

The x86 Architecture

ICS312 - Spring 2009 Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

The 80x86 Architecture

- To learn assembly programming we need to pick a processor family with a given ISA (Instruction Set Architecture)
- We will pick the Intel 80x86 ISA (x86 for short)
 - The most common today in existing computers
 - For instance in my laptop
- We could have picked others
 - Old ones: Sparc, VAX
 - Current ones: PowerPC, Itanium, MIPS
 - In ICS431 you'll (likely) be exposed to MIPS
- Some courses in some curricula subject students to two or even more ISAs, but in this course we'll just focused on one more in depth

X86 History

- In the late 70s Intel creates the 8088 and 8086 processors
 - 16-bit registers
 - 1 MB of memory, divided into 64KB segments
- In 1982: the 80286
 - Added some instructions
 - 16 MB of memory, divided into 64KB segments
- In 1985: the 80386
 - 32-bit registers
 - 5 GB of memory, divided into 4GB segments
- 1989: 486; 1992: Pentium; 1995: P6
 - Only incremental changes to the architecture
- 1997: MMX extensions
 - New instructions to speed up graphics (for instance)
- 1999: Pentium III
 - SSE extension (new cache instructions, new floating point operations)
- 2001: SSE2 extension

X86 History

- It's quite amazing that this architecture has witnessed so little (fundamental) change since the 8086
 - Backward compatibility
 - Imposed early as the ISA (Intel was the first company to produce a 16-bit architecture)
- Some argue that it's an ugly ISA
 - Due to it being a set of add-ons rather than a modern re-design
- But it's easy to implement in hardware, and Intel's was successful in making faster and faster x86 processors for decades
- Intel's new ISA is IA64 (used in Itanium processors), which is radically different
 - And closer to the ISA you'll see in ICS431

The 8086 Registers

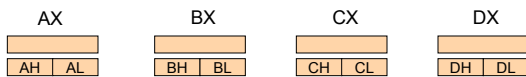
- To write assembly code for an ISA you must know the name of registers
 - Because registers are places in which you put data to perform computation and in which you find the result of the computation (think of them as variables for now)
 - The registers are identified by binary numbers, but assembly languages give them "easy-to-remember" names
- The 8086 offered 16-bit registers
- **Four general purpose 16-bit registers**
 - AX
 - BX
 - CX
 - DX

The 8086 Registers



- Each of the 16-bit registers consists of 8 "low bits" and 8 "high bits"
 - Low: least significant
 - High: most significant
- The ISA makes it possible to refer to the low or high bits individually
 - AH, AL
 - BH, BL
 - CH, CL
 - DH, DL

The 8086 Registers



- The xH and xL registers can be used as 1-byte register to store 1-byte quantities
- Important: both are “tied” to the 16-bit register
 - Changing the value of AX will change the values of AH and AL
 - Changing the value of AH or AL will change the value of AX

The 8086 Registers

- Two 16-bit index registers:
 - SI
 - DI
- These are basically general-purpose registers
- But by convention they are often used as “pointers”, i.e., they contain addresses
- And they **cannot** be decomposed into High and Low 1-byte registers

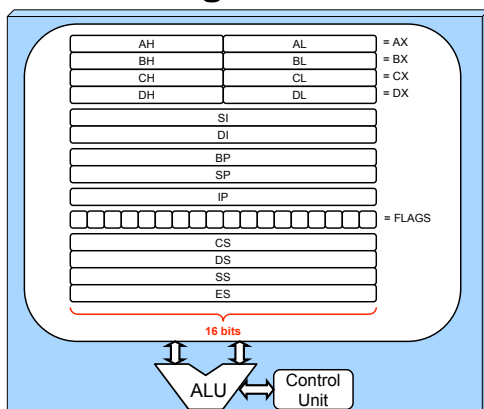
The 8086 Registers

- Two 16-bit special registers:
 - BP: Base Pointer
 - SP: Stack Pointer
 - We’ll discuss these at length later
- Four 16-bit segment registers:
 - CS: Code Segment
 - DS: Data Segment
 - SS: Stack Segment
 - ES: Extra Segment
 - We’ll discuss these later as well

The 8086 Registers

- The 16-bit Instruction Pointer (IP) register:
 - Points to the next instruction to execute
 - Typically not handled directly when writing assembly code
- The 16-bit FLAGS registers
 - Information is stored in individual bits of the FLAGS register
 - Whenever an instruction is executed and produces a result, it may modify some bit(s) of the FLAGS register
 - Example: Z (or ZF) denotes one bit of the FLAGS register, which is set to 1 if the previously executed instruction produced 0, or 0 otherwise
 - We’ll see many uses of the FLAGS registers

