CHAPTER 16:
BIOS-LEVEL PROGRAMMING

Chapter Overview

- Introduction
- Keyboard Input with INT 16h
- VIDEO Programming with INT 10h
- Drawing Graphics Using INT 10h
- Memory-Mapped Graphics
- Mouse Programming

PC-BIOS

- The BIOS (Basic Input-Output System) provides low-level hardware drivers for the operating system.
  - accessible to 16-bit applications
  - written in assembly language, of course
  - source code published by IBM in early 1980's
- Advantages over MS-DOS:
  - permits graphics and color programming
  - faster I/O speeds
  - read mouse, serial port, parallel port
  - low-level disk access

BIOS Data Area

- Fixed-location data area at address 00400h
  - this area is also used by MS-DOS
  - this area is accessible under Windows 98 & Windows Me, but not under Windows NT, 2000, or XP.
- Contents:
  - Serial and parallel port addresses
  - Hardware list, memory size
  - Keyboard status flags, keyboard buffer pointers, keyboard buffer data
  - Video hardware configuration
  - Timer data

What's Next

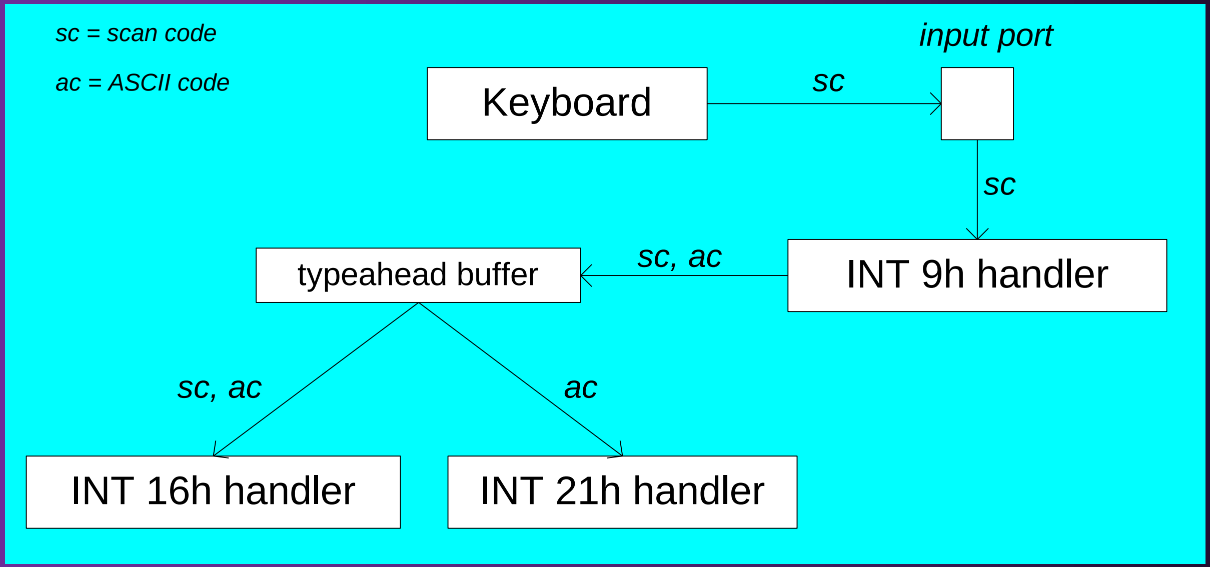
- Introduction
- **Keyboard Input with INT 16h**
- VIDEO Programming with INT 10h
- Drawing Graphics Using INT 10h
- Memory-Mapped Graphics
- Mouse Programming

Keyboard Input with INT 16h

- How the Keyboard Works
- INT 16h Functions

How the Keyboard Works

- Keystroke sends a scan code to the keyboard serial input port
- Interrupt triggered: INT 9h service routine executes
- Scan code and ASCII code inserted into keyboard typeahead buffer



Keyboard Flags

16-bits, located at 0040:0017h – 0018h.

Bit	Description
0	Right Shift key is down
1	Left Shift key is down
2	Either Ctrl key is down
3	Either Alt key is down
4	Scroll Lock toggle is on
5	Num Lock toggle is on
6	Caps Lock toggle is on
7	Insert toggle is on
8	Left Ctrl key is down

Bit	Description
9	Left Alt key is down
10	Right Ctrl key is down
11	Right Alt key is down
12	Scroll key is down
13	Num Lock key is down
14	Caps Lock key is down
15	SysReq key is down

INT 16h Functions

- Provide low-level access to the keyboard, more so than MS-DOS.
- Input-output cannot be redirected at the command prompt.
- Function number is always in the AH register
- Important functions:
  - set typematic rate
  - push key into buffer
  - wait for key
  - check keyboard buffer
  - get keyboard flags

Function 10h: Wait for Key

If a key is waiting in the buffer, the function returns it immediately. If no key is waiting, the program pauses (blocks), waiting for user input.

```
.data
scanCode  BYTE ?
ASCIICode BYTE ?

.code
mov ah,10h
int 16h
mov scanCode,ah
mov ASCIICode,al
```

Function 12h: Get Keyboard Flags

Retrieves a copy of the keyboard status flags from the BIOS data area.

```
.data
keyFlags WORD ?

.code
mov ah,12h
int 16h
mov keyFlags,ax
```

Clearing the Keyboard Buffer

Function 11h clears the Zero flag if a key is waiting in the keyboard typeahead buffer.

```
L1: mov ah,11h                    ; check keyboard buffer
    int 16h                       ; any key pressed?
    jz  noKey                     ; no: exit now
    mov ah,10h                    ; yes: remove from buffer
    int 16h
    cmp ah,scanCode               ; was it the exit key? ESC_key=1
    je  quit                      ; yes: exit now (ZF=1)
    jmp L1                        ; no: check buffer again

noKey:                            ; no key pressed
    or  al,1                      ; clear zero flag
quit:
```

What's Next

- Introduction
- Keyboard Input with INT 16h
- **VIDEO Programming with INT 10h**
- Drawing Graphics Using INT 10h
- Memory-Mapped Graphics
- Mouse Programming

