Lecture 2: Memory Addressing
8086 Basics and Bus Timing
Asynchronous I/O Signaling

Zeshan Chishti
Electrical and Computer Engineering Dept
Maseeh College of Engineering and Computer Science

Source: Lecture based on materials provided by Mark F.
Basic I/O – Part I
Outline for next few lectures

- Simple model of computation
- Memory Addressing (Alignment, Byte Order)
- 8088/8086 Bus
- Asynchronous I/O Signaling
- Review of Basic I/O
  - How is I/O performed
    - Dedicated/Isolated /Direct I/O Ports
    - Memory Mapped I/O
  - How do we tell when I/O device is ready or command complete?
    - Polling
    - Interrupts
  - How do we transfer data?
    - Programmed I/O
    - DMA
Simplified Model of a Computer

Control

Data Path

Microprocessor
  [Fetch]
  [Decode]
  [Execute]

Address, Data, Control

Memory

I/O Device
  Keyboard
  Mouse
  Video display
  Printer
  Hard disk drive
  Audio card
  Ethernet
  WiFi
  WiFi
  CD R/W
  DVD
Memory Addressing

- **Size of operands**
  - Bytes, words, long/double words, quadwords
    - 16-bit half word (Intel: word)
    - 32-bit word (Intel: doubleword, dword)
    - 64-bit double word (Intel: quadword, qword)
  - Note: names are non-standard
    - SUN Sparc word is 32-bits, double is 64-bits

- **Alignment**
  - Can multi-byte operands begin at any byte address?
    - Yes: non-aligned
    - No: aligned. Low order address bit(s) will be zero
Memory Operand Alignment

...Intel IA speak (i.e. word = 16-bits = 2 bytes)
Memory Operand Alignment

Why do we care?

- Unaligned memory references
  - Can cause multiple memory bus cycles for a single operand
  - May also span cache lines
    - Requiring multiple evictions, multiple cache line fills
  - Complicates memory system and cache controller design

- Some architectures restrict addresses to be aligned

- Even in architectures without alignment restrictions (e.g. Intel x86) assembler directives (ex: `align 2`) are typically used (automatically by compiler) to force alignment, improving efficiency of generated code
Byte Order: Big Endian or Little Endian

Little Endian

<table>
<thead>
<tr>
<th>0x107</th>
<th>0x106</th>
<th>0x105</th>
<th>0x104</th>
<th>0x103</th>
<th>0x102</th>
<th>0x101</th>
<th>0x100</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Most significant byte</th>
<th>Least significant byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x107</td>
<td>0x100</td>
</tr>
<tr>
<td>0x106</td>
<td>0x101</td>
</tr>
<tr>
<td>0x105</td>
<td>0x102</td>
</tr>
<tr>
<td>0x104</td>
<td>0x103</td>
</tr>
<tr>
<td>0x103</td>
<td>0x102</td>
</tr>
<tr>
<td>0x102</td>
<td>0x103</td>
</tr>
<tr>
<td>0x101</td>
<td>0x100</td>
</tr>
<tr>
<td>0x100</td>
<td>0x103</td>
</tr>
</tbody>
</table>

Big Endian

<table>
<thead>
<tr>
<th>0x107</th>
<th>0x106</th>
<th>0x105</th>
<th>0x104</th>
<th>0x103</th>
<th>0x102</th>
<th>0x101</th>
<th>0x100</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Most significant byte</th>
<th>Least significant byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x107</td>
<td>0x102</td>
</tr>
<tr>
<td>0x106</td>
<td>0x103</td>
</tr>
<tr>
<td>0x105</td>
<td>0x102</td>
</tr>
<tr>
<td>0x104</td>
<td>0x103</td>
</tr>
<tr>
<td>0x103</td>
<td>0x102</td>
</tr>
<tr>
<td>0x102</td>
<td>0x103</td>
</tr>
<tr>
<td>0x101</td>
<td>0x100</td>
</tr>
<tr>
<td>0x100</td>
<td>0x103</td>
</tr>
</tbody>
</table>
Byte Order

- Pros and cons often exaggerated
- There are successful architectures based on both
  - Big Endian: Motorola 680x0, Sun Sparc, PDP-11
  - Little Endian: VAX, Intel IA32
  - Configurable: MIPS, ARM, Intel i960
- Really only matters when
  - Communicating between two systems
  - Employing networks and serial interfaces
  - Sharing data between computers
A Simple Example: Intel 8086/8088

- 16-bit data bus
- 20-bit address ($2^{20} = 1M$)
- Data/Address Multiplexed
  - AD0..AD15 + A16..A19
- 8088 (first PC) had 8 bit data bus

Minimum/Maximum Mode
Support co-processor
Affects I/O signals

Package pin count constraints result in multiplexed pins and/or “modes”
How is 8086 different?

