actually one of the most efficient optimizations between all Bochs CPU emulation tricks.

BX_DEBUG/BX_INFO/BX_ERROR/BX_PANIC -- logging macros

go through a generic tracing mechanism. Can be switched individually on/off. Sometimes might eat a lot of CPU time – BX_DEBUG and BX_ERROR can be eliminated at compilation time using configure option.

BX_TICK1, BX_TICKN(n), BX_TICK1_IF_SINGLE_PROCESSOR, BX_SYNC_TIME

`BX_TICK1_IF_SINGLE_PROCESSOR`, only used in cpu.cc -- and only confuses the matter. It calls `BX_TICK1` on a single-processor and nothing on SMP.

Memory - An Introduction

Both RAM and BIOS'es. BIOSes can be loaded individually. The CPU is accessing memory through direct host access (see Guest2Host TLB) or through access_read_physical() /access_write_physical() interface. The interface is the wrapper around BX_MEM_C::readPhysicalPage and BX_MEM_C::writePhysicalPage which handles the local APIC access. Each processor has its own local APIC which is memory mapped to a specific 4Kb physical page. The accesses to local APIC page go into local APIC device which is physically part of the CPU and not seen as access to the RAM.

CPU masks the A20 bit by itself if required but memory also does the A20 masking for every memory access, this is mainly affects DMA access by devices which don't care for A20 by themselves.

Go over:

1. ReadPhysicalMemory, WritePhysicalMemory
2. SMM
3. MONITOR/MWAIT
4. Direct memory access getGuest2HostAddr
5. Advanced: Swapping
6. Register memory handlers (supported: read, write an execute handlers)

The Basic CPU

Simple CPU: no data caches! Does have TLBs and decoded instruction trace cache.

Some real IA-32 implementations distinguish between TLBs for code and for data -- we don't. We save some time on having 1024 TLB entries, which is a lot more than almost all real CPUs have at the moment.

Some of the things we have to emulate - The IA-32

Real mode. Protected mode. Virtual 8086 mode. Long mode. 16/32-bit code, segments, instruction prefetch queue, TLBs, writes to memory can be executed "immediately" (makes things a bit harder for us later on), extraordinarily complex and varied instruction formats. Four different privilege levels, and then a system management mode or hardware virtualozation on top of that for some of the CPUs. Segment registers, capability of overriding the default segment register for the instruction, usually DS, but sometimes SS. Prefixes, address and operand size changes, tons of flags, tons of special cases about which registers can be used for what purpose. Totally free alignment of both code and data. Self modifying code support. Instructions can be one to sixteen bytes. IO Privilege level, IO privilege map, paging, vector ISA extensions.

Yada yada, you get the picture...

The main CPU loop

In general, a CPU emulation loop looks very similar to that of a hardware non-pipelined CPU would have. Every emulated instruction passes through the same basic stages during the emulation. See the basic diagram. The Bochs is not very basic so the diagram will get more complicated with a time ☺

