x86 architecture et similia

FREELY INSPIRED FROM
CLASS 6.828, MIT

PC architecture

- A full PC has:
  - an x86 CPU with registers, execution unit, and memory management
  - CPU chip pins include address and data signals
  - memory
  - disk
  - keyboard
  - display
  - other resources: BIOS ROM, clock, …

CPU

- CPU runs instructions:

```
for(;;) {
    run next instruction
}
```

- We will start with the original 16-bit 8086 CPU (1978)

CPU & Memory

- Needs work space: registers
  - four 16-bit data registers: AX, CX, DX, BX
  - each in two 8-bit halves, e.g. AH and AL
  - very fast, very few
- More work space: memory
  - CPU sends out address on address lines (wires, one bit per wire)
  - Data comes back on data lines
  - *or* data is written to data lines

CPU & Memory

- Add address registers: pointers into memory
  - SP - stack pointer
  - BP - frame base pointer
  - SI - source index
  - DI - destination index

IP (program counter)

- Instructions are in memory too!
  - IP - instruction pointer (PC on PDP-11, everything else)
  - increment after running each instruction
  - can be modified by CALL, RET, JMP, conditional jumps

Flags

- Want conditional jumps
  - FLAGS - various condition codes
- whether last arithmetic operation overflowed
    - ... was positive/negative
    - ... was [not] zero
    - ... carry/borrow on add/subtract
    - ... overflow
    - ... etc.
    - whether interrupts are enabled
    - direction of data copy instructions
  - JP, JN, J[N]Z, J[N]C, J[N]O ...

I/O

- Still not interesting - need I/O to interact with outside world
  - Original PC architecture: use dedicated *I/O space*
    - Works same as memory accesses but set I/O signal
    - Only 1024 I/O addresses
  - Memory-Mapped I/O
    - Use normal physical memory addresses
      - Gets around limited size of I/O address space
      - No need for special instructions
      - System controller routes to appropriate device
    - Works like “magic” memory:
      - *Addressed* and *accessed* like memory, but ...
      - ... does not *behave* like memory!
      - Reads and writes can have “side effects”
      - Read results can change due to external events

Example

Example: write a byte to line printer:

```
#define DATA_PORT 0x378
#define STATUS_PORT 0x379
#define BUSY 0x80
#define CONTROL_PORT 0x37A
#define STROBE 0x01

void lpt_putc(int c) {
/* wait for printer to consume previous byte */
while((inb(STATUS_PORT) & BUSY) == 0);

/* put the byte on the parallel lines */
outb(DATA_PORT, c);

/* tell the printer to look at the data */
outb(CONTROL_PORT, STROBE);
outb(CONTROL_PORT, 0);
}
```

More memory

- What if we want to use more than 2^16 bytes of memory?
  - 8086 has 20-bit physical addresses, can have 1 Meg RAM
  - each segment is a 2^16 byte window into physical memory
  - virtual to physical translation: pa = va + seg*16