The 8086 Registers



Addresses in Memory

- We mentioned several registers that are used for holding **addresses of memory locations**
- Segments:
 - CS, DS, SS, ES
- Pointers:
 - SI, DI: indices (typically used for pointers)
 - SP: Stack pointer
 - BP: (Stack) Base pointer
- Let’s look at the structure of the address space

Address Space

- In the 8086 processor, a program is limited to referencing an **address space** of size 1MB, that is 2^{20} bytes
- Therefore, addresses are 20-bit long!
- A **d-bit long address allows to reference 2^d different "things"**
- Example:
 - 2-bit addresses
 - 00, 01, 10, 11
 - 4 "things"
 - 3-bit addresses
 - 000, 001, 010, 011, 100, 101, 110, 111
 - 8 "things"
- In our case, these things are "bytes"
 - One cannot address anything smaller than a byte
- Therefore, a 20-bit address makes it possible to address 2^{20} individual bytes, or 1MB

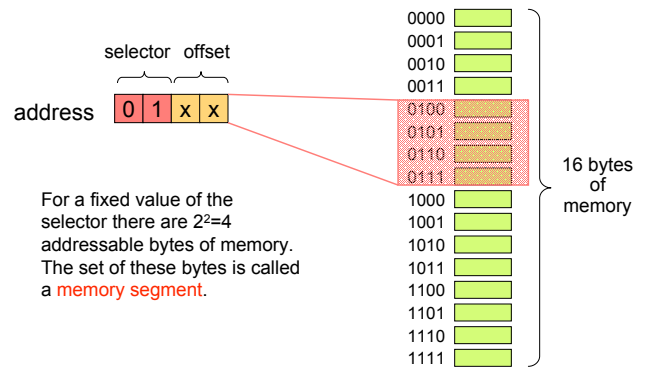
Address Space

- One says that a running program has a 1MB **address space**
- And the program needs to use 20-bit addresses to reference memory content
 - Instructions, data, etc.
- Problem: registers are at 16-bit long! How can they hold a 20-bit address???
- The solution: split addresses in two pieces:
 - The **selector**
 - The **offset**

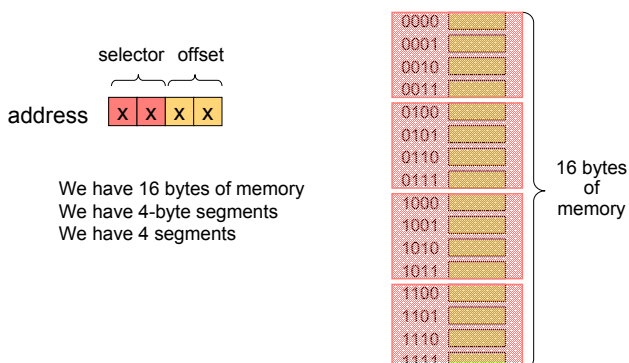
Simple Selector and Offset

- Let us assume that we have an address space of size $2^4=16$ bytes
 - Yes, that would not be a useful computer
- Addresses are 4-bit long
- Let's assume we have a 2-bit selector and a 2-bit offset
 - As if our computer had only 2-bit registers
- We take such small numbers because it's difficult to draw pictures with 2^{20} bytes!

Selector and Offset Example



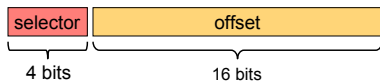
Selector and Offset Example



Selector and Offset

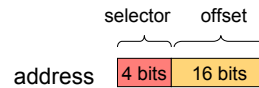
- The way in which one addresses the memory content is then pretty straightforward
- First, set the bits of the selector to "pick" a segment
- Second, set the bits of the offset to address a byte within the segment
- This all makes sense because a program typically addresses bytes that are next to each other, that is within the same segment
- So, the selector bits stay the same for a long time, while the offset bits change often
 - Of course, this isn't true for tiny 4-byte segments as in our example...