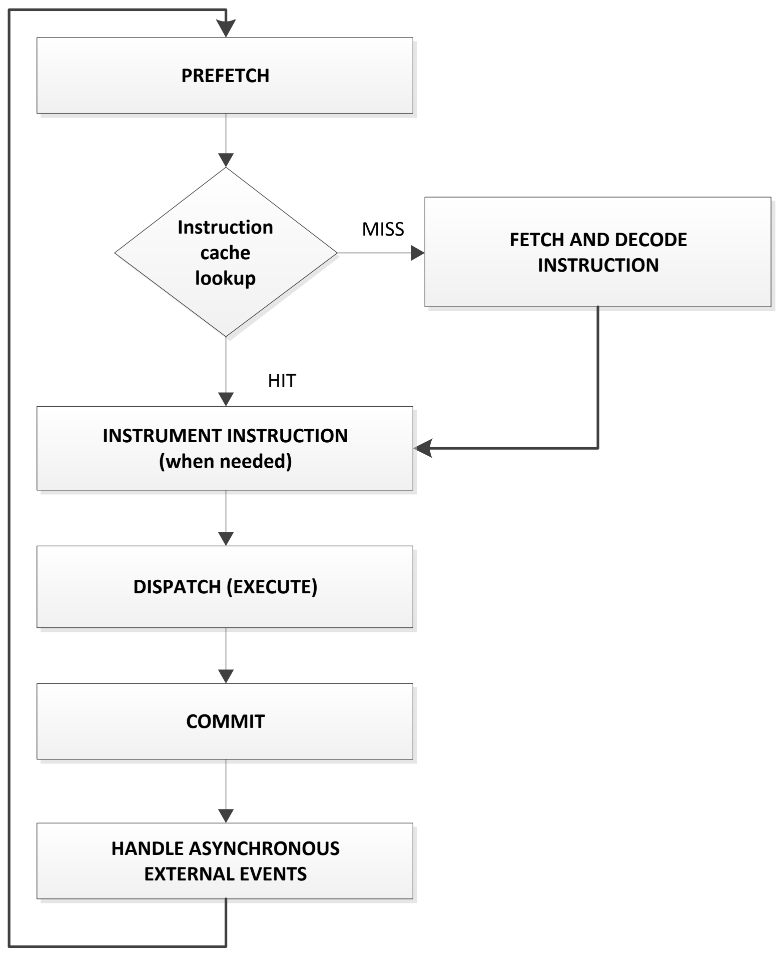
PREFETCH
Instruction cache lookup
MISS
FETCH AND DECODE INSTRUCTION
HIT
INSTRUMENT INSTRUCTION (when needed)
DISPATCH (EXECUTE)
COMMIT
HANDLE ASYNCHRONOUS EXTERNAL EVENTS

Figure 1: Basic CPU emulation loop

Decoding instructions is *HARD*

On nicer processors, decoding instructions is an easy task. It's especially nice on the MIPS and the Alpha. On IA-32 it's just about as lousy as it can get :/

In order to reduce the complexity a bit, all the decoding of the operand fields is done first and then the instruction is executed by one of many hundred small methods that don't have to care (much) about their operands.

bxInstruction_c

All relevant instruction information stored in structure called bxInstruction_c. It has all fields like instruction length, dispatch function pointer, opcode, source and destination registers, immediate.

Individual register index is stored in Bit8u variable. Can't pack two registers to same variable, 4 bits is already not enough for a register. We have 16 registers in x86-64 mode, RIP and NULL register.

The sizeof(bxInstruction_c) == 32 on 32-bit architectures and it is even more on 64-bit host. Too much, could be reduced to 16 bytes. This has real performance behind it.

Every instruction is emulated by or two functions. All common and performance critical opcodes are emulated within single function handler. For less critical and more complex instructions, like LOAD+OP there might be two handler functions, one for LOAD and another for the OP. Pointers to both handler functions also stored in the bxInstruction_c (*execute* and *execute2*). The *execute2* is directly called by *execute*.

FetchDecode

Decoding is done with table-based decoder. It is not very optimal. With the time tables grew too much and became too complicated for understanding. Looking up decoder tables can be optimized for space – should help to reduce decoder host cache miss rate.

How exceptions are implemented?

All instructions restartable from the register state + bxInstruction_c. Never possible to generate exception /after/ changing the visible state except for RIP (and in some cases RSP) registers.

The CPU loop sets up return point using setjmp(), exception will call longjmp() and appear at the beginning of CPU loop.

Optimizations for performance

These and other Bochs optimizations were presented in 1st AMAS-BT workshop during ISCA-35 in Beijing by Stanislav Shwartsman and Darek Mihocka. The paper “Virtualization without direct execution - designing a portable VM” and presentation foils could be downloaded from Bochs website.

Guest –to-Host TLB

Similar to all hardware x86 processors Bochs uses TLB to cache linear to physical address translations. Bochs TLB is organized as direct mapped cache which can hold up to 1024 4K pages, so it easy to find a translation for a linear address using just single memory access and a comparison.

Bochs TLB entry:

```
typedef struct {
   bx_address lpf;                     // linear page frame
   bx_phy_address ppf;                 // physical page frame
   bx_hostpageaddr_t hostPageAddr;     // direct host pointer for efficient memory access
   Bit32u accessBits;                  // access control bits
   Bit32u lpf_mask;                    // linear address mask of the page size
} bx_TLB_entry;
```

- The *lpf* or a Linear Page Frame is the tag of the TLB cache, all other fields are the TLB data that required for correctness and used on almost every memory access. The *lpf* address is always 4K aligned.