VIDEO Programming with INT 10h

- Basic Background
- Controlling the Color
- INT 10h Video Functions

Video Modes

- Graphics video modes
  - draw pixel (像素) by pixel
  - multiple colors
- Text video modes
  - character output, using hardware or software-based font table
  - mode 3 (color text) is the default
  - default range of 80 columns by 25 rows.
  - color attribute byte contains foreground and background colors

Three Levels of Video Access

- MS-DOS function calls
  - slow, but they work on any MS-DOS machine
  - I/O can be redirected
- BIOS function calls
  - medium-fast, work on nearly all MS-DOS-based machines
  - I/O cannot be redirected
- Direct memory-mapped video
  - fast – works only on 100% IBM-compatible computers
  - cannot be redirected
  - does not work under Windows NT, 2000, or XP

Controlling the Color

- Mix primary colors: red, green, blue
  - called subtractive mixing
  - add the intensity (亮度) bit for 4th channel
- Examples:
  - red + green + blue = light gray (0111)
  - intensity + red + green + blue = white (1111)
  - green + blue = cyan (0011)
  - red + blue = magenta (0101)
- Attribute byte:
  - 4 MSB bits = background
  - 4 LSB bits = foreground
- Color constants defined in files

INT 10h Video Functions

- AH register contains the function number
- 00h: Set video mode
- 01h: Set cursor lines
- 02h: Set cursor position
- 03h: Get cursor position and size
- 06h: Scroll window up
- 07h: Scroll window down
- 08h: Read character and attribute

INT 10h Video Functions *(cont)*

- 09h: Write character and attribute
- 0Ah: Write character
- 10h (AL = 03h): Toggle blinking/intensity (闪烁/亮度)bit
- 0Fh: Get video mode
- 13h: Write string in teletype mode

Displaying a Color String

Write one character and attribute:

```
mov  si,OFFSET string
. . .
mov  ah,9                  ; write character/attribute
mov  al,[si]               ; character to display
mov  bh,0                  ; video page 0
mov  bl,color              ; attribute
or   bl,10000000b          ; set blink/intensity bit
mov  cx,1                  ; display it one time
int  10h
```

What's Next

- Introduction
- Keyboard Input with INT 16h
- VIDEO Programming with INT 10h
- **Drawing Graphics Using INT 10h**
- Memory-Mapped Graphics
- Mouse Programming

Drawing Graphics Using INT 10h

- INT 10h Pixel-Related Functions
- DrawLine Program
- Cartesian Coordinates Program
- Converting Cartesian Coordinates to Screen Coordinates

INT 10h Pixel-Related Functions

- Slow performance
- Easy to program
- 0Ch: Write graphics pixel
- 0Dh: Read graphics pixel

DrawLine Program

- Draws a straight line, using INT 10h function calls
- Saves and restores current video mode
- Excerpt from the *DrawLine* program (DrawLine.asm):

```
mov ah,0Ch          ; write pixel
mov al,color        ; pixel color
mov bh,0            ; video page 0
mov cx,currentX
int 10h
```

Cartesian Coordinates Program

- Draws the X and Y axes of a Cartesian coordinate system
- Uses video mode 12 (640 x 480, 16 colors)
- Name: Pixel2.asm
- Important procedures:
  - DrawHorizLine
  - DrawVerticalLine

Converting Cartesian Coordinates to Screen Coordinates

- Screen coordinates place the origin (0,0) at the upper-left corner of the screen
- Graphing functions often need to display negative values
  - move origin point to the middle of the screen
- Let *sOrigX* and *sOrigY* be the screen coordinates of the origin of the Cartesian coordinate system.
- For Cartesian coordinates X, Y screen coordinates *sx* and *sy* are calculated as:
  - *sx* = (*sOrigX* + X)
  - *sy* = (*sOrigY* – Y)

What's Next

- Introduction
- Keyboard Input with INT 16h
- VIDEO Programming with INT 10h
- Drawing Graphics Using INT 10h
- **Memory-Mapped Graphics**
- Mouse Programming

Memory-Mapped Graphics

- Binary values are written to video RAM
  - video adapter must use standard address
- Very fast performance
  - no BIOS or DOS routines to get in the way

Mode 13h: 320 X 200, 256 Colors

- Mode 13h graphics (320 X 200, 256 colors)
  - Fairly easy to program
  - read and write video adapter via IN and OUT instructions
  - pixel-mapping scheme (1 byte per pixel)

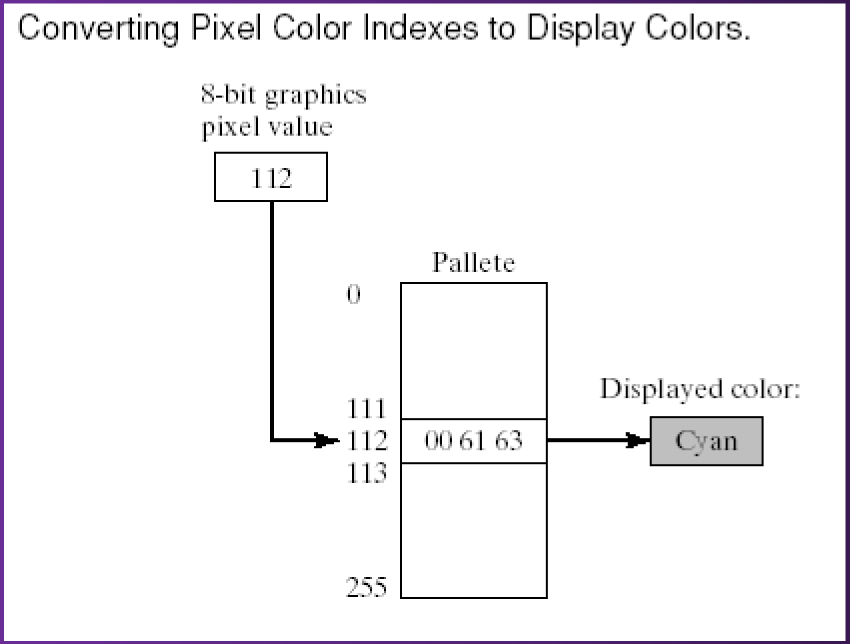
Mode 13h Details

- OUT Instruction
  - 16-bit port address assigned to DX register
  - output value in AL, AX, or EAX
  - Example:

```
mov  dx,3c8h            ; port address
mov  al,20h             ; value to be sent
out  dx,al              ; send to the port
```

- Color Indexes
  - color integer value is an index into a table of colors called a palette (调色板)

Color Indexes in Mode 13h



RGB Colors

Additive mixing of light (red, green, blue). Intensities vary from 0 to 255.