8088 Address, Data, Control Signals

- MN/MX
- A19
- A8
- ALE
- AD7
- AD0
- IO/M
- RD
- WR

Latch EN

Address Bus
A19
A8
A7
A6
A0

Data Bus
D7
D6
D5
D4
D0

Control Signals
MEMR
MEMW
IOR
IOW
8088 Timing

- Instruction Cycle
  - Fetch/Decode/Execute
  - Comprised of one or more Machine Cycles
    - Comprised of one or more states
Block Diagram of 8086-based System
Block Diagram of 8086-based System (cont’d)

8086 can access bytes or words
References do not need to be aligned

Address alone is not enough to convey both location and size of data

BHE# (Byte High Enable)

<table>
<thead>
<tr>
<th>BHE#</th>
<th>A0</th>
<th>Refers to</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>16 bits (D0..D15)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8 bits (D8..D15)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>8 bits (D0..D7)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>&lt;reserved&gt;</td>
</tr>
</tbody>
</table>
8086 Bus Timing (Simplified)

Read Cycle

- **Memory access time**

Write Cycle

- **Memory write time**
Wait State Generation
8086 Bus Timing

- **CLK**
- **ALE**
- **M/IO**
- **ADDR/STATUS**
  - BHE, A19-A16
  - S7-S3
- **ADDR/DATA**
  - A15-A0
  - BUS RESERVED FOR DATA IN
  - D15-D0 VALID
  - A15-A0
  - DATA OUT D15-D0
- **RD**
- **READY**
  - WAIT
  - READY
- **DT/R**
- **DEN**
- **WR**

Memory Access Time
Asynchronous Handshaking
Asynchronous I/O Protocols

Use when sender/receiver may operate at differing speeds
Examples below: level sensitive (not edge triggered)

Source-controlled, no command communication
Suitable for display (e.g. signal to LED, temp value to 7-segment display) where no issue of lost data

Source-controlled, one-way command communication
Assumes destination is capable of keeping up (e.g. keyboard to processor). Simple, fast. No validity verification.

Destination-controlled, one-way command communication
Assumes source is capable of keeping up. Can add Data Error signal for verification.
Asynchronous I/O Protocols (cont’d)

Non-interlocked, request/acknowledge communication
Suitable for interfacing between devices of different speeds, helps to ensure both devices ready

Half-interlocked, request/acknowledge communication
Suitable for interfacing between devices of different speeds, more reliable
Asynchronous I/O Protocols (cont’d)

Fully-interlocked, request/acknowledge communication
Suitable for interfacing between devices of different speeds, even more reliable

“Double Handshake”
Asynchronous I/O Protocols

- Simple I/O
  - Read temperature sensor value
  - Write to LED

- Simple Strobe I/O
  - Keyboard input
  - Suitable for low speed
  - Sender can’t know if OK to send next byte

- Single Handshake
  - Feedback from receiving device
  - Sender: /STB (“Here’s valid data”)
  - Receiver: ACK (“I’ve got it; send the next byte”)

- Double Handshake
EX: Double Handshake

Sender

1. I’ve got data for you. (Asserts /STB)

3. (Transmits Data)
   Here it is. (De-asserts /STB)

Receiver

2. OK. Send it. (Asserts ACK)

4. (Receives Data)
   Got it; awaiting your request to send another byte (De-asserts ACK)
I/O Subsystems – Things to Think About

- What instructions does the processor use to communicate with I/O devices? Example: disk read: address, number of bytes, “read”
  - Direct (Isolated) I/O
  - Memory Mapped I/O

- How do we know if an I/O device is ready or an I/O operation is complete
  - Polling
  - Interrupts

- How do we transfer data between the I/O device and memory?
  - Programmed I/O (PIO)
  - Direct memory access (DMA)
  - Bus Mastering
Direct (Isolated) I/O

- I/O address space separate from Memory Address Space
- I/O “Ports” are numbered (each has its own address)
- Dedicated I/O Instructions
  - IN <destination>, <port> ; fixed or variable port
  - OUT <port>, <source> ; fixed or variable port

Pros:
- Very easy to interface peripherals
- Port Device ICs (e.g. 8255)

Cons:
- Additional H/W and instructions required
- May not be as flexible as memory-mapped I/O
  - Depending upon addressing modes implemented for IN, OUT
## EX: I/O Port Addresses