- The *ppf* or a Physical Page Frame is a emulated physical address corresponding to the *lpf* and it is also always 4K aligned.

- The *lpf_mask* field is required for x86 emulation correctness. The x86 architecture allows allocating of pages >4K (with 2M, 4M or even 1Gb per page). In hardware processors usually there are several TLBs, a separate TLB for each possible page size. Bochs automatically fractures all large pages to the 4K chunks and caches only fractured 4K pages. The 4K entries in the Bochs TLB are really parts of the architectural large pages and should be handled accordingly.

  For example when INVLPG instruction is executed for a large page, all chunks of the large page have to be invalidated from the Bochs TLB. The *lpf_mask* field holds a bitmask to remember a size of architectural page that the TLB entry belongs to. For 4K pages it will be simply 0xfff, for 2M – 0x1fffff and etc. Bochs TLB entry has to be invalidated if INVLPG linear address matches TLB entry with *lpf_mask.*

- The *hostPageAddr* is the direct pointer to the Bochs’ host memory where emulated physical address page *ppf* can be found. The pointer is also 4K aligned. In most of the cases it is possible to calculate direct pointer *hostPageAddr* and access directly the Bochs’ host memory instead of calling expensive BX_MEM_C::writePhysicalPage or BX_MEM_C::readPhysicalPage.

The direct pointer is allocated by BX_MEM_C::getHostMemAddr function. If it returns zero, it means that a pointer to the host space could not be generated, likely because that page of memory is not standard memory (it might be memory mapped IO, ROM, etc). Consult BX_MEM_C::getHostMemAddr for complete list of reason to veto direct host operations.

- The accessBits is used for a very efficient paging permissions check. The goal is to be able, using only the current privilege and access type, to determine if the page tables allow the access to occur or at least should rewalk the page tables. On the first read access, permissions are set to only read, so a rewalk is necessary when a subsequent write fails the tests. This allows for the dirty bit to be set properly, but for the test to be efficient. Note that the CR0.WP flag is not present. The values in the following flags is based on the current CR0.WP value, necessitating a TLB flush when CR0.WP changes.

  The TLB access permissions checks are:

  AccessOK = (accessBits & ((E<<2) | (W<<1) | U)) == 0

E:	*1=Execute, 0=Data*
W:	*1=Write, 0=Read*
U:	*1=CPL3, 0=CPL0-2*

  Thus for reads, it is:
  AccessOK = (accessBits & U) == 0
  for writes:
  AccessOK = (accessBits & (0x2 | U)) == 0
  and for code fetches:
  AccessOK = (accessBits & (0x4 | U)) == 0

Note that the TLB should have TLB_NoHostPtr bit set in the lpf[11] when direct access through host pointer is NOT allowed for the page. A memory operation asking for a direct access through host pointer does not have TLB_NoHostPtr bit set in its lpf (which is 4K always aligned) and thus will get TLB miss result when the direct access is not allowed.
A memory operation that do not care for a direct access to the memory through host pointer should implicitly clear lpf[11] bit in the TLB before comparison with memory access lpf.

Guest –to-Host TLB invalidation

The Bochs' guest to host TLB has to be invalidated in all the cases mentioned in Intel Software Development Manual:

- Flush all TLB entries when any of CR0.PE/CR0.PG/CR0.WP are modified.
- Flush all non-global TLB entries on any write to CR3 register
- Flush all TLB entries when any of CR4.PAE, CR4.PSE, CR4.PGE, CR4.PCIDE or CR4.SMEP are modified.
- What about EFER.NXE ?

and also in some extra Bochs' specific cases:

- Write to MSR_APICBASE flush the TLB.
  It is possible that the TLB had an entry pointing to the new APIC location.

- Write to DR0-DR3 register invalidates all pages that collide with the new DR register value from the TLB.
- Write to DR7 register flush the TLB.
  Instead we could invalidate all pages that collide with the DR0-DR3 values but flushing of the whole TLB simply faster.