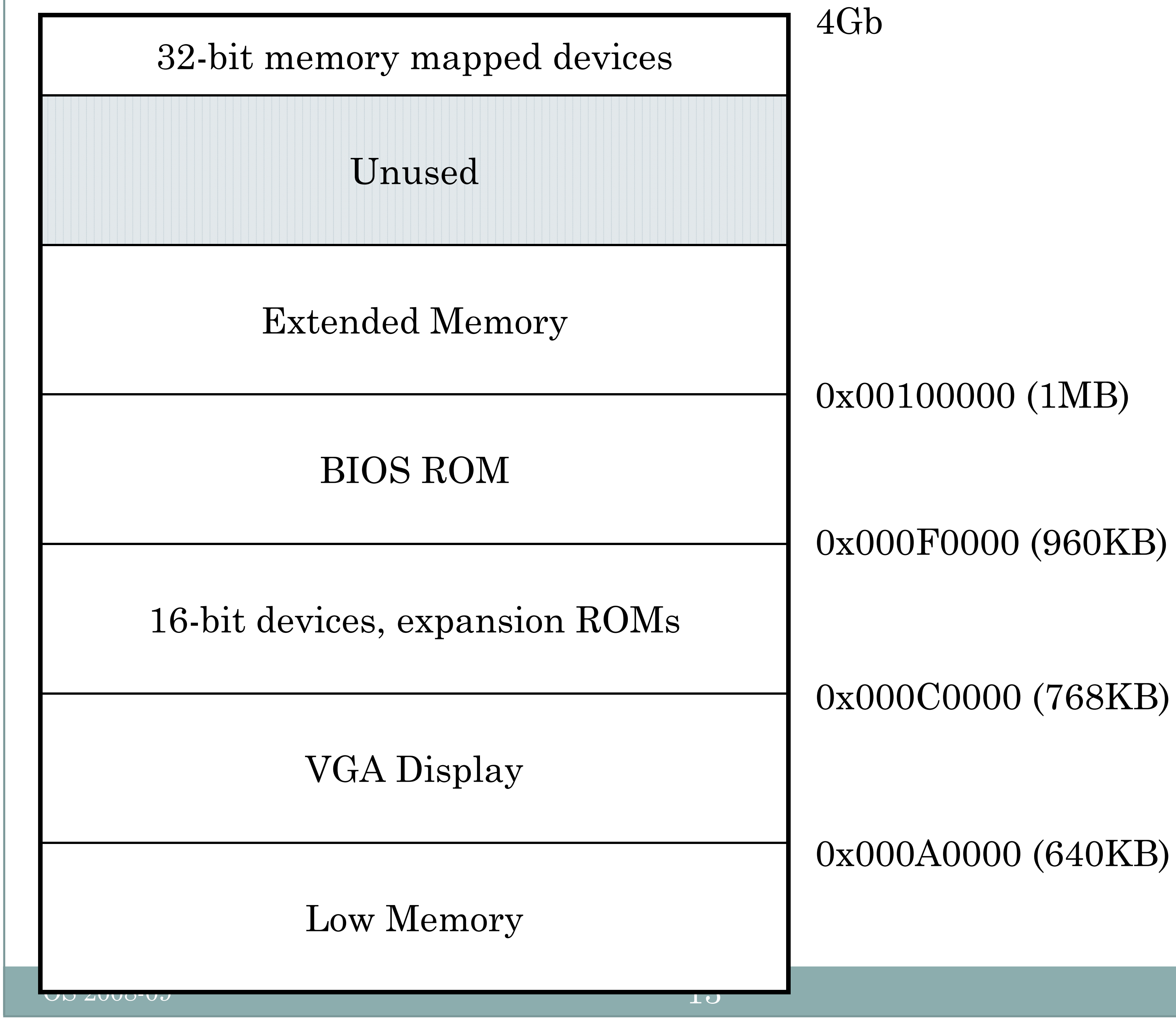
  - the segment is usually implicit, from a segment register
  - CS - code segment (for fetches via IP)
  - SS - stack segment (for load/store via SP and BP)
  - DS - data segment (for load/store via other registers)
  - ES - another data segment (destination for string operations)
  - tricky: can't use the 16-bit address of a stack variable as a pointer (still need 20 bits to identify an address in memory)
  - but a *far pointer* includes full segment:offset (16 + 16 bits)

More memory is needed

- But 8086's 16-bit addresses and data were still painfully small
  - 80386 added support for 32-bit data and addresses (1985)
  - boots in 16-bit mode, then switches to 32-bit mode
  - registers are 32 bits wide, called EAX rather than AX
  - operands and addresses are also 32 bits, e.g. ADD does 32-bit arithmetic
  - prefix 0x66 gets you 16-bit mode: MOVW is really 0x66 MOVW
  - 80386 also changed segments and added paged memory...

Memory map

- **x86 Physical Memory Map**
  - The physical address space mostly looks like ordinary RAM
  - Except some low-memory addresses actually refer to other things
  - Writes to VGA memory appear on the screen
  - Reset or power-on jumps to ROM at 0x000ffff0



32-bit memory mapped devices

Unused

Extended Memory

BIOS ROM

16-bit devices, expansion ROMs

VGA Display

Low Memory

4Gb

0x00100000 (1MB)

0x000F0000 (960KB)

0x000C0000 (768KB)

0x000A0000 (640KB)

Assembly

- Two-operand instruction set
  - Intel syntax: op dst, src
  - AT&T (gcc/gas) syntax: op src, dst
    - uses b, w, l suffix on instructions to specify size of operands
  - Operands are registers, constant, memory via register, memory via constant
  - Examples:

AT&T syntax	"C"-ish equivalent	
movl %eax, %edx	edx = eax;	*register mode*
movl $0x123, %edx	edx = 0x123;	*immediate*
movl 0x123, %edx	edx = *(int32_t*)0x123;	*direct*
movl (%ebx), %edx	edx = *(int32_t*)ebx;	*indirect*
movl 4(%ebx), %edx	edx = *(int32_t*)(ebx+4);	*displaced*

Assembly (instr. classes)

- Instruction classes
  - data movement: MOV, PUSH, POP, ...
  - arithmetic: TEST, SHL, ADD, AND, ...
  - i/o: IN, OUT, ...
  - control: JMP, JZ, JNZ, CALL, RET
  - string: REP MOVSB, ...
  - system: IRET, INT

GCC (a particular compiler)

Example instruction	What it does
pushl %eax	subl $4, %esp; movl %eax, (%esp)
popl %eax	movl (%esp), %eax; addl $4, %esp
call $0x12345	pushl %eip; movl $0x12345, %eip
ret	popl %eip

Examples

- GCC dictates how the stack is used. Contract between caller and callee on x86:
  - after call instruction:
    - %eip points at first instruction of function
    - %esp+4 points at first argument
    - %esp points at return address
  - after ret instruction:
    - %eip contains return address
    - %esp points at arguments pushed by caller
    - called function may have trashed arguments
    - %eax contains return value (or trash if function is void)
    - %ecx, %edx may be trashed
    - %ebp, %ebx, %esi, %edi must contain contents from time of call
  - Terminology:
    - %eax, %ecx, %edx are "caller save" registers
    - %ebp, %ebx, %esi, %edi are "callee save" registers

GCC (cntd.)

