9-22-17

Week 1 - Lecture 1

Intro to hardware, software and languages

* Views by levels
  + Natural languages
    - Used by humans
    - Translated to programming languages
    - Word processors
  + High-level computer languages
    - IDEs
    - Text editors
    - Compilers
    - Linker
    - Loader
    - Debuggers
    - IE: Visual C++, Eclipse
  + Low level programming languages
    - Text editor
    - Libraries
    - Linker
    - Loaders
    - Debuggers
    - IE: text editor together with MASM, Visual C++
  + Machine level computer languages
    - Some way to assign machine instructions into computer memory
    - IE: set individual bits, loader
  + Assembly language provides
    - Set of mnemonics, for machine instructions
      * opcodes and addressing modes
    - Mechanism for naming memory addresses and other constants
      * name memory address is usually called a variable
  + Assembly language program goes to assembler to create target machine code
  + Operating systems provide interface among users, programs and devices
  + Implemented for specific architecture in the host comps machine language
  + Each type of comp has unique instruction set, language will only work on specific comp
    - Cross-assemblers(software) can be used to convert machine language to another machine language
    - Virtual Machines(software) can be used to simulate another comps architecture
  + Hardware: Physical devices
    - circuits, chips, disk drives, printers
  + Software: Instructions that control hardware
    - games, word processors, compilers, operating systems
  + Sometimes the line between hardware and software is not clear
    - parts of an operating system might be implemented in hardware
  + Anything that can be implemented in software could be implemented in hardware, and would execute much faster
  + Why use assembly language?
  + Easier than machine code
  + Access to all features of target machine
  + Potentially increase performance, can be optimized
  + Using mixed languages
  + Assembly tends to evolve towards a high-level language
    - Advanced features such as loops control, etc
    - Libraries
* Common uses of assembly language
  + Embedded systems
    - Efficiency is critical
  + Real-time applications
    - Timing is critical
  + Interactive games
    - Speed is critical
  + Low-level tasks
    - Direct control is critical
  + Device drivers
    - Direct control is critical

9-22-17

Week 1 - Lecture 2

How computer hardware works

* Everything in comp is represented with electrical signals, on and off
* Off is 0, on is 1
  + Binary digit = bit
* Different combinations of switches represent different information
  + Group of 8 bits is called a byte
* CISC
  + Peripheral devices = physical hardware
    - External devices
      * Store/retrieve data
      * Convert data between human-readable and machine readable forms
  + i/o = input output unit = hardware/software functions
    - communicate between cpu/memory and peripheral devices
    - virtual memory interface
    - virtual file system interface
    - i/o buffers
    - network interface
  + Main memory unit = cell with address
    - Store programs and data
      * Currently being used by the cpu
    - Operating system
    - Device drivers
    - System stack
    - System heap
    - User programs
    - User data
    - Has unique address – if electrical power is interrupted, data is lost
  + CPU = central processing unit
    - Execute machine instructions
    - Bus
      * Parallel wires, transfers electrical signals
        + Internal = transfers signals among cpu components

Three busses total, measured by capacity like real bus

16, 32, 64, 128

Transfers bits from one components to another

* + - * + Control = carries signals for memory and I/O operations
        + Address = links to specific memory locations
        + Data = carries data between cpu and memory
    - Register
      * Fast local memory inside cpu
      * Control
        + Dictates current state of the machine
        + Determines which signals go where
        + Set every tick of system clock
        + Send or receive data
      * Status
        + Indicates status of operation (error, overflow, etc)
        + Multiple bits sets by results of various operations

These bits are used by control register to determine subsequent operations are applied

IE if X = Y do this, or if not do that

Can indicate when memory operations have been completed

Can also prevent further execution in things like division by 0

* + - * MAR
        + Memory address register (holds address of memory location currently referenced)
        + Holds address to be accessed connects to that address via address BUS
      * MDR
        + Memory data registers holds data being sent to or retrieved from the memory address in the MAR
        + Transfer takes place via data BUS
      * IP
        + Instruction pointer (holds memory address of the next instruction)
        + Holds address of next instruction to be copied into IR