Code "Prefetching"

Instruction fetch – might be it sounds simple but in fact the CPU has to perform a lot of complicated operations before from current instruction pointer (EIP) it can obtain the actual instruction bytes.

The fetch is a virtual memory access and must follow all the rules of a regular memory access. The EIP has to be checked for segment limit and segment permissions (in 64-bit mode for canonical) and translated to physical address through page tables, paging execution permissions need to be checked as well. With the physical address need to access the memory and bring the instruction bytes.

Of course we don't want to do all of this for every instruction.

Might be only once in 4-Kb page …

We would like to keep a host pointer to the current 4Kb page that instructions being fetched from. Thus if we know to translate quickly from EIP to the offset in that page we can save the expensive walk to the memory object. The trick is to keep a pre-calculated value (*eipPageBias*) which gonna help us to perform the translation.

Skip the following part if you believe in pure magic and don't like dirty math.

So we have the EIP and want to have the offset inside the 4-Kb physical page instead. In better architectures (not like x86) we would take 12 LSB of the EIP and it would be enough. But in x86 the EIP is not necessary aligned with page because of segment base which is not necessary 4-Kb aligned. After taking the segment base into account we get the LIP or linear EIP.

LIP := EIP + CS.base

The code is located in the linear page address *linPageBase* which is translated to the physical address *phyPageBase*.

So we have:

LIP := linPageBase + pageOffset = EIP + CS.base

pageOffset = EIP + CS.base – linPageBase

taking the EIP out and get:

eipPageBias := pageoffset – EIP = CS.base – linPageBase

As you can see the *eipPageBias* value is the same for any address inside the 4-Kb page so it can be kept in the CPU variable and used to obtain *pageOffset* from the EIP when needed.

pageOffset := eipPageBias + EIP

We can also store in the CPU *phyPageBase* value and obtain instruction physical address when required:

Instruction physical address := phyPageBase + pageOffset

So far, so good. But in order to use this trick we need to know when the EIP crossed a 4-Kb page in order to recalculate *eipPageBias* value. But this is very easy to do. The *pageOffset* indicate the distance between *linPageBase* and current LIP. The distance more than 4K means we crossed a page.

Every time we cross a 4-Kb page the CPU has to call the BX_CPU_C::prefetch(). The function checks the EIP for segment limits and canonical, re-calculates *eipPageBias,* translates current *linPageBase* to *phyPageBase* and prepares direct pointer to the code page for future fetches.

This is detected using simple comparison:

eipBiased >= BX_CPU_THIS_PTR eipPageWindowSize

where *eipBiased* is calculated from the current RIP:

bx_address eipBiased = RIP + BX_CPU_THIS_PTR eipPageBias;

When CS segment is reloaded for any reason (for example because a far branch was executed) the BX_CPU_C::prefetch() call has to be requested explicitly by calling the BX_CPU_C::invalidate_prefetch_q() function. The function simply clears *eipPageWindowSize* so next time the *eipBiased* check will fail and call the BX_CPU_C::prefetch() as required.

The Anatomy of Memory Accesses

x86: instructions access memory through linear address – segment:offset pair. The offset called virtual address:

*Effective address = (BASE + INDEX*SCALE + DISPLACEMENT) MOD ADDRSIZE*

Linear address = (SEGMENT.BASE + Effective Address) MOD OPSIZE

Segment limit and protection checks have to be done on effective address level, alignment and canonical checks on linear address. There are four different paging modes to translate linear address into physical and paging also can be disabled.

Lookup the TLB – TLB keeps physical address and permissions bits. Also keep host pointer for direct access if possible.

Memory access functions are critical for performance so split them as much as possible. There is different function for each possible data size. Also split for 32-bit and 64-bit mode. 32-bit accesses functions check for segment limit and permissions, 64-bit check for canonical.

Lookup the TLB – TLB keeps physical address and permissions bits. Also keep host pointer for direct access if possible. In case of TLB miss or direct access is not possible for some reason go slow path – call access_linear (read or write) function. The access_linear correctly handles page split and linear to physical address translation.