For 20-bit Addresses

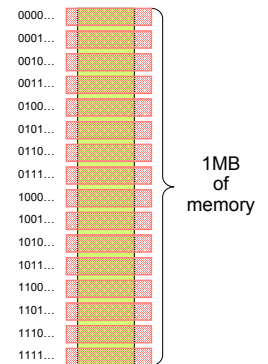


- On the 8086 the offset is 16-bit long
 - And therefore the selector is 4-bit
- We have $2^4 = 16$ different segments
- Each segment is 2^{16} bytes = **64KB**
- For a total of 1MB of memory, which is what the 8086 used

For 20-bit Addresses



We have 1MB of memory
We have 64K segments
We have 16 segments



The 8086 Selector Scheme

- So far we've talked about the selector as a 4-bit quantity, for simplicity
- This leads to 16 **non-overlapping** segments
- The designers of the 8086 wanted more flexibility
- E.g., if you know that you need only an 8K segment, why use 64K for it? Just have the "next" segment start 8K after the previous segment
 - We'll see why segments are needed in a little bit
- So, for the 8086, the selector is NOT a 4-bit field, but rather the address of the beginning of the segment
- But now we're back to our initial problem: Addresses are 20-bit, how are we to store an address in a 16-bit register???

The 8086 Selector Scheme

- What the designers of the 8086 did is pretty simple
- Enforce that the beginning address of a segment can only be a multiple of 16
- Therefore, its representation in binary always has its four lowest bits set to 0
- Or, in hexadecimal, its last digit is always 0
- So the address of a beginning of a segment is a 20-bit hex quantity that looks like: XXXX0
- Since we know the last digit is always 0, no need to store it
- Therefore, we need to store only 4 hex digits
- Which, lo and behold, fits in a 16-bit register!

The 8086 Selector Scheme

- So now we have two 16-bit quantities
 - **The 16-bit selector**
 - **The 16-bit offset**
- The selector must be stored in one of the "segment" registers
 - CS, DS, SS, ES
- The offset is typically stored in one of the "index" registers
 - SI, DI
 - But could be stored in a general purpose register
- Address computation is straightforward
- Given a 16-bit selector and a 16-bit offset, the 20-bit address is computed as follows
 - Multiply the selector by 16
 - This simply transforms XXXX into XXXX0, thanks to the beauty of hexadecimal
 - Add the offset
 - And voila

In-class Exercise

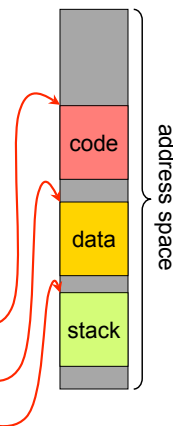
- Consider the byte at address 13DDE within a 64K segment defined by selector value 10DE. What is its offset?

In-class Exercise

- Consider the byte at address 13DDE within a 64K segment defined by selector value 10DE. What is its offset?
- $13DDE = 10DE * 16_{10} + \text{offset}$
- $\text{offset} = 13DDE - 10DE0$
- $\text{offset} = 2FFE$ (a 16-bit quantity)

Code, Data, Stack

- Although we'll discuss these at length later, let's just accept for now that the address space has **three regions**
- A program constantly references all three regions
- Therefore, the program constantly references bytes in three different segments
 - For now let's assume that each region is fully contained in a single segment, which is in fact not always the case
- **CS**: points to the beginning of the code segment
- **DS**: points to the beginning of the data segment
- **SS**: points to the beginning of the stack segment



How come it ever survived?

- If you code and your data are <64K, segments are great
- Otherwise, they are a pain
- Unfortunately, one often has more than 64K data
- Given the horror of segmented programming, one may wonder how come it stuck?
- From the linked article: "Under normal circumstances, a design so twisted and flawed as the 8086 would have simply been ignored by the market and faded away."
- But in 1980, Intel was lucky that IBM picked it for the PC!
- And we've been stuck with it ever since
- Luckily the segment issue isn't as terrible with 32-bit architectures
 - Segments are 4GB, and thus typically "big enough"
- Not to criticize IBM or anything, but they are also the reason why we got stuck with FORTRAN for so many years :)
- Probably the fate of giant companies: whenever they make a bad choice it can have large repercussions

Address Computation Example

- Consider the whole 1MB address space
- Say that we want a 64K segment whose end is 8K from the end of the address space
- The address at the end of the address space is FFFFF
- 8K in binary is 10000000000000, that is 02000 in 20-bit hex
- So the address right after the end of the segment is $FFFFF - 02000 + 1 = FDFFF + 1 = FE000$
- The length of the segment is 64K
- 64K in binary is 1000000000000000, that is 10000 in 20-bit hex
- So the address at the beginning of the segment is $FE000 - 10000 = EE000$
- So the value to store in a segment register is EE00
- To reference the 43th byte in the segment, one must store 002A (= 42₁₀) in an index register
- The address of that byte is: $EE000 + 002A = EE02A$
- The address of the last byte in the segment is: $EE000 + 0FFFF = FDFFF$
 - Which is right before FE000, the beginning of the last 8K of the address space