Examples:

Red	Green	Blue	Color
0	30	30	cyan
30	30	0	yellow
30	0	30	magenta
40	0	63	lavender

Red	Green	Blue	Color
0	0	0	black
20	20	20	dark gray
35	35	35	medium gray
50	50	50	light gray
63	63	63	white

Red	Green	Blue	Color
63	0	0	bright red
10	0	0	dark red
30	0	0	medium red
63	40	40	pink

What's Next

- Introduction
- Keyboard Input with INT 16h
- VIDEO Programming with INT 10h
- Drawing Graphics Using INT 10h
- Memory-Mapped Graphics
- **Mouse Programming**

Mouse Programming

- MS-DOS functions for reading the mouse
- Mickey – unit of measurement (200^{th} of an inch)
  - mickeys-to-pixels ratio (8 x 16) is variable
- INT 33h functions
- Mouse Tracking Program Example

Reset Mouse and Get Status

- INT 33h, AX = 0
- Example:

```
mov   ax,0
int   33h
cmp   ax,0     ; AX = FFFFh or 0
je    MouseNotAvailable
mov   numberOfButtons,bx
```

Show/Hide Mouse

- INT 33h, AX = 1 (show), AX = 2 (hide)
- Example:

```
mov   ax,1        ; show
int   33h
mov   ax,2        ; hide
int   33h
```

Get Mouse Position & Status

- INT 33h, AX = 3
- Example:

```
mov   ax,3
int   33h
test bx,1
jne Left_Button_Down
test bx,2
jne Right_Button_Down
test bx,4
jne Center_Button_Down
mov   xCoord,cx     ; X-position
mov   yCoord,dx     ; Y-position
```

Set Mouse Position

- INT 33h, AX = 4
- Example:

```
mov   ax,4
mov   cx,200        ; X-position
mov   dx,100        ; Y-position
int   33h
```

Get Button Press Information

- INT 33h, AX = 5
- Example:

```
mov   ax,5
mov   bx,0           ; button ID
int   33h
test  ax,1           ; left button down?
jz    skip           ; no - skip
mov   X_coord,cx     ; yes: save coordinates
mov   Y_coord,dx
```

Other Mouse Functions

- AX = 6: Get Button Release Information
- AX = 7: Set Horizontal Limits
- AX = 8: Set Vertical Limits

Mouse Tracking Program

- Tracks the movement of the text mouse cursor
- X and Y coordinates are continually updated in the lower-right corner of the screen
- When the user presses the left button, the mouse’s position is displayed in the lower left corner of the screen
- Source code

CHAPTER 17:
EXPERT MS-DOS PROGRAMMING

Chapter Overview

- **Defining Segments**
- Runtime Program Structure
- Interrupt Handling
- Hardware Control Using I/O Ports

Defining Segments

- Simplified Segment Directives
- Explicit Segment Definitions
- Segment Overrides
- Combining Segments

Simplified Segment Directives

- .MODEL – program memory model
- .CODE – code segment
- .CONST – define constants
- .DATA – near data segment
- .DATA? – uninitialized data
- .FARDATA – far data segment
- .FARDATA? – far uninitialize data
- .STACK – stack segment
- .STARTUP – initialize DS and ES
- .EXIT – halt program

Memory Models (review)

Model	Description
tiny	A single segment, containing both code and data. This model is used by .com programs.
small	One code segment and one data segment. All code and data are near, by default.
medium	Multiple code segments and a single data segment.
compact	One code segment and multiple data segments.
large	Multiple code and data segments.
huge	Same as the large model, except that individual data items may be larger than a single segment.
flat	Protected mode. Uses 32-bit offsets for code and data. All data and code (including system resources) are in a single 32-bit segment.

NEAR and FAR Segments

- NEAR segment
  - requires only a 16-bit offset
  - faster execution than FAR
- FAR segment
  - 32-bit offset: requires setting both segment and offset values
  - slower execution than NEAR

.MODEL Directive

- The .MODEL directive determines the names and grouping of segments
- .model tiny
  - code and data belong to same segment (NEAR)
  - .com file extension
- .model small
  - both code and data are NEAR
  - data and stack grouped into DGROUP
- .model medium
  - code is FAR, data is NEAR

.MODEL Directive

- .model compact
  - code is NEAR, data is FAR
- .model huge & .model large
  - both code and data are FAR
- .model flat
  - both code and data are 32-bit NEAR

.MODEL Directive

- Syntax:

  .MODEL *type*, *language*, *stackdistance*

- *Language* can be:
  - C, BASIC, FORTRAN, PASCAL, SYSCALL, or STDCALL (details in Chapters 8 and 12).
- *Stackdistance* can be:
  - NEARSTACK: (default) places the stack segment in the group DGROUP along with the data segment
  - FARSTACK: stack and data are **not** grouped together

.STACK Directive

- Syntax:

  .STACK [*stacksize*]

- *Stacksize* specifies size of stack, in bytes
  - default is 1024
- Example: set to 2048 bytes:
  - .stack 2048

.CODE Directive

- Syntax:

  .CODE [*segname*]

  - optional *segname* overrides the default name
- Small, compact memory models
  - NEAR code segment
  - segment is named _TEXT
- Medium, large, huge memory models
  - FAR code segment
  - segment is named *modulename*_TEXT

Whenever the CPU executes a FAR call or jump, it loads CS with the new segment address.