Example of read memory access function (yes, look pretty big):

```
Bit32u BX_CPU_C::read_virtual_dword_32(unsigned s, Bit32u offset)
{
    Bit32u data;
    bx_segment_reg_t *seg = &BX_CPU_THIS_PTR sregs[s];

    if (seg->cache.valid & SegAccessROK) {
        if (offset < (seg->cache.u.segment.limit_scaled-2)) {
accessOK:
        Bit32u laddr = get_laddr32(s, offset);
        unsigned tlbIndex = BX_TLB_INDEX_OF(laddr, 3);
        Bit32u lpf = AlignedAccessLPFOf(laddr, (3 & BX_CPU_THIS_PTR alignment_check_mask));
        bx_TLB_entry *tlbEntry = &BX_CPU_THIS_PTR TLB.entry[tlbIndex];
        if (tlbEntry->lpf == lpf) {
            // See if the TLB entry privilege level allows us read access from this CPL
            if (! (tlbEntry->accessBits & USER_PL)) { // Read this pl OK.
                bx_hostpageaddr_t hostPageAddr = tlbEntry->hostPageAddr;
                Bit32u pageOffset = PAGE_OFFSET(laddr);
                Bit32u *hostAddr = (Bit32u*) (hostPageAddr | pageOffset);
                ReadHostDWordFromLittleEndian(hostAddr, data);
                return data;
            }
        }
        if (BX_CPU_THIS_PTR alignment_check()) {
            if (laddr & 3) {
                BX_ERROR(("read_virtual_dword_32(): #AC misaligned access"));
                exception(BX_AC_EXCEPTION, 0);
            }
        }
        access_read_linear(laddr, 4, CPL, BX_READ, (void *) &data);
        return data;
  }
  else {
      BX_ERROR(("read_virtual_dword_32(): segment limit violation"));
      exception(int_number(s), 0);
  }
}

if (!read_virtual_checks(seg, offset, 4))
  exception(int_number(s), 0);
  goto accessOK;
}
```

The single functional is almost a masterpiece developed by Bochs authors and it demonstrates a collection of tricks to make the common case of the memory access as fast as possible:

- The first if statement checks for (seg->cache.valid & SegAccessROK) != 0

Every memory access in real, protected or compatibility mode has to check the segment descriptor cache for access permissions (including valid bit, present bit and segment descriptor type checking). Avoid the check for every memory access by caching the AccessROK and AccessWOK in seg->cache.valid bit. In rare cases when cached accessOK bit is clear, call for read_virtual_checks.

- Access the Guest-To-Host TLB. Calculate the TLB entry index using the last byte of the memory access:

  ```
  uint tlbIndex = BX_TLB_INDEX_OF(laddr, 3 /* mem access size = 4 */);
  ```

  this way all page-split accesses are guaranteed to fail (miss in TLB lookup). The page split accesses require access to two physical pages and they are handled separately in *access_read_linear* code.

- Similarly filter misaligned memory accesses when x86 hardware alignment check is enabled. The LPF values (Linear Page Frame) stored in the TLB are always 4K-aligned. The *AlignedAccessLPFOf* macro will keep few LSB when calculating the LF for unaligned access and cause TLB miss.

- Check Guest-to-Host TLB accessBits for U/S and R/W permissions and host direct access (see Guest-to-Host TLB section).

- The actual memory fast access is only five lines of code:

  ```
  bx_hostpageaddr_t hostPageAddr = tlbEntry->hostPageAddr;
  Bit32u pageOffset = PAGE_OFFSET(laddr);
  Bit32u *hostAddr = (Bit32u*) (hostPageAddr | pageOffset);
  ReadHostDWordFromLittleEndian(hostAddr, data);
  return data;
  ```

  All uncommon scenarios (like page split access, HW alignment check enabled, direct host access is not allowed and etc) are handled later, almost everything inside *access_read_linear/access_write_linear* functions.

  x86 hardware breakpoints are also checked in *access_read_linear/access_write_linear*, the Guest-To-Host TLB guarantees that there is no direct host access allowed to pages with x86 hardware breakpoints enabled.

In long 64-bit mode there are no segment access checks but everything else remains the same. There is an assumption that non-canonical access cannot be allocated in the TLB so canonical check can also be moved to *access_read_linear/access_write_linear*.