- Functions can do anything that doesn't violate contract with the GCC

- C code

```
int main(void) { return f(8)+1; }
int f(int x) { return g(x); }
int g(int x) { return x+3; }
```

- assembler

```
_main:
                              prologue
        pushl %ebp
        movl %esp, %ebp
                              body
        pushl $8
        call _f
        addl $1, %eax
                              epilogue
        movl %ebp, %esp
        popl %ebp
        ret
_f:
                              prologue
        pushl %ebp
        movl %esp, %ebp
                              body
        pushl 8(%esp)
        call _g
                              epilogue
        movl %ebp, %esp
        popl %ebp
        ret

_g:
                              prologue
        pushl %ebp
        movl %esp, %ebp
                              save %ebx
        pushl %ebx
                              body
        movl 8(%ebp), %ebx
        addl $3, %ebx
        movl %ebx, %eax
                              restore %ebx
        popl %ebx
                              epilogue
        movl %ebp, %esp
        popl %ebp
        ret
```

Compilation

- Compiling, linking, loading:
  - *Compiler* takes C source code (ASCII text), produces assembly language (also ASCII text)
  - *Assembler* takes assembly language (ASCII text), produces .o file (binary, machine-readable!)
  - *Linker* takes multiple '.o's, produces a single *program image* (binary)
  - *Loader* loads the program image into memory at run-time and starts it executing

x86 & the OS

- One way to think about an operating system interface is that it extends the hardware instructions with a set of "instructions" that are implemented in software. These instructions are invoked using a system call instruction **(INT and RETI on the x86)**. In this view, a task of the operating system is to provide each application with a *virtual* version of the interface; that is, it provides each application with a virtual computer.

- One of the challenges in an operating system is multiplexing the physical resources between the potentially many virtual computers. What makes the multiplexing typically complicated is an additional constraint: isolate the virtual computers well from each other.

Virtual x86 (the OS)

- To give each application its own set of virtual processor, we need to virtualize the physical processors. One way to do is to multiplex the physical processor over time: the operating system runs one application for a while, then runs another application for while, etc. We can implement this solution as follows: when an application has run for its share of the processor, unload the state of the physical processor, save that state to be able to resume the application later, load in the state for the next application, and resume it.
- What needs to be saved and restored? That depends on the processor, but for the x86:
  - IP
  - SP
  - The other processor registers (eax, etc.)

- To enforce that a virtual processor doesn't keep a processor, the operating system can arrange for a periodic interrupt, and switch the processor in the interrupt routine.

Interrupt

- Big picture: kernel is trusted third-party that runs the machine. Only the kernel can execute privileged instructions (e.g., changing MMU state). The processor enforces this protection through the ring bits in the code segment. If a user application needs to carry out a privileged operation or other kernel-only service, it must ask the kernel nicely. How can a user program change to the kernel address space? How can the kernel transfer to a user address space? What happens when a device attached to the computer needs attention? These are the topics for today's lecture.

Continued…

- There are three kinds of events that must be handled by the kernel, not user programs: (1) a system call invoked by a user program, (2) an illegal instruction or other kind of bad processor state (memory fault, etc.). and (3) an interrupt from a hardware device.
- Although these three events are different, they all use the same mechanism to transfer control to the kernel. This mechanism consists of three steps that execute as one atomic unit. (a) change the processor to kernel mode; (b) save the old processor somewhere (usually the kernel stack); and (c) change the processor state to the values set up as the “official kernel entry values.” The exact implementation of this mechanism differs from processor to processor, but the idea is the same.

Mutual exclusion x86

Here is one way we can implement acquire and release using the x86 xchgl instruction:

```
struct Lock {
    unsigned int locked;
};

acquire(Lock *lck)  {
    while(TSL(&(lck->locked)) != 0) ;
}

release(Lock *lck) {
    lck->locked = 0;
}
```

xchg

```
int
TSL(int *addr)
{
        register int content = 1;
        // xchgl content, *addr
        // xchgl exchanges the values of its two operands, while
        // locking the memory bus to exclude other operations.
        asm volatile ("xchgl %0,%1" :
        "=r" (content),
        "=m" (*addr) :
        "0" (content),
        "m" (*addr));
        return(content);
}
```

the instruction "XCHG %eax, (content)" works as follows:

1. freeze other CPUs' memory activity
2. temp := content
3. content := %eax
4. %eax := temp
5. un-freeze other CPUs

steps 1 and 5 make XCHG special: it is "locked" special signal lines on the inter-CPU bus, bus arbitration