Near Data Segments

- .DATA directive creates a Near segment
  - Up to 64K in Real-address mode
  - 16-bit offsets are used for all code and data
  - automatically creates segment named DGROUP
  - can be used in any memory model
- Other types of data:
  - .DATA? (uninitialized data)
  - .CONST (constant data)

Far Data Segments

- .FARDATA
  - creates a FAR_DATA segment
- .FARDATA?
  - creates a FAR_BSS segment
- Code to access data in a far segment:

```
 .FARDATA
myVar
 .CODE
    mov ax,SEG myVar
    mov ds,ax
```

The SEG operator returns the segment value of a label. Similar to @data.

Explicit Segment Definitions

- Use them when you cannot or do not want to use simplified segment directives
- All segment attributes must be specified
- The ASSUME directive is required

SEGMENT Directive

Syntax:

```
name SEGMENT [align] [combine] ['class']
    statements
name ENDS
```

- *name* identifies the segment; it can either be unique or the name of an existing segment.
- *align* can be BYTE, WORD, DWORD, PARA, or PAGE.
- *combine* can be PRIVATE, PUBLIC, STACK, COMMON, MEMORY, or AT *address*.
- *class* is an identifier used when identifying a particular type of segment such as CODE or STACK.

Segment Example

```
ExtraData SEGMENT PARA PUBLIC 'DATA'
    var1 BYTE 1
    var2 WORD 2
ExtraData ENDS
```

- name: ExtraData
- paragraph align type (starts on 16-byte boundary)
- public combine type: combine with all other public segments having the same name
- 'DATA' class: 'DATA' (load into memory along with other segments whose class is 'DATA')

ASSUME Directive

- Tells the assembler how to calculate the offsets of labels
- Associates a segment register with a segment name

Syntax:

```
ASSUME segreg:segname [,segreg:segname] ...
```

Examples:

```
ASSUME cs:myCode, ds:Data, ss:myStack
ASSUME es:ExtraData
```

Multiple Data Segments (1 of 2)

```
cseg SEGMENT 'CODE'
ASSUME cs:cseg, ds:data1, es:data2, ss:mystack

main PROC
    mov ax,data1                ; DS points to data1
    mov ds,ax
    mov ax,SEG val2             ; ES points to data2
    mov es,ax
    mov ax,val1                 ; data1 segment assumed
    mov bx,val2                 ; data2 segment assumed

    mov ax,4C00h                ; (same as .exit)
    int 21h
main ENDP
cseg ENDS
```

Multiple Data Segments (2 of 2)

```
data1 SEGMENT 'DATA'
    val1 WORD 1001h
data1 ENDS

data2 SEGMENT 'DATA'
    val2 WORD 1002h
data2 ENDS

mystack SEGMENT PARA STACK 'STACK'
    BYTE 100h DUP('S')
mystack ENDS

END main
```

Combining Segments

- Segments can be merged into a single segment by the linker, if . . .
  - their names are the same,
  - and they both have combine type PUBLIC,
  - . . . even when they appear in different source code modules
- Example:
  - cseg SEGMENT PUBLIC 'CODE'

What's Next

- Defining Segments
- **Runtime Program Structure**
- **Interrupt Handling**
- Hardware Control Using I/O Ports

Interrupt Handling

- Overview
- Hardware Interrupts
- Interrupt Control Instructions
- Writing a Custom Interrupt Handler
- Terminate and Stay Resident Programs

Overview

- Interrupt handler (interrrupt service routine) – performs common I/O tasks
  - can be called as functions
  - can be activated by hardware events
- Examples:
  - video output handler
  - critical error handler
  - keyboard handler
  - divide by zero handler
  - Ctrl-Break handler
  - serial port I/O

Interrupt Vector Table

- Each entry contains a 32-bit segment/offset address that points to an interrupt service routine
- Offset = *interruptNumber* * 4
- The following are only examples:

Interrupt Number	Offset	Interrupt Vectors
00-03	0000	02C1:5186 0070:0C67 0DAD:2C1B 0070:0C67
04-07	0010	0070:0C67 F000:FF54 F000:837B F000:837B
08-0B	0020	0D70:022C 0DAD:2BAD 0070:0325 0070:039F
0C-0F	0030	0070:0419 0070:0493 0070:050D 0070:0C67
10-13	0040	C000:0CD7 F000:F84D F000:F841 0070:237D

Hardware Interrupts

- Generated by the Intel 8259 Programmable Interrupt Controller (PIC)
  - in response to a hardware signal
- Interrupt Request Levels (IRQ, 中断请求优先级)
  - priority-based interrupt scheduler
  - brokers simultaneous interrupt requests
  - prevents low-priority interrupt from interrupting a high-priority interrupt

Common IRQ Assignments

IRQ	
0	System timer
1	Keyboard
2	Programmable Interrupt Controller
3	COM2 (serial)
4	COM1 (serial)
5	LPT2 (printer)
6	Floppy disk controller
7	LPT1 (printer)

Common IRQ Assignments

8	CMOS real-time clock
9	modem, video, network, sound, and USB controllers
10	(available)
11	(available)
12	mouse
13	Math coprocessor
14	Hard disk controller
15	(available)

Interrupt Control Instructions

- STI – set interrupt flag
  - enables external interrupts
  - always executed at beginning of an interrupt handler
- CLI – clear interrupt flag
  - disables external interrupts
  - used before critical code sections that cannot be interrupted
  - suspends the system timer

Writing a Custom Interrupt Handler

- Motivations
  - Change the behavior of an existing handler
  - Fix a bug in an existing handler
  - Improve system security by disabling certain keyboard commands
- What's Involved
  - Write a new handler
  - Load it into memory
  - Replace entry in interrupt vector table
  - Chain to existing interrupt hander (usually)

Get Interrupt Vector

- INT 21h Function 35h – Get interrupt vector
  - returns segment-offset addr of handler in ES:BX

```
.data
int9Save LABEL WORD
DWORD ?                   ; store old INT 9 address here
.code
mov ah,35h                          ; get interrupt vector
mov al,9                            ; for INT 9
int 21h                             ; call MS-DOS
mov int9Save,BX                     ; store the offset
mov [int9Save+2],ES                 ; store the segment
```