Self-modifying code (SMC) detection

SMC is a pain of the x86 architecture. Every store can potentially modify the instruction flow. There is only one relaxation – instruction cannot modify itself. For repeated instruction, once decoded it can run forever and even overwrite its own instruction bytes, nothing should happen. Once repeat flow was interrupted for any reason (for example by breakpoint or any external event) and it must detect the SMC.

Bochs uses huge data structure called pageWriteStampTable for self-modifying code detection. The pageWriteStampTable includes a single Bit32u value for every 4-Kb page in the physical address space. The physical address space is:

- 32-bit (4GB) without PSE-36 or PAE
- 36-bit (64GB) with PSE-36 or PAE legacy mode
- 40-bit with long mode enabled

Each Bit32u value is a bitmap. A bit is set if corresponding 128-byte block of the 4K page was decoded to an instruction and cached in the trace cache. Any store to physical memory must check the pageWriteStampTable and see if it modifies any instruction bytes.

In case a SMC is detected:

1. Invalidate corresponding entries in the trace cache
2. Clear corresponding bits in bitmap stored in the pageWriteStampTable entry
3. Set async_event indication BX_ASYNC_EVENT_STOP_TRACE which will cause currently executed trace to be stopped and re-fetched.

Bochs always allocates 1048576 Bit32u entries for pageWriteStampTable which can fully cover 32-bit address space. For guests with larger physical address space the pageWriteStampTable wraps so single pageWriteStampTable entry might represent multiple physical pages. And yes, this is too much (pageWriteStampTable uses 4Mb of host RAM) because most of the emulation targets don't have so much physical memory. But I'd prefer to have more memory allocated than have an extra run-time check for every executed store operation.

Trace Cache

```
                "Almost all programming can be viewed as an exercise in caching"
                                                           -- Terje Mathisen
```

Decoded instructions form traces. Trace is not allowed to cross 4K boundary and must end with any taken branch instruction. Tracing over non-taken branches is not allowed (yet). The traces are formed on decode time and cached in the Trace Cache structure. The trace cache is physically + mode indexed. The index for trace cache is formed by XOR'ing current instruction physical address obtained by:

bx_phy_address pAddr = BX_CPU_THIS_PTR pAddrPage + eipBiased

and current execution mode (*fetchModeMask*) of the emulated CPU. The *fetchModeMask* is currently 4-bit wide and includes the following fields:

0	CS.D_B (16/32-bit mode switch)
1	CS.L (64-bit mode switch)
2	SSE OK (allowed to execute SSE instructions) SSE instructions #UD if CR0.TS=1, CR0.EM=1 or CR4.OSFXSR=0
3	AVX OK (allowed to execute AVX instructions) AVX instructions #UD if CR0.TS=1, CR4.OSXSAVE=0, AVX is disabled in XCR0 or not in protected mode.

Thus the same opcode might be decoded to completely different Bochs handler depending on current CPU mode. The *fetchModeMask* has to be re-calculated every time the CPU mode is changed, CS segment is reloaded when CR0, CR4 or XCR0 is modified.

The trace cache is organized as Direct Mapped array of 64K entries. Every entry in the trace cache holds a sequence of instructions without branches with maximum length 32 instructions. Storing all these instructions directly in the trace cache entries would require huge amount of host memory, enough to hold 2M of instructions. This is would be enormous waste of space because average trace length hardly exceeds 5 instructions. In order to solve that problem the trace cache manages its own memory pool of *bxInstruction_c* which is large enough but still significantly smaller than 2M of instructions (currently memory pool is configured to hold 384K of instructions). Instead of storing a whole 32 instructions trace only a pointer to the memory pool location is stored in the trace cache entry. When the memory pool becomes full the trace cache simply invalidated.

In order to attack aliasing issues in the direct mapped cache, the trace cache includes also a victim cache of 8 entries. Every replaced trace pointer is saved in the victim cache and every trace miss first looks up the victim cache before starting to fetch and decode instructions, which is much heavier task.