Transfer takes place in addressing unit

* + - * IR
        + Instruction register (holds current machine instruction)
        + Transfer to instruction decoder which determines hardware machine instruction to be executed
      * SAG
        + Starting address generator determines where in micro memory the corresponding micro program is implemented
        + Micro instruction pointer sets up next micro instruction in the control register per tick of system clock
      * Operand\_1, Operand\_2, Result
        + ALU registers (for calcs and comparisons)
      * General
        + Fast temp. storage
        + Instantaneous access
    - ALU
      * Arithmetic/Logic unit
      * Store current operands 1 and 2
      * Control register signals ALU to perform operation sets results register and status register
    - Microprogram
      * Sequence of micro-instructions (implemented in hardware\_ required to execute a machine instruction
    - Micromemory
      * The actual hardware circuits that implement the machine instructions as micro programs
* Cache
  + An area of comparatively fast temporary storage for info copied from slower storage.
  + Takes place at several levels in a computer system
  + Main memory is a cache for secondary storage, programs sent to main memory
  + Sent to general registers
* Stored program concept
  + VonNeumann architecture
    - Program is stored in memory and is executed under the control of the operating system
    - Instruction Execution cycle
      * 1. Fetch next instruction (at address in IP) into IR
      * 2. Increment IP to point to next instruction
      * 3. Decode instruction in IR
      * 4. If instruction requires memory access,
        + A. Determine memory address
        + B. Fetch operand from memory into a CPU register, or send operand from a CPU register to memory.
      * Execute micro-program for instruction
      * Go to step 1
* EX.
  + ADD R1, mem1
    - Meaning: Add value in memory location mem1 to value in register R1
  + EX ADD Program
    - 1. Copy contents of R1 to ALU Operand\_1
    - 2. Move address mem1 to MAR
    - 3. Signal memory fetch (gets contents of memory address currently in MAR into MDR)
    - 4. Copy contents of MDR into ALU Operands\_2
    - 5. Signal ALU addition
    - 6. Set Status Register and Copy contents of ALU to register R1

9-22-17

Week 1 - Lecture 3

Introduction to Intel IA-32 architecture

* Preliminary: Metrics, measurements
  + Speed ( distance/time )is measured in electronic units
    - K = 10^3, M = 10^6, G = 10^9
    - Ie: network speed of 8 Mbps means 8,000,000 bits per second
  + Size in bits, Bytes measured in Binary units
    - Commonly used: K = 2^10, M = 2^20,G = 2^30
    - This course, Ki = 2^10, etc
    - IE: disk size of 200 GiB means
      * 200 x 2^30 Bytes = 214,748,364,800 Bytes
      * = 1,717,986,918,400 bits
  + Bytes and bits (abbreviations)
    - User lower-case b for bits
    - Use upper-case B for Bytes
      * Ex: 1 Mib = 128 KiB
* CISC architecture, implements machine via micro programs
  + Two modes of operation
    - Protected
      * Certain areas or memory and certain registers cannot be accessed by user programs
    - Real-address
      * Allows programs do about anything they want
  + Two processors in one
    - Integer unit
    - Floating point unit
    - Two processors can work in parallel(co-processors)
      * Separate instruction sets
      * Separate data registers
      * Separate ALUs
  + Specific hardware implementations
    - Registers
    - Memory addressing scheme
  + Specific instruction set and micro programs
  + Specific assembly languages
    - MASM, NASM, TASM, etc.
  + Specific operating systems
    - Windows, Linus, DOS, etc.
* Memory
  + Up to 4GiB
  + Byte addressable
    - Unique addresses
  + Little-endian
* 32-bit machine
  + Registers
  + Buses
  + ALU
* Bytes is the smallest unit of data can be manipulated directly in the IA 32 architecture
* Operating system and instruction decoder determine how byte codes are interpreted
  + Internal interpreter, representation
* Integer Unit Registers
  + 32-bit general purpose registers
    - EAX
    - EBX
    - ECX
    - EDX
  + 32-bit multi-purpose registers
    - EBP
    - ESP
    - ESI
    - EDI
  + 32-bit special-purpose registers
    - EFL(status)
    - EIP(instruction pointer)
  + 16-bit segment registers
    - CS, ES
    - SS, FS
    - DS, GS
  + Most of the 32-bit registers are visible during MASM debugging
    - 32-bit “general” and “multi” registers may be manipulated directly
    - 32-bit “special” registers are manipulated by the micro-programs that implement the instructions
  + Some “general-purpose” and “multi-purpose” registers are used for special purposes:
    - EAX and EDX are automatically used to integer multiplication and division instructions
    - ECX is automatically used as a counter for some looping instructions
    - ESP is used for referencing the system stack.
  + Multipurpose registers
    - Some 32-bit registers have only 16-bit “sub registers”
      * ESI, EDI, EBP, ESP
      * Ex: sub-registers of ESI
        + SI refers to the least-significant 16-bits of ESI