Set Interrupt Vector

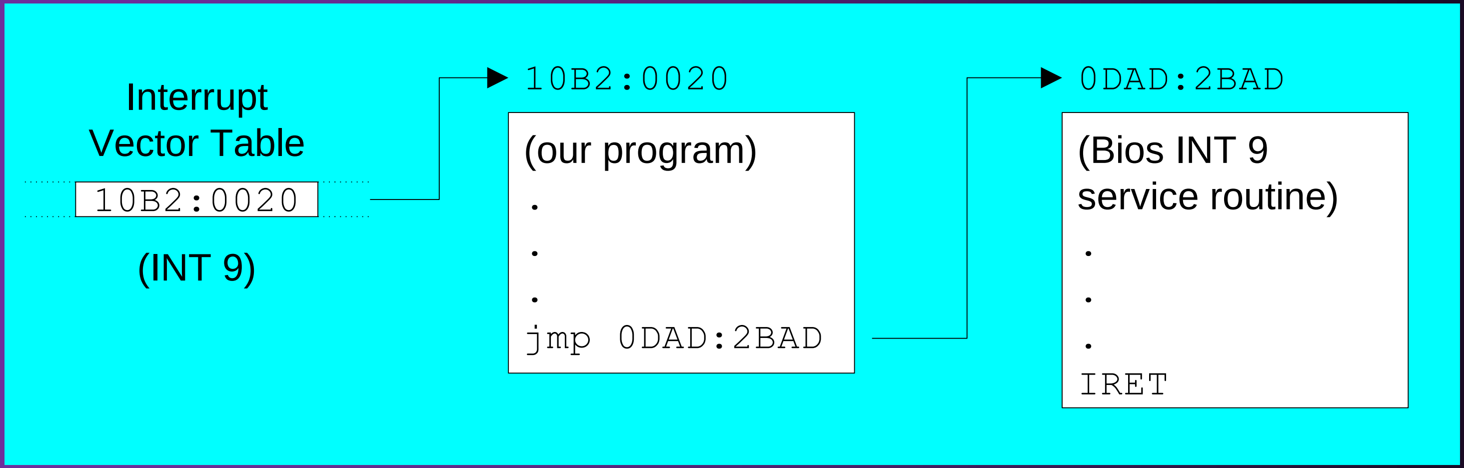
- INT 21h Function 25h – Set interrupt vector
  - installs new interrupt handler, pointed to by DS:DX

```
mov ax,SEG kybd_rtn            ; keyboard handler
mov ds,ax                      ; segment
mov dx,OFFSET kybd_rtn         ; offset
mov ah,25h                     ; set Interrupt vector
mov al,9h                      ; for INT 9h
int 21h
.
.
kybd_rtn PROC                  ; (new handler begins here)
```

See the CtrlBrk.asm program.

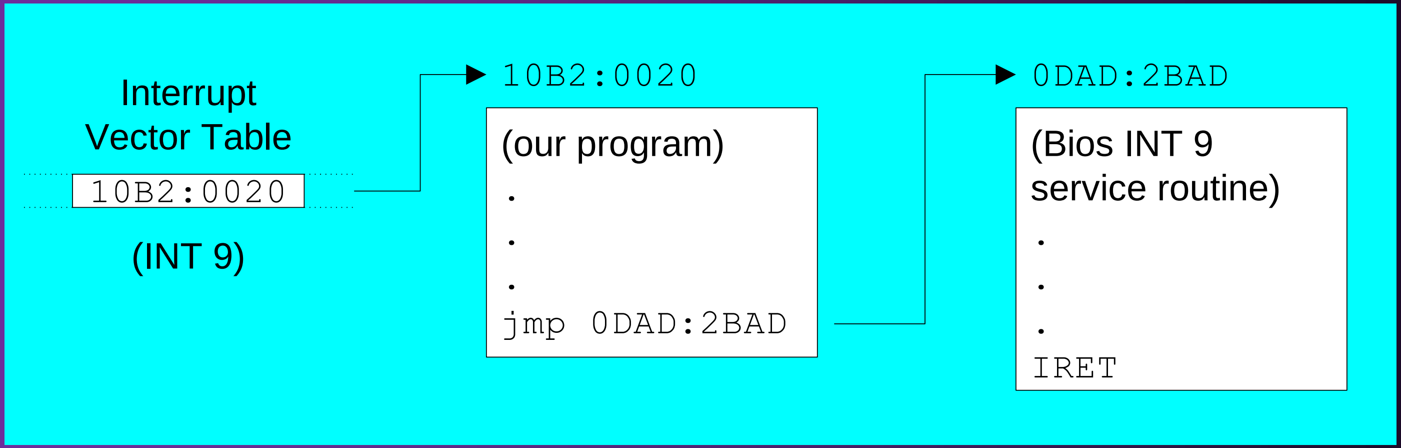
Keyboard Processing Steps

1. Key pressed, byte sent by hardware to keyboard port
2. 8259 controller interrupts the CPU, passing it the interrupt number
3. CPU looks up interrupt vector table entry 9h, branches to the address found there



Keyboard Processing Steps

4. Our handler executes, intercepting the byte sent by the keyboard
5. Our handler jumps to the regular INT 9 handler
6. The INT 9h handler finishes and returns
7. System continues normal processing



Terminate and Stay Resident Programs

- (TSR): Installed in memory, stays there until removed
  - by a removal program, or by rebooting
- Keyboard example
  - replace the INT 9 vector so it points to our own handler
  - check, or filter certain keystroke combinations, using our handler
  - forward-chain to the existing INT 9 handler to do normal keyboard processing

What's Next

- Defining Segments
- Runtime Program Structure
- Interrupt Handling
- **Hardware Control Using I/O Ports**

Hardware Control Using I/O Ports

- Two types of hardware I/O
  - memory mapped
    - program and hardware device share the same memory address, as if it were a variable
  - port based
    - data written to port using the OUT instruction
    - data read from port using the IN instruction

Input-Ouput Ports

- ports numbered from 0 to FFFFh
- keyboard controller chip sends 8-bit scan code to port 60h
  - triggers a hardware interrupt 9
- IN and OUT instructions:

```
IN accumulator, port
OUT port, accumulator
```