Trace Cache and Handlers Chaining

The handlers chaining optimization was presented in the 4th AMAS-BT workshop (http://amas-bt.cs.virginia.edu/program-2011.htm) during ISCA-38 in San Jose, California by Jens Troeger and Darek Mihocka in the paper “Fast Microcode Interpretation with Transactional Commit/Abort”.

Bochs adapted the handlers chaining algorithm since release Bochs 2.5.

In the naïve interpreter implementation all handler functions are called from single place (usually from the main interpretation cpu_loop) through indirect function call. In this case the indirect call would have many different targets and as result become very hard to predict even for modern processors with sophisticated branch prediction hardware. The easiest way to help the branch prediction is to spreading the indirect calls branches over many call sites. This makes each call site much more predictable and also gives more information to the host’s branch predictor.

With handlers chaining optimization each handler computes a pointer to the next handler to execute (by singly incrementing the bxInstruction_c* poiner using pointer arithmetic). Then the next handler is called directly from the current instruction handler.

Instructions that must end trace have two options:

- Compute a pointer to a next instruction (by lookup up or even fetch a next trace) and call the next trace first instruction handler directly.
- Return to the main cpu_loop.

Current Bochs implementation simply returns to the main CPU loop. Bochs is designed to be portable and therefore cannot rely on host compiler to optimize call stack. We must return to the main cpu_loop eventually because the host stack growth caused by endless handlers chaining cannot continue forever. Also Bochs increments the system timers after each trace (wish list: timer is still not decoupled from instructions execution).

Typical instruction handler

There are different kinds of handlers. The handlers that must end trace and return to the main cpu_loop behave a little different of the others.

Typical handler looks like:

```
BX_INSF_TYPE BX_CPU_C::HANDLER_NAME (bxInstruction_c *i)
{
   /* handler code */
   BX_NEXT_INSTR(i);
}
```

The handler that ends a trace looks very similar:

```
BX_INSF_TYPE BX_CPU_C::HANDLER_NAME (bxInstruction_c *i)
{
   /* handler code */
   BX_NEXT_TRACE(i);
}
```

All the magic is in macros. Ignore BX_INSF_TYPE for a moment, it is currently void, the macro was created in preparation for case when a handler would like to return a value.

Let's look closer to the BX_NEXT_INSTR and BX_NEXT_TRACE macros.

Things that are done after every instruction:

- Commit the instruction by advancing the prev_rip variable.
- Call for instrumentation callbacks if required (another macro).
- Increment emulated instructions counter (icount).
- Check for async events (also takes care for self and cross modifying code).

The BX_NEXT_TRACE ends here and transfers control back to the caller and all the way back to heart of cpu_loop. The BX_NEXT_INSTR calls for next instruction directly.

Lazy flags handling

X86 has 6 arithmetic flags (OSZAPC for Overflow, Sign, Zero, Aux, Parity and Carry). Almost every logical and arithmetic instruction (such as ADD, CMP, MUL, SHL, etc.) updates these flags but fewer instructions read these flags, and usually only one or two of the flags. Calculating the exact bit values of these 6 flags for every instruction which generates them can be very complicated and expensive; and wasteful if the flags are not needed.

"Lazy flags" computation is a technique of storing information about the last instruction that modified the arithmetic flags in a way that permits any specific flag to be determined later. It is relatively inexpensive to store such information compared to actually deriving the flags completely.

Earlier Bochs versions stored 4 values for arithmetic operations (src1, src2, dest and opcode) and 2 values for logic operations (when src1 and src2 are not required) to describe the flags. The flags were computed using giant switch statement which went over possible opcodes that modify flags. In effect, the instruction which generates the arithmetic flags was replayed. This was expensive due to the fact that several flow controls were required to call the helper function and dispatch the switch statement.

The latest Bochs implementation is much more optimal. It uses the insight that the 6 arithmetic flags are derived in one of two ways:

1. Zero, Sign, and Parify flags (ZF, SF, PF) are based simply on the result of an operation. ZF is set if the result is zero, cleared otherwise. SF is set based on the high bit of the result, which is the sign flag of a signed integer. PF is the even parity of the low 8 bits of the result. These 3 flags can be derived *without* replaying the instruction by simply recording the last result of an operation.
2. Carry, Overflow, and Aux (CF, OF, AF) are all based on the carry-out value of specific bits. CF is the carry-out value of the high bit of the result, which for a 32-bit integer operation is bit 31. OF is the XOR of the carry-out of the two high bits (which for a 32-bit integer operation are bits 31 and 30). AF is the carry-out of the low nibble, bit 3. These 3 flags can be derived without replaying the instruction if the carry-out vector is stored.