The trouble with segments

- It is well-known that programming with segmented architectures is really a pain
 - Don't panic, for our purposes we won't really suffer from this (too much)
- You constantly have to make sure segment registers are set up correctly
- What happens if you have data/code that's more than 64K?
 - You must then switch back and forth between selector values, which can be really awkward
- Something that can cause complexity also is that two different (selector, offset) pairs can reference the same address
 - Example: (a,b) and (a-1, b+16)
- There is an interesting on-line article on the topic:
<http://world.std.com/~swmcd/steven/rants/pc.html>

"Real Mode"

- The addressing scheme we just described is called **real mode**
- Called "real" because addresses being computed are *physical* addresses
 - that reference physical locations in the memory chips
- This can cause problems if we have **multiple programs** running at the same time
 - Which is something all modern computers do
- **First problem**: a program may inadvertently modify the memory that belongs to another program
 - This could be very dangerous
- **Second problem**: the total memory used by all running programs must be no bigger than the total physical memory
 - This is limiting and may be difficult to enforce

“Protected Mode”

- With the 286, Intel introduced **protected mode**, also known as **virtual memory**
- This is a complicated topic that you will see in your ICS412 and ICS431
 - I may give a short Virtual Memory lecture if time allows
- With virtual memory, the addresses computed and issued by the CPU are not physical addresses
- Although the CPU thinks they are, instead they are virtual addresses
- Virtual addresses are translated into physical addresses by the hardware/OS
- Benefits
 - Memory protection between different programs
 - Use of the disk as extended memory
- All modern computers use protected mode

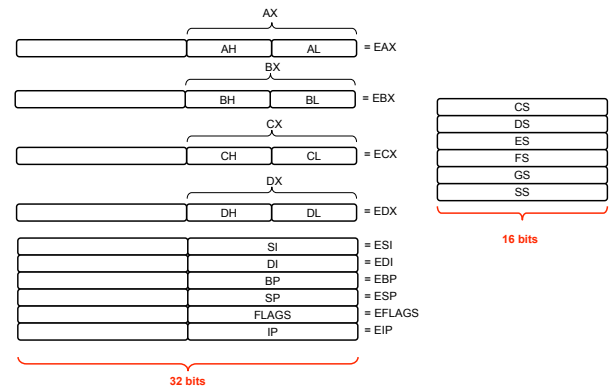
32-bit Protected Mode

- With the 80386 Intel introduced a processor with 32-bit registers
- **Addresses are 32-bit long**
 - Segments are 4GB
- All modern operating systems running on 32-bit Intel processors use paged 32-bit protected mode
- Let’s have a look at the 32-bit registers

The 80386 32-bit registers

- The General purposes were extended to 32-bit
 - EAX, EBX, ECX, EDX
 - For backward compatibility, AX, BX, CX, and DX refer to the 16 low bits of EAX, EBX, ECX, and EDX
 - AH and AL are as before
 - There is no way to access the high 16 bits of EAX separately
- Similarly, other registers are extended
 - EBX, EDX, ESI, EDI, EBP, ESP, EFLAGS
 - For backward compatibility, the previous names are used to refer to the low 16 bits
- The segment registers stay the same
- Two new segment registers: FS and GS

The 8386 Registers



Segment registers

- In 32-bit protected mode, segment registers are still 16-bit
- This may seem surprising, but in fact segmenting in the 386 is very different from segmenting in the 8086
- Each segment register points to an address
- At that address is a small data structure that describes the corresponding segment
 - Begin, end
- So when the CPU issues an address in a segment, the address computation is just a bit more complicated than in the 8086
 - Go look up the data structure
 - Find the beginning of the segment
 - Add the offset to it

Conclusion

- From now on we’ll keep referring to the register names, so make sure you absolutely know them
- We’re ready to move on to writing assembly code for the x86 architecture in 32-bit protected mode
- The registers are, in some sense, the variables that we can use



In-class Quiz

- Quiz #2 will be on the last two sets of slides
 - Numbers and Computers
 - The x86 Architecture
- Note that there will be questions on the quiz like:
 - Converting numbers from one base to another
 - Computing 2's complements
 - Arithmetic in different bases

- The quiz will be next...