  - accumulator is AL, AX, or EAX
  - port is a constant between 0 and FFh, or a value in DX betweeen 0 and FFFFh

Summary (Chap 16)

- Working at the BIOS level gives you a high level of control over hardware
- Use INT 16h for keyboard control
- Use INT 10h for video text
- Use memory-mapped I/O for graphics
- Use INT 33h for the mouse

Summary (Chap 17)

- Explicit segment definitions used often in custom code libraries
- Directives: SEGMENT, ENDS, ASSUME
- Interrupt handlers, interrupt vector table
- Hardware interrupt, 8259 Programmable Interrupt Controller, interrupt flag
- Terminate and Stay Resident (TSR)
- Memory-mapped and port-based I/O

Principles of Assembly and Compilation
汇编与编译原理
44100593

王朝坤
THSS@Tsinghua

Part II: Principles of Compilation

王朝坤
IISE@Tsinghua

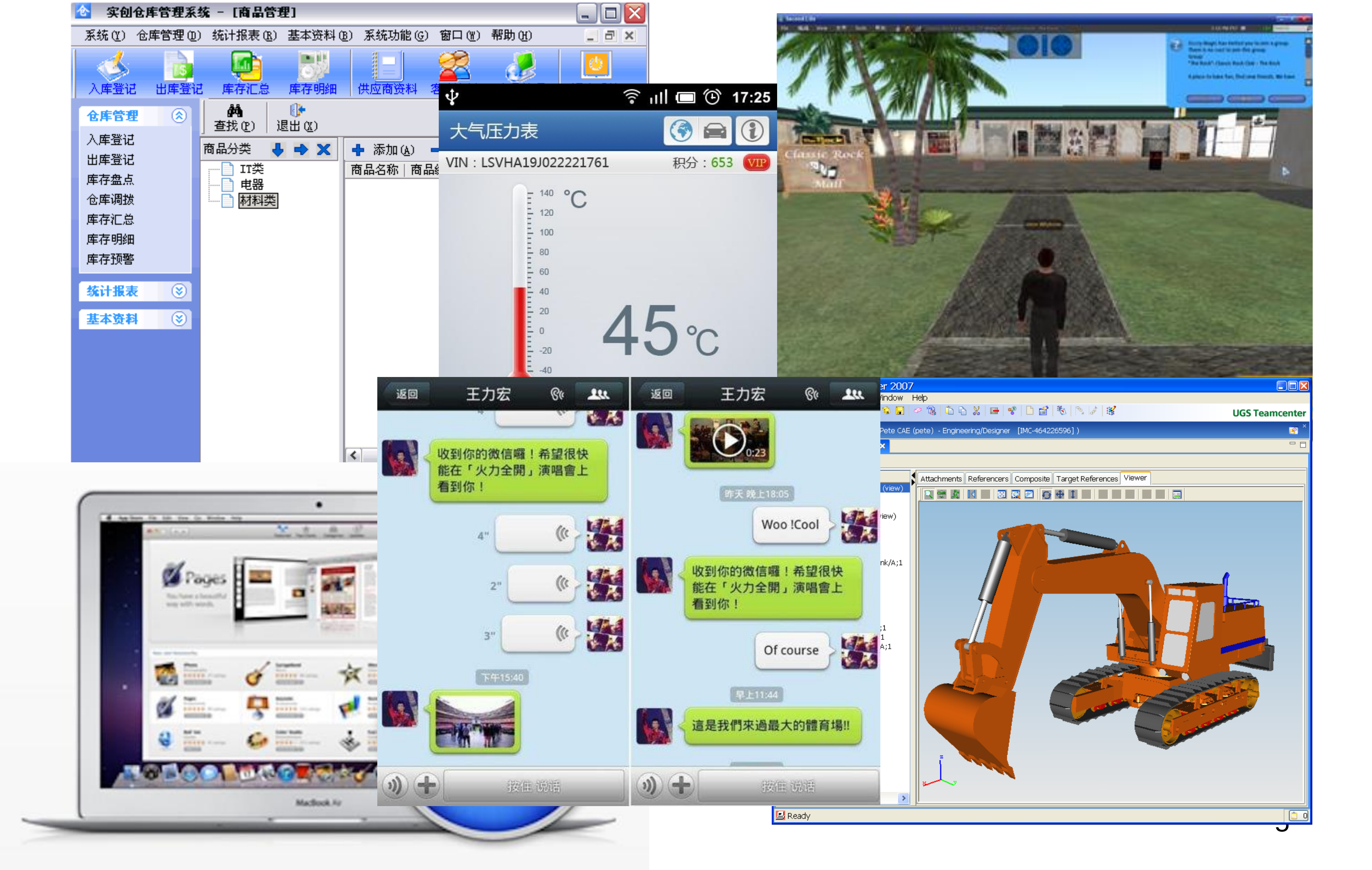
Chap. 1 Introduction

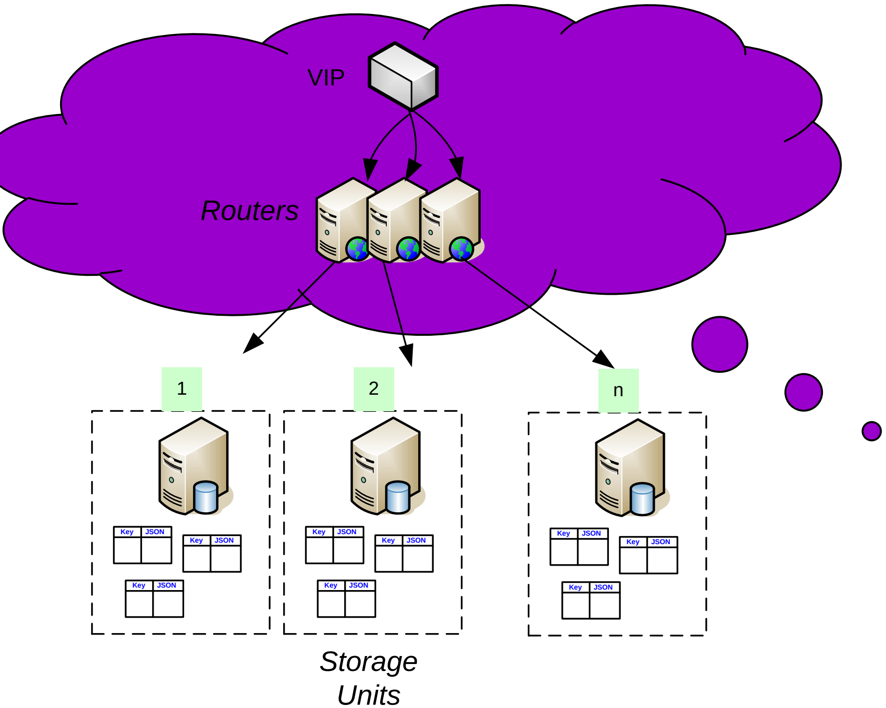
Chaokun Wang
IISE@Tsinghua

Outline

- **What***
  - 什么是编译和编译器
- **Why**
- **How**
  - **Info. of this Part**
- **PL****
  - 编译与解释的区别和联系
- **History**
- **Contents***
  - 编译各阶段的内容

1.1 Computer & Software





- **Pig Latin**
  - Pig Latin is a data flow language rather than procedural or declarative.
  - User code and existing binaries can be included almost anywhere.
  - Metadata not required, but used when available.
  - Support for nested types.
  - Operates on files in HDFS.

How It Works

Pig Latin

```
A = LOAD 'myfile'
    AS (x, y, z);
B = FILTER A by x > 0;
C = GROUP B BY x;
D = FOREACH A GENERATE
    x, COUNT(B);
STORE D INTO 'output';
```

pig.jar:

- parses
- checks
- optimizes
- plans execution
- submits jar to Hadoop
- monitors job progress