9-22-17

Week 1 - Lecture 4

Introduction to MASM assembly language

* Instruction types
  + Move data
    - Copy data from source to destination
  + Arithmetic
  + Compare two values
  + Conditional/unconditional branch
  + Call procedure, return
  + Loop control
  + I/O (input/output)
* MASM is NOT case sensitive
  + ; Comments start with ;
  + A procedure is essentially a function
* Segments start with .
  + Main should be the first procedure in the .code segment
  + Beginning of next segment (or END main) is end of segment
* Comments start with ;
  + Can start anywhere in a line
  + Remainder of line is ignored by assembler
  + End of line is end of comment
* Use indentation and sufficient white space to make sections easy to find and ID
* Identifiers: names for variable constants, procedures and labels
* 1 to 247 characters (no spaces)
  + Use concise, meaningful names
* Memory locations
  + May be named – can refer to a variable name or program label
* Interpretation of contents depends on program instructions
  + Numeric data
    - Integer, floating point
  + Non-numeric data
    - Character, string
  + Instruction
  + Address
* .data segment
  + Label data\_type initializer ;comment
* Data in memory
  + “variables” are laid out in memory in the order declared
* Literals
  + Integers – default is decimal, base 10
  + Floating points (decimal real)
    - Must have decimal point
  + Characters
    - Single characters in quotes, recommended single
  + String
    - 2 or more characters in quotes
    - Double quotes recommended
    - Must be null-terminated, always end with zero-byte
      * Ex “always”, 0
  + Syntax
    - Each instruction line has 4 fields
      * Label
      * Opcode
      * Operands
      * Comment
    - Depending on the opcode, on or more operands may be required
      * Otherwise any field may be empty
      * If empty opcode field, operand field must be empty
    - Opcode (specifies what to do)
      * Mnemonic (e.g. ADD, MOV, CALL, etc.)
    - Zero, one or two operands (specify the opcodes target)
      * Different number of operands for different opcodes
        + Opcode
        + Opcode destination
        + Opcode destination, source
      * EX:  
        ADD EAX, EBX
  + Addressing Modes
    - Specific “addressing modes” are permitted for the operands associated with each opcode
    - Basic (used in first programming assignment)
      * Intermediate
        + Constant, literal, absolute address
      * Register
        + Contents of register
      * Direct
        + Contents of referenced memory address
      * Offset
        + Memory address may be calculated
    - Advanced
      * Register indirect
        + Access memory through address in a register
      * Indexed
        + “Array” element, using offset in register
      * Base-indexed
        + Start address in one register; offset in another, add and access memory
      * Stack
        + Memory area specified and maintained as a stack; stack pointer in ESP register

9-22-17

Week 1 - Lecture 5b

Rules and regulations of MASM

* Know how to use:
  + Mov
  + Add
  + Sub
  + Mul
  + Div
  + Call
* Implied operands
  + Mul implied operand must be in EAX
    - Mul op2 ;result is in EDX:EAX
  + EX
    - Mov eax, 10
    - Mov ebx, 12
    - Mul ebx ; result is in eax (120), with possible overflow in edx
      * Edx is changed!
  + Div implied is in EDX:EAX
    - So extend EAX into EDX before division
    - Div op2 ;quotient is in EAX
      * + Remainder I sin EDX
* Reg
  + Any general puirpose or multi-purpose register
* Accum
  + AL, AX, or EAX (depending on operand size)
* Mem
  + Contents of specified memory location
  + 8-bit, 16-bit or 32-bit memory location
* Imm
  + Some literal value