One can therefore avoid the helper function and large switch statement to replay an instruction by recording only two values – the result of an arithmetic or logical operation, and the carry-out bits from bit position 31, 30, and 3 (or 63, 62, and 3 for 64-bit operations).

This approach reduces the lazy flags state from 4 values to 2 values, and avoids unnecessary flow control, thus reducing branch mispredictions. The lazy flags in Bochs are encoded in two variables – *result* and *auxbits*. The *result* variable always holds the sign-extended result of the last logical or arithmetic operation. The *auxbits* variable holds the three carry-out bits used to derive CF, OF, and AF, as well as some additional bits.

The lazy flags layout for a 32-bit guest is shown:

	31	30	29..16	15..8	7..4	3	2	1	0
result	sign-extended last result derives Zero Flag								
	SF				Parity Flag byte				
auxbits	CF	PO	x	PDB	x	AF	x	x	SFD

<u>Legend</u>:
x = don't care.
SF = Sign Flag bit from last result.
SFD = Sign Flag Delta bit, the true Sign Flag returned is SF XOR SFD.
PDB = Parity Delta Byte, the true PF returned is derived from the parity of Parity Flag Byte XOR Parity Delta Byte.
PO = Partial Overflow. This is the carry-out of bit 30. Overflow Flag is derived from CF XOR PO.

Ideally, all logical and arithmetic operations would always update all 6 flags, and no instruction would ever produce an inconsistent state (such as ZF=1 and SF=1 indicating that a result was both zero and negative at the same time!). However, instructions such as POPF and SAHF can in fact produce such states. To handle this, additional bits are stored in the *auxbits* variable which are combined with bits in the *result* variable to derive the SF and PF.

A key optimization is to make the ZF and CF easiest to extract, since traces of real-world Windows and Linux code has shown that almost 90% of all EFLAGS reads access the ZF and most of the rest access the CF. It is permitted to have PF SF OF and AF derivation be more complex, since those are much more rarely read.

When a logical instruction sets the lazy flags state, this requires only two instructions: one two store the result into the *result* variable, and one to store zero into *auxbits* variable. This is due to the fact that most logical instructions such as AND, OR, XOR, TEST are defined to always clear AF CF OF. Arithmetic instructions generate the carry-out vector by using either the ADD_COUT_VEC or SUB_COUT_VEC macros in lazy_flags.h which can produce the carry-out vector in at most 5 logical operations. This compiles into very efficient code that has no branches.

Instructions such as POPF and SAHF and some shift and rotate instructions which can set arbitrary EFLAGS state must manipulate those extra bits in *auxbits* such that PF and SF are derived correctly. Whenever the SF state needs to be set opposite to the high bit in *result* variable, the SFD ("sign flag delta") bit in *auxbits* is set. Bochs will derive the true Sign Flag based on *result*.SF XOR *auxbits*.SFD. Similarly, the parity bits can be overridden the same way, allowing the Parify Flag to be set independently of the last result.

Unlike older Bochs versions new Bochs OSZAPC flags are always lazy. This means that the architectural EFLAGS register in the CPU never have the most updated copy of the flags and they have to be computed on demand. This approach saves an extra check that older Bochs versions had to do to determine if a particular flag is up to date in EFLAGS register or lazy which further reduces branch mispredictions.

Repeat speedups

TBD

Things I lied about

extend down segments, x86 hardware debugger, temporary disabling of interrupts (after SS changes) ;; might have to go below the following section

Ideas for future

Squish out flags handling

BX_NEED_FLAGS, BX_SETS_FLAGS

How to be lazy with addresses

only retranslate seg:ofs -> linear -> physical when strictly necessary

combine segment limits with TLB pages

A bit that says if everything is ok or the address has to be reevaluated