Execution Plan

Map:
  Filter

Reduce:
  Count

软件开发

- 程序开发工具？？
- 如何从编辑器中的文本到可执行程序的？？

A language-processing system

- gcc -E hi.c

  -E Preprocess only;
  do not compile,
  assemble or link

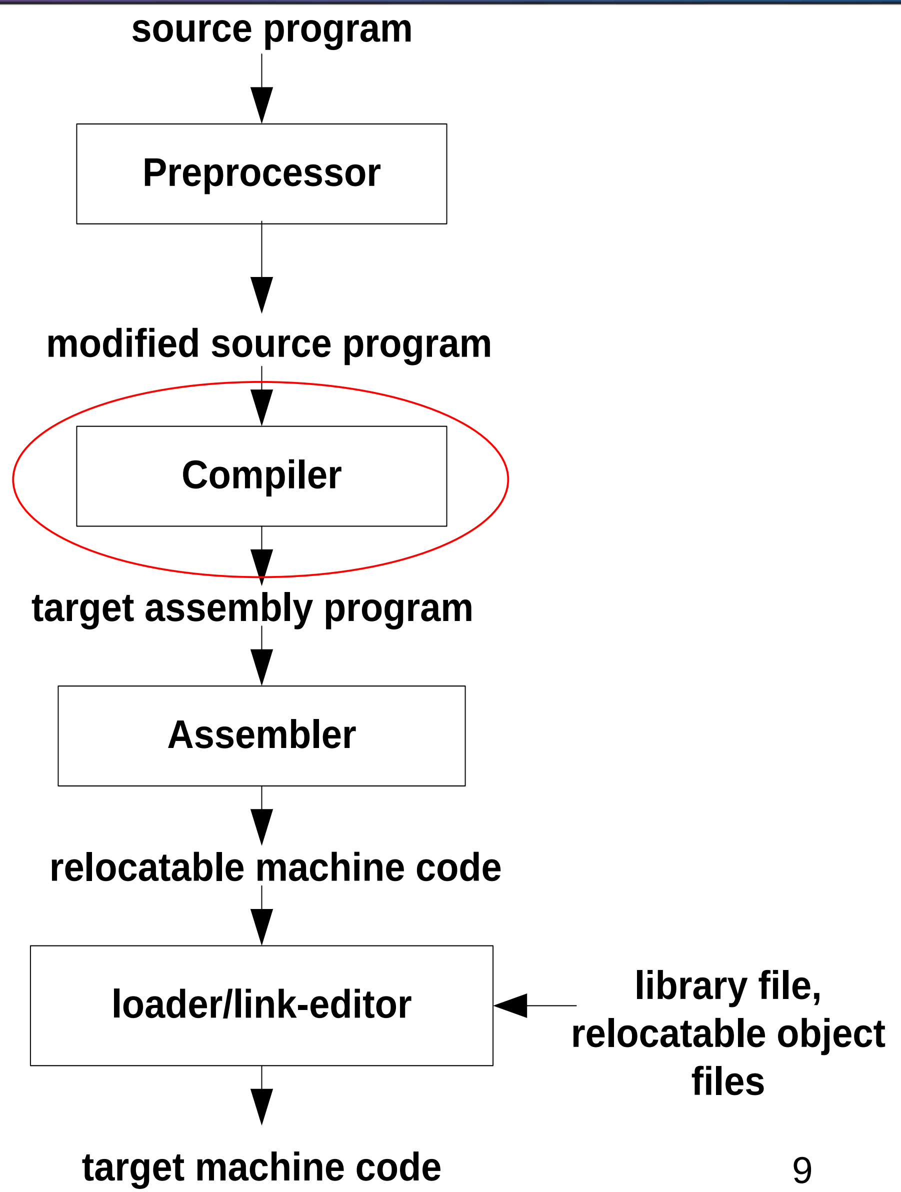
- gcc -S hi.c

  -S Compile only;
  do not assemble or link

- gcc -c hi.c

  -c Compile and assemble,
  but do not link

- gcc hi.c

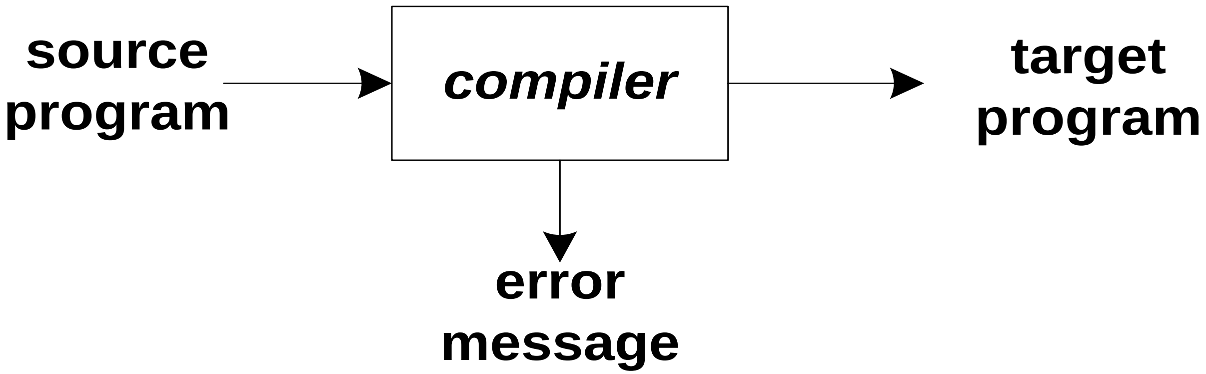


What is …

- **Compilation（编译）**
  - **Compilation is the process whereby one computer language is translated into another (usually simpler and more low-level i.e. machine orientated) language**
  - **Traditionally, programs in a high-level computer language (e.g. Pascal, C, Objective-C, Lisp) are compiled into assembly language (essentially machine code)**
- **Compiler（编译器）**
  - **The program that does the translation is known as a compiler.**

Compilers

- **Simply stated, a compiler is a program that reads a program written in one language — the source language — and translates it into an equivalent program in another language — the target language. As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.**



Preprocessors

- Preprocessors produce input to compilers. They may perform the following functions.
- Macro inclusion.
- File inclusion.
- “Rational” preprocessors.
- Language extensions.

Assemblers

- Some compilers produce assembly code that is passed to an assembler for further processing.
- Other compilers perform the job of the assembler, producing relocatable machine code that can be passed directly to the loader/link-editor.
- The relationship between assembly and machine code
  - Assembly code is mnemonic version of machine code,
  - names are used instead of binary codes for operations
  - names are also given to memory address.