<table>
<thead>
<tr>
<th>Hex Range</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 - 01F</td>
<td>DMA controller 1, 8237A-5</td>
</tr>
<tr>
<td>020 - 03F</td>
<td>Interrupt controller 1, 8259A, Master</td>
</tr>
<tr>
<td>040 - 05F</td>
<td>Timer, 8254-2</td>
</tr>
<tr>
<td>060 - 06F</td>
<td>8042 (keyboard)</td>
</tr>
<tr>
<td>070 - 07F</td>
<td>Real-time clock, NMI mask</td>
</tr>
<tr>
<td>080 - 09F</td>
<td>DMA page register, 74LS612</td>
</tr>
<tr>
<td>0A0 - 0BF</td>
<td>Interrupt controller 2, 8237A-5</td>
</tr>
<tr>
<td>0C0 - 0DF</td>
<td>DMA controller 2, 8237A-5</td>
</tr>
<tr>
<td>0F0</td>
<td>Clear math coprocessor busy</td>
</tr>
<tr>
<td>0F1</td>
<td>Reset math coprocessor</td>
</tr>
<tr>
<td>0F8 - 0FF</td>
<td>Math coprocessor</td>
</tr>
<tr>
<td>1F0 - 1F8</td>
<td>Fixed disk</td>
</tr>
<tr>
<td>200 - 207</td>
<td>Game I/O</td>
</tr>
<tr>
<td>20C - 20D</td>
<td>Reserved</td>
</tr>
<tr>
<td>21F</td>
<td>Reserved</td>
</tr>
<tr>
<td>278 - 27F</td>
<td>Parallel printer port 2</td>
</tr>
<tr>
<td>280 - 2DF</td>
<td>Alternate enhanced graphics adapter</td>
</tr>
<tr>
<td>2E1</td>
<td>GPIB (adapter 0)</td>
</tr>
<tr>
<td>2E2 &amp; 2E3</td>
<td>Data acquisition (adapter 0)</td>
</tr>
<tr>
<td>2F8 - 2FF</td>
<td>Serial port 2</td>
</tr>
<tr>
<td>300 - 31F</td>
<td>Prototype card</td>
</tr>
<tr>
<td>360 - 363</td>
<td>PC network (low address)</td>
</tr>
<tr>
<td>364 - 367</td>
<td>Reserved</td>
</tr>
<tr>
<td>368 - 36B</td>
<td>PC network (high address)</td>
</tr>
<tr>
<td>36C - 36F</td>
<td>Reserved</td>
</tr>
<tr>
<td>378 - 37F</td>
<td>Parallel printer port 1</td>
</tr>
<tr>
<td>380 - 38F</td>
<td>SDLC, bisynchronous 2</td>
</tr>
<tr>
<td>390 - 393</td>
<td>Cluster</td>
</tr>
<tr>
<td>3A0 - 3AF</td>
<td>Bisynchronous 1</td>
</tr>
<tr>
<td>3B0 - 3BF</td>
<td>Monochrome display and printer adapter</td>
</tr>
<tr>
<td>3C0 - 3CF</td>
<td>Enhanced graphics adapter</td>
</tr>
<tr>
<td>3D0 - 3DF</td>
<td>Color/graphics monitor adapter</td>
</tr>
<tr>
<td>3F0 - 3F7</td>
<td>Diskette controller</td>
</tr>
<tr>
<td>3F8 - 3FF</td>
<td>Serial port 1</td>
</tr>
<tr>
<td>6E2 &amp; 6E3</td>
<td>Data acquisition (adapter 1)</td>
</tr>
<tr>
<td>790 - 793</td>
<td>Cluster (adapter 1)</td>
</tr>
<tr>
<td>AE2 &amp; AE3</td>
<td>Data acquisition (adapter 2)</td>
</tr>
<tr>
<td>B90 - B93</td>
<td>Cluster (adapter 2)</td>
</tr>
<tr>
<td>EE2 &amp; EE3</td>
<td>Data acquisition (adapter 3)</td>
</tr>
<tr>
<td>1390 - 1393</td>
<td>Cluster (adapter 3)</td>
</tr>
<tr>
<td>22E1</td>
<td>GPIB (adapter 1)</td>
</tr>
<tr>
<td>2390 - 2393</td>
<td>Cluster (adapter 4)</td>
</tr>
<tr>
<td>42E1</td>
<td>GPIB (adapter 2)</td>
</tr>
<tr>
<td>62E1</td>
<td>GPIB (adapter 3)</td>
</tr>
<tr>
<td>82E1</td>
<td>GPIB (adapter 4)</td>
</tr>
<tr>
<td>A2E1</td>
<td>GPIB (adapter 5)</td>
</tr>
<tr>
<td>C2E1</td>
<td>GPIB (adapter 6)</td>
</tr>
<tr>
<td>E2E1</td>
<td>GPIB (adapter 7)</td>
</tr>
</tbody>
</table>
Memory Mapped I/O

- I/O treated exactly like memory access
- I/O devices share address space with memory
- Use normal instructions for I/O
  - MOV AL, MemAddr ; read byte
  - MOV MemAddr, AL ; write byte
  - Full addressing modes available (++, [])
  - Caution: I/O registers read as side-effect
- I/O devices do Address decoding
Memory-Mapped I/O

Pros:
- “Regular” architecture
- No special I/O instructions
  - Simplifies processor architecture
  - Can write I/O device drivers in C
- Full addressing modes available
  - Increment, indirect
  - Memory to memory instructions
    - Can test/set bits in a control word without using an additional I/O instruction to bring the control word into a register
- Can protect against random process accessing I/O by using page tables

Cons:
- Must have solution to avoid caching device registers!
- Address space consumed (not really an issue!)

What does Intel IA32 architecture use?
Next Time

- Lecture Topics:
  - Polling vs. Interrupts
  - Programmed I/O and DMA
  - Interrupts