Loaders and Link-Editors

- Usually, the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine.

- The linker (link-editor) resolves external memory addresses, where the code in one file may refer to a location in another file.

- The loader then puts together all of the executable object files into memory for execution.

1.2 Why is PoC?

- Why Learn it ?
  - Software Engineering/Computer Science
    - OS (shell)
    - DBS (sql, xquery)
    - WEB (wsdl, soap)
    - SE (basis)
  - Career
    - Programmer (javac, gcj)
    - Scientist (wm, osn)
- Why research it?
  - Live
    - 1956~
    - POPL (1973~)
    - cc, etc.
  - Impact
    - PLDI

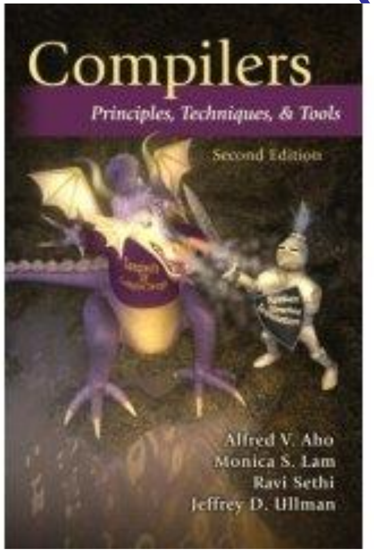
Why Study Compilers

- Excellent software-engineering example
  - theory meets practice.
- Essential software tool
- Influences hardware design
  - RISC
  - VLIW
- Tools (mostly "optimization") for enhancing software reliability and security
  - memory leak
- Applications
  - Pig Latin
  - JCVM
  - …

1.3 Textbooks for this Part

Compilers: Principles, Techniques and Tools (2ed)

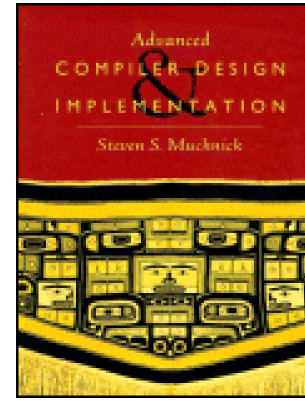
Alfred V. Aho,
Monica S. Lam,
Ravi Sethi,
Jeffrey D. Ullman
Addison-Wesley, 2006 (77, 86)
ISBN 0-32148-6811



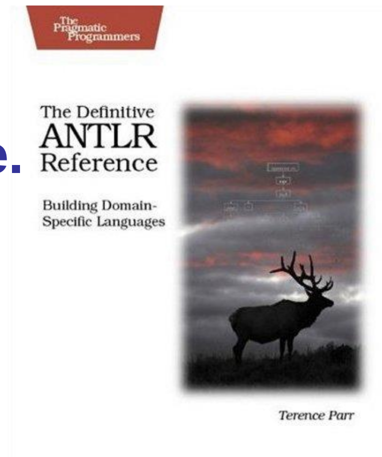
编译原理第2版.本科教学版

Alfred V. Aho, Monica S. Lam, Ravi Sethi,
Jeffrey D. Ullman
译者: 赵建华 郑滔 戴新宇
机械工业出版社, 2009.5
ISBN 9787111269298

Ref.



Advanced Compiler Design and Implementation
Steven Muchnick
Morgan Kaufman Publishers, 1997
ISBN 1-55860-320-4



The Definitive ANTLR Reference.
Terence Parr.
The Pragmatic Programmers, 2007
ISBN-10: 0-9787392-5-6

Lex and Yacc (2nd edition)
John R Levine, Tony Mason, Doug Brown
O'Reilly & Associates, October 1992
ISBN: 1-56592-0007

… …

Thank you!